Universidade de Brasília
Instituto de Ciências Exatas
Departamento de Ciência da Computação

GrAMoS: Serviço para a Monitoração de Acordos em Grid

Glauber Scorsatto

Dissertação apresentada como requisito parcial
para obtenção do grau de Mestre em Informática

Orientadora
Prof.ª Dr.ª Alba Cristina Magalhães Alves de Melo

Brasília
2007

Universidade de Brasília – UnB
Instituto de Ciências Exatas
Departamento de Ciência da Computação
Mestrado em Informática

Coordenador: Prof.ª Dr.ª Alba Cristina Magalhães Alves de Melo

Banca examinadora composta por:

Prof.ª Dr.ª Alba Cristina Magalhães Alves de Melo (Orientadora) – CIC/UnB
Prof. Dr. Henrique Mongelli – DCT/UFMS
Prof.ª Dr.ª Célia Ghedini Ralha – CIC/UnB

CIP – Catalogação Internacional de Publicação

Glauber Scorsatto.

GrAMoS: Serviço para a Monitoração de Acordos em Grid / Glauber Scorsatto. Brasília:UnB, 2007. 86 p. ; il. ; 29,5 cm.

Tese (Mestre) – Universidade de Brasília, Brasília, 2007.

1. *grid computing*, 2. qualidade de serviço, 3. serviço de monitoração

CDU 004

Endereço: Universidade de Brasília
Campus Universitário Darcy Ribeiro – Asa Norte
CEP 70910-900
Brasília – DF – Brasil

Universidade de Brasília
Instituto de Ciências Exatas
Departamento de Ciência da Computação

GrAMoS: Serviço para a Monitoração de Acordos em Grid

Glauber Scorsatto

Dissertação apresentada como requisito parcial
para obtenção do grau de Mestre em Informática

Prof.ª Dr.ª Alba Cristina Magalhães Alves de Melo (Orientadora)
CIC/UnB

Prof. Dr. Henrique Mongelli
DCT/UFMS

Prof.ª Dr.ª Célia Ghedini Ralha
CIC/UnB

Prof.ª Dr.ª Alba Cristina Magalhães Alves de Melo
Coordenadora do Mestrado em Informática

Brasília, 17 de setembro de 2007

Agradecimentos

Agradeço à minha família, principalmente a meus pais, por todo o apoio e força que me deram para a conclusão deste trabalho, sem os quais sei que isso não seria possível.

Agradeço também à Dra. Alba Cristina Magalhães Alves de Melo pela orientação que me foi dada e por me encorajar em momentos que o trabalho não parecia possível.

Agradeço ainda ao Dr. Pedro Vieira da Silva Filho, pela compreensão em relação às minhas conflitantes obrigações profissionais e acadêmicas.

Finalmente, agradeço a Deus, pois sei que sem Ele nada disso teria sido possível.

Resumo

Um sistema distribuído consiste em um conjunto de elementos computacionais que se comunicam através de uma rede de interconexão e que utilizam um software que permite que trabalhem de maneira cooperativa e coordenada. Visando o compartilhamento de recursos em larga escala, foi proposta a Computação em *Grid*, um tipo de sistema distribuído cujo objetivo é o compartilhamento de recursos geograficamente distribuídos, autônomos e pertencentes a diferentes domínios administrativos.

Algumas aplicações executadas em *grid* podem necessitar de garantias de disponibilidade de recursos. Para que tais aplicações sejam satisfeitas, faz-se necessário o fornecimento de garantias de Qualidade de Serviço (QoS). QoS consiste em um conjunto de características quantitativas e qualitativas de um sistema necessárias para que seja alcançado o nível de serviço esperado pelas aplicações. Um meio de se fazer a negociação dos níveis de QoS fornecidos às aplicações é a utilização de acordos de serviço, pelos quais a parte fornecedora se compromete a prover o nível de serviço desejado.

A presente dissertação propõe o GrAMoS (*Grid Agreement Monitoring Service*), um serviço capaz de monitorar acordos firmados em *grid*, assim como de detectar possíveis violações desses acordos e de tomar ações em caso de detecção de violações. A monitoração é feita em instantes randômicos dentro de intervalos fixos e as ações tomadas na situação de quebra de acordo são flexíveis e podem ser fornecidas pelo usuário. A análise do protótipo do GrAMoS, implementado no *Globus Toolkit* 4 (GT4), mostra que o overhead causado pelo monitoramento é baixo (abaixo de 2,8%), causando um impacto mínimo na aplicação de *grid*.

Palavras-chave: g*rid computing*, Qualidade de Serviço (QoS), serviço de monitoração.

Abstract

A distributed system consists in a set of computational elements, which communicate through an interconnected network, making use of software that allows the elements to work in a cooperative and coordinate way. Aiming a large scale resource sharing, Grid Computing has emerged, a kind of distributed system which has as objective the sharing of autonomous, geographically distributed resources, belonging to different administrative domains.

Some applications executed in a grid system may demand resource availability in some specific levels. For these applications to be satisfied, it is necessary to supply guarantees of Quality of Service (QoS). QoS consists of several quantitative and qualitative system characteristics which are necessary for the level of service demanded by applications to be achieved. A means of negotiating the QoS levels supplied for the applications is to use service agreements, by which the resource supplier compromises to supply the demanded level of service.

The present dissertation proposes GrAMoS (*Grid Agreement Monitoring Service*), a service capable of monitoring agreements firmed in a grid, which can detect agreement violations and take actions. In our service, monitoring is done at random moments inside fixed intervals and the actions taken in the case of agreement violations are flexible and can be provided by the user. The evaluation of the GrAMoS prototype, which was implemented on top of the *Globus Toolkit* 4 (GT4), shows that the monitoring overhead is low (below 2.8%), causing a very small impact on the grid applications.

Keywords: grid computing, Quality of Service (QoS), monitoring service.

Sumário

Lista de Figuras

Lista de Tabelas

Capítulo 1

Introdução

Os sistemas de computação vêm sofrendo uma evolução desde o seu surgimento. No período compreendido entre 1945 e por volta de 1985, os computadores eram normalmente muito grandes e tinham um alto custo, o que limitava bastante sua quantidade na maioria das organizações. Além disso, por não ser simples e efetiva a conexão entre computadores, operavam independentes uns dos outros, geralmente de forma centralizada.

Por volta de 1985, porém, com o surgimento dos microprocessadores, tornou-se viável a utilização de uma grande quantidade de computadores. O surgimento das redes de computadores de alta velocidade possibilitou então que diversas máquinas fossem interligadas de forma a trabalharem cooperativamente. Surgiram assim os chamados Sistemas Distribuídos [10].

Para que aplicações possam ser executadas em sistemas computacionais, é necessária a disponibilidade de recursos. A Qualidade de Serviço (QoS) é uma maneira de garantir às aplicações a disponibilidade dos recursos dos quais necessitam. QoS pode ser definida como o conjunto de características quantitativas e qualitativas necessárias para que seja alcançado o nível de serviço esperado pelos usuários de um sistema ou aplicação [5]. Para que aplicações possam ser executadas utilizando os recursos de que necessitam, são necessários mecanismos capazes de lhes dar garantias de QoS.

Um sistema em *Grid* é um tipo de sistema distribuído surgido na década de 1990 cujo objetivo é o compartilhamento de recursos autônomos, heterogênos, geograficamente distribuídos e pertencentes a diferentes domínios administrativos para a resolução de problemas de larga escala [41]. Para que tal objetivo seja alcançado, faz-se necessária a utilização de *software* que forneça funcionalidades básicas ao *grid*. Para tanto, foi desenvolvido um conjunto de ferramentas conhecido como *Globus Toolkit*, o qual se tornou um padrão *de facto* para a implementação de *grids*. A versão mais atual do *Globus Toolkit* é a versão 4, conhecida como GT4. O *grid* pode ser utilizado para a execução de uma diversidade de aplicações. Para um subconjunto delas, pode ser necessário o fornecimento de garantias de QoS.

Em sistemas de *grid*, normalmente, QoS é fornecida às aplicações na base do melhor esforço, ou seja, sem garantias de que os recursos necessários estarão disponíveis para a sua execução. Para algumas aplicações, porém, o fornecimento de QoS dessa forma não é suficiente. Torna-se necessária então a existência de mecanismos capazes de garantir QoS de forma determinística às aplicações de *grid*.

Dentre os gerentes de recursos em *grid* estudados, o *Job Submission Service* (JSS) oferece um mecanismo para a reserva de recursos às aplicações, mais especficamente de CPU [49]. As reservas são firmadas com a utilização de acordos de serviço do tipo WS-*Agreement* [27]. Garante, dessa forma, a QoS de forma determinística no início da execução das aplicações. Além disso, é o único sistema

dentre os estudados que utiliza o GT4 como *middleware* de *grid*. O JSS, porém, não faz o monitoramento dos acordos firmados. O monitoramento se faz importante devido às características altamente dinâmicas do *grid*, que podem causar oscilações na quantidade de recursos disponíveis às tarefas. Dessa forma, se faz importante a existência de um mecanismo capaz de detectar violações de acordos. É importante também que tal mecanismo seja capaz de executar ações em resposta às violações detectadas.

O objetivo da presente dissertação de mestrado é projetar e avaliar o *Grid Agreement Monitoring Service* (GrAMoS), um mecanismo de monitoração de acordos executado junto ao JSS. O serviço proposto possui três características básicas: (i) faz a monitoração periódica da utilização de recursos pelas aplicações utilizando a técnica de Amostragem Randômica Estratificada [18]; (ii) é flexível, permitindo que o usuário defina quais ações tomar caso seja detectada uma violação de acordo; e (iii) gera um baixo *overhead* para as tarefas em execução. A nosso conhecimento, esse é o primeiro serviço de monitoração de acordos que utiliza o GT4 e que monitora acordos do tipo WS-*agreement*.

Os resultados obtidos com a avaliação de um protótipo desenvolvido mostram que o *overhead* gerado para aplicações com tempo de execução superiores a 5 minutos, tendo um intervalo de amostragem definido em 5 segundos, é da ordem de 2 por cento do tempo de execução total da aplicação. Já para um intervalo de 10 segundos, o *overhead* é da ordem de 1 por cento do tempo total de execução. Para aplicações com tempo de execução de aproximadamente 1 minuto, os *overheads* gerados para intervalos definidos de 5 e 10 segundos são, respectivamente, 2.6 por cento e 1.3 por cento do tempo de execução. Os resultados mostram, assim, que o *overhead* relativo (porcentagem) gerado pelo mecanismo é praticamente constante para aplicações de maior duração. Os resultados mostram ainda que o *overhead* absoluto gerado pelo impacto na aplicação de cada verificação de cumprimento de um acordo é também praticamente constante, independente de quanto tempo o mecanismo leva para fazer a verificação.

O restante deste documento está organizado da seguinte maneira. O Capítulo 2 apresenta conceitos fundamentais sobre Qualidade de Serviço. O Capítulo 3 apresenta conceitos fundamentais sobre *Grid Computing* e ferramentas que garantem a infra-estrutura necessária para este tipo de sistema. O Capítulo 4 apresenta conceitos acerca de QoS relacionada a *Grid Computing* e também o estado da arte no que se refere a gerentes de recursos para o *grid*. No Capítulo 5 é apresentada a proposta de um mecanismo para a monitoração de acordos de serviço, verificação de violações dos acordos e tomada de ações na ocorrência de tais violações. O Capítulo 6 apresenta os resultados experimentais obtidos a partir da implementação e avaliação do mecanismo proposto. No Capítulo 7 são apresentadas as conclusões obtidas com o trabalho realizado e sugestões de trabalhos futuros.

Capítulo 2

Qualidade de Serviço

Diversas definições para o termo Qualidade de Serviço (*Quality of Service* - QoS) podem ser encontradas na literatura. De maneira genérica, QoS pode ser definida como o conjunto de características quantitativas e qualitativas que são necessárias para alcançar o nível de serviço esperado pelos usuários de um sistema ou aplicação [5]. Pode também ser definida como um conjunto de requisitos impostos por um usuário sobre o comportamento dos serviços providos a uma aplicação por uma camada de suporte inferior [9]. Outra definição possível é a de Bochmann et al. [5], que a define, no contexto de multimídia, como o conjunto de características quantitativas e qualitativas de um sistema distribuído de multimídia necessárias para atingir a funcionalidade requerida de uma aplicação.

Em suma, QoS é uma maneira de garantir ao cliente a disponibilidade dos recursos de que este necessita. Define características não funcionais de um sistema, afetando diretamente a produção dos resultados gerados pelo mesmo.

O termo foi inicialmente utilizado em referência à tecnologia de redes, e atualmente se expandiu para uma diversidade de recursos computacionais, tais como memória, ciclos de processador, entre outros.

Uma abordagem para o fornecimento de QoS a aplicações é a realização de acordos de serviço em *Web service* (WS-*Agreement*) [27], os quais consistem em acordos entre a parte responsável pelos recursos utilizados para a execução e a parte cliente, de forma a garantir a disponibilidade dos recursos necessários. Por tais acordos, a parte cliente expressa os requisitos de QoS da aplicação a ser executada e a parte fornecedora se compromete a prover os recursos necessários à execução.

2.1. Especificação de QoS

O papel da especificação de QoS é a definição dos níveis de QoS exigidos pelas aplicações. A especificação pode ser feita com o uso de interfaces configuráveis ou de linguagens declarativas, como a QoS Modeling Language – QML [2]. Os aspectos da QoS podem ser divididos em várias categorias, as quais são caracterizadas por dimensões[3], nas quais são definidos valores para os diferentes aspectos de QoS. Uma dimensão pode ser definida como um atributo quantitativo ou qualitativo de uma categoria de QoS, sendo um atributo quantitativo especificado em função de alguma métrica (e.g. taxa de transmissão de dados na rede: 1 Mb/s), e um atributo qualitativo especificado de maneira descritiva (e.g. política de segurança: política A). Um exemplo de dimensão para a categoria *desempenho* é o tempo de resposta.

A definição de uma especificação de QoS geralmente é feita de forma declarativa, ou seja, as aplicações apenas informam o que é requerido por estas e, de

acordo com o princípio da transparência, toda a complexidade das camadas de QoS inferiores é mantida de forma transparente às aplicações. Dessa forma, pode-se espeficar apenas o que é requerido, sem que seja necessário definir como satisfazer os requerimentos.

Uma abordagem para o uso de especificações é a sua utilização em tempo de execução do sistema, de forma que os componentes conheçam a QoS demandada e que possam manipular e negociar os parâmetos de QoS especificados.

Dentre os aspectos que podem ser especificados, pode-se exemplificar [3]:

- Desempenho: parâmetros relativos ao desempenho dos serviços prestados. Engloba fatores como tempo de resposta, latência, e *throughput*;
- Confiabilidade: parâmetros relativos a falhas, tais como tempo necessário para se recuperar de uma falha, o tempo previsto para a ocorrência de uma falha e a quantidade de falhas esperadas;
- Segurança: diz respeito à segurança dos dados, tais como encriptação e acesso autenticado a dados;
- Nível de serviço: define o grau de dedicação exigido dos recursos utilizados pelas aplicações, como recursos dedicados ou compartilhados;
- Custo: especificação dos valores a serem pagos para que aplicações tenham a garantia de um determinado nível de QoS.

2.1.1. Camadas de especificação

Um sistema com suporte à QoS pode ser dividido em algumas camadas de abstração, cada uma delas incluindo diferentes serviços [6]. Segundo Jingwen e Nahrstedt [4], especificações feitas em camadas diferentes apresentam funcionalidades bastante distintas. As camadas de abstração em que pode-se dividir um sistema são [4, 6]: camada de usuário, de aplicação e de recursos do sistema (Figura 1).

Figura 1: Camadas de especificação de QoS.

Camada de Usuário

A camada de usuário é a camada mais acima na estrutura, como pode ser observado na figura 1. Nesta camada, a QoS é geralmente descrita com relação às características do sistema de acordo com a percepção humana [6].

Os parâmetros de QoS devem ser definidos por um usuário, de maneira bem abstrata. É necessário que seja provida uma interface entre o usuário e a aplicação para que esse possa definir os parâmetros da forma desejada [4].

É importante, porém não fundamental, que a camada de usuário forneça mecanismos para a definição da qualidade dos serviços a serem utilizados [4]. Os mecanismos devem ser apresentados ao usuário de forma simples, uma vez que não se espera que este possua conhecimentos específicos da área de QoS.

Camada de aplicação

A camada de aplicação é uma camada intermediária, localizada entre as camadas de usuário e de recursos. É responsável por traduzir os parâmetros recebidos pela camada de usuário em especificações de QoS para a aplicação.

Nesta camada, a QoS é descrita com parâmetros orientados à aplicação, os quais são resultados dos requisitos do usuário e das características da aplicação [6]. É importante notar que uma especificação de QoS nesta camada é dependente apenas da aplicação, sendo totalmente independente das camadas de *hardware* sobre as quais a aplicação é executada [4].

Uma especificação de QoS nesta camada pode ser avalidada de acordo com os seguintes parâmetros [4]:

- Expressividade: deve ser capaz de especificar uma grande variedade de serviços, os recursos necessários e as regras de adaptação correspondentes;

- Declaratividade: uma especificação deve ser declarativa, de modo que as aplicações não necessitem tratar de diversos mecanismos complexos de gerência de recursos necessários para garantir a QoS requerida;

- Independência: especificações devem ser desenvolvidas independentes dos códigos funcionais para propósitos de legibilidade e facilidade de desenvolvimento e manutenção. Permite também que uma aplicação seja associada a diferentes especificações em momentos diferentes;

- Extensibilidade: relaciona-se à facilidade de se estender uma especificação para incluir novas dimensões;

- Reusabilidade: capacidade de uma especificação de ser reusada por outra especificação. É necessário, porém não suficiente, que uma especificação apresente independência para que possa ser reusada.

Os parâmetros definidos nesta camada podem ser de dois tipos: específicos de desempenho e específicos de comportamento [4]. Parâmetros do primeiro tipo expressam características quantitativas do sistema, tais como taxa de transmisão de dados e nível de segurança. Já os do segundo tipo expressam características qualitativas, como qual comportamento adotar em uma situação de escassez de recursos.

Camada de recursos de sistema

Depois de definidas as especificações da camada de aplicação, é necessário que essas sejam traduzidas em especificações mais concretas e dependentes de *hardware* [4], como quais políticas de alocação de recursos devem ser utilizadas e quais desses recursos devem ser utilizados. Tal tradução ocorre entre as camadas de aplicação e de recursos de sistema.

A camada de recursos de sistema é a camada mais inferior na estrutura de uma arquitetura de QoS. As especificações de QoS nesta camada são totalmente dependentes do *hardware* utilizado. Os parâmetros de QoS nesta camada descrevem os requisitos de comunicação e de sistema operacional que devem ser adotados. Tais parâmetros devem ser definidos quantitativamente, quando a utilização dos recursos for mensurável (e.g. *clock* de processador), e qualitativamente, quando se definir um comportamento esperado (e.g. política de alocação de recursos) [6]. Em relação aos requisitos de comunicação, os parâmetros de QoS devem ser definidos em termos de desempenho e carga da rede; já em relação ao sistema operacional, os parâmetros de QoS devem ser definidos de acordo com os requisitos de utilização de recursos para a execução de tarefas, tais como CPU, acesso a dispositivos e à memória principal [6].

Especificações de QoS dessa camada podem ser classificadas em dois grupos de acordo com sua granulosidade: granulosidade grossa e granulosidade fina, sendo que para o primeiro grupo é esperada apenas uma meta-especificação, enquanto que para o segundo são necessárias descrições concretas dos recursos utilizados [4].

Em especificações de granulosidade grossa os parâmetros são definidos de maneira abstrata. Podem ser especificados quais os recursos devem ser utilizados, sem que seja necessário especificar quando os recursos devem ser alocados, ou como o sistema deve agir caso não se possa atingir a quantidade de recursos necessária.

Em especificações de granulosidade fina são esperadas descrições quantitativas e qualitativas dos requisitos, além de parâmetros de utilização dos recursos com relação ao tempo, como quando e por quanto tempo os recursos devem ser alocados e regras de adaptação [4].

2.1.2. Linguagens de especificação de QoS

Existem diversas linguagens para a especificação de QoS a nível de aplicação. Dentre elas temos [4]: *Quality of Service Modeling Language* – QML e *Hierarchical QoS markup Language* - HQML.

2.1.2.1. *Quality of Service Modeling Language* – QML

A QML [12] é uma linguagem para a especificação de QoS para componentes em sistemas de objetos distribuídos. É uma linguagem de propósito geral, não estando ligada a nenhum domínio em particular, tais como sistemas de tempo real ou sistemas multimídia, e tampouco a quaisquer categorias de QoS, como confiabilidade ou desempenho [12]. Para tanto, certos requisitos devem ser satisfeitos [2].

Primeiramente, especificações de QoS devem ser sintaticamente separadas de outras partes de especificações de serviços, o que permite que diferentes propriedades de QoS sejam especificadas para diferentes implementações de uma mesma interface.

Em segundo lugar, deve ser possível especificar separadamente as propriedades de QoS requeridas pelo cliente e as providas pelo servidor, de forma que exista uma especificação do que é requerido para o cliente e uma outra do que é provido pelo servidor. Isto permite que sejam especificadas características de QoS providas e requeridas por um componente, sem que seja especificada uma interconexão entre componentes. Tal característica é fundamental para a especificação de atributos de QoS de componentes que são reutilizados em diferentes contextos [2].

Além dos pontos destacados acima, deve existir uma maneira de determinar se a especifiação de QoS de um serviço provido satisfaz os requisitos de um cliente, requisitos estes que são uma conseqüência direta da separação de especificações [2].

A QML também suporta o refinamento de especificações de QoS, o que é atingido pelo mecanismo de herança entre interfaces através do suporte a tal conceito da orientação a objetos pela linguagem.

Por fim, a QML permite a definição de propriedades de granulosidade fina, tais como especificações de QoS para interfaces, atributos, parâmetros de operação e resultados de operações [2].

Contratos e perfis

Quanto a mecanismos de abstração para especificações de QoS, a QML possui três principais: tipo de contrato, contrato e perfil [2, 12]. A relação entre tais conceitos é ilustrada na Figura 2.

Em QML, um tipo de contrato é uma especificação das dimensões de QoS que podem ser utilizadas para caracterizar determinado aspecto de QoS [4], sendo uma dimensão formada por um nome e um determinado valor pertencente a um certo domínio. Cada dimensão especificada é classificada como numérica, conjunto ou enumeração, de acordo com seu domínio de valores, sendo enumerações e conjuntos domínios definidos pelo usuário [12]. A figura 2 mostra a definição dos tipos de contrato *Reliability* e *Performance*, cada qual com suas respectivas dimensões especificadas.

```
type Reliability = contract {
  NumberOfFailure: decreasing numeric no/year;
  TTR: decreasing numeric sec;
  Availability: increasing numeric;
};
type Performance = contract {
  delay: decreasing numeric msec;
  throughput: inceasing numeric mb/sec;
};

systemReliability = Reliability contract {
  numberOfFailures < 10 no/year;
  TTR {
       Percentile 100 < 2000;
       Mean < 500;
       Variance < 0.3
  };
  availability > 0.8;
};

ServerProfile for ServiceInterface = profile {
  require systemReliability;
  from operation1 require Performance contract {
       delay {
         percentile 50 < 10 msec;
         percentile 80 < 20 msec;
       };
  };
  from operation2 require Performance contract {
       delay < 4000 msec
  };
}
```

Figura 2: Exemplo de contratos e perfis escritos em QML [4].

Um contrato é uma instância de um tipo de contrato, podendo definir limitações para todas ou algumas dimensões especificadas no tipo de contrato instanciado. Representa uma especificação em particular dentro de determinada categoria de QoS. Os contratos são usados para a captura de especificações para interfaces, sendo que uma interface pode estar associada a um ou mais contratos, uma para cada categoria de QoS pertinente [12]. Pode ser vista na Figura 2 a definição do contrato *SystemReliability*, o qual é uma instância do tipo de contrato *Reliability*, sendo especificadas restrições às suas dimensões.

Um perfil descreve uma associação de contratos com interfaces e operações [4]. Enquanto uma interface define operações e atributos exportados por um serviço, um perfil descreve as propriedades de QoS de um serviço. Um perfil é definido com relação a uma determinada interface e especifica contratos para os atributos e operações descritos na interface. Múltiplos perfis podem ser definidos para uma única interface, o que é necessário para que uma mesma interface possa ter múltiplas implementações com diferentes propriedades de QoS [12]. A Figura 2 mostra a definição do perfil *ServiceProfile*, o qual associa a interface *ServiceInterface* com os contratos previamente

definidos, sendo a associação feita para o serviço em geral com o contrato *SystemReliability* e para operações específicas com o contrato *Performance*.

Por suportar o conceito de herança, a QML permite o refinamento de estruturas da linguagem [4]. Um contrato pode ser declarado como um refinamento de outro, assim como um perfil pode ser declarado como um refinamento de outro perfil. Novos parâmetros podem ser definidos para o refinamento, assim como também podem ser sobrescritos parâmetros definidos previamente. Exemplos de refinamento de contratos e perfis são ilustrados na Figura 3. Na figura, é definido o contrato *B*, o qual herda do também definido contrato *A*, redefinindo o parâmetro *failureMasking*. Ainda na figura 3, são também definidos os perfis P1 e P2, sendo P2 um refinamento de P1.

```
A = Reliability contract {
   failureMasking <= {omission, response};
   dataPolicy == valid;
   availability > 99.9%;
};
B = A refined by {
   failureMasking <= {omission};
   ...
};

P1 for I1 = profile {
   require D1;
   from E require C1;
};
P2 for I2 = P1 refined by {
   require D2;
   from E require C2;
};
```

Figura 3: Exemplo de refinamento de contratos e perfis em QML [19].

2.1. 2.2. *Hierarchical QoS Markup Language* – HQML

A HQML é uma liguagem para a definição de QoS baseada no padrão XML, definindo um conjunto de *tags* relevantes da QoS para aplicações multimídia [4]. Apesar de classificada por [4] como uma linguagem de QoS em nível de aplicação, a HQML pode ser utilizada para a especificação de QoS nas camadas de usuário, aplicação e recursos de sistema [13].

No nível do usuário, a linguagem apresenta *tags* para a definição de parâmetros qualitativos de QoS, quantias monetárias a serem cobradas pelos serviços oferecidos e modelos de quantias monetárias aos quais o serviço se adapta. Especificações neste nível são utilizadas em tempo de execução para verificar qual o padrão no qual o serviço melhor se encaixa em termos de condição econômica, nível de QoS preferido e níveis de QoS disponibilizados por diferentes provedores [13].

Em nível de aplicação, a linguagem provê *tags* para a especificação de tipos de parâmetros de QoS relativos a este nível, e de políticas de QoS (tais como regras de adaptação do sistema) específicas de aplicação. As especificações feitas neste nível são utilizadas por entidades *middleware*, ou *proxies* de QoS, para configurar e reforçar a QoS em favor da aplicação.

No nível do sistema, são providas *tags* para a especificação de diferentes requisitos de recursos. Se os serviços de sistema operacional e rede estiverem disponíveis, os *proxies* de QoS podem iniciar a reserva dos recursos requisitados em favor da aplicação de acordo com as especificações de QoS deste nível [13]. Em tempo de execução, os *proxies* escolhem o melhor mapeamento entre as especificações de QoS neste nível e os recursos disponíveis, sendo o melhor encaixe a configuração de QoS que é capaz de ser provida pelo sistema e que tenha a especificação mais alta a nível de usuário [14].

A HQML possui um módulo chamado HQML *Executor*, o qual realiza a tradução das especificações HQML nas estruturas de dados necessárias e coopera com os *proxies* de QoS para prover aplicações multimídia integradas com QoS [13].

Por ser uma linguagem baseada em XML, a HQML possui um grande poder de extensibilidade, podendo-se facilmente realizar a inclusão de novos parâmetros de QoS [4]. Ainda segundo [4], a linguagem possui uma boa declaratividade, expressividade e independência, não possuindo porém mecanismos para o reuso de especificações pré-existentes.

```
<AppConfig id = "1">
  <ServerCluster>
    ...
  </ServerCluster>
  <ClientCluster>
    <Client type = "required">
      <Hardware>  Pentium PC 500 </Hardware>
      <Software>  Windows 2000 </Software>
      ...
    </Client>
  </ClientCluster>
..<LinkList>
    <Link type = "FixedLink">
      <Start> Server </Start>
      <End> Client </End>
      ...
  </LinkList>
  <ReconfigRuleList>
    <ReconfigRule>
      <Condition type = "Bandwidth"> very low </Condition>
      <ReconfigAction type = "switch to"> 2 </ReconfigAction>
    </ReconfigRule>
  </ReconfigRuleList>
</AppConfig>
```

Figura 4: Exemplo de especificação de QoS em HQML [13].

Um exemplo de especificação na HQML pode ser visualizado na Figura 4. Na figura, a especificação é iniciada com a tag <*AppConfig*>, possuindo um atributo *id* associado, e terminada com a tag <*/AppConfig*>. Dentro da especificação são inseridas tags, tais como <*ServerCluster*> e <*ClientCluster*>, para a definição dos requisitos de QoS. São definidas também regras de adaptação a determinadas condições utilizando-se as tags <*ReconfigRuleList*> e <*/ReconfigRuleList*>.

2.2. Arquiteturas de QoS

Arquiteturas de QoS são responsáveis pela integração de mecanismos de QoS em sistemas computacionais de modo a organizar os recursos providos pelo sistema de maneira consistente com o intuito de satisfazer os requisitos impostos pelo usuário da aplicação. São responsáveis por preencher a lacuna entre os protocolos de reserva de recursos, os quais se situam em baixo nível, e a camada de aplicação [9]. Descrevem uma camada *middleware* que provê às aplicações mecanismos para a especificação de QoS.

Uma arquitetura de QoS pode ser definida pelos seguintes princípios [1]: integração, separação, transparência e desempenho. Segundo o princípio da integração, a QoS deve ser configurável, previsível e possível de ser mantida em todos os módulos (componentes de hardware) pelos quais passa um fluxo de execução. O da separação diz que a transferência, controle e gerência dos dados são atividades distintas dentro de uma arquitetura de QoS, e devem ser realizadas por diferentes componentes arquiteturais. De acordo com o princípio da transparência, a complexidade das camadas de QoS inferiores deve ser totalmente transparente às aplicações. Por último, o princípio do desempenho define um conjunto de técnicas e regras para a construção de sistemas de comunicação dirigidos a QoS.

Como exemplos de arquiteturas para o provimento de QoS, são detalhadas as arquiteturas *QoS-Architecture* (QoS-A) [8] e *Quartz* [11].

2.2.1. QoS-A

A *QoS Architecture*, QoS-A, é uma arquitetura de serviços e mecanismos divididos em camadas para gerência e controle de fluxos de dados contínuos em redes multiserviços [8]. Incorpora as seguintes noções chave [1]:

- Fluxos, os quais caracterizam a produção, transmissão e consumo de fluxos de mídia com uma QoS associada;
- Acordos de serviço, os quais são acordos entre os usuários e os provedores sobre os níveis de QoS fornecidos;
- Gerência de fluxos, o que provê mecanismos para a monitoração e garantia dos níveis de QoS contratados.

Em termos funcionais, a QoS-A é composta de camadas e planos [8]. A camada mais superior é uma plataforma para aplicações distribuídas, a qual contém serviços para o provimento de comunicações multimídia e especificação de QoS em um ambiente baseado em objetos. Logo abaixo, existe uma camada de *orchestration*, a qual possui mecanismos para sincronização de multimídia e controle de *jitters*. Mais abaixo existe uma camada de transporte, a qual possui mecanismos e serviços de QoS configuráveis. Ainda mais abaixo existem as camadas de rede, *data link* e física, as quais dão suporte à QoS de ponta-a-ponta.

Nesta arquitetura, a gerência de QoS é realizada em três planos verticais: protocolo, manutenção e gerência de fluxo. O plano de protocolo, formado por subplanos de usuário e controle, é motivado pelo princípio da separação. O plano de manutenção contém gerentes de QoS específicos de cada camada. Cada um deles é reponsável pela monitoração de granulosidade fina e manutenção de suas entidades de protocolos associadas. O plano de gerência de fluxo é responsável pelo estabelecimento de fluxos, mapeamento de QoS e escala de QoS.

A Figura 5 ilustra a configuração das camadas e planos da QoS-A.

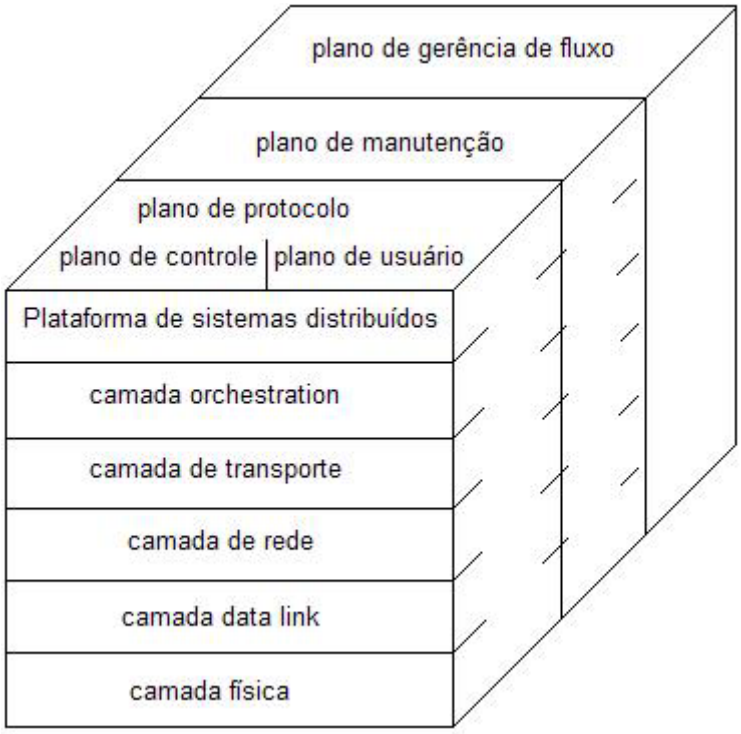

Figura 5: A arquitetura QoS-A [8].

2.2.2. A arquitetura *Quartz*

Quartz é uma arquitetura baseada em um projeto flexível, extensível e independente de plataforma, o que permite que seja usada em diferentes áreas de aplicação e em conjunto com diferentes protocolos de reserva de recursos [9]. O

principal objetivo considerado em seu desenvolvimento foi prover suporte a ambientes heterogêneos [11]. Logo, a arquitetura deve ser capaz de lidar com os diversos tipos de *hardware* e protocolos encontrados em um sistema distribuído heterogêneo.

Para poder lidar com a heterogeneidade dos ambientes distribuídos, não é suficiente que a arquitetura seja portável para diferentes plataformas, mas também que possa se adaptar a mudanças ocorridas nos sistemas de reserva de recursos de camadas inferiores sem que seja necessária qualquer recompilação. Para atingir esta capacidade, a arquitetura possui um projeto baseado em componentes, de maneira que componentes relacionados a diferentes mecanismos de reserva podem ser ligados à arquitetura de forma dinâmica [9]. Além disso, para prover suporte para novos protocolos de reserva, um novo componente pode ser desenvolvido de modo a suportar tal protocolo.

A arquitetura possui mecanismos para o mapeamento entre diferentes camadas de abstração, mais especificamente entre as camadas de aplicação e de sistema [9]. Uma vez que a arquitetura se propõe a ser utilizada em diferentes áreas de aplicação, é necessário que tal mecanismo seja capaz de interpretar um conjunto potencialmente infinito de parâmetros de QoS [11]. Para tanto, é necessário que os parâmetros sejam traduzidos em parâmetros que possam ser internamente reconhecidos pela arquitetura. Tal tradução é realizada em três passos [9]. Primeiramente, os parâmetros específicos da aplicação são traduzidos em um conjunto de parâmetros genéricos no nível da aplicação, os quais são reconhecidos pela arquitetura. Tais parâmetros são então traduzidos em um conjunto de parâmetros genéricos a nível de sistema, e são balanceados entre rede e sistema operacional. Por útimo, são traduzidos em parâmetros de sistema específicos que podem ser entendidos pelos protocolos de reserva de recursos presentes no sistema. Uma relação de todos os parâmetros genéricos reconhecidos pela arquitetura pode ser encontrada em [9]. Dessa forma, é contemplado o princípio da transparência, uma vez que os parâmetros da camada de sistema tornam-se transparentes em relação à aplicação.

2.2.2.1. Componentes da arquitetura

Cada componente definido na arquitetura Quartz encapsula uma tarefa em particular no problema da especificação de QoS em um ambiente heterogêneo. A arquitetura pode ser facilmente adaptável a diferentes ambientes pela troca de componentes [11].

Seu principal componente é o *QoS Agent*, que é responsável pela implementação dos mecanismos necessários para a provisão de serviços com a qualidade exigida por um usuário, o que envolve duas tarefas principais: a tradução de parâmetros de QoS entre diferentes camadas de abstração e a interação com os protocolos de reserva de recursos presentes no sistema [9]. Tal componente é composto por uma Unidade de Tradução e múltiplos Agentes de Sistema, os quais são associados aos protocolos de reserva responsáveis pela administração dos recursos providos pelo sistema.

A unidade de tradução contém filtros de QoS e um interpretador de QoS [9]. Os filtros podem ser divididos em filtros de aplicação e de sistema, sendo cada tipo responsável pela tradução entre parâmetros genéricos e específicos de suas respectivas

camadas de abstração. O interpretador é responsável pelo mapeamento entre os parâmetros genéricos da camada de aplicação e da camada de recursos de sistema, realizando também o balanceamento da utilização de recursos entre rede e sistema operacional [11].

Por último, os Agentes de sistema recuperam os valores dos parâmetros de QoS providos pela Unidade de Tradução e realizam a reserva de recursos necessária utilizando o protocolo de reserva de recursos correspondente.

A estrutura da arquitetura Quartz pode ser visualizada na Figura 6.

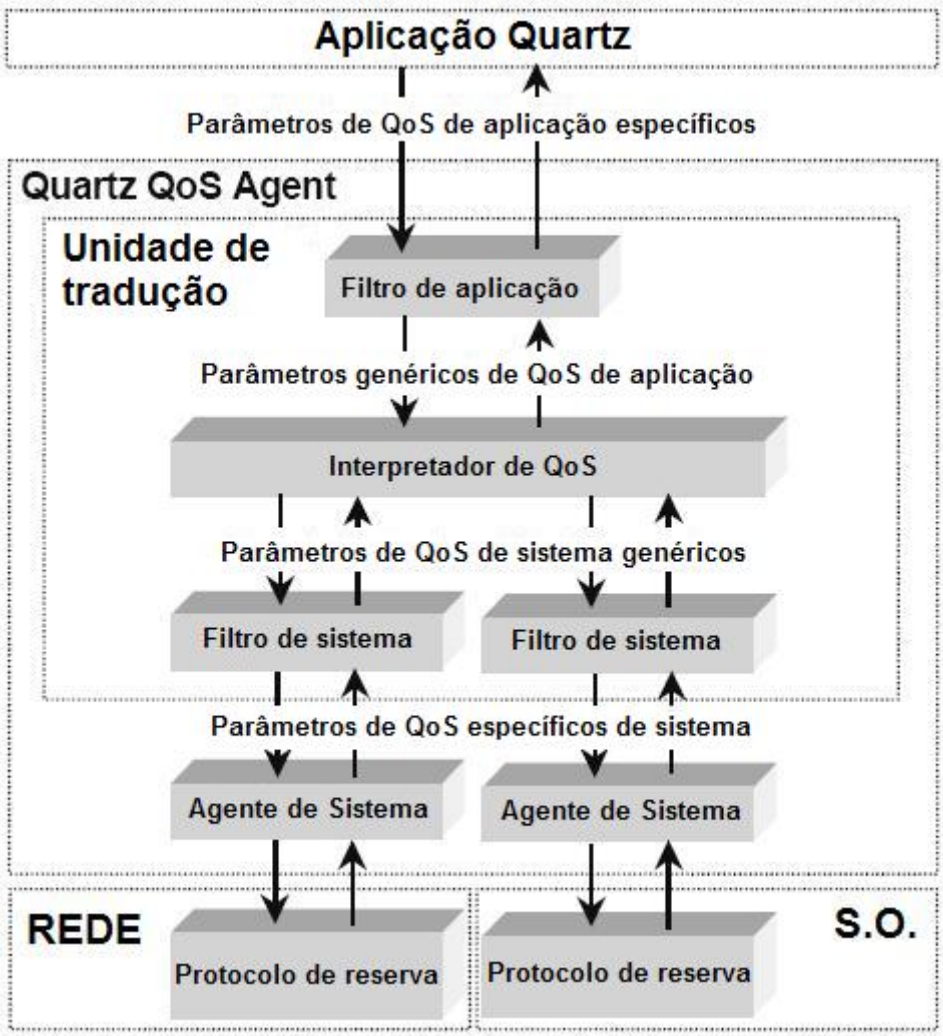

Figura 6: Estrutura da arquitetura Quartz [11].

2.3. QoS em nível de CPU

Em relação à *Central Processing Unit* (CPU), QoS está relacionada basicamente à reserva de uma certa quantidade de tal recurso para a execução de determinada aplicação, de modo a satisfazer seus requisitos de QoS. Mecanismos que possibilitam a reserva devem ser providos pelo sistema operacional. A reserva de CPU

isola a execução de certa aplicação de outras aplicações também executando no sistema, da mesma forma que a proteção de memória isola as aplicações de acesso à sua área de memória por outras [20]. Uma vez que uma aplicação faça uma solicitação de reserva de CPU e tal solicitação seja atendida pelo sistema, deve ser assegurada a disponibilidade da quantidade de CPU solicitada pela aplicação para a sua execução. As aplicações devem executar como se tivessem um processador mais lento, porém dedicado, independente da presença de outras tarefas [21].

Um dos problemas encontrados para a utilização da reserva de CPU é a definição da quantidade do recurso que deve ser reservada devido à dificuldade em se estimar a quantidade de CPU necessária previamente [22]. Além disso, uma reserva baseada na utilização média da CPU pela aplicação resultaria em degradações temporárias, porém inaceitáveis, da QoS fornecida, e uma baseada no pior caso, ou seja, máxima utilização, resultaria em subaproveitamento da CPU. Para a superação deste problema, é interessante que o sistema de reservas suporte a reserva dinâmica do recurso [23], o qual possibilita que a reserva seja feita de acordo com as necessidades temporárias das aplicações e que variem com o tempo. É importante ainda considerar que a especificação das quantidades de recursos que devem ser reservadas às aplicações é inevitavelmente dependente do *hardware* que as executa [20], uma vez que diferentes CPUs executam as tarefas com desempenhos distintos.

Outros problemas surgem quando a reserva de CPU é feita para a execução de aplicações remotas, ou seja, quando há o compartilhamento de recursos em um ambiente distribuído. Um deles refere-se a quantidade e ao tempo pelo qual os recursos devem ser reservados para aplicações remotas, uma vez que se poucos recursos forem reservados, tal aplicação irá executar por bastante tempo, enquanto que se uma grande parte dos recursos for reservada, o desempenho de aplicações locais será fortemente degradado [25].

De acordo com Mercer e outros[20] e Sun e Chen [24], um mecanismo para a reserva de CPU deve ter quatro características fundamentais. Primeiramente, deve prover mecanismos para que as aplicações possam definir seus requisitos de CPU, o que depende da existência de um modelo consistente de escalonamento capaz de acomodar aplicações com diferentes requisitos. Os requisitos de CPU de uma aplicação podem ser definidos, por exemplo, como uma porcentagem da capacidade total do sistema.

Em segundo lugar, deve ser capaz de avaliar os requisitos de processamento de novas aplicações e decidir se devem ser admitidos ou não. Tal característica requer a existência de um *framework* capaz de traduzir or requisitos de CPU especificados por cada aplicação individual em medidas de utilização que possam ser utilizadas em uma política de controle de admissão de novas tarefas.

Uma terceira característica é que as novas aplicações devem ser escalonadas de maneira consistente de acordo com uma política de controle de admissão, de acordo com a qual um *framework* para o escalonamento de tarefas deve ser consistente com todas as políticas de gerência de recursos existentes no sistema; caso contrário, os mecanismos de reserva não funcionarão corretamente.

Por último, o mecanismo deve ser capaz de medir precisamente a quantidade de processamento sendo utilizada por cada uma das aplicações e de controlar a execução destas, de modo a evitar que os recursos utilizados superem as quantidades

reservadas. Para tanto, é necessário que exista um software de monitoração do desempenho das aplicações, capaz de medir com precisão a utilização de CPU por cada aplicação. Faz-se necessário também um mecanismo que sinalize ao escalonador de tarefas quando uma determinada aplicação ultrapassar sua utilização limite dos recursos.

Capítulo 3

Grid Computing

O *Grid* é definido como um sistema distribuído que permite o compartilhamento, seleção e agregação de recursos autônomos e heterogêneos, geograficamente distribuídos e pertencentes a diferentes domínios administrativos em tempo de execução, dependendo de sua disponibilidade, desempenho, custo e da qualidade de serviço requisitada pelo usuário para a resolução de problemas de larga escala [29, 30]. O termo *grid* se deve a uma analogia com a rede elétrica (*power grid*), que garante um acesso consistente, seguro e transparente à energia elétrica, desconsiderando sua fonte.

Um sistema em *grid* pode ser visto como um sistema computacional integrado e colaborativo. Conforme Baker et al [31], deve ser capaz de prover: serviços que permitem a execução de tarefas em recursos de diferentes domínios administrativos; mecanismos de segurança para que apenas usuários autorizados possam ter acesso aos recursos; mecanismos que garantam a segurança dos dados e das aplicações executadas; e ferramentas que realizem a procura e alocação dos recursos, a distribuição das tarefas e a coleta de resultados.

3.1. Principais características

Segundo Baker et al [31], existem alguns aspectos fundamentais do *grid* para que esse possa garantir o acesso da forma desejada aos recursos. Tais aspectos caracterizam o *grid*. São eles:

- Múltiplos domínios administrativos e autonomia: os recursos do *grid* são distribuídos geograficamente por vários domínios administrativos e pertencem a diferentes organizações. A autonomia dos proprietários dos recursos deve ser mantida, assim como sua gerência de recursos e políticas de uso locais;

- Heterogeneidade: um *grid* típico envolve múltiplos recursos de natureza heterogênea, uma vez que engloba diversas tecnologias e diversos tipos de recursos;

- Escalabilidade: um *grid* pode vir a crescer de uma quantidade pequena de recursos a até milhões deles. Isso aumenta o problema da degradação de desempenho à medida que o *grid* aumenta de tamanho. Sendo assim, aplicações que necessitam, para a sua execução, de diversos recursos geograficamente distribuídos devem ser projetadas de modo que sejam tolerantes à latência e banda de transmissão;

- Dinamismo e adaptação: a falha de recursos em um *grid* é bastante comum devido à grande quantidade deles. Sendo assim, gerentes de recursos devem adaptar o *grid* dinamicamente e utilizar os recursos disponíveis de maneira eficiente e efetiva.

3.2. Tipos de serviços

O *Grid* é um ambiente computacional integrado e colaborativo. Os usuários se relacionam com o gerente de recursos do *grid* para solucionarem determinados problemas, o qual, por sua vez, aloca os recursos necessários e processa as tarefas da aplicação nos recursos distribuídos. Do ponto de vista do usuário, o *grid* pode ser utilizado para prover os seguintes serviços[31]:

- Serviços computacionais: possui ênfase no provimento de serviços seguros para a execução de aplicações que necessitam de recursos computacionais distribuídos. *Grids* que provêem esses serviços são chamados *grids* computacionais;

- Serviços de dados: possui ênfase no provimento de acesso seguro a conjuntos de dados distribuídos e possibilita a gerência de tais dados. O processamento dos conjuntos de dados é feito com o auxílio de *grids* computacionais. Essa combinação entre os conjuntos de dados e os *grids* computacionais é conhecida como *Data Grid*;

- Serviços de aplicação: consiste na gerência de aplicações e acesso remoto a *softwares* e bibliotecas de funções de maneira transparente;

- Serviços de informação: responsáveis pela extração e apresentação de dados significativos, fazendo uso dos serviços computacionais, de dados e de aplicação, acima descritos;

- Serviços de conhecimento: sua principal preocupação é a maneira pela qual o conhecimento é adquirido, usado, recuperado, publicado e mantido para assistir usuários no alcance de suas metas e objetivos particulares.

3.3. Benefícios

Certamente a tecnologia *Grid* traz diversos benefícios se comparada à computação convencional. Os principais benefícios alcançados são [33,36]: (i) possibilitar uma maior cooperação, sem barreiras, entre comunidades, científicas ou comerciais, geograficamente dispersas; (ii) transparência, sob o ponto de vista do usuário, de acesso aos recursos, isolando-o da complexidade do *grid*; (iii) possibilitar o alcance de soluções para problemas computacionais em menos tempo; (iv)tornar possível a resolução de problemas que antes não tinham solução viável, devido à necessidade de uma quantidade de recursos que não poderia ser facilmente acoplada; e

(v) permitir que pessoas ou empresas separadas geograficamente possam formar organizações virtuais e assim compartilhar recursos.

Conforme [33], além dos benefícios citados acima, o *grid* traz também alguns benefícios tecnológicos:

- Otimização da infra-estrutura, pois fornece ao sistema capacidade de execução de aplicações de alta demanda computacional, além de reduzir o tempo de ciclo (os ciclos ociosos são compartilhados, levando a um efeito semelhante a uma redução no tempo de ciclo);

- Aumento da colaboração, à medida em que suporta uma colaboração multidisciplinar e permite que diversas organizações e empresas colaborem entre si;

- Alta disponibilidade, uma vez que a ocorrência de falhas nos recursos do *grid* não devem indisponibilizar o acesso a outros recursos.

Pode-se considerar também um benefício de ordem econômica, uma vez que empresas têm percebido que podem fazer economias significativas terceirizando elementos não essenciais de seu ambiente computacional junto a diferentes provedores de serviços.

3.4. Organizações virtuais

Conforme dito anteriormente, o objetivo do *Grid Computing* é o compartilhamento de recursos para a resolução de problemas de larga escala. Tal compartilhamento, entretanto, não trata apenas de um intercâmbio de arquivos, mas sim de um acesso direto a computadores, compartilhando assim memória, dados, softwares, ciclos ociosos de processador, entre outros recursos. Tais compartilhamentos devem ser altamente controlados, seguir regras bem determinadas, de modo que consumidores e provedores de recursos tenham bem definidos quais recursos são compartilhados, quais usuários ou instituições possuem permissão para compartilhar seus recursos, além das condições sob as quais o compartilhamento pode ocorrer.

Um conjunto de usuários ou instituições utilizando um *grid* sob tais regras de compartilhamento é chamado de organização virtual (VO – *Virtual Organization*). Uma organização virtual pode ser também definida, segundo [35], como uma entidade abstrata que agrupa usuários, instituições e recursos em torno de um mesmo objetivo.

Uma organização real pode fazer parte de uma ou mais organizações virtuais compartilhando alguns ou todos os recursos sobre os quais a organização possui o controle. As relações de compartilhamento podem variar dinamicamente através do tempo em relação aos recursos compartilhados, às regras de compartilhamento e aos participantes aos quais o acesso aos recursos é permitido, isso porque a qualquer momento uma organização real pode começar a fazer parte de uma organização virtual ou mesmo sair, deixando de compartilhar seus recursos.

As organizações virtuais podem variar enormemente de acordo com propósito, escopo, duração, tamanho, estrutura e comunidade. Apesar disso, existe entre elas um amplo conjunto de requisitos e interesses comuns. Em particular, é necessária uma relação de compartilhamento altamente flexível, assim como níveis sofisticados e precisos de controle sobre a política de compartilhamento de recursos. Existe também a necessidade de compartilhamento de uma variedade de recursos, indo desde programas, arquivos e dados, até computadores e redes, além da necessidade de diversos modos de uso dos recursos, indo de mono-usuário a multi-usuário ou de uso baseado em desempenho a baseado em custo [32].

Diferentes organizações virtuais podem diferir em vários aspectos, porém possuem alguns aspectos em comum. Por exemplo, em toda organização virtual existem diferentes usuários compartilhando seus recursos de modo a atingir algum resultado. O compartilhamento de recursos é condicional, ou seja, ocorre de acordo com condições impostas pelos proprietários dos recursos. Além disso, os usuários (consumidores) podem impor interesses sobre as propriedades dos recursos com os quais estão preparados para trabalhar.

As relações de compartilhamento podem variar dinamicamente com o tempo, o que leva à necessidade da existência de mecanismos para descobrir e caracterizar a natureza das relações existentes em determinado momento.

3.5. Escalonamento em ambiente de *grid*

O escalonamento de tarefas em um *grid* consiste em fazer um mapeamento de tarefas a um grupo de recursos disponíveis, os quais podem pertencer a diferentes domínios administrativos, de modo a executá-las [54]. Para que o *grid* possa atingir um de seus objetivos, o de agregar diferentes recursos e prover aos usuários serviços não-triviais, um escalonador eficiente torna-se uma parte fundamental do sistema [54]. Apesar disso e da existência de diversos sistemas de escalonamento em *grid*, sendo o problema do escalonamento em *grid* de grande complexidade, são providas apenas soluções *ad hoc* e específicas de domínio, não existindo um sistema de escalonamento genérico e que supra todas as necessidades em quaisquer situações [55], uma vez que o problema é NP-completo.

O escalonamento de recursos em *grid* é feito geralmente em dois níveis [56]. Primeiramente, uma tarefa é alocada a um nó em particular no *grid*, o qual pode ser uma única CPU, um supercomputador ou um conjunto de estações de trabalho, e então é escalonada localmente. O tipo do escalonamento local (ou de rede local) depende da natureza do nó em questão, podendo ser monoprocessador, multiprocessador, entre outros. Este tipo de abordagem é conhecido como meta-escalonamento e a entidade que realiza a alocação de determinado nó como meta-escalonador. O funcionamento desse tipo de escalonamento é visualizado na figura 7.

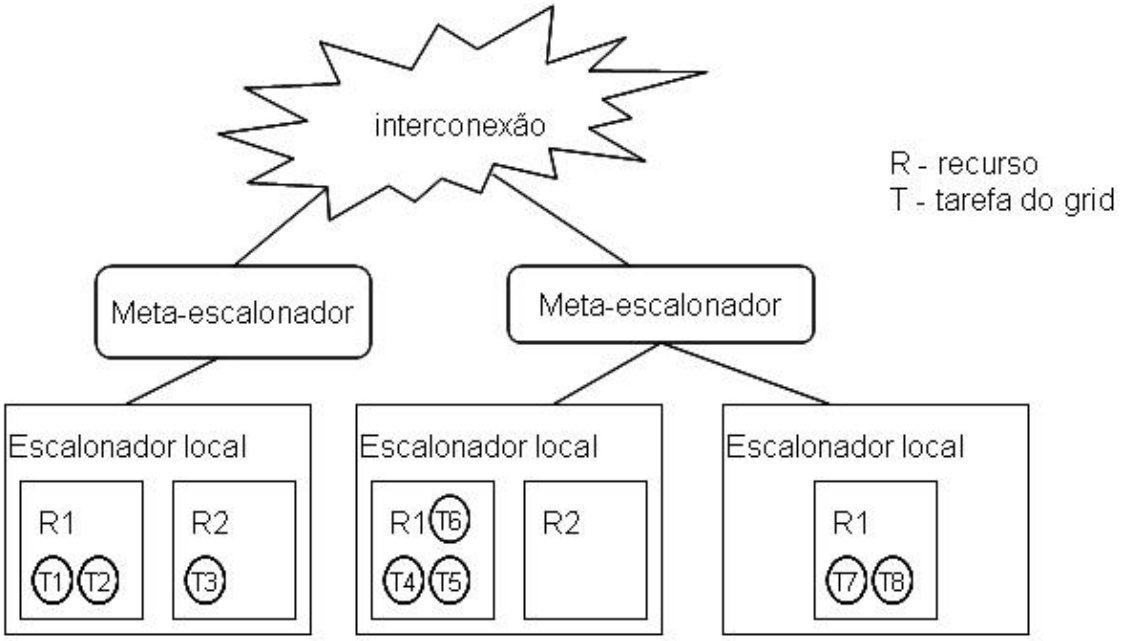

Figura 7: Esquema de meta-escalonamento.

É desejável que um meta-escalonador tenha as seguintes capacidades [55]:

- Interação com gerentes de recursos locais;
- Interação com serviços do *grid*, tais como serviços de informação, segurança e execução;
- Receber um problema de escalonamento, calcular um escalonamento e retornar uma decisão;
- Dividir um problema em subproblemas, receber decisões de escalonamento e combiná-las em uma nova decisão;
- Encaminhar o problema a outros escalonadores.

Um meta-escalonador pode exibir apenas um subconjunto de tais capacidades, dependendo de suas interações com outros escalonadores e serviços e de seu comportamento esperado. Caso interaja com outros escalonadores, deve ser capaz de repassar problemas ou subproblemas e receber resultados.

Diferentemente de outros tipos de sistemas distribuídos, escalonadores no *grid* não são capazes de controlar diretamente os recursos. Ao invés disso, funcionam como *brokers*, apenas selecionando os recursos desejados [54]. É importante também salientar que não estão necessariamente localizados no mesmo domínio administrativo que os recursos visíveis a eles.

Alguns desafios são encontrados na realização do escalonamento no *grid*. Segundo [57], alguns destes desafios são:

- Os recursos disponíveis são heterogêneos. Essa heterogeneidade faz que tenham diferentes capacidades para o processamento das aplicações. Um escalonador adequado deve ser capaz de considerar e superar as diferenças entre as capacidades computacionais dos diferentes recursos;

- Autonomia dos recursos. Os recursos podem ter políticas próprias de escalonamento, as quais devem ser respeitadas, o que complica o processo de escalonamento;

- Os recursos não são dedicados. A autonomia dos recursos pode causar contenção nos recursos, fazendo que seus comportamentos e desempenhos variem com o tempo;

- Diversidade das aplicações. As aplicações no *grid* podem ser originadas de diversos usuários diferentes e terem os mais diversos requisitos. Isso torna a construção de um escalonador genérico bastante complexa;

Para que seja realizado um escalonamento adequado, é necessário acesso a informações sobre os recursos disponíveis no *grid* [54]. Por isso há a necessidade de comunicação do escalonador com serviços de informação, como dito anteriormente. São necessárias também informações sobre propriedades das aplicações e desempenho dos recursos para diferentes tipos de aplicação.

3.6. Passos para a execução em *grid*

O processo de escalonamento e execução de tarefas em *grid*, segundo [38], pode ser dividido em três fases: (i) descoberta de recursos, o que gera uma lista de recursos potencialmente disponíveis; (ii) seleção de sistema, que consiste na recuperação de informações sobre os recursos descobertos e a escolha de um conjunto deles; e (iii) a execução de tarefas. Tais fases, assim como os passos executados em cada uma delas, podem ser visualizadas na Figura 8.

3.6.1. Descoberta de Recursos

Esta fase consiste em identificar um conjunto de recursos em potencial a ser utilizado para a execução de tarefas no *grid*, ou seja, recursos que passaram em uma avaliação quanto aos requisitos mínimos exigidos. Esta fase é executada em três passos [38]: (i) filtragem de autorizações, (ii) definição de requisitos de tarefas e (iii) filtragem de modo a serem alcançados os requisitos mínimos.

O primeiro passo, a filtragem de autorizações, consiste em determinar um conjunto de recursos nos quais o usuário possui permissão para a execução de tarefas, uma vez que sem tal permissão a tarefa simplesmente não será executada.

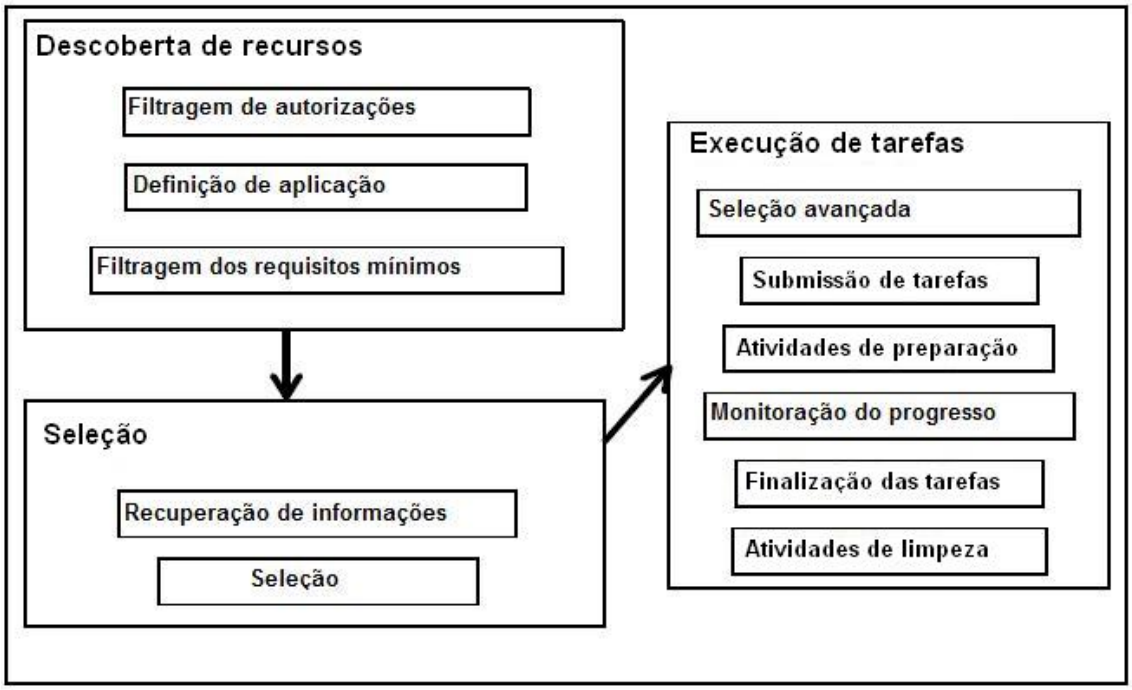

Figura 8: Passos para a execução de tarefas em *grid* [38].

O próximo passo é a definição dos requisitos mínimos para a execução da tarefa ou aplicação, para que os recursos levantados no passo anterior possam ser filtrados de forma a manter apenas os que podem ser úteis na execução. O conjunto de requisitos pode ser bastante amplo e varia de acordo com a tarefa a ser executada, podendo conter tanto requisitos estáticos, como sistema operacional e arquitetura de *hardware*, quanto dinâmicos, como quantidade mínima de memória e conectividade, sendo que quanto mais detalhes dos requisitos forem dados, melhor se dará a execução da tarefa [38]. O grande problema nesta fase é que os requisitos da tarefa podem mudar de acordo com o sistema escolhido. Por exemplo, os requisitos de memória ou de bibliotecas podem mudar de acordo com a arquitetura ou o algoritmo escolhido.

Após levantados os requisitos mínimos e os recursos aos quais o usuário possui permissão de acesso, o próximo passo é filtrar os recursos que não satisfazem os requisitos mínimos das tarefas a serem executadas. Ao final deste passo, o usuário possuirá apenas um conjunto reduzido de recursos a serem investigados mais a fundo. Após este passo, é iniciada então a próxima fase.

3.6.2. Seleção de sistema

Após descobertos os recursos, é necessário que um ou mais deles sejam selecionados para que possam ser alocados às tarefas a serem executadas. Tal seleção é realizada em dois passos [38]: recuperação de informações e seleção dos recursos.

A recuperação de informações dinâmicas sobre os recursos é importante para que sejam escolhidos recursos de forma a executar a tarefa o melhor possível. Uma

vez que as informações levantadas variam de acordo com a tarefa sendo escalonada, é possível que nenhuma solução funcione para todos os requisitos, ou até mesmo para muitos deles. É necessário levar em consideração as informações sobre políticas de alocação locais aos recursos, sendo que tais dados devem fazer parte do conjunto de informações dinâmicas coletadas.

Após coletadas e analisadas as informações sobre os recursos, alguns deles são finalmente escolhidos para a execução das tarefas. É iniciada então a próxima fase, a execução de tarefas.

3.6.3. Execução de tarefas

O primeiro passo desta fase é a reserva prévia de recursos, sendo este opcional [38]. Tal reserva pode ser necessária para um ou mais recursos para que o sistema seja utilizado de forma otimizada. Dependendo do recurso em questão, pode ser mais fácil ou difícil realizar a reserva prévia, sendo que pode ser realizada através de meios mecânicos ou humanos.

Realizada ou não a reserva prévia, uma vez que os recursos já estão escolhidos, as tarefas podem ser submetidas para execução, o que constitui o próximo passo. Este passo pode se tornar bastante complicado devido à falta de um padrão para a submissão de tarefas em ambiente de *grid*.

O próximo passo na seqüência é a execução de atividades de preparação [38]. Dentre tais atividades, podem estar contidas atividades de instalação, configuração, pedidos de reserva de recursos ou outras atividades referentes à preparação do recurso para a execução das tarefas. Executadas tais atividades, dependendo da aplicação e de seu tempo de execução, o usuário pode monitorar o progresso da execução e possivelmente decidir realizar alterações quanto à localidade e à maneira da execução.

Após o término da execução da tarefa, o próximo passo a ser seguido é a notificação do usuário que a ordenou. Por último, devem ser realizadas atividades de limpeza do ambiente. O usuário pode recuperar arquivos dos recursos para análise de resultados, remover arquivos temporários utilizados na execução da aplicação, entre outras. Em suma, devem ser desfeitas as alterações realizadas no passo de preparação do ambiente as quais causaram modificações no sistema.

3.7. Arquitetura do *grid*

Inicialmente foi proposta para o *grid* uma arquitetura de protocolos dividida em camadas. Esta arquitetura, porém, possuía problemas de flexibilidade. Foram então incluídos mecanismos de *Web services* [10], surgindo assim a *Open Grid Services Architecture* (OGSA) [34]. Posteriormente, um componente da OGSA, a *Open Grid Services Infrastructure* (OGSI) [41], a qual define um conjunto de convenções para a utilização de *Web services* com manutenção de estados, evoluiu para o *Web Services*

Resource Framework (WSRF) [39], de forma a se adequar melhor ao padrão de *Web services*.

3.7.1. Arquitetura de protocolos

O principal objetivo de um *grid* é o compartilhamento coordenado de recursos entre diversas organizações virtuais. Esses recursos podem estar espalhados por uma ampla área geográfica, além de ser provável que possuam proprietários diferentes. Além disso, tais recursos podem possuir diferentes plataformas, sistemas, linguagens de programação e políticas de acesso e segurança, entre outros. Por isso é necessário que se tenham mecanismos que permitam uma interoperabilidade dos recursos. Essa interoperabilidade pode ser alcançada com a definição de algumas regras, implementadas em protocolos. Sendo assim, a arquitetura proposta inicialmente para o *grid* é, antes de tudo, uma arquitetura de protocolos, onde tais protocolos definem os mecanismos pelos quais os usuários das organizações virtuais negociam, estabelecem, gerenciam e exploram as relações de compartilhamento; definem também como os elementos do sistema devem interagir de modo a atingir um comportamento específico e qual a estrutura da informação trocada durante tal interação.

Porém, apenas protocolos não são suficientes para a operação do *grid*. É necessária a definição de *Application Programming Interfaces* (APIs), que definem interfaces padrão para a chamada de um conjunto específico de funcionalidades, e *Software Development Kits* (SDKs), conjuntos de código projetados para serem ligados com ou chamados de uma aplicação de modo a fornecer uma funcionalidade específica. A definição destes permite que sejam criadas abstrações de programação de modo a se criar um *grid* operável. O uso das APIs e SDKs permite que as organizações virtuais atinjam também um objetivo além da interoperabilidade. Permite que desenvolvedores sejam capazes de implementar aplicações em ambientes de execução complexos e dinâmicos, e que os usuários sejam capazes de executar tais aplicações. Eles aceleram o desenvolvimento de aplicações, permitem o compartilhamento de código e aumentam a portabilidade das aplicações.

A arquitetura de protocolos do *grid* proposta por Foster [32] é baseada no modelo da ampulheta, onde, por definição, tem-se no gargalo um número pequeno de protocolos. A arquitetura é dividida em camadas, onde componentes pertencentes a elas possuem características em comum, mas podem apresentar comportamentos e capacidades providas por qualquer outra camada. O modelo da ampulheta se explica por existir uma pequena quantidade de protocolos nas camadas *Resource* e *Connectivity,* que compõem o gargalo, sobre os quais diversos comportamentos de alto nível podem ser implementados, configurando o topo da ampulheta, e por eles poderem ser implementados sobre uma diversidade de tecnologias.

A ampulheta possui a seguinte configuração [32]:

- Base: camada *Fabric*;
- Gargalo: camadas *Resource* e *Connectivity*;
- Topo: camadas *Collective* e *Application*.

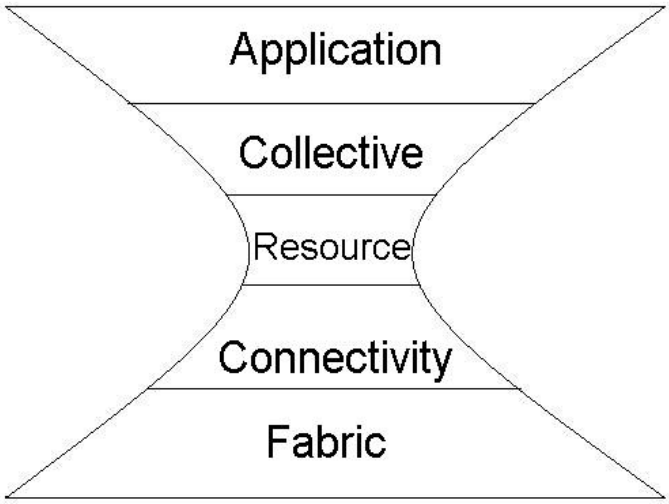

Figura 9: O modelo da ampulheta para a arquitetura do *grid* [32].

3.7.1.1. A camada *Fabric*

A camada *Fabric* provê os componentes que serão compartilhados pelos usuários do *grid*, tais como recursos computacionais, de armazenamento e recursos de rede. Os componentes desta camada são responsáveis pela implementação de operações específicas, dependentes de recursos específicos, e que são resultado das operações de compartilhamento.

Os recursos devem implementar mecanismos que permitam que se descubra sua estrutura, estado e suas capacidades, além de mecanismos de gerência dos recursos que permitam algum controle sobre a qualidade do serviço que é fornecido. Os componentes dessa camada são dos seguintes tipos e possuem as seguintes características [32]:

- Recursos computacionais: mecanismos necessários para iniciar a execução de aplicações e para monitorar os processos resultantes. Deve haver também mecanismos que permitam descobrir informações sobre o estado dos componentes, além de mecanismos de gerência;

- Recursos de armazenamento: são necessários mecanismos para a leitura e escrita de arquivos, assim como mecanismos para a leitura e escrita de subconjuntos de arquivos e para a execução de dados remotos;

- Recursos de rede: mecanismos de gerência que fornecem controle sobre recursos alocados nas operações de transferência em rede. Devem existir funções que permitam identificar características e carga da rede;

- Depósitos de código: requer mecanismos para gerenciar diferentes versões de códigos fonte e objeto;

- Catálogos: esse tipo de armazenamento requer mecanismos que permitam a consulta e atualização de catálogos.

3.7.1.2. A camada *Connectivity*

É a camada na qual são definidos os protocolos de autenticação, ou segurança, e comunicação, necessários para operações de rede no *grid*. Os protocolos de comunicação possibilitam a troca de dados entre os recursos da camada *Fabric* e os de autenticação fornecem mecanismos seguros para a verificação de identidade de usuários e recursos.

Os protocolos de comunicação devem prover suporte a operações de transporte, para as quais podem ser utilizados os protocolos de Internet e transporte (como IP, ICMP, UDP, TCP), operações de roteamento e de identificação de endereços através de nomes, podendo-se utilizar para isso o protocolo do DNS (*Domain Name Service*).

Quanto à parte de autenticação ou segurança, é importante que os mecanismos fornecidos sejam baseados em padrões existentes [32]. A autenticação e autorização de usuários e recursos deve ser uniforme, e deve possuir as características:

- *Single sign-on*: deve ser possível que o usuário tenha acesso aos recursos do *grid* definidos na camada *Fabric* tendo sido autenticado uma única vez, ou seja, com uma única autenticação criar uma credencial *proxy* que programas podem utilizar para autenticar com qualquer serviço remoto em benefício do usuário;

- Delegação: deve ser possível que uma aplicação de usuário possa ser executada em favor desse usuário, utilizando-se dos recursos os quais o usuário tem permissão de uso;

- Integração com várias soluções de segurança locais;

- Relações de confiança entre usuários: para que um usuário possa utilizar-se de diversos recursos pertencentes a fornecedores diferentes, não deve ser necessário que os diferentes recursos interajam entre si quanto a aspectos de segurança.

Os mecanismos de segurança do *grid* devem prover flexibilidade à proteção de comunicações e devem permitir que os interessados tenham controle sobre as decisões de autorização.

3.7.1.3. A camada *Resource*

Essa camada define, sobre os protocolos da camada *Connectivity*, protocolos, APIs e SDKs para a negociação segura, inicialização, controle e monitoração das operações de compartilhamento de recursos individuais. Esses protocolos, APIs e SDKs fazem chamadas a funções definidas na camada *Fabric* para fazer acesso e controlar recursos locais. Os protocolos desta camada se preocupam unicamente com os recursos individuais, ignorando o estado global do sistema distribuído.

Podemos distinguir dentro da camada *Resource* dois tipos fundamentais de protocolos [32]:

- Protocolos de informação, utilizados na obtenção de informações sobre o estado e a estrutura de recursos;
- Protocolos de gerência, usados para negociações de acesso a recursos compartilhados.

Como essa camada faz parte do "gargalo da ampulheta", apenas um pequeno conjunto de protocolos deve ser definido.

3.7.1.4. A camada *Collective*

Diferente da camada *Resource*, que se preocupa com as operações em recursos individuais, a camada *Collective* define protocolos, APIs e SDKs que não estão ligados a nenhum recurso individual específico, mas sim a recursos de natureza global, capturando interações entre conjuntos de recursos. Como essa camada é implementada acima do "gargalo da ampulheta", ela pode englobar uma variedade de protocolos, sem que seja necessária a definição de novos requisitos para os recursos compartilhados. Alguns exemplos de serviços fornecidos por essa camada são:

- Serviços de monitoração e diagnósticos, que dão suporte à monitoração de falhas em recursos;
- Serviços de replicação de dados;
- Serviços de descobrimento de *software* (*software discovery services*)[32], que descobrem e selecionam a melhor implementação de software e plataforma de execução baseados no problema sendo considerado.

Estes exemplos ilustram uma parte da variedade de protocolos e serviços que podem ser implementados na camada *Collective*, sendo uns bastante genéricos, e outros bastante específicos para uma certa aplicação ou usuário. Tal especificidade pode levar a camada a existir apenas em organizações virtuais específicas.

3.7.1.5. A camada *Applications*

Essa camada, a mais ao topo do modelo da arquitetura, inclui as aplicações de usuário que são executadas dentro do ambiente de uma organização virtual. Tais aplicações são construídas de acordo com os serviços definidos em todas as outras camadas, podendo fazer chamadas a esses serviços. A cada camada, certos protocolos são utilizados para garantir determinados serviços, e são usados para a execução de ações desejadas. As APIs são implementadas por SDKs, que por sua vez utilizam os protocolos do *grid* para interação com os serviços de rede que provêem certas capacidades ao usuário final. SDKs de alto nível podem prover uma funcionalidade que

não está mapeada diretamente a um protocolo específico, mas que combina diversas operações de protocolos, fazendo chamadas a aplicações adicionais, implementando uma funcionalidade local.

3.7.1.6. Implementação – *Globus Toolkit* 2

O *Globus Toolkit* 2 (GT2) será apresentado segundo [41]. Trata-se de um conjunto de serviços e bibliotecas de arquitetura e código abertos que dá suporte ao *grid* e a suas aplicações. Implementa a arquitetura de protocolos e surgiu como um padrão *de facto* para a implementação de *grids*. Alguns aspectos tratados pelo GT2 são: segurança, descoberta de informações, gerência de recursos e de dados, comunicação, portabilidade e detecção de falhas [41].

Em relação à camada *Fabric*, o GT2 assume a existência de mecanismos que realizem as funções relativas a ela. Porém, inclui componentes projetados para facilitar a interconexão com protocolos da camada *Resource*. Inclui mecanismos para a descoberta do estado e estrutura de recursos e para o empacotamento das informações descobertas de forma a facilitar a implementação de protocolos de níveis superiores [41].

A camada *Connectivity* é definida com base em uma estrutura de segurança baseada em chave pública chamada *Grid Security Infrastructure* (GSI). A estrutura provê suporte à autenticação única (*single sign-on*), delegação e proteção às comunicações [41. O GSI define um formato padrão para as credenciais e um protocolo de delegação remota para a transferência das credenciais aos serviços remotos, visando com isto à interoperabilidade no *grid* [41]. Provê ainda meios para que aplicações invoquem operações de autenticação utilizando APIs de alto-nível, ou seja, sem realizar operações de protocolo diretamente.

Em relação à camada *Resource,* o GT2 utiliza o protocolo *Grid Resource Allocation and Management* (GRAM) para a criação segura e confiável de computações remotas [41]. Utiliza mecanismos providos pelo GSI para a autenticação, autorização e delegação de credenciais a computações remotas. Sua implementação do GRAM utiliza um processo *gatekeeper* para a criação de computações remotas, um gerente de tarefas para gerenciá-las e o GRAM *reporter* para a monitoração e publicação de informações sobre identidade e estado das computações locais.

O GT2 utiliza também uma implementação do *Monitoring and Discovery Service* (MDS-2), o qual provê meios para a descoberta e o acesso a informações de configuração e status de serviços [41]. Tal implementação provê dois componentes principais: um registro local, responsável por gerenciar a coleta e publicação de informações em um local em particular, e um registro coletivo, responsável por dar suporte a consultas de informações partindo de múltiplas localidades. O funcionamento da camada *Resource* pode ser visualizado na Figura 10.

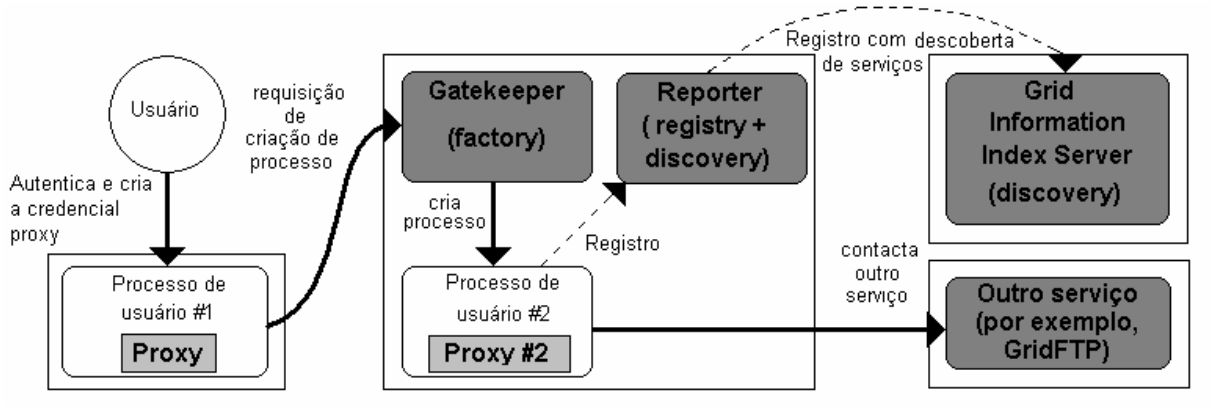

Figura 10: Mecanismos da camada *Resource* no *Globus Toolkit* 2 [34].

Na camada *Resource*, como ilustra a Figura 10, inicialmente o usuário se autentica e é criada uma credencial para o processo de usuário #1, o qual requisita, utilizando a credencial criada e o protocolo GRAM, a criação de um processo remoto. A requisição é processada pelo *gatekeeper* do GRAM e um novo processo é criado, o processo de usuário #2, com novas credenciais. Propriedades relativas ao processo criado são registradas pelo MDS-2.

3.7.2. *Open Grid Services Architecture* (OGSA)

A OGSA é uma evolução da arquitetura de protocolos descrita na Seção 3.7.1 que visa o suporte à criação, manutenção e aplicação dos serviços mantidos pelas organizações virtuais [34]. É uma arquitetura orientada a serviços, ou seja, todos os recursos disponíveis no *grid* são representados como serviços, sendo estes definidos como entidades que provêem determinadas funcionalidades aos clientes através de trocas de mensagens. É uma arquitetura construída sobre *Web Services*, um paradigma de computação distribuída que se baseia em padrões baseados em Internet para atingir uma computação distribuída heterogênea, sendo independentes de modelo e de linguagem de programação, assim como de sistema de software. A OGSA e seus principais componentes podem ser visualizados na Figura 11.

Figura 11: Arquitetura OGSA e seus principais componentes [41].

A orientação a serviços da OGSA facilita a virtualização dos serviços, ou seja, o encapsulamento das diversas implementações possíveis por trás de uma interface. A virtualização é importante pois torna possíveis as seguintes operações:

- Acesso consistente aos recursos através de diversas plataformas heterogêneas de maneira transparente;
- Mapeamento de diversos recursos lógicos referentes a um mesmo recurso físico;
- Gerência de recursos dentro de uma organização virtual baseada na composição a partir de recursos de baixo nível (recursos de alto nível são construídos pela composição de diversos recursos de baixo nível);
- Composição de serviços de modo a se formarem serviços mais sofisticados.

A arquitetura provê suporte à transparência local e remota no que diz respeito à localização e à invocação de serviços. Permite também que sejam feitas ligações de interfaces a diversos protocolos.

A OGSA define o conceito fundamental de *Grid Service*, um *web service* que implementa interfaces e comportamentos padrão que permitem que os serviços sejam criados, destruídos e tenham estados bem definidos. Tais interfaces e comportamentos são definidos pela *Open Grid Services Infrastructure* (OGSI), a qual é estruturada sobre mecanismos de *Web services* [41].

Logo acima da OGSI, tem-se na arquitetura um conjunto de serviços providos pela OGSA, os quais são constituídos utilizando-se funcionalidades providas pela OGSI. Tais funcionalidades são combinadas de forma a se construir interfaces de mais alto nível para que se possa prover novas funcionalidades não suportadas diretamente pela OGSI [41]. Os componentes desta camada são serviços tão fundamentais que espera-se sua utilização em quase todos ambientes de *grid*, tais como descoberta e gerência de serviços, monitoração, segurança e acesso a dados.

O padrão de *Web service* mais importante para a OGSA é o *Web Service Definition Language* (WSDL) [42]. É fundamental que no *grid* haja suporte à descoberta dinâmica e composição de serviços em ambientes heterogêneos, o que necessita mecanismos para o registro e descoberta de definições de interfaces, ao que a WSDL dá suporte provendo mecanismos para a criação de definições de interfaces separadas de sua parte concreta [41]. Uma definição de serviço WSDL é um documento XML que descreve interfaces de serviço para acesso a *Web services*, contendo também as mensagens trocadas no acesso ao serviço, dentre outras informações.

A OGSA não possui como objetivo apresentar detalhes de implementação, tais como linguagem de programação, ferramentas de implementação ou ambiente de execução. Isso torna possível que sejam especificadas interações entre serviços de maneira que sejam totalmente independentes do ambiente em que os serviços estão hospedados.

3.7.2.1. *Open Grid Services Infrastructure* (OGSI)

A OGSI define um conjunto de interfaces padrão e semânticas associadas para interações entre serviços permitindo a construção de componentes reutilizáveis e interoperantes [41]. Define ainda mecanismos para a criação, nomeação, monitoração, agrupamento, troca de informações entre serviços e gerência do tempo de vida destes. Um *web service* que adere aos padrões definidos na OGSI é denominado *Grid Service*. A OGSI define ainda interfaces padrão de fábrica e registro de grupos, de forma a possibilitar a criação de descoberta de *grid services*.

A OGSI possibilita que se tenha uma separação entre a definição de uma interface de serviço e uma instância de tal definição [41], requisito fundamental para que os serviços possam ser instanciados dinamicamente. Define ainda mecanismos para a representação de metadados e dados de estado relativos aos serviços como parte de sua definição de interface e para o acesso a tais dados relativos a uma instância de serviço. Tais mecanismos são denominados *service data*, e permitem a descoberta, monitoração e introspecção dos serviços.

O esquema de nomeação (*naming*) de serviços utilizado pela OGSI é dividido em dois níveis [41]. No primeiro nível tem-se os *Grid Service Handles* (GSH), que são identificadores abstratos que identificam unicamente cada instância de *grid service* existente, diferenciando as instâncias entre si, e que são representados por uma *Uniform Resource Identifier* (URI) [10]. No segundo nível estão os *Grid Service Resources* (GSR), que são referências a recursos concretos no *grid*. Um GSH é mapeado a um GSR por um serviço de resolução de nomes, o qual deve implementar a interface OGSI *HandleResolver*.

São também definidos mecanismos para o controle do ciclo de vida de instâncias de serviços [41]. Quando ocorre a criação de um novo serviço, pode ser definido um período de duração do serviço, ao final do qual o serviço é destruído e os recursos alocados liberados. É permitido também que este intervalo de duração seja renegociado a qualquer momento da vida do recurso. Existe ainda um mecanismo de destruição explícita de serviços. São definidas também interfaces para a gerência de grupos de serviços. Por último, a OGSI define também um tipo básico para todas as possíveis mensagens de erro que possam ser retornadas por um *grid service* [41].

Implementação – *Globus Toolkit 3*

O *Globus Toolkit 3* (GT3) é uma implementação em código aberto da OGSI. Foi desenvolvido sobre a tecnologia de *Web services* e consiste em um conjunto de ferramentas consideradas fundamentais a qualquer aplicação de *grid* por seus desenvolvedores [43]. É composto por uma infra-estrutura de segurança e serviços de sistema. Oferece um ambiente para a hospedagem de *Grid Services* (como definido pela OGSI), o qual medeia a interação entre as aplicações e as camadas inferiores de rede e protocolos de transporte. Para possibilitar a integração no *grid* e a interoperabilidade entre serviços utiliza o protocolo *Simple Object Access Protocol* (SOAP) [45] – para a codificação de mensagens.

A arquitetura do GT3 pode ser visualizada na Figura 12. As partes em branco na figura constituem o núcleo do GT3, pois incluem as funcionalidades mais essenciais aos *grid services*. Os Serviços Básicos incluem diversos tipos de serviços, tais como gerência de dados, execução de aplicações e serviços de informação, e são normalmente construídos sobre os componentes OGSI e de segurança (GSI). Já os serviços definidos por usuários, como o próprio nome diz, não são inclusos no GT3. São serviços desenvolvidos por terceiros e que podem ser construídos sobre quaisquer conjuntos de componentes do GT3. Os Serviços de Sistema constituem serviços de infraestrutura genéricos o bastante para serem utilizados por todos os outros *grid services*, ou em conjunto com estes, e são construídos sobre a implementação OGSI e a infra-estrutura de segurança.

A infra-estrutura de segurança provê segurança nos níveis de transporte e mensagem. Para a proteção em nível de transporte é utilizado um protocolo HTTP com mecanismos GSI habilitados, o qual é semelhante ao HTTPS, porém suporta a delegação de credenciais [43]. O uso deste tipo de segurança, porém, não é recomendado pelos projetistas do GT3. Para as mensagens, a segurança é feita pelo protocolo SOAP, podendo ser utilizado com qualquer protocolo de transporte que o suporte.

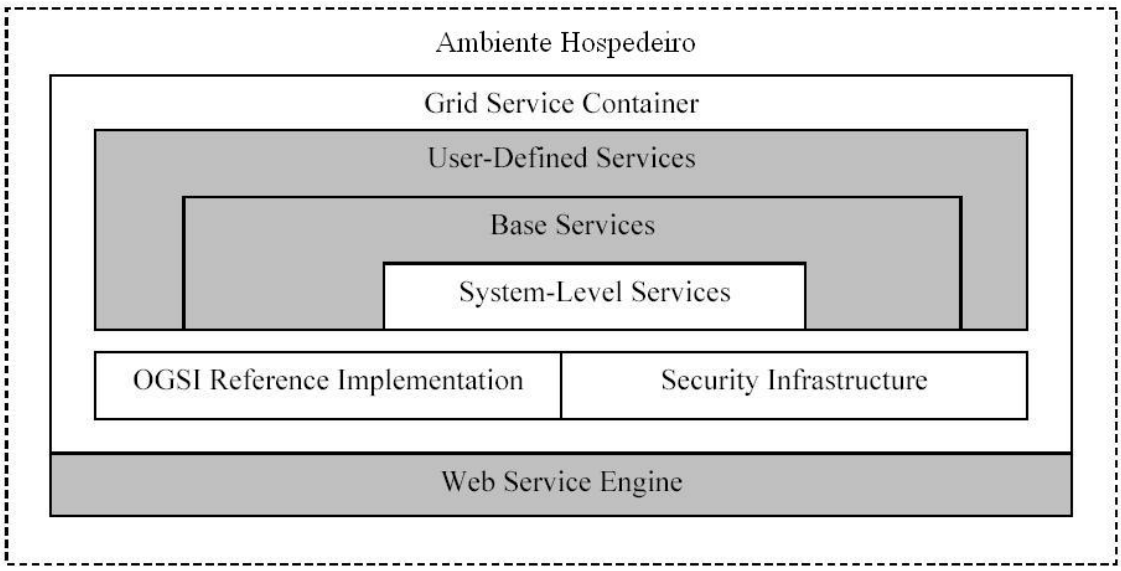

Figura 12: Arquitetura do Globus Toolkit 3 [43].

3.7.3. *WS - Resource Framework* (WSRF)

O *WS-Resource Framework* (WSRF) é uma evolução da OGSI que possui como objetivos principais a criação, endereçamento, inspeção e gerência de recursos com estados bem definidos [40]. Codifica a relação entre *Web services* e os recursos em termos do padrão *implied resource*. Por esse padrão, o identificador de um recurso é encapsulado em uma referência *endpoint*, a qual pode conter, além do endereço do *Web service*, outros metadados associados a este, tais como a descrição e propriedades do

serviço. Tal referência é então usada para a identificação do recurso a ser utilizado na execução de trocas de mensagens entre *Web services*.

A composição entre um recurso e um *Web service* de acordo com o padrão *implied resource* é denominada *WS-Resource*. O WSRF permite a declaração, criação, acesso, monitoramento e destruição de *WS-Resources* através de mecanismos convencionais de *Web Services* [39], e é descrito por cinco especificações normativas [39]. São elas:

- *WS-ResourceLifetime*. Consiste em mecanismos para a destruição de *WS-Resources*, incluindo trocas de mensagens que permitem que um requerente execute a destruição tanto de forma imediata quanto de forma agendada;

- *WS-ResourceProperties*. Consiste na definição de um *WS-Resource*, além de mecanismos para a recuperação, mudança e remoção de propriedades;

- *WS-RenewableReferences*. Consiste em informações associadas a uma referência *endpoint* contendo dados necessários para a recuperação de uma nova versão da referência caso ela se torne inválida;

- *WS-ServiceGroup*. Consiste em uma interface para coleções heterogêneas de *Web services*;

- *WS-BaseFaults*. Um tipo XML básico para falhas para ser utilizado quando for necessário retornar falhas em uma troca de mensagens pelos *Web services*.

WS-Resource é uma abordagem proposta para a modelagem de estados de recursos em um contexto de *Web services* [39]. É definido como a composição de um *Web Service* e um recurso com estados bem definidos, sendo expresso por uma associação entre um documento XML, o qual possui um tipo definido, e um *Web services portType*, e sendo também endereçado e recebendo acessos de acordo com o padrão *implied resource*, um conjunto de convenções com relação à tecnologia de *Web services*, particularmente a XML, WSDL e *WS-Addressing* [39]. Tais convenções permitem que o estado de um recurso seja definido e associado à descrição de uma interface de *Web Service*. Tal estado é definido com relação a um documento que especifica as propriedades do recurso em questão.

WS-Addressing padroniza uma referência *endpoint* para a representação da localização de um *Web service* instalado em um determinado *endpoint* na rede [39]. O padrão *implied resource* define um uso convencional para o *WS-Addressing* no qual um recurso é tratado como uma entrada implícita para o processamento de trocas de mensagens implementadas por um *Web service* [39].

3.7.3.1. *Globus Toolkit* 4

O *Globus Toolkit* 4 – GT4 é uma evolução do GT3 e faz uso extensivo de mecanismos de *Web services* para a definição de suas interfaces e para a estruturação de seus componentes [44]. Possui em sua composição um conjunto de implementações de serviços, containers para a hospedagem de serviços desenvolvidos por usuários e um conjunto de bibliotecas que permitem a chamada de métodos do GT4 e de serviços de usuário.

O conjunto de implementações de serviços contidas no GT4 contém serviços de infra-estrutura [44]. Existem serviços para a gerência de execução, acesso e movimentação de dados, gerência de credenciais, monitoração e descoberta de recursos e serviços, entre outros. A maioria dos serviços, porém não todos, são *Web services* Java. Para a gerência de execução, o GT4 provê uma implementação do GRAM, o qual provê uma interface *Web services* para a iniciação, monitoração e a gerência de computações arbitrárias em computadores remotos, e que permite a um cliente expressar, entre outras coisas, a quantidade de recursos desejada, o executável e seus parâmetros. Para a descoberta e monitoração de recursos e serviços, provê mecanismos padronizados para a associação de arquivos de propriedades baseados em XML a entidades da rede e para o acesso a tais propriedades. Tais mecanismos são basicamente implementações das especificações WSRF e *WS-Notification* e estão acoplados a todos os serviços e containers contidos no GT4 [44]. Provê ainda serviços para a coleta de informações recentes sobre o estado de diversos serviços registrados e para o acesso às informações coletadas. A arquitetura do GT4 é ilustrada na Figura 13.

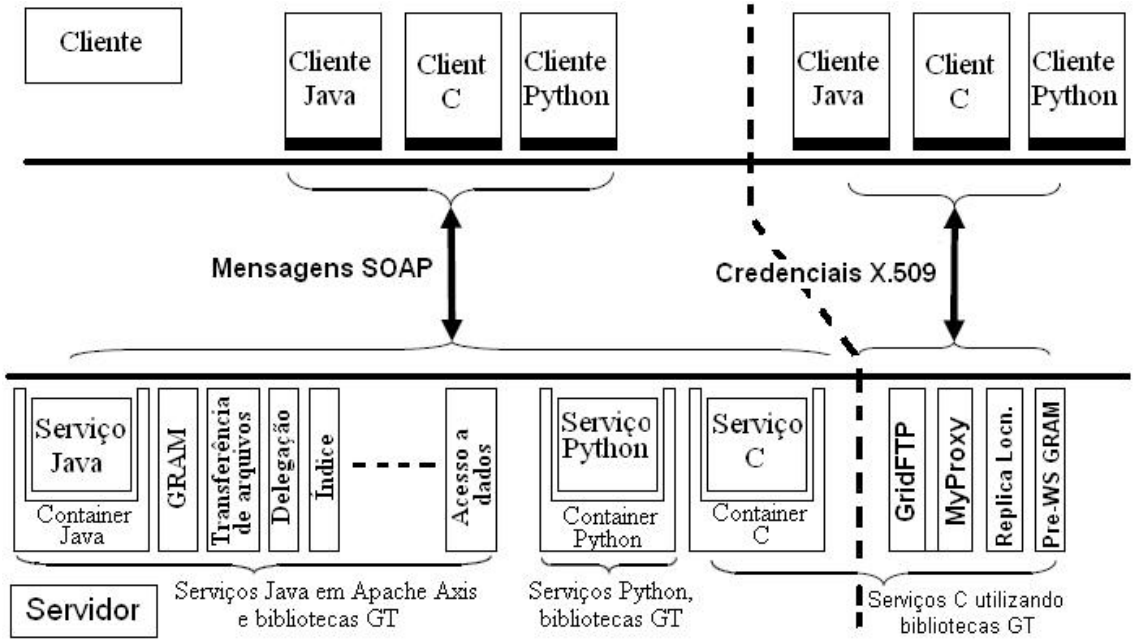

Figura 13: Arquitetura do GT4 [44].

Como é mostrado na Figura 13, o GT4 implementa uma diversidade de serviços de infra-estrutura ao *grid*. A maioria dos serviços são *Web services* Java, mas alguns, localizados na parte direita-inferior da figura, são implementados em outras linguagens. O GT4 provê três containers para a hospedagem de serviços desenvolvidos pelo usuário nas linguagens de programação Java, Python e C. Tais containers possuem implementações de segurança, gerência, descoberta, gerência de estado, entre outros mecanismos necessários na construção de serviços, além de darem suporte a diversas especificações de *web services*, tais como o WSRF, *WS-Notification* e *WS-Security*. O GT4 provê ainda um conjunto de bibliotecas cliente que permitem que aplicações cliente escritas em Java, C e Python façam chamadas a serviços do GT4 e serviços desenvolvidos por usuários [44].

Capítulo 4

QoS em *Grid Computing*

Algumas aplicações necessitam de garantias de QoS para que sejam executadas da maneira desejada. O *grid* porém trabalha tradicionalmente na base do melhor esforço, não garantindo, por *default*, QoS às aplicações [46]. Usualmente, as aplicações submetem seus requisitos aos serviços de gerência de recursos do *grid*, e as tarefas são escalonadas para execução de acordo com a disponibilidade dos recursos através do tempo. Porém, para aplicações com necessidade de garantias de QoS não é aceitável esperar pelos recursos até que estes se tornem disponíveis. Dessa forma, é importante a introdução de mecanismos que garantam a utilização dos recursos compartilhados por essas aplicações da forma que estas necessitam, ou seja, mecanismos que possam prover QoS às aplicações de forma determinística.

Uma solução para o problema mencionado é a introdução de mecanismos que permitam que um provedor particione seus recursos entre clientes de acordo com diferentes critérios [46]. Pode ser necessária a reserva de recursos [46], prévia ou por demanda, para a execução das aplicações. Estratégias mais elaboradas também podem ser utilizadas, onde acordos (*agreements*) são estabelecidos. Dessa forma, QoS pode ser provida às aplicações numa abordagem diferente da abordagem tradicional de melhor esforço.

Aplicações executadas em *grid* possuem requisitos de QoS que vão além dos relativos à rede, tais como CPU e memória, entre outros, e por isso são denominadas aplicações *high-end* [47].

4.1. Requisitos para QoS em ambiente de *grid*

Em sistemas tradicionais, a QoS de ponta-a-ponta é basicamente definida por parâmetros de rede como largura de banda, perda de pacotes, atraso e *jitters* [47], parâmetros estes que juntos formam a matriz de medição de QoS em rede [46]. Porém, considerando-se aplicações *high-end* em um ambiente de *grid,* devido à complexidade do sistema, os requisitos de QoS tornam-se mais diversificados e difíceis de serem satisfeitos, sendo inadequado considerar fatores relativos a QoS apenas para a rede. É importante também considerar fatores que levam a um melhor desempenho do sistema final, tais como reserva de recursos e escalonamento de serviços, ou seja, *end-system* QoS [48]. Dessa forma, para se garantir QoS em *grid* devem ser levados em conta fatores de QoS rede e de sistema final (*end-system*).

Segundo von Lanzewski [46], um sistema de gerência de recursos para *grid computing* deve tentar satisfazer os seguintes requisitos relacionados a QoS:

- Reserva prévia de recursos: o sistema deve suportar mecanismos para a reserva prévia, imediata ou por demanda, de recursos. A reserva prévia é

particularmente importante quando se está lidando com recursos escassos;

- Política de reserva: o sistema deve suportar mecanismos que facilitem aos donos dos recursos compartilhados o reforço de políticas que dizem respeito a quando, como ou quem pode utilizar tais recursos, ao mesmo tempo que desacopla entidades de reserva e de políticas, o que aumenta a flexibilidade dos mecanismos de reserva;

- Protocolo de concordância (*agreement protocol*): o sistema deve assegurar aos clientes a sua reserva prévia de recursos e a qualidade esperada durante a utilização dos serviços. Tal segurança pode estar contida em protocolos de concordância;

- Segurança: o sistema deve evitar que usuários maliciosos penetrem ou alterem repositórios de dados que guardam informações sobre a reserva dos recursos, políticas de reserva ou protocolos de concordância;

- Simplicidade: a estrutura de QoS deve ter um projeto simples e um *overhead* mínimo em termos de computação, infra-estrutura, armazenamento e complexidade de mensagens;

- Escalabilidade: a abordagem deve ser escalável a um grande número de entidades, uma vez que o *grid* é uma estrutura de escala global, e haverá recursos e usuários participando do *grid* de forma dinâmica.

Usualmente um sistema em *grid* é composto por diversos recursos heterogêneos distribuídos em uma ampla rede, possivelmente a Internet. Existem alguns aspectos relativos a QoS de redes e de sistema final que devem ser considerados [47]. Quanto a redes, tem-se:

- Reserva e gerência de banda: o sistema deve utilizar mecanismos para gerenciar e alocar banda de rede, além de reservar recursos para o usuário;

- Estabilidade de roteamento: devido a instabilidades que podem existir na rede que interliga o *grid* e que podem degradar o nível de QoS de rede, devem ser providos mecanismos apropriados de modo a se manter um roteamento estável;

- Compartilhamento de *link*: aplicações devem ser capazes de compartilhar seus canais de transmissão, o que é um pré-requisito para aplicações de *grid* genéricas;

- Controle de congestionamento: devem existir na rede mecanismos para o controle de congestionamento, de modo a manter a transmissão de dados desbloqueada, assegurar eficiência na transmissão e garantir QoS na rede.

Quanto à QoS de sistema, os seguintes aspectos devem ser considerados [47]:

- Descoberta de serviços: no *grid*, todo recurso é geralmente abstraído como algum tipo de serviço. O sistema deve prover mecanismos para a descoberta de serviços de modo a atender aos clientes de acordo com seus requisitos de QoS;

- Escalonamento de serviços: o sistema deve ser capaz de alocar recursos aos clientes, de forma a escolher o serviço mais adequado para cada cliente;

- Controle em nível de aplicação: um conjunto de APIs podem ser providas à aplicação, de modo que a mesma possa monitorar o estado dos serviços, controlar recursos alocados e modificar os requisitos de QoS de forma dinâmica.

4.2. Serviços de Acordos

Um requisito para a execução de aplicações em um ambiente de *grid* é o acesso a recursos distribuídos geograficamente e pertencentes a diferentes domínios administrativos. Porém, as partes consumidora e fornecedora possuem interesses conflitantes na relação de compartilhamento. O cliente precisa conhecer o comportamento dos recursos e afetá-lo, muitas vezes exigindo garantias de nível do serviço prestado, enquanto o fornecedor deseja manter suas políticas locais de uso, sem divulgar determinadas informações sobre estas. Dessa forma, torna-se fundamental a existência de mecanismos e protocolos que permitam a negociação dinâmica entre clientes e fornecedores de políticas de acesso aos recursos [28]. Para tanto, podem ser utilizados os serviços de acordos (*agreements*).

Um cliente pode negociar com os serviços de acordos de forma a alcançar objetivos específicos em um ambiente de *grid*. O serviço avalia as requisições dos clientes no contexto de um conjunto de políticas de acesso, e um acordo é então criado representando uma concretização de tais políticas de forma a satisfazer os requisitos dos clientes. O acordo representa dessa forma um compromisso entre as partes para o fornecimento de uma quantidade mensurável de determinado serviço ou para a execução de determinada tarefa. Permite então que o cliente saiba o que esperar dos recursos sem que seja necessário o conhecimento detalhado sobre a carga do recurso ou sobre políticas de compartilhamento do fornecedor. Além disso, pode ser posteriormente renegociado e dinamicamente atualizado.

Como exemplos de serviços de acordos, tem-se os *Service Level Agreements* - SLAs, propostos por [38], e o *WS-Agreement* [27], proposto pelo *Global Grid Forum* – GGF.

4.2.1. *Service Level Agreements* – SLAs

Um SLA representa um acordo firmado entre as partes consumidora e provedora para o compartilhamento de recursos, definindo explicitamente os termos do acordo, as expectativas e as obrigações existentes entre ambas [41,48]. Os SLAs

comandam o compartilhamento de recursos específicos entre múltiplos grupos de usuários.

Confome Foster [41, existem definidos três diferentes tipos de SLAs:

- *Task Service Level Agreements* (TSLAs). É utilizado para negociações quanto ao desempenho de uma atividade ou tarefa. Representa um compromisso de execução de uma tarefa satisfazendo seus requisitos de recursos. Caracteriza uma tarefa em termos dos requisitos de recursos e passos seguidos pelo serviço utilizado;

- *Resource Service Level Agreements* (RSLAs). É utilizado para a negociação do direito de utilizar recursos. Representa um compromisso para o provimento de um recurso quando exigido. Pode ser negociado sem que seja especificada a tarefa executada com a utilização dos recursos. Utilizado para a reserva prévia de recursos;

- *Binding Service Level Agreements* (BSLAs). Utilizado para negociar a aplicação de um recurso a uma determinada tarefa. Representa um compromisso de utilizar um determinado recurso para a execução de determinada tarefa. Um BSLA associa uma tarefa definida por um TSLA a um recurso potencialmente capaz de suprir o que foi definido no RSLA.

Os três tipos de SLAs descritos acima dão suporte a um modelo de gerência de recursos interativo no qual pode-se independentemente submeter tarefas, obter promessas de recursos e ligar uma tarefa a um recurso [38]. Diferentes combinações deles podem representar diferentes abordagens de gerência de recursos.

Todos os três tipos de SLAs representam promessas de disponibilidade de recursos, porém com algumas diferenças. Os RSLAs representam apenas a promessa de disponibilidade dos recursos, sem que haja um planejamento para a utilização desses. Já os BSLAs representam uma promessa de alocação de recursos a tarefas. Por último, os TSLAs representam um compromisso do gerente de recursos em executar determinada tarefa com determinada quantidade de recursos. A figura 14 sintetiza os três tipos de SLAs.

Um RSLA corresponde a uma reserva de recursos, a qual pode ser utilizada futuramente na negociação de novos SLAs. Nenhum RSLA possui efeito a menos que seja utilizado na negociação de algum TSLA ou BSLA [38]. Um BSLA permite que se tenha um controle sobre a alocação de recursos a tarefas. A negociação de um BSLA não inicia nenhuma tarefa, sendo que esta deve ser criada separadamente. Por último, a negociação de um TSLA representa o compromisso de um gerente de recursos de executar a tarefa descrita com o nível de recursos descrito.

No momento da negociação de um SLA, são aplicadas as políticas locais de compartilhamento definidas pelos donos dos recursos a serem compartilhados. Apesar disso, não é necessário que todas as informações relativas a tais políticas sejam publicadas a possíveis consumidores, uma vez que um conhecimento detalhado das políticas não é necessário para a negociação de um SLA. É permitido que o dono dos recursos publique apenas as informações que deseje aos consumidores, as quais serão

utilizadas na descoberta de recursos no *grid*. Porém, mesmo que mantidos ocultos, aspectos referentes à política de compartilhamento podem ainda ser utilizados no ato da negociação do SLA.

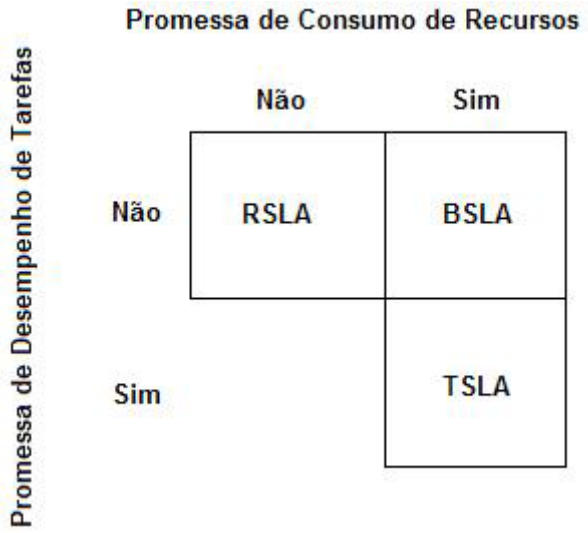

Figura 14: Os três tipos de SLAs e as promessas que representam [38].

4.2.2. *WS-Agreement*

O WS-*Agreement* é a especificação feita pelo *Global Grid Forum* (GGF) de um protocolo que utiliza mecanismos de *Web services* para o estabelecimento de acordos entre duas partes, como fornecedor e consumidor, utilizando também a linguagem XML para a especificação dos acordos e de modelos para esses [27]. O objetivo da especificação do WS-*Agreement* é a definição de uma linguagem e um protocolo para sinalizar a capacidade de provedores de serviços, para a criação de acordos baseados em ofertas de criação e para a monitoração de tais acordos em tempo de execução.

No âmbito do WS-*Agreement*, um acordo é definido como uma relação dinamicamente estabelecida e gerenciada entre duas partes, cujo objetivo é o provimento de algum serviço por uma das partes no contexto do acordo [27]. A criação de acordos é realizada de acordo com modelos de acordo definidos, os quais especificam aspectos customizáveis dos acordos, além de regras que devem ser seguidas na criação desses.

A estrutura de um acordo definido pelo WS-*Agreement* é composta de várias partes distintas, e é ilustrada na Figura 15 [27]. As partes componentes são: nome, contexto e termos. O nome é uma parte opcional, a qual faz apenas a identificação do acordo criado. O contexto é a parte responsável por guardar informações sobre o acordo. São armazenadas informações sobre as partes participantes, sobre o tempo de vida do acordo e sobre o modelo no qual sua criação foi baseada. Por último, o acordo é composto também pelos termos que o definem, os quais são sua principal parte. Os termos podem ser de serviço e de garantia.

Termos de serviços são componentes fundamentais dos acordos, uma vez que esses se referem aos serviços descritos por eles. São tais termos que definem quais funcionalidades serão providas sob um acordo de serviço. Já os termos de garantias definem as garantias de QoS associadas aos serviços descritos pelos termos de serviço do acordo. Cada termo de garantia deve estar associado a um de serviço, o que permite que um único acordo seja utilizado para a definição de níveis de QoS providos por diferentes recursos ou serviços.

Figura 15: Estrutura de um acordo WS-Agreement [27].

São definidos, para cada um dos tipos de termos dos acordos, diferentes estados de execução. Para os termos de serviço, são definidos os estados: não *pronto*, o qual indica que o serviço não pode ser usado no momento; *pronto*, o qual indica que o serviço está pronto para ser utilizado; e *completo*, segundo o qual o serviço não está sendo utilizado e quaisquer atividades requeridas a este estão finalizadas. O estado pronto se divide ainda em dois sub-estados: *processando*, o qual indica que o serviço está pronto e em atividade; e *parado*, que significa que o serviço está pronto, porém não está sendo usado.

Para os termos de garantia, são definidos os seguintes estados: *satisfeito*, o qual indica que a garantia está sendo satisfeita até o momento; *violado*, que indica que a garantia não está sendo cumprida; e *indeterminado*, que indica que no momento nenhuma atividade com respeito à garantia está ocorrendo de forma que não é possível determinar o seu cumprimento ou não.

O protocolo do WS-Agreement é definido por uma rodada de troca de mensagens de oferta e aceitação [27]. Pelo protocolo, a parte que inicia o acordo envia

uma oferta contendo os termos do acordo a ser criado, o qual pode ser aceito ou rejeitado por uma decisão unilateral da outra parte. Caso o acordo seja aceito, a parte que o iniciou recebe uma referência a ele; caso contrário, é retornada uma mensagem indicando uma falha.

4.3. Exemplos de Gerentes de Recursos para o *grid*

Nesta seção são apresentados alguns exemplos de gerentes de recursos existentes para utilização em ambiente de *grid* que possuem algum tipo de suporte à QoS.

4.3.1. *General-purpose Architecture for Reservation and Allocation* (GARA)

GARA é uma arquitetura para gerência de recursos desenvolvida sobre mecanismos fornecidos pelo *Globus Toolkit* 2 que provê aos usuários mecanismos para atingir QoS de ponta-a-ponta em aplicações computacionais [50, 52]. Provê mecanismos para a reserva de diferentes tipos de recursos, tais como rede, armazenamento e processamento. O mecanismo de reserva da GARA é uma garantia de que a aplicação que requisita uma reserva irá receber a QoS desejada do gerente de recursos [51]. Possibilita que uma reserva seja manipulada de forma a ser modificada, cancelada ou enfileirada.

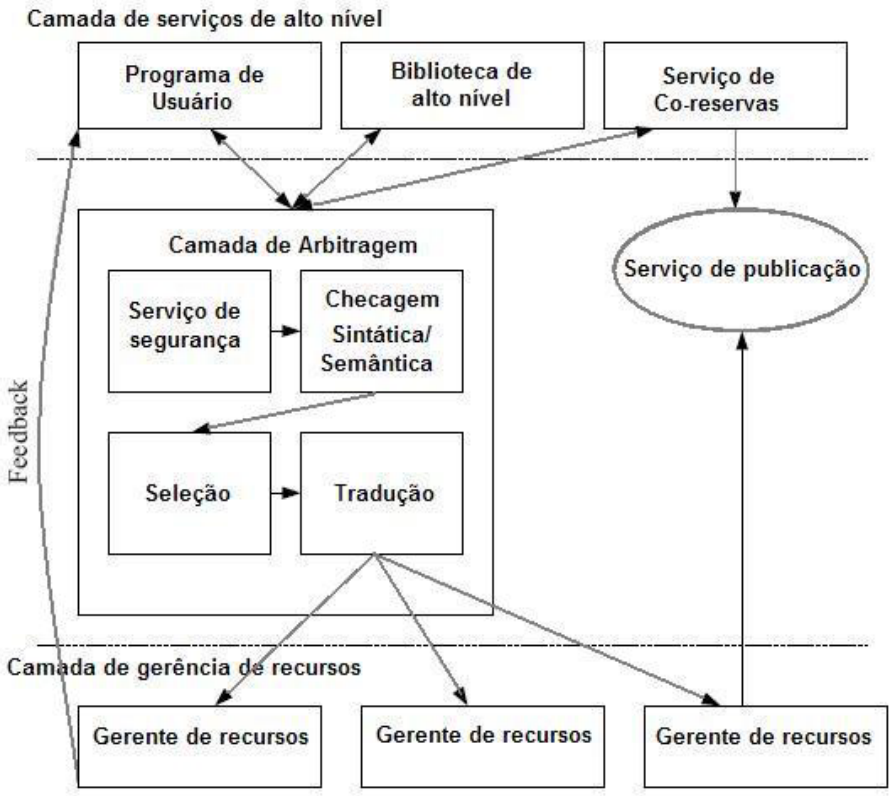

Figura 16: Arquitetura do GARA [52].

A arquitetura do GARA, vista na Figura 16, possui quatro características principais [52]. Primeiramente, fornece uma interface única para a reserva prévia ou imediata de diferentes tipos de recursos. Tal interface é provida pela camada de arbitragem. Em segundo lugar, devido à interface unificada para o alcance de diferentes tipos de QoS, torna-se simples a construção de serviços de alto nível sobre sua arquitetura. Outra característica de destaque é atingida devido à arquitetura ser dividida em camadas. Dessa forma, torna-se relativamente simples extendê-la quando do surgimento de novos tipos de recursos. Por último, é utilizada uma infra-estrutura de segurança que permite que todas as requisições de reservas sejam autenticadas e autorizadas.

Apesar de suas características e de seu mecanismo de reserva, a GARA possui algumas limitações para a utilização em ambiente de *grid* [51]. Primeiramente, ela não é compatível com a OGSA, ou seja, aplicações que executam utilizando a OGSA não podem fazer uso da GARA. Além disso, não suporta o conceito de protocolos de concordância (Seção 4.2). Para a reserva de diferentes recursos, diferentes chamadas à GARA devem ser feitas pela aplicação e a responsabilidade por gerenciar suas reservas passa a ser da própria aplicação. Além destas limitações, é importante ressaltar que o desenvolvimento da GARA está inativo, o projeto foi descontinuado.

4.3.2. *Grid Application Development Software* (GrADS)

Segundo [58], o GrADS é um projeto de pesquisa que envolve diversas instituições e cujo objetivo é simplificar a computação distribuída heterogênea. Possui uma arquitetura que possibilita a execução de aplicações no *grid*, contendo mecanismos que possibilitam o escalonamento e o reescalonamento dos recursos. Possui um metaescalonador responsável por receber possíveis escalonamentos feitos por escalonadores de aplicações e implementar políticas de escalonamento de modo a balancear os interesses de diferentes aplicações. Os objetivos de seu metaescalonador são [58]:

- interromper a execução de tarefas longas e que consumam muitos recursos para que tarefas curtas sejam acomodadas no sistema;
- facilitar a execução de novas aplicações parando a execução de certas aplicações concorrentes, fazendo que essas possam executar mais rapidamente;
- minimizar o impacto que a criação de novas tarefas pode ter em tarefas preexistentes;
- migrar aplicações em execução para diferentes nós em resposta a mudanças na carga do sistema, de forma a melhorar o desempenho ou evitar sua degradação.

O GrADS possui um *framework* para o desenvolvimento de aplicações que tenta minimizar o impacto da complexidade do *grid* sobre o código das aplicações [59]. As aplicações são encapsuladas em objetos chamados *Configurable Object Programs* (COP), os quais podem ser otimizados para a execução em um conjunto específico de

recursos. Cada COP possui o código da aplicação, um mapeador, o qual determina como mapear as tarefas de uma aplicação em um conjunto de recursos, e um modelo de desempenho, que estima o desempenho da aplicação em um conjunto de recursos. Além disso, o *framework* contém os chamados acordos de desempenho, os quais definem o desempenho esperado dos recursos disponíveis. Os principais módulos do GrADS estão ilustrados na Figura 17.

A execução de aplicações funciona da seguinte maneira [58,59]. Primeiramente, o usuário invoca uma rotina passando a aplicação a ser executada, encapsulada em um COP, assim como os parâmetros recebidos pela aplicação. A rotina chamada invoca um componente chamado *Resource Selector*, o qual faz acesso ao *Globus MetaDirectory Service* (MDS-2) e recupera uma lista das máquinas ativas. É feito então acesso a um serviço chamado *Network Weather Service* (NWS)[26], usado para monitorar periodicamente os componentes do sistema e descobrir o desempenho que pode ser fornecido por estes. Dessa forma consegue-se informações sobre as máquinas. O próximo passo é então conseguir permissão para que a aplicação possa utilizar o *grid*. Para isso, é contactado um serviço chamado *Permission Service*, o qual garante ou não permissão para que as aplicações utilizem o *grid*, o que ocorre de acordo com a quantidade de memória livre nos recursos. Caso a permissão seja negada à aplicação, uma nova seleção de recursos deve ocorrer.

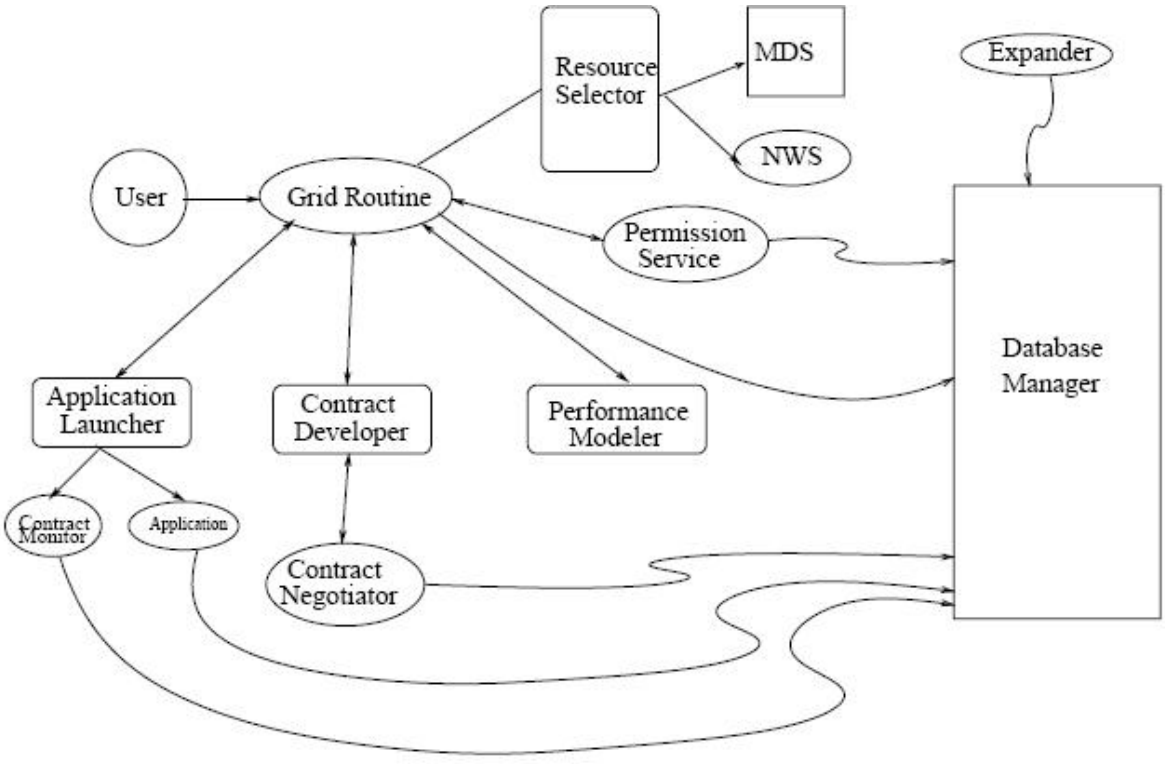

Figura 17: Arquitetura do GrADS [58].

Considerando que a aplicação obteve permissão para prosseguir, um componente do GrADS chamado *Performance Modeler*, juntamente com o mapeador e

o modelo de desempenho do COP relativo à tarefa, define a alocação dos recursos para a aplicação, chamada escalonamento específico de aplicação. A alocação definida, juntamente com os parâmetros da aplicação, é então passada ao componente *Contract Developer* como um contrato, o qual contacta o serviço *Contract Negotiator*, o qual aprova ou não o contrato. Um contrato pode ser rejeitado nas seguintes situações [58]:

- a aplicação conseguiu informações através do NWS antes de uma aplicação atualmente em execução ser iniciada;
- caso o desempenho de uma nova aplicação possa ser consideravelmente melhorado na ausência de uma outra aplicação em execução;
- caso uma aplicação em execução seja severamente afetada pela nova aplicação.

Uma vez aprovado o contrato, o componente *Application Launcher* recebe a aplicação, seus parâmetros e o escalonamento em nível de aplicação. Ele então distribui a aplicação nos recursos escalonados e inicia sua execução. A execução é monitorada pelo *Contract Monitor*, que detecta anomalias na execução e invoca, se necessário, um reescalonador, o qual toma ações corretivas. Dessa forma, o GrADS tenta garantir QoS às aplicações fazendo o reescalonamento das tarefas que não tenham suas exigências atendidas.

Existem ainda dois componentes que completam a estrutura do metaescalonador [58]: *Database Manager* e *Expander*. O *Database Manager* mantém um registro para cada aplicação submetida no *grid*, o qual contém informações sobre o estado da aplicação, sobre os recursos no instante em que a aplicação deu entrada no sistema, a lista de máquinas nas quais a aplicação encontra-se em execução, entre outras. Tais informações são utilizadas por outros componentes para a tomada de decisões de escalonamento. O componente *Expander* é um *daemon* que tenta melhorar o desempenho das aplicações em execução. Consulta o *Database Manager* periodicamente em busca de aplicações finalizadas. Cada vez que uma aplicação termina sua execução, o *expander* determina se aplicações em execução podem obter ganhos de desempenho caso sejam expandidas para os recursos liberados pela aplicação finalizada. Em caso positivo, interrompe a execução da aplicação e a continua com o novo conjunto de recursos.

4.3.3. Nimrod-G

O Nimrog-G é um sistema de gerência de recursos em *grid* desenvolvido por Buyya [60]. Inicialmente foi desenvolvido o sistema Nimrod, o qual utiliza uma linguagem declarativa parametrizada para a definição de tarefas. O Nimrod foi utilizado para conjuntos estáticos de recursos distribuídos, porém não suporta a utilização em *grids* dinâmicos, devido à grande variedade nas características dos recursos, políticas locais, entre outros. Foi então proposto o Nimrod-G, o qual utiliza o *Globus Toolkit* 2 para a descoberta dinâmica de recursos, execução e distribuição de tarefas no *grid*. É um escalonador baseado em economia computacional, ou seja, podem ser estabelecidos custos monetários para a utilização dos recursos compartilhados no *grid* e prazos para o término da execução de aplicações.

Possui uma arquitetura modular e baseada em componentes. Os principais componentes da arquitetura são ilustrados na figura 18: cliente, *Parametric Engine*, escalonador, *Dispatcher* e Empacotador de Tarefas.

O cliente funciona como uma interface de usuário para o controle e supervisão da aplicação em consideração. O usuário pode variar os parâmetros da aplicação relativos a custo e tempo de execução, os quais têm influência direta sobre as decisões do escalonador na seleção de recursos. Serve também como uma ferramenta de monitoração das tarefas, mostrando o *status* de todas elas, as quais podem ser vistas e controladas.

O *Parametric Engine* funciona como um agente de controle de tarefas e é o componente central onde a aplicação é gerenciada. É responsável pela parametrização da aplicação, pela criação das tarefas e manutenção do estado destas. Recebe como entrada um plano para a aplicação, descrito em linguagem parametrizada, e com isso gerencia a aplicação. Mantém o estado de toda a aplicação e garante que esse estado seja persistido em um repositório, o que permite que a aplicação seja reiniciada em caso de falha.

O Escalonador é responsável pela descoberta e seleção dos recursos e por atribuir tarefas aos recursos selecionados. A descoberta de recursos é feita com a utilização do MDS-2, sendo identificada uma lista de recursos e mantidas informações sobre o estado dos recursos. São selecionados então os recursos que respeitam o prazo definido para o término da aplicação e que minimizam o custo monetário das computações.

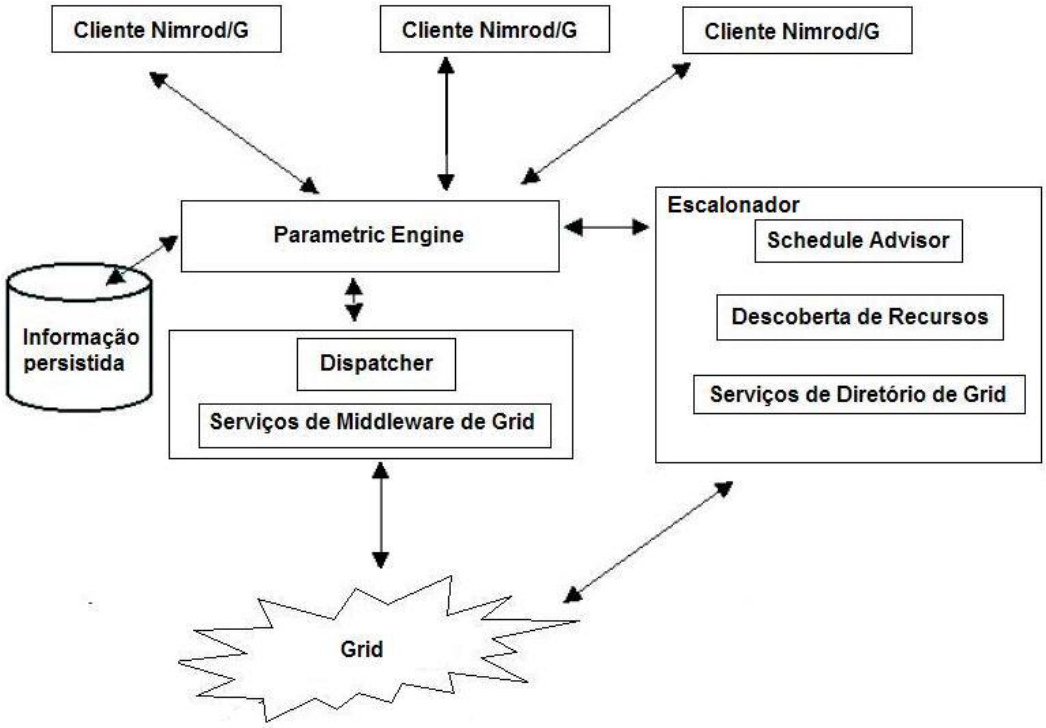

Figura 18: Arquitetura do Nimrod-G [60].

O *Dispatcher* é responsável, primeiramente, por iniciar a execução das tarefas nos recursos selecionados. Após iniciada a execução, é responsável por atualizar

periodicamente a situação das tarefas em execução para a *Parametric Engine*. É responsável também por iniciar o componente Empacotador de Tarefas.

O último componente é o Empacotador de Tarefas. Este componente é responsável por servir de base às tarefas das aplicações e seus dados, iniciar a execução de tarefas nos recursos definidos e por enviar os resultados da execução à *Parametric Engine* através do *Dispatcher*.

No Nimrod-G, enquanto as aplicações definem prazos para o término de suas execuções e a quantia que se está disposto a pagar pela execução, os recursos definem qual o custo para o cliente para que possam ser utilizados. Dessa forma, os recursos são escalonados de acordo com os seguintes passos [61]:

- descoberta: o conjunto de recursos de mais baixo custo capazes de respeitar o prazo de execução é identificado. Este passo produz uma lista dos recursos aos quais tarefas devem ser submetidas, ordenados pelo custo de utilização;

- alocação: as tarefas desalocadas são alocadas aos recursos identificados no passo anterior;

- monitoração: o tempo de finalização de cada tarefa submetida é monitorado, estabelecendo uma taxa de execução para cada recurso;

- refinamento: informações sobre as taxas definidas na monitoração são utilizadas para atualizar as estimativas do tempo de execução em diferentes recursos e, consequentemente, o tempo esperado para o término da execução das tarefas.

Os passos acima são executados até que o prazo de execução da aplicação expire ou até que o custo ultrapasse o preço que o usuário se dispôs a pagar.

4.3.4. O *Framework* AGMetS

O *framework* AGMetS, proposto por Nainwal et. al. [64], tem como objetivo realizar um meta-escalonamento adaptável com provimento de QoS no *grid*, de forma que seja tolerante a falhas. O escalonamento é realizado com base na disponibilidade dos recursos. A cada nó componente do *grid* é associado um vetor indexado contendo probabilidade de falha de seus recursos individualmente, o que pode ser usado para a previsão de falhas. As decisões de escalonamento de recursos são tomadas de acordo com as disponibilidades dos recursos indicadas nos elementos do vetor.

Cada vez que uma nova aplicação deve ser escalonada no *grid*, a disponibilidade de diferentes recursos requerida pela aplicação é computada a partir de seus requisitos de QoS. O AGMetS então recolhe informações sobre todos os nós do *grid* e as compara com os requisitos da aplicação de forma a encontrar o nó mais adequado para a execução. A aplicação é então submetida ao nó escolhido e a disponibilidade dos recursos utilizados monitorada até que a execução esteja terminada.

A disponibilidade necessária então é garantida pelo AGMetS durante toda a execução da aplicação. Caso os recursos utilizados para a execução da aplicação não sejam capazes de fornecer a disponibilidade necessária, a infraestrutura do AGMetS migra a aplicação para a execução em outros recursos que o façam.

Para que as aplicações possam executar com tolerância a falhas, o AGMetS faz uso da técnica de *checkpointing*, ou seja, em determinado ponto salva o estado de execução das aplicações de modo que possa reiniciar sua execução a partir do ponto em que o estado foi salvo. Porém, diferentemente da maioria dos casos [64], o estado da aplicação não é salvo periodicamente, mas sim quando ocorre uma diminuição na disponibilidade dos recursos, o que evita o *overhead* causado quando o estado é salvo periodicamente. Quando há a ocorrência de falhas dos recursos utilizados pela aplicação ou os requisitos de QoS não são alcançados, novos recursos são dinamicamente escalonados e a aplicação é migrada para a utilização desses, continuando sua execução.

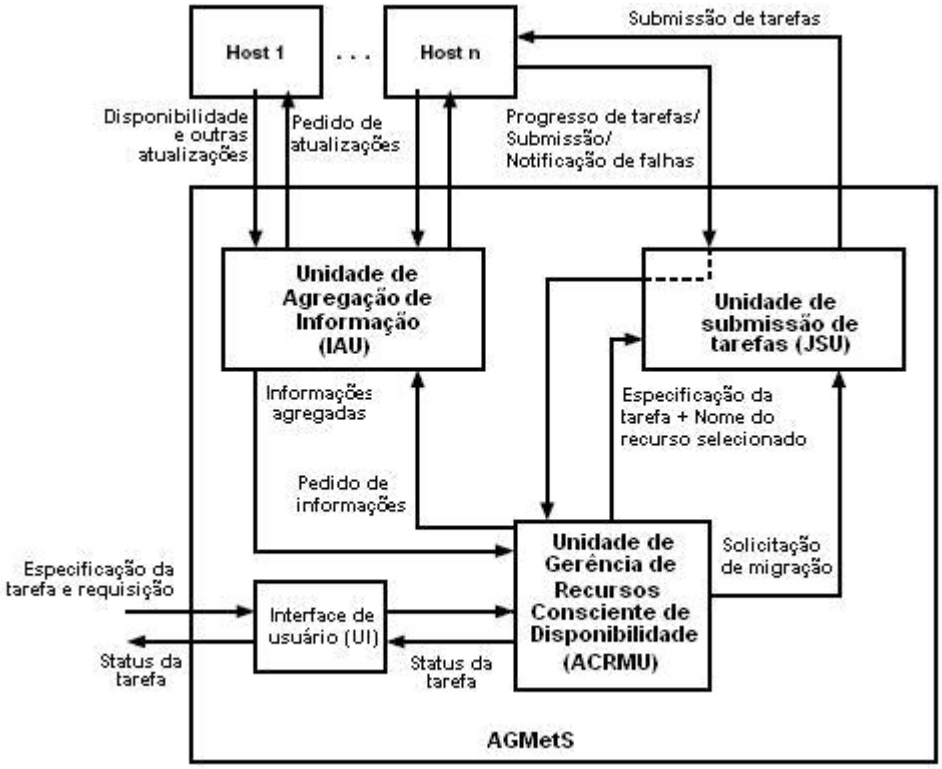

Figura 19: Arquitetura do AGMetS [64].

A arquitetura do AGMetS, ilustrada na Figura 19, é composta por quatro componentes principais [64]: uma interface de usuário (UI), utilizada para que o usuário possa interagir com o sistema; a unidade de agregação de informação (IAU), reponsável por coletar informações sobre os recursos componentes dos nós do *grid*; a unidade de gerência de recursos consciente de disponibilidade (ACRMU), a qual compara as informações coletadas pela IAU com os requisitos da aplicação e escolhe o nó mais adequado para a execução, além de passar esta informação à ultima unidade

componente da arquitetura; e a unidade de submissão de tarefas (JSU), a qual submete as aplicações aos nós selecionados.

Uma idéia central ao projeto do AGMetS é a existência do vetor indexado de disponibilidade dos recursos, o qual indica suas probabilidades de falha. Dessa forma, é imprescindível que se tenha em execução em todo nó componente do *grid* um programa que determine a disponibilidade dos recursos.

4.3.5. *Job Submission Service* (JSS)

O JSS [49] consiste em um *broker* e um serviço de submissão de tarefas projetado de forma a que possa ser utilizado com diferentes *middlewares* de *grid*, sendo as partes dependentes de *middleware* concentradas em componentes de menor importância em sua arquitetura, estando disponíveis componentes para a utilização com o GT4. Os serviços propostos são fortemente baseados em padrões de *grid* e *Web Services*, tais como *Job Submission Description Language* – JSDL [63], WSRF (Seção 3.7.3), WS-Agreement (Seção 4.2.2) e GLUE [62], e implementados com a utilização do GT4.

O *broker* funciona de maneira descentralizada, ou seja, sem a influência de nenhum controle central e independente de quaisquer outros *brokers* atuando sobre os mesmos recursos, e seleciona os recursos que tenham o menor *Total Time to Delivery* (TTD) estimado, o qual é composto pela estimativa dos tempos de execução, transferência de arquivos e realização de reservas dos recursos desejados.

Para a descrição das tarefas submetidas por clientes e seus requisitos é utilizada a JSDL. As especificações componentes do WSRF são utilizadas da seguinte maneira: tarefas e reservas de recursos, ou seja, garantias de disponibilidade de recursos às aplicações, são representadas por WS-*Resources*; informações sobre tarefas submetidas e reservas de recursos criadas são modeladas por WS-*ResourceProperties*; mecanismos definidos na especificação WS-*ResourceLifetime* são utilizados para a implementação de reservas em dois passos; por último, WS-*BaseFaults* são utilizados para representação de erros. Para a negociação de acordos é utilizado o padrão WS-*Agreement*. Para a representação de informações sobre os recursos, coletadas na fase de descoberta de recursos, é utilizado o formato GLUE.

A arquitetura dos serviços é composta por sete componentes: *JobSubmissionService*, *Dispatcher*, *Reserver*, *Broker*, *DataManager*, *InformationFinder* e *Submitter*. Existe ainda uma implementação da especificação WS-*Agreement*, a qual deve residir nos recursos do *grid*. A arquitetura e os fluxos de informações entre os componentes podem ser visualizados na Figura 20.

O *JobSubmissionService* é o único componente do módulo de submissão de tarefas ao qual os clientes têm acesso. É responsável por armazenar informações sobre as tarefas submetidas e provê uma interface *Web Service* para acesso. A única operação exportada pela interface é a submissão de tarefas, a qual recebe uma descrição JSDL da tarefa submetida e um documento opcional contendo preferências do usuário. Para cada tarefa submetida com sucesso é criado um WS-*Resource*, com suas informações descritas por WS-*ResourceProperties*.

O componente *InformationFinder* possui como objetivo descobrir quais recursos estão disponíveis no *grid* e por recuperar informações detalhadas sobre esses. Primeiramente é realizada a descoberta de recursos. Cada recurso identificado é então consultado por informações mais detalhadas. Informações sobre *hardware* e *software*, políticas de uso, entre outras informações, são recuperadas.

O *Broker* é responsável por selecionar os melhores recursos para a execução de cada tarefa, de acordo com características das tarefas e preferências do usuário. Realiza três operações: validação das descrições das tarefas submetidas; filtragem dos recursos que preenchem os requisitos da aplicação; e classificação dos recursos quanto à adequabilidade para a execução das tarefas, a qual é definida de acordo com o TTD estimado para a execução em determinado recurso.

O componente *DataManager* é responsável por gerenciar todas as operações de transferência de dados relacionadas à submissão de tarefas. Realiza operações de resolução da localização física de arquivos replicados, previsão do tempo de duração de transferências de arquivos, além de determinar o tamanho de arquivos físicos.

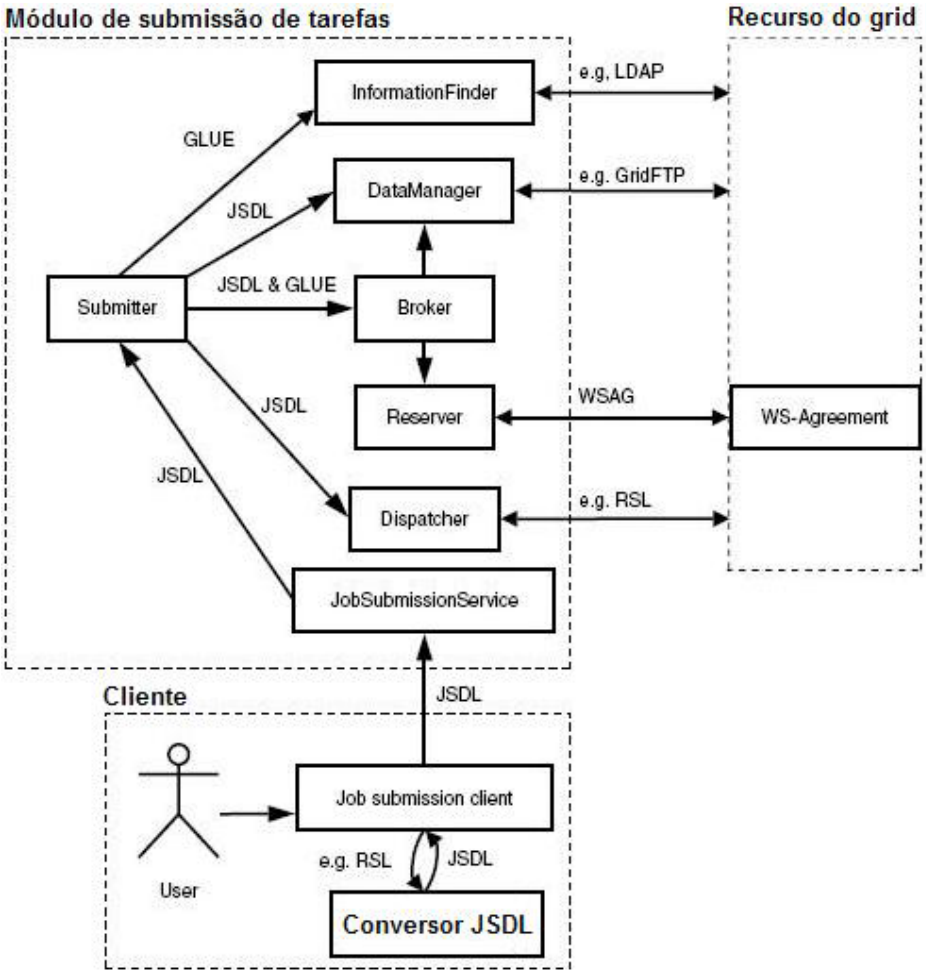

Figura 20: Arquitetura do JSS [49].

O *Reserver* possui uma API para a reserva avançada de CPU nos recursos selecionados. Três operações são suportadas: a criação, confirmação e cancelamento de reservas. As reservas são então criadas em duas fases, sendo a primeira a criação e a segunda a confirmação. Caso uma reserva não seja confirmada, ela é desfeita. Tais

operações são implementadas utilizando chamadas ao módulo WS-*Agreement*. O módulo possui ainda um repositório contendo as reservas criadas. Uma vez que uma reserva seja confirmada, todas as outras relativas à mesma tarefa são então canceladas.

O *Dispatcher* é responsável pelo envio da requisição de execução de tarefa aos recursos selecionados. Por ser uma operação dependente de *middleware* de *grid*, o componente exporta uma interface que deve ser implementada de acordo com o *middleware* utilizado no recurso. A interface possui uma operação *dispatch*, para o envio das requisições aos recursos, e uma operação de tradução da descrição da tarefa em JSDL para outra linguagem de descrição usada pelo recurso.

O *Submitter* coordena todo o processo de submissão de tarefas. Quando o *JobSubmissionService* recebe uma requisição, esta é passada ao *Submitter*. Este então invoca o *Broker* para a validação da descrição da tarefa requisitada. Caso seja válida, o *Submitter* faz então uma chamada ao *InformationFinder* para a recuperação dos recursos disponíveis no *grid*. O *Broker* é novamente invocado então para a filtragem dos recursos inadequados para a execução da tarefa e para a ordenação dos recursos de acordo com a adequabilidade para a execução, passo esse que pode requerer a realização de reservas de recursos, feita pelo *Reserver*. O *Submitter* chama então o *Dispatcher* para que a requisição seja passada ao recurso mais adequado. Terminados os passos acima, o *Submitter* retorna então ao *JobSubmissionService* um identificador da tarefa submetida.

O módulo WS-*Agreement* inclui a implementação de interfaces e *porttypes* definidos na especificação WS-*Agreement* (Seção 4.2.2). É responsável pela negociação de reservas e pela representação das reservas avançadas criadas nos escalonadores locais. Diferentemente dos módulos descritos anteriormente, os quais se localizam nas máquinas executando o serviço de submissão de tarefas, o WS-*Agreement* deve estar localizado nos recursos compartilhados no *grid*. A implementação inclusa no serviço porém não inclui o *porttype AgreementState*, responsável pela monitoração do estado dos acordos.

Para que se possam realizar reservas pelo serviço de submissão de tarefas, é necessário que o escalonador de recursos local suporte tais reservas. Além disso, é necessária a existência de *plugins* específicos ao escalonador para a realização das reservas, o que permite que o serviço se adapte a diferentes escalonadores. Além disso, na descrição das requisições submetidas, a realização das reservas deve estar especificada. Outra particularidade é que apenas a reserva de CPU é suportada pelo serviço, não sendo suportadas reservas de outros tipos de recursos.

4.3.6. G-QoSm

O *Grid Quality of Service Management* (G-QoSm) é um *framework* desenvolvido para dar suporte à gerência de QoS em ambientes de *grid* no contexto da OGSA [53]. Possui mecanismos para a realização de reservas de recursos, tanto de forma avançada quanto por demanda. Seu funcionamento é dividido em três fases operacionais: estabelecimento, atividade e término. Na primeira fase, o cliente define qual o serviço desejado e seus requisitos de QoS. É executada então a descoberta de recursos de acordo com os requisitos definidos e negociados acordos para a aplicação do usuário. Na fase de atividade, quando está sendo executada a aplicação, podem ser

realizadas operações de QoS adicionais, tais como a monitoração dos acordos, adaptação e ainda renegociações de acordos. A monitoração dos acordos firmados não é realizada, porém, diretamente pela G-QoSm. A fase de término consiste na finalização da sessão de QoS devido à expiração de reservas, violação de acordos ou o término do serviço prestado, sendo os recursos liberados para a utilização por outros clientes.

A arquitetura do G-QoSm pode ser visualizada na Figura 21. A arquitetura consiste em uma parte cliente e uma provedora de serviços. O principal componente que compõe o *framework* é o QoS *Grid Service* (QGS), o qual é um *grid service* compatível com a OGSA e que provê funcionalidades de QoS, tais como negociação, reserva e alocação de recursos com determinados níveis de qualidade. O acesso a cada recurso capaz de prover QoS no sistema é feito através de um QGS. Um QGS suporta dois tipos de alocação de recursos: domínio de recursos, na qual o cliente especifica uma porcentagem de utilização dos recursos compartilhados; e domínio de tempo, quando o cliente especifica um intervalo de tempo pelo qual o recurso é alocado de maneira exclusiva.

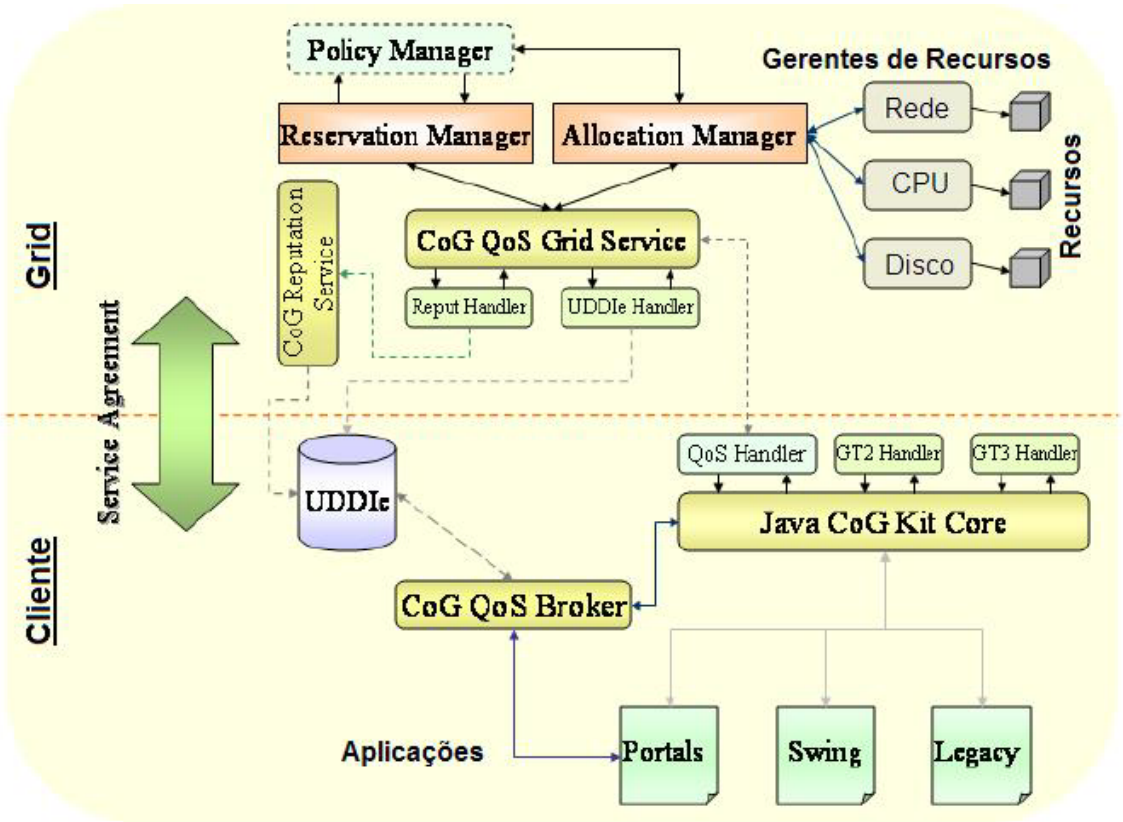

Figura 21: Arquitetura do G-QoSm [53].

Quando o QGS recebe uma requisição de reserva, tal requisição é repassada ao componente *Reservation Manager*, o qual contacta o componente *Policy Manager* para a validação do pedido de reserva e verifica então a disponibilidade dos recursos requisitados. Caso a reserva seja possível, o *Reservation Manager* retorna uma resposta positiva ao QGS, o qual pode então propor um acordo de serviço no sistema. Caso o

acordo seja aceito, os recursos requisitados são reservados e um acordo contendo a especificação da reserva é gerado e retornado ao cliente.

Quando é recebida pelo QGS uma requisição de alocação de recursos, este verifica se o usuário fez a reserva dos recursos de acordo com o acordo criado. Em caso positivo, o QGS repassa ao *Allocation Manager*, componente reponsável por interagir com os gerentes de recursos para a alocação e liberação de recursos, a requisição de alocação, sendo que esse a repassa ao gerente dos recursos a serem alocados.

O *framework* G-QoSm é baseado na implementação da OGSI. Dessa forma, o QGS e outros serviços existentes no container OGSI devem ser publicados em algum registro para que possam ser utilizados por outros serviços. Para tanto, é utilizado o registro *Universal Description Discovery and Integration Extension* (UDDIe) [37], um registro de *Web Services* que possibilita aos provedores de serviços meios de publicar seus serviços com propriedades de QoS, possibilitando a busca por serviços com base em propriedades de QoS.

Para a implementação do G-QoSm é utilizado o Java CoG Kit Core [7], o qual é um *middleware* baseado em Java utilizado para prover acesso a diferentes implementações de *grid*, tais como GT2 e GT3. Provê APIs contendo funcionalidades do *grid*, tais como execução remota de tarefas e transferência de arquivos, sem qualquer consideração à implementação de *grid* utilizada. Possui ainda suporte a operações de negociação e renegociação de QoS, execução de tarefas e redirecionamento de dados entre as aplicações e o QGS.

É definido no G-QoSm um protocolo para a negociação de QoS, o qual define a sintaxe e a semântica das mensagens trocadas entre as entidades envolvidas no processo de negociação, as quais são o cliente, o serviço de QoS (QGS) e o provedor de serviços, objetivando alcançar um acordo entre estas. O serviço de QoS é a entidade central, coordenando o processo de negociação entre o cliente e o provedor. Suporta algumas operações a serem utilizadas pelo cliente, sendo a interação entre os dois baseada na troca de mensagens XML. As operações suportadas são: consulta, reserva, atualização e cancelamento. As garantias de QoS negociadas são representadas por acordos de serviço (Seção 4.2), mais especificamente por RSLAs (Seção 4.2.1).

A operação de consulta consiste em realizar uma pesquisa no registro de forma a encontrar serviços com determinadas características de QoS. Caso seja encontrado algum serviço, este é reservado temporariamente e é retornada uma referência à consulta; caso contrário, é realizada uma nova consulta por serviços que tenham algumas das características desejadas. Após a consulta, é realizada então a operação de reserva, a qual confirma as reservas feitas na operação de consulta.

A operação de atualização é utilizada para a renegociação de parâmetros de QoS. Caso seja utilizada para a diminuição das exigências de QoS, a operação é garantida. Porém, quando se é necessária a alocação de mais recursos, a renegociação é tratada como uma nova aquisição, sendo equivalente a uma operação de consulta seguida por uma de reserva. Por último, a operação de cancelamento é utilizada para cancelar uma operação de reserva, o que deve ser feito antes que o serviço comece a ser prestado.

As operações descritas acima são os elementos fundamentais do processo de negociação e definem o protocolo de negociação. Quando o serviço de QoS recebe uma operação de consulta, verifica se o recurso pode ser reservado de acordo com sua política de acesso e sua capacidade de provimento de serviços. Caso a reserva seja possível, retorna uma referência utilizada pela operação de reserva, executada posteriormente, e que de fato realiza a reserva dos recursos e retorna uma referência ao acordo (*agreement*) realizado. O cliente pode então cancelar o acordo utilizando a operação de cancelamento, ou mesmo renegociar o acordo, pela operação de atualização.

4.4. Quadro comparativo

Sistema	**Reserva**	**Acordos**	**QoS**	**Middleware de *grid***	**Monitoração de Acordo**
GARA	Sim	Não	CPU, rede e disco	GT2	Não
GrADS	Não	Sim	desempenho geral da aplicação	GT2	Sim
Nimrod-G	Não	Não	Não	GT2	Não
AGMetS	Não	Não	CPU e disco	GT3	Não
Job Submission Service - JSS	Sim	WS-Agreement	CPU	GT4	Não
G-QoSm	Sim	RSLA	CPU	GT2, GT3	Não

Tabela 1: Comparação entre características dos gerentes de recursos estudados.

Fazendo uma análise dos sistemas em relação a garantias de QoS às aplicações executadas, nota-se que apenas três dos sistemas estudados realizam a reserva de recursos. Os outros sistemas que provêem QoS tentam fazê-lo reescalonando as aplicações que não encontrem os níveis de recursos disponíveis para a execução. Há ainda o Nimrod-G, qual não provê QoS às tarefas submetidas. Dentre os que realizam reservas, apenas o JSS e o G-QoSm utilizam também o conceito de acordos de serviço (Seção 4.2), sendo que o primeiro utiliza o WS-Agreement (Seção 4.2.2) e o segundo RSLA (Seção 4.2.1). O GrADS, apesar de trabalhar com acordos, os quais são de um tipo específico deste gerente de recursos, não realiza a reserva. Dentre os sistemas que provêem QoS, todos são capazes de garantir a disponibilidade de pelo menos CPU.

Em relação à *middleware* de *grid*, todos os gerentes utilizam o *Globus Toolkit*. Porém, apenas o JSS utiliza sua versão mais nova (GT4). Dentre os sistemas estudados, apenas o GrADS realiza a monitoração de acordos.

Capítulo 5

Projeto do GrAMoS

O objetivo desta dissertação é o projeto e avaliação do GrAMoS (*Grid Agreement Monitoring Service*), um mecanismo para a monitoração da execução de tarefas submetidas ao *grid,* de modo a possibilitar a verificação do cumprimento de acordos de serviço firmados entre as partes consumidora e provedora (Seção 4.2). Além de monitorar os recursos envolvidos no acordo, o GrAMoS possibilita a execução de determinadas ações quando ocorre o não cumprimento do mesmo. O serviço de monitoração descrito neste capítulo foi projetado para ser facilmente integrável a um sistema de gerenciamento de recursos em *grid*.

O projeto do GrAMoS baseou-se em duas premissas básicas. Primeiramente, a detecção de violações de acordos de serviço deve ser feita levando-se em conta que, em ambientes de *grid*, há freqüentemente a ocorrência de variações temporárias e dinâmicas de carga nos recursos. Tais variações não implicam necessariamente em violações de acordos de QoS. Em segundo lugar, a ação tomada quando é verificada a violação de acordos deve ser especificada pelo usuário.

Para que seja garantida a primeira premissa, são determinados aleatoriamente instantes dentro de intervalos definidos para a verificação de acordos e um número configurável de violações a serem detectadas para que seja constatada a quebra do acordo. Para a garantia da segunda premissa, a ação a ser tomada na quebra de acordo é definida de forma flexível. Essas três características nortearam o projeto do GrAMoS e são descritas nas Seções 5.1, 5.2 e 5.3, respectivamente.

5.1. Medidas em tempo aleatório dentro de um intervalo

Em sistemas homogêneos e com pouca variação de carga local, é comum determinar que sistemas de monitoração façam medidas de utilização de recursos a cada *n* segundos. Em um ambiente altamente dinâmico e heterogêneo como o *grid*, tal política pode levar a conclusões errôneas, no caso da ocorrência de picos periódicos e instantâneos de carga [18]. Por esta razão, técnicas mais elaboradas devem ser utilizadas.

Dentre as técnicas de monitoração estudadas, a técnica de Amostragem Randômica Estratificada [18] permite que o efeito dos picos periódicos seja bastante reduzido e, por esta razão, foi escolhida. A técnica consiste na definição de um intervalo de tempo dentro do qual deve ser feita uma amostragem da utilização de recursos por uma certa aplicação. Dentro do intervalo definido, é então escolhido, de maneira aleatória, um instante para a realização da amostragem. Cada vez que se repete o intervalo, um novo instante é definido aleatoriamente e uma nova amostragem é realizada.
Uma comparação entre o funcionamento de técnicas tradicionais de monitoração e a técnica escolhida pode ser visualizada na Figura 22.

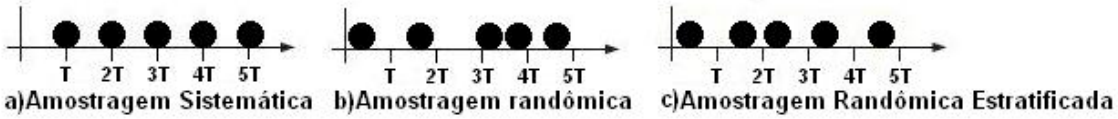

Figura 22: Técnicas de monitoração [18].

A Figura 22 ilustra três diferentes técnicas de monitoração [18]. A técnica de Amostragem Sistemática (Figura 22 (a)) consiste em definir os instantes de monitoração baseando-se em alguma função determinística. Já a técnica de Amostragem Randômica (Figura 22 (b)) define os instantes de monitoração de forma totalmente aleatória. Por fim, a técnica de Amostragem Randômica Estratificada (Figura 22 (c)), utilizada pelo serviço proposto, consiste na definição de um intervalo de tempo fixo, dentro do qual um instante para a amostragem é definido de forma aleatória.

5.2. Número configurável de quebras de acordo detectadas

A detecção de violação de acordo implica em um alto *overhead* de comunicação no *grid*, pois deve ser tomada alguma providência, como, por exemplo, a realização de uma nova negociação. Além disso, em diversos sistemas, a execução da tarefa do *grid* onde foi detectada a violação é suspensa, causando potenciais retardos na obtenção de resultados.

Muitas vezes, o instante da medida ocorre exatamente em uma variação anormal e temporária de carga. Caso seja concluído, nesse momento, que houve uma quebra de acordo, um alto *overhead* será gerado. Por essa razão, foi definido o parâmetro *unfulfillmentTimes*, que determina a quantidade de vezes que a quebra de acordos deve ser detectada para que o sistema de monitoração conclua que houve a violação do acordo e que medidas devem ser executadas. Caso o usuário deseje que a detecção seja instantânea, basta especificar o valor de *unfulfillmentTimes* como 1. Na maioria dos casos, porém, é aconselhável que um valor maior que 1 seja especificado para o parâmetro. Essa técnica, aliada à descrita na Seção 5.1, permite que os efeitos de variações bruscas e intermitentes de carga sejam bastante reduzidos.

A cada amostragem realizada, os valores de utilização de recursos recuperados são então comparados com os definidos no acordo firmado, podendo então verificar se as exigências estão ou não sendo cumpridas.

5.3. Ação flexível na quebra de acordo

Ao se detectar uma quebra de acordo, diversas ações podem ser tomadas, dentre elas [41]: forçar a finalização da tarefa no *grid*; suspender a execução da tarefa, solicitando a renegociação do acordo; continuar a execução da tarefa do *grid*, registrando a violação do acordo em um *log*.

Foi considerado que a ação a ser tomada é dependente da aplicação de *grid*, das políticas de cada organização virtual e das políticas de gerência de recursos do *grid*. Dessa forma, foi projetada uma interface genérica de ação, denominada *AgreementUnaccomplishmentAction* (Seção 5.4.1), que permite a definição de diferentes ações, desde que sua interface seja respeitada.

5.4. Projeto do serviço de monitoração

O GrAMoS foi projetado como um *Web service* e é composto por uma parte cliente e pelo serviço de monitoração propriamente dito. A parte cliente é responsável apenas por fazer a chamada, utilizando mecanismos de *Web service*, ao serviço de monitoração, enquanto a parte servidora é responsável por monitorar a execução e o consumo de recursos das aplicações submetidas, bem como por executar ações definidas pelo usuário, no caso de violações de acordos.

O serviço foi projetado e desenvolvido para ser hospedado no container *Web services* do *Globus Toolkit* 4 (Seção 3.7.3.1) e para ser utilizado em conjunto com a ferramenta JSS (Seção 4.3.5). A Figura 23 ilustra a disposição do serviço de monitoração junto aos demais serviços e aplicações utilizadas.

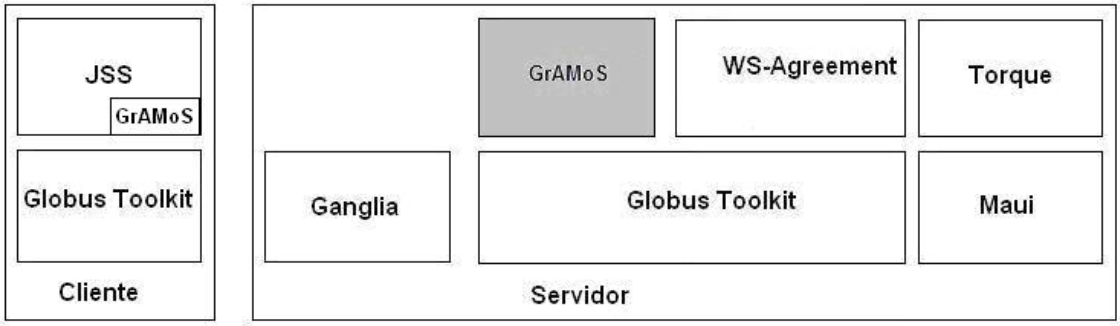

Figura 23: Disposição das aplicações instaladas para a utilização do GrAMoS.

Conforme mostrado na Figura 23, estão instalados na máquina cliente o componente JSS e o Globus Toolkit 4. O JSS se executa sobre o Globus, uma vez que o primeiro está hospedado no container de *Web services* do último. A parte cliente do serviço de monitoração foi integrada ao JSS. Foi realizada uma alteração no módulo *Submitter* (Seção 4.3.5) para que quando ocorra a submissão de tarefas, caso seja criado um acordo de serviço, seja feita uma chamada ao GrAMoS de modo que seja monitorado o cumprimento de tal acordo.

Já na máquina servidora, conforme dito anteriormente, há instalados o módulo WS-Agreement (Seção 4.3.5), o Globus Toolkit 4, o Ganglia Monitoring Service [17], o Torque Resource Manager [16] e o escalonador Maui [15]. O WS-*Agreement* executa-se sobre o Globus por estar hospedado em seu container de *Web services*. O Ganglia, por sua vez, interage com o Globus, fazendo a recuperação de informações junto a esse. O Globus submete as tarefas ao Torque para a execução

propriamente dita. E finalmente, o Torque utiliza as decisões de escalonamento tomadas pelo Maui para que possa realizar a execução das tarefas submetidas a ele, ou seja, a execução de aplicações pelo Torque depende diretamente das decisões tomadas pelo Maui.

O serviço proposto exporta uma interface *Web service* para o acesso e requisições de monitoração de tarefas submetidas. A interface possui apenas um método acessível, o *initiateJobMonitoring*, o qual recebe como parâmetros a identificação da tarefa submetida e uma chave para a recuperação do acordo de serviço firmado:

public void initiateJobMonitoring(*String task_id, int agreement_key*);

A identificação da tarefa (*task_id*) é obtida a partir do Globus e a chave de recuperação do acordo é obtida através do JSS,o qual a recupera a partir do módulo *WS-Agreement* em execução na máquina servidora.

5.4.1. Componentes do Serviço

O serviço proposto possui quatro componentes: o *MonitoringService* (MS), ou núcleo do serviço, responsável por receber requisições de monitoração de tarefas e por iniciar tais monitorações; o *Monitoring Thread* (MT), que é uma linha de execução, ou *thread*, criada para a monitoração de cada tarefa submetida; o *Scheduler Information Fetcher* (SIF), responsável por recuperar junto ao escalonador da rede local informações sobre a utilização de recursos na máquina onde a tarefa monitorada está sendo executada e por repassar tais informações ao módulo *Monitoring Thread*; e, por fim, o *Unaccomplishment Action* (UA), responsável pela tomada de ações na ocorrência de quebras de acordos.

O componente *Unaccomplishment Action* (UA) é definido pela interface genérica *AgreementUnaccomplishmentAction*, a qual exporta dois métodos:

public void agreementStartTimeAction(*String tak_id*)*;*

public void agreementUnfulfillmentAction(*String task_id*).

Ambos os métodos recebem como parâmetro o identificador da tarefa (*task_ id*) definido pelo Globus no momento da submissão e devem ser chamados em resposta ao não cumprimento de acordos, porém em situações distintas (ver Seção 5.4.3). Diferentes implementações da interface podem ser utilizadas de acordo com a configuração do serviço (Seção 5.4.2), o que permite que o administrador possa escolher entre diferentes ações a se tomar em caso de quebra de acordo, garantindo assim a premissa exposta na Seção 5.3. Foi feita uma implementação *default* para a interface, na qual os métodos apenas registram em arquivo de *log* que ocorreu uma quebra de acordo.

O componente *Scheduler Information Fetcher* (SIF) é definido pela interface *SchedulerInformationFetcher*, a qual exporta apenas um método:

public JobExecutionInfo getJobExecutionInfo();

O método não recebe parâmetros e deve retornar um objeto do tipo *JobExecutionInfo,* contendo diversas informações sobre a execução de determinada tarefa (Seção 5.4.3). A implementação do módulo também é um parâmetro configurável do serviço (Seção 5.4.2), o que permite que se tenha uma independência de escalonador local utilizado, necessitando apenas de diferentes implementações para diferentes escalonadores. Para este trabalho, porém, apenas uma implementação para comunicação com o escalonador Maui [15] foi feita.

A interação entre os módulos está ilustrada na Figura 24.

5.4.2. Configurações do Serviço

Quando o serviço de monitoração é iniciado, diversas configurações são carregadas. Tais configurações são definidas em um arquivo no formato XML chamado *monitoring-config.xml*. Os parâmetros de configuração que definem o comportamento do serviço são:

- *unfulfillmentTimes*: define quantas vezes o serviço deve constatar que o acordo não está sendo cumprido antes que seja constatada a quebra contratual (Seção 5.2);

- *timeInterval*: define o intervalo de tempo fixo para a utilização da técnica de monitoração Amostragem Randômica Estratificada (Seção 5.1);

- *agreementUnacomplishmentActionClass*: define qual classe, sendo uma implementação da interface *AgreementUnaccomplishmentAction*, deve ser utilizada como o módulo UA;

- *schedulerInformationFetcherClass*: define qual classe, implementação da interface *SchedulerInformationFetcher*, deve ser utilizada como o módulo SIF.

5.4.3. Funcionamento do serviço

Na Figura 24 está ilustrada a interação do serviço de monitoração com os demais serviços e aplicações utilizados no ambiente de *grid*. Inicialmente, quando o serviço de monitoração (MS) é iniciado, diversas configurações gravadas em arquivo, tais como o parâmetro *unfulfillmentTimes*, o intervalo para a realização de amostragens e as classes utilizadas como os módulos SIF e UA (Seção 5.4.2), são carregadas do repositório local.

Quando uma tarefa é submetida e há a criação de um acordo de serviço, a parte cliente do serviço faz uma chamada via mecanismo de *Web service* ao serviço de monitoração para que a execução da tarefa seja monitorada, passando então o identificador da tarefa e uma chave para a recuperação do acordo de serviço criado. Feita a requisição de monitoração, o serviço de monitoração cria e inicia uma nova

instância do módulo *Monitoring Thread*, passando diversos parâmetros de configuração, o acordo criado e o identificador da tarefa monitorada.

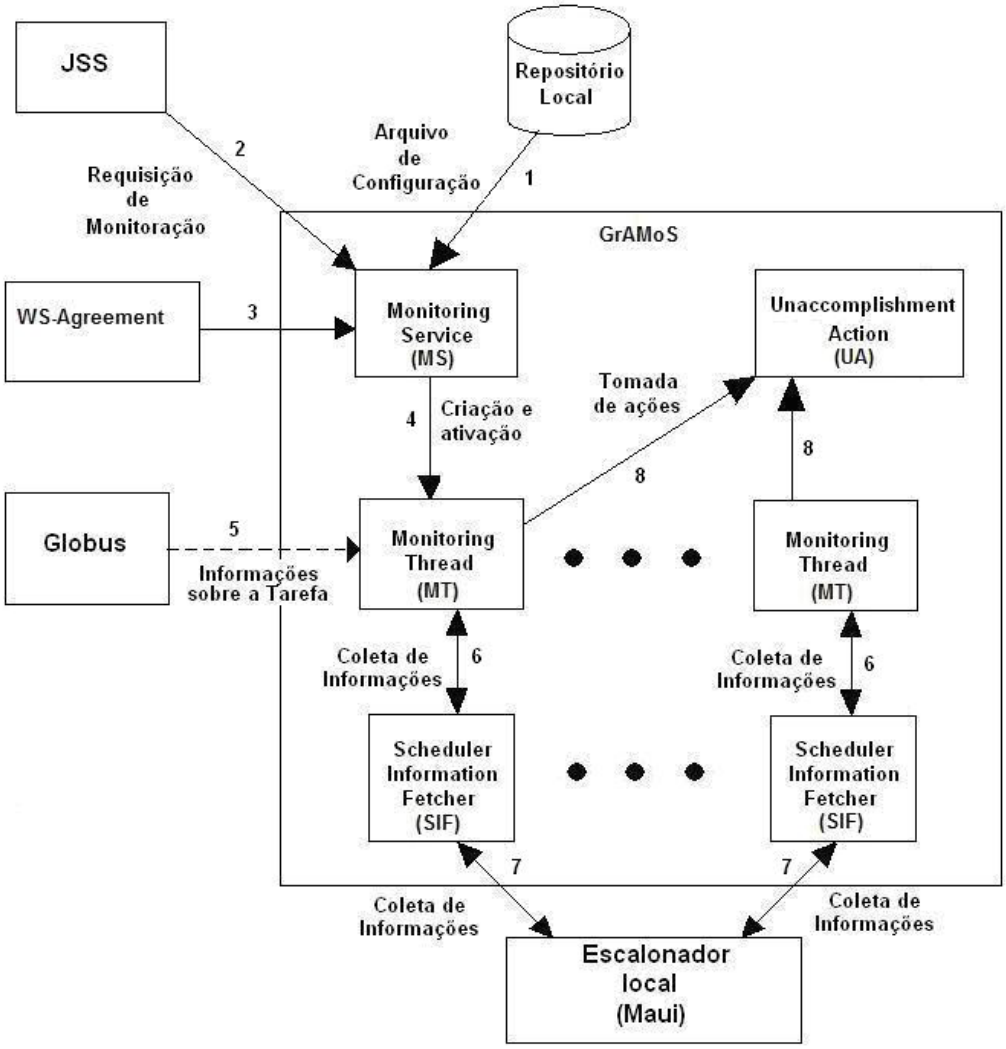

Figura 24: Interações entre os módulos do GrAMoS e entre o serviço e demais componentes do ambiente.

Iniciada a nova instância do módulo *Monitoring Thread*, são então recuperadas junto ao Globus informações locais sobre a tarefa, como seu identificador local e o usuário responsável por sua execução. Tais informações não são recuperadas por chamadas feitas ao Globus, mas sim por uma leitura direta de arquivos gravados por esse quando é realizada a submissão da tarefa. O módulo em questão instancia então os módulos *Scheduler Information Fetcher* e *Unaccomplishment Action*, de acordo com as configurações do serviço de monitoração, e fica responsável por, em intervalos de tempo aleatórios (Seção 5.1), coletar junto ao escalonador local, por intermédio do *Scheduler Information Fetcher*, informações sobre a execução da tarefa monitorada. Por *default*, as seguintes informações são coletadas:

- estado de execução da tarefa: indica se a tarefa está em espera, em execução ou se está finalizada;

- quantidade de recursos utilizados pela tarefa: indica o consumo de recursos pela tarefa em questão no momento da amostragem. Devido a limitações do JSS (Seção 4.3.5), apenas a quantidade de processadores alocados à tarefa é retornada;

- quantidade de recursos em disponibilidade: a quantidade de recursos disponíveis para serem alocados à tarefa caso esta necessite utilizá-los;

- instante em que se iniciou a execução da tarefa monitorada, caso esta tenha sido iniciada;

- instante em que foi realizada a última coleta das informações relativas à execução da tarefa em questão.

Recuperadas as informações acima, a *thread* do módulo MT executa diversas verificações. Primeiramente, verifica se a tarefa está em execução. Caso a tarefa esteja em espera, ou seja, não tenha sido iniciada, nenhuma ação é tomada e a monitoração fica inativa até o instante da próxima amostragem; caso a tarefa tenha sido finalizada, a *thread* é terminada; no último caso, se a tarefa estiver em execução, é feita a verificação de cumprimento do acordo.

A primeira verificação determina se a tarefa teve sua execução iniciada no intervalo entre o instante mínimo acordado e o instante máximo presentes no acordo de serviço. Caso a tarefa não tenha sua execução iniciada nesse intervalo, é detectada uma violação de acordo e feita uma chamada ao método *agreementStartTimeAction* do módulo UA, o qual se comporta de acordo com a configuração do serviço de monitoração. Dessa forma, fica à escolha do administrador do recurso qual ação deve ser tomada neste tipo de situação.

É verificada então a validade do acordo de serviço, de acordo com o instante de início da execução da tarefa e a duração do acordo, se este ainda está em vigor. Caso tenha expirado a duração do acordo, nenhuma ação é tomada, a linha de execução responsável pela monitoração é finalizada e a tarefa de *grid* continua sua execução, porém sem vinculação a acordos; caso contrário, é verificado se a quantidade de recursos definida no acordo está disponível para a utilização pela aplicação. Isso é feito comparando-se a quantidade de processadores sendo utilizada pela aplicação, somada à quantidade de processadores em disponibilidade, com a quantidade de processadores definida no acordo. O serviço considera que houve uma quebra de acordo se constatar, pelo menos um número *n* de vezes, que a quantidade de processadores disponíveis à execução da tarefa é menor que a definida no acordo, sendo *n* o valor do parâmetro de configuração *unfulfillmentTimes,* definido nos parâmetros de configuração do serviço. Esse comportamento garante a premissa exposta na Seção 5.2.

Caso seja constatada a quebra do acordo, é feita uma chamada ao método *agreementUnfulfillmentAction* do módulo *Unaccomplishment Action* (UA), cujo comportamento é definido de acordo com a configuração do serviço. Dessa forma, diferentes ações podem ser tomadas em caso de violação de acordo, conforme a implementação que se deseje usar, o que garante a premissa exposta na Seção 5.3.

Capítulo 6

Resultados experimentais

O serviço de monitoração descrito no Capítulo 5 foi implementado, gerando um protótipo que teve sua avaliação feita com a execução de aplicações desenvolvidas especificamente para testes. No presente capítulo são expostos o ambiente configurado para a execução dos testes e aplicações utilizadas, assim como os resultados obtidos.

6.1. Ambiente de testes

O ambiente criado para a realização dos testes do GrAMoS foi composto por duas máquinas: uma cliente, responsável pela submissão de tarefas junto ao JSS; e uma servidora, responsável pela execução e monitoramento das aplicações executadas. Para o escopo deste trabalho, é suficiente a utilização de apenas duas máquinas, isso porque o foco da análise é a monitoração da execução de aplicações na máquina servidora, não sendo assim necessário que as aplicações sejam escalonadas para a execução em diferentes recursos.

A configuração de *hardware* da máquina cliente é a seguinte: processador Intel Pentium 4, executando a uma freqüência de 3.0 Ghz e com a tecnologia *Hiper Threading* – HT; 512 MB de memória RAM. A máquina servidora possui a seguinte configuração: processador AMD Athlon XP 1.7, executando a uma freqüência de 1.3 GHZ; 768 MB de memória RAM.

As duas máquinas possuem o sistema operacional Linux, distribuição Ubuntu, instalado. As versões, porém, são diferentes. Na máquina cliente está instalada a versão 6.10, com kernel versão 2.6.17. Já na máquina servidora, está instalada a versão 7.04 do sistema, com kernel versão 2.6.20.

Foram instalados em ambas as máquinas os seguintes softwares, pré-requisitos para a instalação e execução do JSS:

- Globus Toolkit versão 4.0.4 para a infraestrutura do *grid*;
- Java Development Kit versão 1.5.0_08, para a execução do container *web services* do Globus;

Além dos *softwares* acima citados, foram instalados também na máquina servidora:

- Maui Cluster Scheduler versão 3.2.6p13, responsável pelo escalonamento local das tarefas submetidas;

- Torque Resource Manager versão 2.1.7, responsável pela execução das tarefas submetidas;

- Ganglia Monitoring System versão 3.0.4, para a coleta de informações sobre os recursos existentes no *grid* no formato GLUE.

As máquinas estão configuradas de forma que o Globus instalado no servidor se registra no serviço de índice de recursos (*index service*) do cliente, o qual é hospedado no container *web services* do Globus e é utilizado na descoberta de recursos.

6.2. Aplicações utilizadas para testes

Foram desenvolvidas aplicações específicas para a realização dos testes do mecanismo de monitoração. As aplicações foram desenvolvidas com a utilização da linguagem de programação C e não realizam nenhum trabalho significativo, apenas demandam processamento. Foram desenvolvidas cinco aplicações semelhantes, porém com diferentes tempos de execução. Os tempos de execução das aplicações A, B, C, D e E desenvolvidas são, respectivamente, de aproximadamente um, cinco, dez, quinze e trinta minutos.

O tempo de execução das aplicações usadas para teste é medido com a utilização de código interno a elas, sendo feita a comparação entre o instante de início e o instante de finalização da execução. Tais instantes são capturados com o uso da função *time*() do Linux, existente na biblioteca *time.h* da linguagem C, função a qual tem sua precisão da ordem de segundos. Os instantes aleatórios foram gerados com a utilização da função pseudo-randômica *random*(), presente na classe *java.util.Random*. Para a função, é passada como semente um objeto contendo os valores de data e hora do instante de geração do instante de amostragem.

As tarefas são submetidas para a execução com o uso do JSS, através do comando *submit-job*, o qual é um executável gerado na instalação do JSS e localizado na pasta *bin* existente na pasta de instalação do Globus, que no caso, para a instalação feita, é */usr/local/globus-4.0.4*. A Figura 25 ilustra uma chamada ao comando *submit-job* executada na máquina cliente.

```
submit -job -f jobdescription.jsdl  \
-i https://cliente:8443/wsrf/services/DefaultIndexService \
-p /usr/local/globus-4.0.4/etc/jobsubmission/job-preferences.xml
```

Figura 25: Submissão de tarefa utilizando o JSS.

Na Figura 25 é mostrada a chamada de submissão de tarefas utilizando o JSS, assim como os parâmetros passados à chamada. O primeiro parâmetro passado (-f), o arquivo *jobsubmission.jsdl*, contém a descrição da tarefa submetida. O segundo (-i), indica qual o índice de recursos deve ser utilizado para a descoberta de recursos. Para os testes, o serviço utilizado foi o da máquina cliente. Por último, há também um

parâmetro que indica um arquivo contendo preferências para a execução da tarefa [49]. Tais preferências incluem, entre outras, informações tais como o menor e o maior instante em que a tarefa deve ser iniciada, qual o objetivo da tarefa entre iniciar sua execução o mais cedo possível (*earliest start*) e ser finalizada o quanto antes (*earliest completion*), além de informações utilizadas na seleção de recursos para a execução.

O acordo monitorado para a realização dos testes, o qual representa a reserva de 1 processador por 32 minutos, está ilustrado na Figura 26.

```
<wsag:AgreementType ... >

<wsag:Name>Agreement</wsag:Name>
<wsag:Context>
     ...
</wsag:Context>
<wsag:Terms>
     <wsag:All>
          <wsag:ServiceDescriptionTerm
               wsag:Name="numberOfCPUs"
               wsag:ServiceName="Reservation">

     <res:numberOfCPUs>1</res:numberOfCPUs>
          </wsag:ServiceDescriptionTerm>
          <wsag:ServiceDescriptionTerm
               wsag:Name="duration"
               wsag:ServiceName="Reservation">
               <res:duration>1920</res:duration>
          </wsag:ServiceDescriptionTerm>
               ...
     </wsag:All>
</wsag:Terms>

</wsag:AgreementOffer>
```

Figura 26: Acordo monitorado para a realização dos testes.

6.3. Resultados experimentais

Como dito anteriormente, para a realização de testes do GrAMoS foram utilizadas as aplicações A, B, C, D e E, cujas durações aproximadas são, respectivamente, um, cinco, dez, quinze e trinta minutos. Primeiramente, foram testadas as execuções das aplicações sem haver monitoração. Foram também realizados testes de execução com o parâmetro *timeInterval* do serviço de monitoração (Seção 5.4.2) definido com os valores 5 segundos e 10 segundos. Para a coleta de resultados, cada aplicação foi executada 10 vezes, sendo calculada a média aritmética dos tempos de execução constatados. Os resultados das médias dos tempos de execução das aplicações encontram-se na Tabela 2.

Aplicação	Monitoração			Overhead (%)	
	sem	5 s	10 s	5 s	10 s
A	61,7 s	63,4 s	62,5 s	2,75	1,29
B	307,4 s	314,3 s	311,0 s	2,24	1,17
C	611,8 s	625,3 s	618,8 s	2,20	1,14
D	916,4 s	936,4 s	927,0 s	2,18	1,15
E	1808,4 s	1848,4 s	1829,0 s	2,21	1,13

Tabela 2: Resultados coletados com a execução das tarefas A, B, C, D e E.

Analisando-se os resultados obtidos, pode-se perceber que o *overhead* gerado pela monitoração do acordo de serviço para a aplicação A, de menor duração, foi ligeiramente maior que o gerado para as demais aplicações. Isso ocorre porque, para aplicações de pouca duração, a criação de objetos utilizados na monitoração realizada assim que a tarefa é submetida possui um maior peso, gerando um *overhead* mais significativo que para aplicações de longa duração. Porém, a execução de aplicações de curta execução em ambiente de *grid* não é comum, o que faz que tal diferença no *overhead* não tenha grande significado.

Para as demais aplicações, porém, o *overhead* gerado manteve-se praticamente constante. Percebe-se que o *overhead* gerado pela criação inicial de objetos para a monitoração é muito pequeno para aplicações de maior duração. Isso leva a crer que o *overhead* gerado pelas operações de verificação de acordos mantém-se praticamente constante, independente do tempo de execução da tarefa.

Outro fato observado é que, para um intervalo de monitoração de 5 segundos, o *overhead* é bem próximo do dobro do encontrado para um intervalo de 10 segundos. Sendo que para o intervalo de 5 segundos são realizadas o dobro de amostragens que para o de 10 segundos, pode-se perceber que o *overhead* gerado é inversamente proporcional ao tamanho do intervalo definido. Com base nisso, acredita-se também que o *overhead* gerado por cada uma das amostragens realizadas é o mesmo durante toda a execução da aplicação.

A Figura 27 ilustra um gráfico do *overhead* gerado versus a aplicação utilizada. A figura ilustra exatamente o que foi dito anteriormente, pois pode ser visto um maior *overhead* do mecanismo para a aplicação A e praticamente constante para as demais aplicações, assim como um *overhead* para o intervalo de 5 segundos duas vezes maior que para o intervalo de 10 segundos.

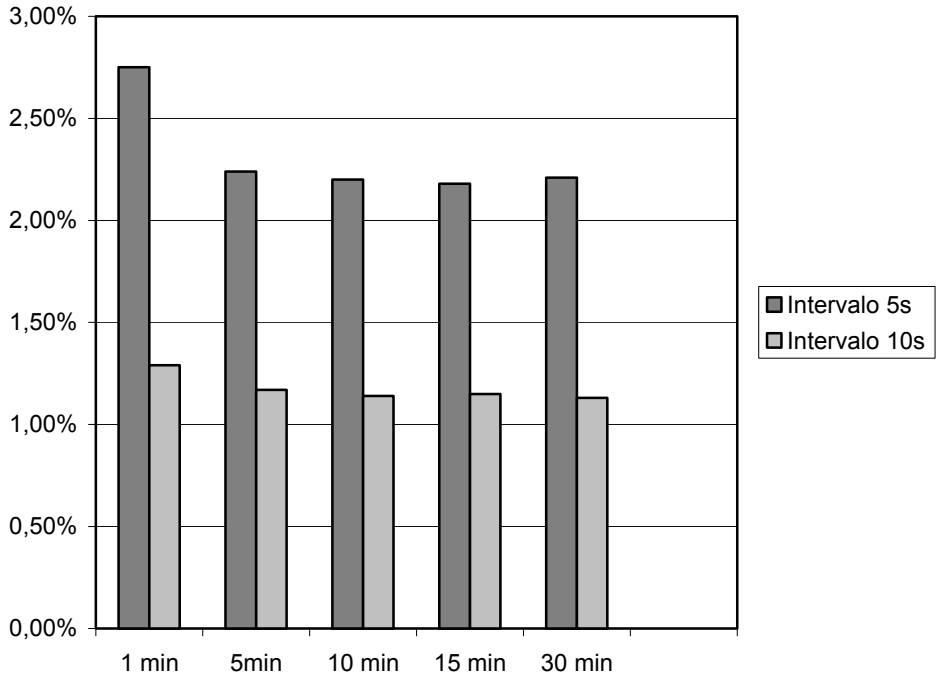

Figura 27: *overhead* gerado versus aplicação.

Além dos dados sobre o tempo de execução das tarefas, foram também coletados e analisados dados sobre as amostragens realizadas na verificação dos acordos, tais como o instante em que é realizada a amostragem dentro do intervalo definido, e o tempo de duração delas. Dados sobre o instante de verificação são apresentados na Tabela 3, e sobre a duração das amostragens na Tabela 4.

Aplicação	Instante de verificação					
	5 s			10 s		
	menor	médio	maior	menor	médio	maior
A	0,01 s	2,33 s	4,95 s	0,16 s	3,57 s	9,59 s
B	0,01 s	3,08 s	4,85 s	0,16 s	4,32 s	9,92 s
C	0,01 s	2,53 s	4,99 s	0,01 s	4,88 s	9,97 s
D	0,01 s	2,43 s	4,99 s	0,01 s	4,83 s	9,65 s
E	0,01 s	2,48 s	4,98 s	0,01 s	4,94 s	9,87 s

Tabela 3: Instantes de realização das amostragens.

A tabela 3 mostra os dados relativos aos instantes em que foram realizadas as amostragens durante a monitoração dos acordos relativos à execução de cada aplicação. São mostrados os valores dos menores e maiores instantes, assim como o valor da média dos instantes. Pode-se perceber que, para todas as aplicações executadas, o menor instante no qual foi realizada a verificação fica bem próximo do zero, quando tem início o intervalo.

Para o intervalo definido em 5 segundos, para todas as aplicações executadas, o maior instante de verificação ocorre bem próximo dos 5 segundos, quando é finalizado o intervalo. O instante médio de realização das amostragens para este intervalo está perto dos 2,5 segundos para as aplicações A, C, D e E, instante que se encontra na metade do intervalo. Para tais aplicações, pode-se perceber que os instantes de amostragem estão bem distribuídos dentro do intervalo de 5 segundos definido. Para a aplicação B, porém, ficou 0,58 segundo acima da metade do intervalo, uma quantidade de tempo bem superior ao das outras aplicações executadas. Considerando-se os resultados produzidos pelas outras tarefas, pode-se atribuir à diferença encontrada à maneira como são definidos os instantes de amostragem, que é randômica. A Figura 28 ilustra a distribuição dos instantes de amostragem em um trecho da execução da aplicação D com o intervalo definido em 5 segundos.

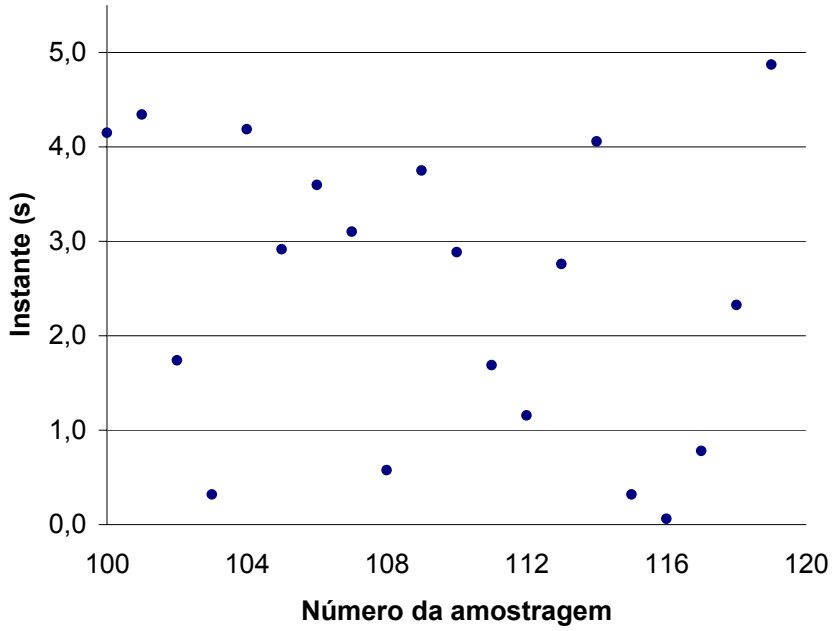

Figura 28: Distribuição constatada dos instantes de amostragens para o intervalo de 5 segundos.

Para o intervalo de 10 segundos, os maiores instantes de verificação ocorrem perto dos 10 segundos, quando é finalizado o intervalo. O instante médio de verificação, para as aplicações B, C, D e E encontra-se bem perto dos 5 segundos, metade do intervalo. Para essas aplicações então, pode-se concluir que os instantes de amostragem estão bem distribuídos no intervalo definido. Para a aplicação A, porém, o instante médio registrado ficou 1,5 segundos abaixo da metade do intervalo, um desvio de 30%. Considerando-se os resultados constatados com a execução das demais aplicações, pode-se creditar a diferença encontrada à natureza randômica e não determinística da definição dos instantes para a realização das amostragens.

Foram coletados, além dos resultados já expostos, os valores sobre a duração das amostragens realizadas na monitoração de acordos. Os dados coletados são expostos na Tabela 4.

Aplicação	Duração da verificação					
	5 s			10 s		
	menor	médio	maior	menor	médio	maior
A	0,26 s	0,36 s	1,39 s	0,27 s	0,40 s	1,50 s
B	0,26 s	0,37 s	1,39 s	0,26 s	0,37 s	1,40 s
C	0,25 s	0,38 s	1,58 s	0,26 s	0,38 s	1,38 s
D	0,26 s	0,37 s	1,38 s	0,26 s	0,38 s	1,37 s
E	0,25 s	0,37 s	1,42 s	0,26 s	0,37 s	1,45 s

Tabela 4: Dados sobre a duração das amostragens realizadas.

Analisando os resultados contidos na Tabela 4, pode-se perceber que, tanto para os intervalos definidos em 5 segundos e 10 segundos, os valores das menores durações de verificações são bastante próximos. O mesmo ocorre para os valores médios das durações com ambos os intervalos. Para as maiores durações de verificação, a variação entre os valores é um pouco mais significativa. Apesar disso, mesmo havendo variação nos valores da maior duração de verificação, os *overheads* percebidos pelas aplicações é praticamente o mesmo, o que leva a crer que o *overhead* gerado pelas verificações é o mesmo, independente do tempo levado para fazer a verificação. Isso ocorre porque as aplicações executadas estão executando em paralelo ao serviço de monitoração, competindo por processador com este. Dessa forma, mesmo quando uma verificação leva um maior período de tempo para ocorrer, a aplicação continua sua execução, tendo menor influência o tempo levado para a realização da verificação.

6.4. Estudo de caso – Violação de acordo

Foram realizados testes de monitoração de acordos sendo simulada a violação desses. Para tanto, foi desenvolvida uma classe, implementação da interface *SchedulerInformationFetcher*, que a cada cinco verificações realizadas comunica ao módulo *MonitoringThread* que a quantidade de recursos utilizados somada à de recursos disponíveis é menor que a quantidade de recursos definida no acordo firmado. Dessa forma, o MT verifica o não cumprimento do acordo, independente da utilização real dos recursos, sendo possível simular a ocorrência de falhas. Dessa forma, uma vez que o número de violações indicadas supere o valor do parâmetro *unfulfillmentTimes*, a violação do acordo é constatada.

O acordo monitorado para o teste é o ilustrado na Figura 26. O parâmetro *unfulfillmentTimes* foi definido com o valor 10. A implementação da interface *AgreementUnfulfillmentAction* utilizada é programada para apenas fazer o registro em arquivo de *log* em caso de detecção de violação do acordo monitorado. Foi utilizada para os testes a aplicação B, cuja duração aproximada é de 5 minutos.

Foi constatado então com a execução da aplicação que, a partir da qüinquagésima verificação realizada, o MT constatou a violação do acordo e registrou em *log* tal ocorrência, o que era o comportamento esperado do mecanismo de

monitoração. Dessa forma, o GrAMoS mostrou-se capaz de detectar a violação do acordo e da tomada de ação.

Capítulo 7

Conclusão e Trabalhos Futuros

A presente dissertação de mestrado apresentou o GrAMoS, um mecanismo para a monitoração de acordos de serviço em *grid*, assim como para a detecção de violações de acordo e para a tomada de ações na ocorrência de violações.

O GrAMoS foi testado e avaliado a partir da execução de diferentes aplicações, as quais possuem tempos de execução de 1, 5, 10 e 15 minutos, com o mecanismo de monitoração de acordos ativado e desativado. Foram medidos, além do *overhead* gerado pelo mecanismo, os instantes médios nos quais são realizadas as amostragens para a verificação dos acordos e os tempos de duração das verificações realizadas.

Os resultados obtidos mostraram que o *overhead* gerado pelo GrAMoS é praticamente constante para aplicações de longa duração, independente de seu tempo de execução. Para aplicações de curta duração, porém, é ligeiramente maior. Os resultados mostraram também que o *overhead* gerado é inversamente proporcional ao intervalo definido para a realização de amostragens. Pode ser visto ainda que o *overhead* total do mecanismo de monitoração do GrAMoS é pequeno em relação ao tempo total de execução da aplicação.

Analisando-se os resultados, nota-se ainda que os instantes em que são realizadas as amostragens estão bem distribuídos no intervalo definido nas configurações do GrAMoS. Em relação à duração das verificações realizadas, os resultados levam a crer que, apesar de as durações variarem consideravelmente, o *overhead* gerado para a aplicação em execução por cada uma delas é praticamente o mesmo. Isso ocorre porque o mecanismo de verificação é executado em uma *thread* separada, a qual executa paralelamente às aplicações submetidas.

Como trabalhos futuros, sugerimos os seguintes. Primeiramente, deve ser feita uma adaptação no GrAMoS de forma que esse se torne independente do módulo do JSS instalado em máquinas cliente, ficando dependente apenas do módulo WS-*Agreement*.

O GrAMoS deve também ser alterado de forma que possa ser utilizado com diferentes *middlewares* de *grid*, assim como o JSS.

Sugerimos ainda que sejam realizados testes como GrAMoS em plataformas mais complexas, compostas de uma quantidade maior de recursos compartilhados que a utilizada na presente dissertação. Além disso, podem ainda ser realizados testes utilizando aplicações de mais longa duração que as utilizadas no presente trabalho, além de utilizar também intervalos de amostragem maiores que 10 segundos.

Sugerimos, por fim, que o GrAMoS seja ser alterado de forma a suportar a monitoração de acordos com relação a outros tipos de recursos além da quantidade de processadores, tais como memória alocada às aplicações e espaço de armazenamento.

Referências

[1] Aurrecoechea, C. et al. A survey of QoS architectures. *Multimedia Systems,* vol.6, no.3, pgs. 138-151, 1998.

[2] Frolund, S., Koistinen, J. Quality-of-service specifications in distributed object systems. *Distributed Systems Engineering, IEE*, No. 5, pgs. 179-202. Reino Unido. 1998.

[3] Bochmann, G. V.; Kerhervé, B.; Mohamed-Salem, M. O. Quality of service management issues in electronic commerce applications. *IBM Workshop on Electronic Commerce*. Setembro, 1998.

[4] Jingwen, J.; Nahrstedt, K. QoS specification languages for distributed multimedia applications: A survey and taxonomy. *IEEE Multimedia Magazine*, volume 11, issue 3, pgs 74-87. Julho, 2004.

[5] Bochmann, Gregor at. al. Distributed Multimedia and QoS : A Survey. *IEEE Multimedia,* volume 2, Issue 2, pgs 10-19. 1995.

[6] Rutschenreuther, T., Schönberg, S. Layer Integrated Quality of Service Management. *Workshop on System Design Automation,* Dresden, pgs 199-204. Março 1998.

[7] von Laszewski, G. et al. A Java Commodity Grid Kit. *Concurrency and Computation: Practice and Experience*, volume 13, Issue 8-9, pgs 643-662. 2001.

[8] Campbell, A.T., Coulson, G., Hutchison, D. A Quality of Service Architecture. *ACM SIGCOMM Computer Communication Review*, Vol. 24, No. 2, pgs. 6-27. 1994.

[9] Siqueira, F., Cahill, V. Quartz: a QoS architecture for open systems. *IEEE International Conference on Distributed Computing Systems (ICDCS 2000),* pgs 197-204. Taipei, Taiwan. Abril 2000.

[10] Coulouris, G.; Dollimore, J.; Kindberg, T. Distributed Systems – Concepts and Design. 3ª edição. 2000.

[11] Siqueira, F., Cahill, V. Delivering QoS in open distributed systems. *Seventh IEEE Workshop on Future Trends of Distributed Computing Systems,* pg 185. 1999.

[12] Frolund, S.; Koistinen, K. QML: A Language for QoS Specification. *HP Research Report HPL-98-10*, Hewlett Packard, 1998.

[13] Yuan, W. et al., An XML-based Quality of Service Enabling Language for the Web. *Journal of Visual Language and Computing, special issue on Multimedia Languages for the Web*, volume 13(1), pgs 61-95. 2002.

[14] Gu, X. et. al., Visual QoS Programming Environment for Ubiquitous Multimedia Services. *IEEE International Conference on Multimedia and Expo (ICME'01)*, pgs 575-578. 2001.

[15] Maui Cluster Scheduler. www.clusterresources.com/pages/products/maui-cluster-scheduler.php. Visitado em 21 de agosto de 2007.

[16] TORQUE Resource Manager. www.clusterresources.com/pages/products/torque-resource-manager.php. Visitado em 21 de agosto de 2007.

[17] Massie, M. L.; Chun, B. N.; Culler, D. E.; The Ganglia Distributed Monitoring System: Design, Implementation and Experience. *Parallel Computing*, volume 30, issue 7, pgs 817-840. 2004.

[18] Ta,X., Mao, G. Online End-toEnd Quality of Service Monitoring for Service Level Agreement Verification. *The 14th IEEE International Conferences on Networks (ICON'06)*, volume 6, pgs 1-6. 2006.

[19] Jingwen, J.; Nahrstedt, K. Classification and Comparison of QoS specification Languages for Distributed Multimedia Applications. *Technical Report UIUCDCS-R-2002-2302/UILU-ENG-2002-1745*. University of Illinois. 2002.

[20] Mercer, C. W.; Savage, S.; Tokuda, H. Processor capacity reserves for multimedia operating systems. *Technical Report CMU-CS-93-157*, Carnegie Mellon University, Pittsburg. 1993.

[21] Palopoli, L.; Cucinotta, T.; Bicchi, A. Quality of service control in soft real-time applications. *42nd IEEE Conference on Decision and Control*, volume 1, pgs 664-669. 2003.

[22] Fujita, H.; Nakajima, T.; Tezuka, H. A processor reservation system supporting dynamic qos control. *2nd International Workshop on Real-Time Computing Systems and Applications*, pg 224. 1995.

[23] Albeni, L. et al. Adaptive reservations in a Linux environment. *10th IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS'04)*, pg 238. 2004.

[24] Wu, M.; Sun, X.; Chen, Y. QoS Oriented Resource Reservation in Shared Environments. *Sixth IEEE International Symposium on Cluster Computing and the Grid (CCGRID'06)*, pgs 601-608. 2006.

[25] Mercer, C. W.; Savage, S.; Tokuda, H. Processor Capacity Reserves: Operating System Support for Multimedia Applications. *IEEE International Conference on Multimedia Computing and Systems (ICMCS)*, pgs 90-99. 1994.

[26] Wolski, R.; Spring, N.; Hayes, J. The Network Weather Service: A Distributed Resource Performance Forecasting Service for Metacomputing. *Journal of Future Generation Computing Systems*, volume 15, nos 5-6. pgs 757-768. 1999.

[27] Andrieux, A. et. al. Web Services Agreement Specification (WS-Agreement). 2004.

[28] Keahey,K., Araki, T., Lane, P. Agreement-Based Interactions for Experimental Science. *Euro-par 2004*, pgs 399-408. 2004..

[29] Grid Computing Info Centre: Frequently Asked Questions (FAQ): www.cs.mu.oz.au/~raj/GridInfoware/gridfaq.html. Consultado em 06 de junho de 2004.

[30] Enterprise Architect Magazine – Answers to the Enterprise Architect Magazine Query. Rajkumar Buyya. www.cs.mu.oz.au/~raj/press/EA03/EAInterview.pdf. Consultado em 06 de junho de 2004.

[31] Baker, M., Buyya, R., Laforenza, D. Grids and Grid Technologies for Wide-Area Distributed Computing, *International Journal of Software: Practice and Experience (SPE)*, vol. 32, no. 15, pgs. 1437-1466. Dezembro, 2002.

[32] Foster, Ian; Kesselman, Carl; Tuecke, Steven. The Anatomy of the Grid: Enabling Scalable Virtual Organizations. *International Journal of High Performance Computing Applications*, vol. 15, pp. 200-222. Sage Publishers. Londres. 2001.

[33] IBM Grid Computing – Grid benefits: www.ibm.com/grid/about_grid/benefits.shtml. Consultado em 06 de junho de 2004.

[34] Foster, Ian, et. al. The Physiology of the Grid: an Open Grid Services Architecture for Distributed Systems Integration. *Open Grid Service Infrastructure WG, Global Grid Forum*. Junho, 2002.

[35] Alfieri, R., et. al. VOMS, an Authorization System for Virtual Organizations. *1st European Across Grids Conference*, pgs 33-40. 2003.

[36] IBM Grid Computing – Frequently asked questions: www-1.ibm.com/grid/about_grid/faq.shtml. Consultado em 06 de junho de 2004.

[37] Shaikh, A. et al. UDDIe: An Extended Registry for Web Services. *2003 Symposium.on Applications and the Internet Workshops*, pgs 85-89. 2003.

[38] Nabrzyski, J., Schopf, J. M., Weglarz, J. Grid Resource Management : State of the Art and Future Trends. 1ª Edição. Ed. Springer. 2003.

[39] Foster, I., et al. The WS-Resource Framework. Março, 2004.

[40] Foster, I., et al. From Open Grid Services Infrastructure to WS-Resource Framework: Refactoring & Evolution. Março, 2004.

[41] Foster, I. Kesselman, C. Grid – Blueprint for a New Computing Infrastructure. Morgan Kauffman. 2ª Edição. 2003.

[42] Christensen, E., at. al. Web Services Description Language (WSDL) 1.1. W3C note 15. 2001.

[43] Sandholm, T.; Gawor, J. Globus Toolkit 3 Core - A Grid Service Container Framework. Technical report, 2003.

[44] Foster, I. Globus Toolkit Version 4: Software for Service-Oriented Systems. *Journal of Computer Science and Technology*, Volume 21, no. 4, pgs 513-520. Springer Boston, 2005.

[45] Box, D. Simple Object Access Protocol (SOAP) 1.1. W3C note 8. 2000.

[46] von Lawzewski, G. et al. Analysis and Provision of QoS for Distributed Grid Applications. *Journal of Grid Computing*. Vol. 2, No. 2, pgs. 163-182. Junho, 2004.

[47] Yang, Z. et al. A QoS-enabled Services System Framework for Grid Computing. *The 3rd International Conference on Grid and Cooperative Computing (GCC 04')*. *LNCS* volume 3251, pgs 8-16. 2004.

[48] Padgett, J. Djemame, K. Dew, P. Grid Service Level Agreements Combining Resource Reservation and Predictive Run-time Adaptation. *UK e-Science All Hands Meeting*. 2005.

[49] Elmroth, E., Tordsson, J. An Interoperable, Standards-based Grid Resource Broker and Job Submission Service. First International Conference on e-Science and Grid Computing, pgs 212-220. 2005.

[50] Foster, I. et al. End-to-End Quality of Service for High-End Application. *Elsevier Computer Communications Journal*, vol. 27, no. 14, pgs. 1375-1388. 2004.

[51] von Lawzewsky, G. et al. QoS Support for High-Performance Scientific Grid Applications. *Cluster Computing and the Grid*, pgs 134-143. Abril, 2004.

[52] Roy, A. End-to-End Quality of Service for High-End Applications. PhD Thesis. Department of Computer Science. Chicago University. 2001.

[53] von Laszewski, G. Analysis and Provision of QoS for Distributed Grid Applications. *Journal of Grid Computing*, volume 2, no 2, pgs 163-182. 2004.

[54] Dong, F. Akl, S. G. Scheduling algorithms for Grid Computing: State of the Art and Open Problems. Technical Report. Queen's University. Ontario, Canada. 2006.

[55] Tonellotto, N.; Wieder, P.; Yahyapour, R. A Proposal for a Generic Grid Scheduling Architecture. *Integrated Research in Grid Computing Workshop*, pgs 337-346. Pisa, Itália. 2005.

[56] Gradwell, P. Overview of Grid Scheduling Systems. University of Bath. Reino Unido.

[57] Zhu, Y. A Survey on Grid Scheduling Systems. PhD Thesys. Department of Computer Science. Hong Kong University of Science and Technology. 2003.

[58] Vadhiyar, S. Dongarra, J. A Metascheduler for the Grid. *11th IEEE Symposium on High-Performance Distributed Computing*, pgs 343-351. 2002.

[59] Dongarra, J. et al. New Grid Scheduling and Rescheduling Methods in the GrADS Project. *International Parallel and Distributed Processing Symposium*, pg 199. 2004.

[60] Buyya, R. Abramson, D. Giddy, J. Nimrod/G: An Architecture for a Resource Management and Scheduling System in a Global Computational Grid. *4th International Conference and Exhibition on High Performance Computing in Asia-Pacific Region*, volume 1, pgs 283-289. 2000.

[61] Abramson, D., Giddy, J. Kotler, L. High Performance Parametric Modeling with Nimrod/G: Killer Application for the Global Grid? *14th International Parallel and Distributed Processing Symposium*, pgs 520-528. 2000.

[62] Andreozzi, S. et. al. Glue Schema Specification version 1.2. 2005.

[63] Drescher, M. et. al. Job Submission Description Language (JSDL) specification, version 1.0. 2005.

[64] Nainwal, K. C. et. al. A Framework for QoS Adaptive Grid Meta Scheduling. *16th International Workshop on Database and Expert Systems Applications*, pgs 292-296. 2005.