PUC

ISSN 0103-9741

Monografias em Ciência da Computação
n° 37/08

Gerenciamento de Recursos em Ambientes Distribuídos: uma Visão do Escalonamento de Processos

Valéria Quadros dos Reis

Renato Fontoura de Gusmão Cerqueira

Departamento de Informática

PONTIFÍCIA UNIVERSIDADE CATÓLICA DO RIO DE JANEIRO

RUA MARQUÊS DE SÃO VICENTE, 225 - CEP 22453-900

RIO DE JANEIRO - BRASIL

Monografias em Ciência da Computação, No. 37/08 ISSN: 0103-9741
Editor: Prof. Carlos José Pereira de Lucena September, 2008

Gerenciamento de Recursos em Ambientes Distribuídos: uma Visão do Escalonamento de Processos[1]

Valéria Quadros dos Reis e Renato Fontoura de Gusmão Cerqueira

{vreis, rcerq}@inf.puc-rio.br

Abstract. As distributed computing infrastructures get more and more heterogeneous, with a higher quantity of nodes, the role of the resource managers become more important. Resource managers, responsible for ensuring system usage policies, have been through several changes in order to efficiently manage highly diversified environments, such as the ones which are found in grids. This paper presents the research about four different resource managers one can find in the literature; also it identifies the main functionalities provided by the mentioned resource managers. The main goal is that the identified basic profile can be used as a guideline for future implementations of resource managers in distributed computational environments.

Keywords: Distributed computing, resource management, process scheduling

Resumo. Conforme infraestruturas de Computação Distribuída se tornam cada vez mais heterogêneas, com maior quantidades de nós, mais importante se torna o papel do gerenciador de recursos. Gerenciadores de recursos, responsáveis por garantir o bom funcionamento de um sistema computacional, têm sofrido mudanças para administrar infraestruturas altamente diversificadas tal como as encontradas em grades. Este trabalho expõe o estudo de quatro diferentes gerenciadores de recursos encontrados na literatura e identifica as principais funcionalidades providas por eles. O objetivo é que o perfil básico traçado sirva de guia para novas iniciativas de implementação de gerenciadores de recursos em ambientes computacionais distribuídos.

Palavras-chave: Computação distribuída, gerenciamento de recursos, escalonamento de processos

[1]Trabalho patrocinado pelo Ministério de Ciência e Tecnologia da Presidência da República Federativa do Brasil. Processo CNPq número 142035/2006-8.

Responsável por publicações:
Rosane Teles Lins Castilho
Assessoria de Biblioteca, Documentação e Informação
PUC-Rio Departamento de Informática
Rua Marquês de São Vicente, 225 - Gávea
22451-900 Rio de Janeiro RJ Brasil
Tel. +55 21 3527-1516 Fax: +55 21 3527-1530
E-mail: bib-di@inf.puc-rio.br
Web site: http://bib-di.inf.puc-rio.br/techreports/

1 Introdução

Desde a época dos *mainframes*, o compartilhamento de tempo entre os processos de uma máquina obrigava a existência no Sistema Operacional de uma entidade responsável por decidir quando um determinado processo seria executado. Essa entidade, denominada escalonador de processos executa regras preestabelecidas por políticas de escalonamento. No caso dos *mainframes*, a política padrão consistia em compartilhar o tempo de processamento igualmente entre os processos.

Com o surgimento dos Sistemas Distribuídos, as aplicações passaram a poder ser executadas em máquinas distintas e de forma paralela. Identificou-se então, a necessidade da criação de mecanismos para decidir, dentre as máquinas disponíveis, aquelas que seriam "as melhores"para executar determinado trecho da aplicação. Assim como os escalonadores de processos existentes nos Sistemas Operacionais, era preciso um escalonador de processos de mais alto nível, responsável por escalonar processos não em tempos de processamento de uma única CPU, mas em diversas delas. Dessa maneira, as políticas de escalonamento utilizadas pelos escalonadores passaram a decidir não somente quando uma determinada aplicação seria executada, mas também como (seqüencial ou paralelamente) e onde.

A popularização dos *clusters* computacionais na década de 80 incentivou muitas iniciativas de criação de sistemas para o gerenciamento de recursos nesse tipo de infra-estrutura [1, 2, 3, 4, 5]. De maneira geral, esses sistemas apresentam mecanismos de descoberta, seleção e monitoração de recursos [6, 7]. Alguns sistemas para *cluster* sofreram adaptações e hoje são executados em ambientes ainda mais diversos tais como as grades computacionais [8, 9, 10], onde a tarefa de escolher os melhores nós de execução é ainda mais complexa dada a heterogeneidade e dinamicidade desse tipo de ambiente. Nota-se portanto que a tarefa de escalonar processos é de extrema importância para o bom desempenho e para o aumento da utilização dos recursos de um sistema computacional distribuído.

Este trabalho tem por objetivo descrever alguns dos principais sistemas de gerenciamento de recursos encontrados na literatura focando na maneira pela qual eles realizam o escalonamento de processos. As Seções a seguir estão divididas da seguinte forma: a Seção 2 apresenta a terminologia utilizada na área de escalonamento de processos; a Seção 3 descreve a arquitetura básica de gerenciadores de recursos em sistemas computacionais distribuídos; a Seção 4 tem por objetivo apresentar as fases contidas em um ciclo de escalonamento; a Seção 5 introduz o projeto de gerenciamento de recursos Condor, enquanto a Seção 6 descreve o sistema PBS; nas Seções 7 e 8 são descritos os sistemas OAR e o Mosix, respectivamente; por fim, na Seção 9 é apresentada a conclusão deste trabalho.

2 Terminologia

Quando se trata de gerenciamento de recursos em Sistemas Computacionais, referencia-se às entidades de uma máquina que podem ser gerenciadas tais como CPU, memória e disco [11, 12]. Entre as funcionalidades principais de um gerenciador de recurso estão o recebimento de pedidos por recursos e a atribuição de recursos à esses pedidos, realizando o casamento do que é buscado com o que é oferecido pela infra-estrutura. Esse casamento é realizado de maneira a maximizar a qualidade de serviço (QoS) estabelecida pelos usuários clientes do sistema. A qualidade de serviço pode, por exemplo, ser a minimização do tempo

de espera pelos resultados de uma aplicação. Para atingir os objetivos dos clientes, existem políticas de escalonamento que ditam como, quando e onde determinada aplicação deve ser executada. No caso de minimização de tempo de espera, por exemplo, pode-se empregar a regra de atribuir uma aplicação ao servidor com maior capacidade atual de processamento. Uma mesma política pode ser implementada por diferentes algoritmos que, por sua vez, podem necessitar de diferentes fatias de tempos para serem executados.

A qualidade de serviço é dividida em duas partes: o controle admissão e o monitoramento do consumo de recursos da aplicação. O controle de admissão determina se as requisições de recursos de uma aplicação podem ser atendidas, enquanto o monitoramento verifica, dado que a aplicação foi admitida, se o consumo de recurso atual não está violando o que foi acordado durante a admissão. Dessa maneira, um sistema que não permite à aplicação especificar o montante de recursos necessário para a sua execução não suporta QoS. Por outro lado, um sistema que permite a especificação de recursos pela aplicação, mas não garante a reserva desses recursos provê QoS parcial. Esse é o caso da maioria dos sistemas de *cluster* e grades computacionais, visto que a maioria dos Sistemas Operacionais que não são de tempo real, não provêem meios eficientes para a garantia de qualidade de serviço por não apresentarem mecanismos de reserva de recursos.

O escalonamento de processos pode ser guiado por informações a respeito dos estados dos recursos e das características das aplicações sem que um histórico de execuções seja considerado. Nesse caso, há um escalonamento não-preditivo que pode ser baseado em heurísticas a respeito das aplicações ou mesmo em modelos de distribuição de probabilidade baseados em análises das características das aplicações obtidas de fontes externas ao gerenciador de recursos. No caso do escalonamento preditivo, históricos de execução de aplicações são utilizados por heurísticas, modelos de custo e aprendizado de máquina para que seja feita uma estimativa do tempo de execução dessas aplicações e de suas necessidades de recursos.

Muitas vezes, devido a mudanças nos estados dos recursos, é preciso transferir a execução de um processo para outro nó. Essa transferência, denominada migração, geralmente ocorre acompanhada de um ato de *checkpoint*, onde o estado de execução do processo é salvo em um arquivo para posterior reinicialização. Sistemas onde se deseja balancear a carga de trabalho dos nós devem implementar mecanismos de migração.

O atendimento das filas de processos a serem executados pode ser realizado em intervalos periódicos ou disparados por determinados eventos. Essa prática de escalonamento em lote é potencialmente mais eficiente uma vez que mais pedidos podem ser considerados a cada ciclo. No caso de aplicações que necessitam de QoS muito altas, porém, essa abordagem pode ser indesejável já que a violação do nível de serviço solicitado pode não causar um reescalonamento imediato.

Em sistemas multiusuários, é comum haver diferentes objetivos a serem atingidos no escalonamento. Exemplos são aumentar a vazão do sistema, diminuir o tempo de resposta, aumentar a utilização de recursos ou balancear a carga do sistema. Muitas vezes esses objetivos são conflitantes, mas é interessante haver a opção de escolha de qual abordagem se deseja utilizar, ou seja, é desejável que as políticas de escalonamento presentes nos sistemas de gerenciamento de recurso sejam flexíveis e extensíveis.

3 Arquitetura Básica

Dadas as extensões que infraestruturas distribuídas têm atingido, o escalonamento é, muitas vezes, realizado por diversas decisões de atribuição sendo cada atribuição feita mais localmente e com um maior controle dos recursos. A figura 1 ilustra a interação de escalonadores em uma infra-estrutura de grade computacional. Nesse caso, o escalonador do topo da figura consiste em um meta-escalonador o qual é responsável por determinar qual subgrupo de máquinas, possivelmente componentes de um *cluster*, irá executar uma aplicação. Quando o comando de execução chegar ao *cluster*, o mesmo irá, de forma recorrente, escolher qual das máquinas sob seu domínio está mais apta à execução requerida. Uma vez escolhida a máquina, o escalonador do Sistema Operacional agenda a execução da aplicação no processador.

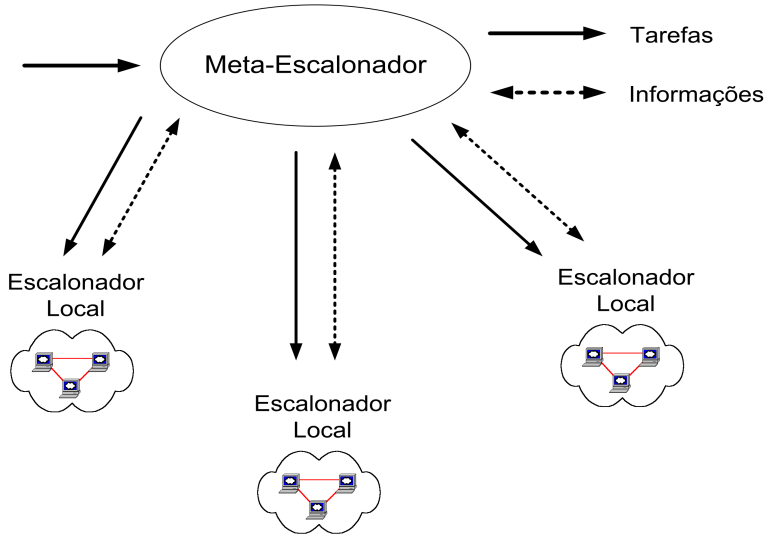

Figura 1: Arquitetura de um gerenciador de recursos hierárquico composta por grupos de clusters.

A hierarquia de escalonadores apresentada na figura 1 torna-se ainda mais complexa à medida que máquinas virtuais e arquiteturas *multi-core* de processamento se integram às infraestruturas distribuídas. Nesses casos, antes de uma aplicação ser executada, ainda é preciso decidir em qual processador e em qual máquina virtual ela irá ser executada.

4 Fases do Escalonamento

Basicamente, o escalonamento consiste de três fases: a descoberta de recursos, a seleção de recursos e a execução da aplicação [12]. A realização dessas fases envolve diferentes entidades de um *middleware* para computação distribuída. A descoberta de recursos, por exemplo, é feita através de uma interação com o Serviço de Monitoramento o qual detém informações a respeito dos estados dos recursos. A figura 2 apresenta, de forma bem definida, a seqüência de fases realizadas por um gerenciador de recursos a fim de escalonar as aplicações recebidas.

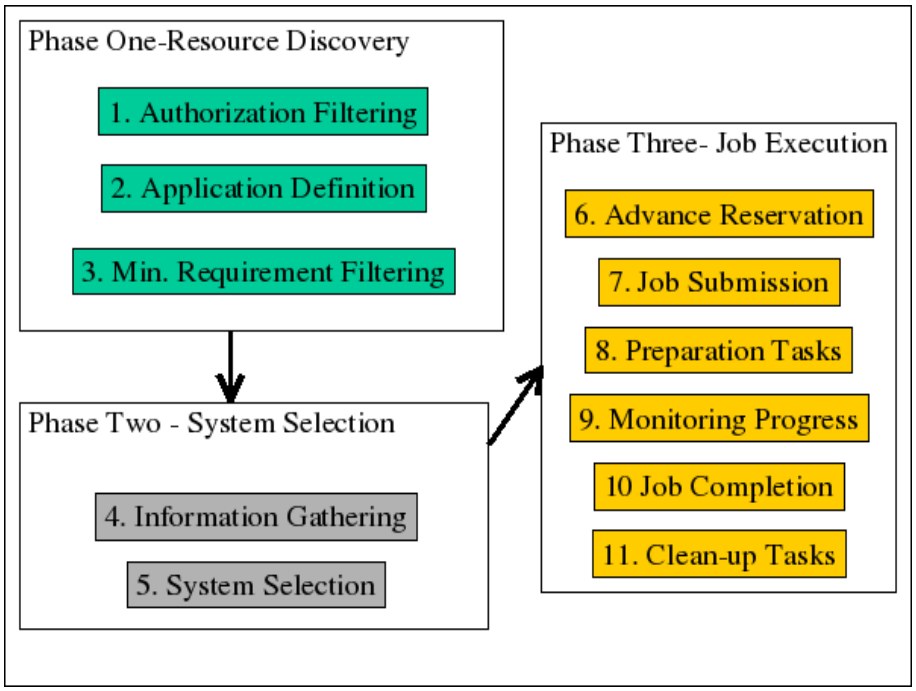

Figura 2: Arquitetura básica de um gerenciador de recursos [12].

4.1 Descoberta de Recursos

A fase de descoberta de recursos é responsável por selecionar um conjunto de recursos para ser investigado mais profundamente na seleção de recursos. Ela se divide em três etapas: a autorização, a definição dos requisitos da aplicação e o casamento dos requisitos mínimos. Na primeira etapa, apenas os recursos para os quais o usuário submissor da aplicação tem acesso são considerados.

Na segunda etapa, o usuário deve definir os requisitos básicos da aplicação tais como Sistema Operacional e arquitetura alvo, além de quantidades de recursos como, por exemplo, montante de memória RAM e processamento a serem utilizados. Por fim, ao final da etapa de casamento dos requisitos mínimos, apenas os recursos para os quais o usuário tem acesso e que apresentem os requisitos mínimos definidos estarão na lista de candidatos a nós executores da aplicação.

4.2 Seleção de Recursos

A seleção de recursos é realizada em duas etapas: a da coleta de informações dinâmicas detalhadas a respeito dos recursos, e a da seleção do(s) melhore(s) recurso(s) para executar a aplicação alvo. Na fase de coleta, a fonte das informações dinâmicas comumente consiste no Serviço de Monitoração do sistema, mas é possível consultar fontes de mais baixo nível como o próprio escalonador local de cada máquina.

A escolha dos melhores recursos pode ser realizada utilizando regras que consideram as informações retornadas na primeira etapa de coleta. Exemplos são a seleção de recursos implementada no *middleware* Condor, políticas multi-critérios e metaheurísticas.

4.3 Execução das Aplicações

Para que a execução de uma aplicação seja realizada com sucesso, é importante que haja uma reserva antecipada dos recursos que essa aplicação irá utilizar. Essa etapa, muitas vezes inexistente em diversos gerenciadores de recursos, é seguida da submissão da aplicação ao nó executor. As opções para a submissão variam da execução de um simples comando à execução de diversos *scripts* para a configuração do ambiente. Em seguida, há a necessidade de transferir os arquivos de entrada e outras configurações necessárias para a execução remota da aplicação.

Quando uma aplicação está sendo executada, é necessário realizar o monitoramento de sua execução a fim de detectar possíveis comportamentos indesejáveis tais como desempenho inferior ao esperado, ou mesmo uma falha no nó de execução. Em casos de falha, pode-se realizar um novo escalonamento.

Ao término da execução é preciso notificar o submissor da aplicação e transferir os arquivos de saída gerados. Os arquivos criados na máquina executora devem então ser removidos.

5 Condor

Desenvolvido na Universidade de Wisconsin-Madison, o Condor consiste em um sistema especializado de gerenciamento de cargas. Ele provê mecanismos de enfileiramento e priorização de aplicações, políticas de escalonamento e monitoração de recursos. Usuários do Condor têm suas aplicações escalonadas, monitoradas e com valor de retorno informado automaticamente [13].

Outras características presentes no Condor são a possibilidade de especificação de dependência entre tarefas, o suporte a modelos seqüenciais e paralelos (MPI e PVM) de aplicações, mecanismos de oferta e procura de recursos (*classAds*), além de *checkpointing* e migração de processos.

Diferentemente de outros sistemas de gerenciamento de carga tais como o PBS, o LoadLeveler e o LSF, o Condor não necessita de um sistema de arquivos compartilhado entre as máquinas que gerencia, mesmo assim, ele é capaz de transferir arquivos de forma transparente ao usuário, sendo que, se uma tarefa estiver em execução em uma determinada máquina e esta for requisitada pelo usuário local, o Condor é capaz de realizar um *checkpoint* no processo e migrá-lo para outra máquina sob seu domínio, ou seja, o usuário não perde a autonomia sobre o seu recurso. Trata-se portanto de um sistema bastante avançado de computação distribuída.

5.1 MatchMaking

O escalonamento de processos no Condor é feito através de um mecanismo denominado *matchmaking* o qual decide quando, como e onde será executada uma determinada tarefa. O ciclo desse casamento se dá conforme ilustrado pela figura 5.1. Primeiramente, cada recurso e tarefa necessitando de execução, através de um componente denominado agente, anunciam suas respectivas existências à entidade de *matchmaker*. Essa atividade é chamada de *Classified advertisements* ou *ClassAds* somente. De posse das informações de oferta e procura, o *matchmaker* cria pares que satisfazem às necessidades e limitações um do outro. Esses pares são notificados à tarefa e recurso envolvidos no passo 3 e, finalmente, no passo

4 essas partes negociam possíveis termos e começam a execução. O mecanismo de ClassAds é feito em cima de uma linguagem própria, da forma *nome=valor*, para a especificação das necessidades de uma tarefa ou de uma máquina. Um exemplo encontra-se na figura 5.1.

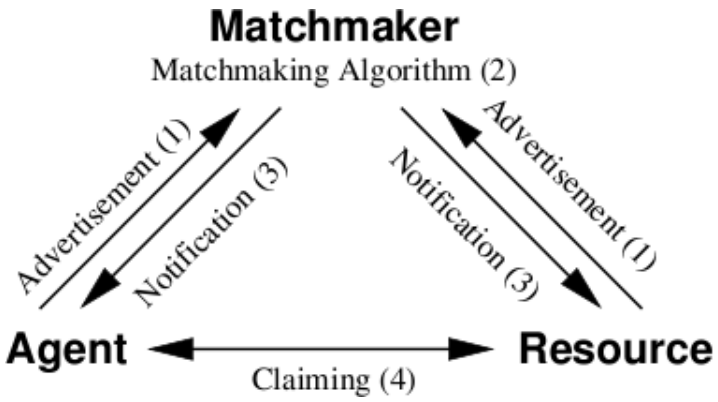

Figura 3: Passos do esquema de *matchmaking* [14].

Job ClassAd	Machine ClassAd
[	[
MyType = "Job"	**MyType** = "Machine"
TargetType = "Machine"	**TargetType** = "Job"
Requirements =	**Machine** = "nostos.cs.wisc.edu"
((other.Arch=="INTEL" &&	**Requirements** =
other.OpSys=="LINUX")	(LoadAvg <= 0.300000) &&
&& other.Disk > my.DiskUsage)	(KeyboardIdle > (15 * 60))
Rank = (Memory * 10000) + KFlops	**Rank** = other.Department==self.Department
Cmd = "/home/tannenba/bin/sim-exe"	**Arch** = "INTEL"
Department = "CompSci"	**OpSys** = "LINUX"
Owner = "tannenba"	**Disk** = 3076076
DiskUsage = 6000	]
]	

Figura 4: Exemplos de utilização de ClassAd para a notificação de tarefas e máquinas [14].

Diferentes casamentos de ofertas e procuras podem acontecer configurando diferentes valores no campo *rank* do *ClassAd*. Esse campo informa como ordenar recursos/aplicações em caso de diversas entidades satisfazerem à uma mesma procura. No caso da figura 5.1, por exemplo, a aplicação do *ClassAd* esquerdo priorizará máquinas com as maiores quantidades de memória e processamento de operações em ponto flutuante, enquanto a máquina do *ClassAd* direito priorizará as aplicações submetidas a partir de máquinas presentes no mesmo departamento a qual ela pertence.

5.2 Ambientes de Execução

O Condor provê seis diferentes tipos de ambientes de execução chamados universos. Os universos consistem em Vanilla, MPI, PVM, Globus, Scheduler e *Standard*. O tipo de universo a ser utilizado deve ser especificado no arquivo de descrição de submissão, caso contrário, o tipo padrão (*Standard*) será utilizado.

O universo Vanilla é utilizado para executar aplicações seriais. Esse ambiente apresenta a vantagem de que qualquer programa que executa fora do Condor pode ser executada nele sem que seja necessário nenhuma alteração, ligação ou recompilação. Por outro lado, esse ambiente não apresenta as características de *checkpoint*, migração, chamada remota ao sistema e *bufferização* de entrada/saída presentes no universo *Standard*. Para que o universo *Standard* proporcione *checkpointing* e chamada remota ao sistema é preciso que a aplicação sofra uma religação com a biblioteca de chamada ao sistema do Condor.

A chamada remota ao sistema presente no universo *Standard* torna desnecessário o acesso a arquivos na máquina remota via algum tipo de sistema de arquivo em rede tal como o NFS. Na chamada remota, quando um processo acessa um arquivo, ele chama uma função do tipo *open(), read()* ou *write()* que sofreram uma ligação com uma biblioteca C do Condor em vez da biblioteca C padrão. Na biblioteca Condor, durante uma chamada ao sistema, os *stubs* empacotam o número da chamada e seus argumentos em uma mensagem enviada pela internet ao processo *condor_ shadow* sendo executado na máquina submissora. O *condor_ shadow* consiste em um processo do tipo *daemon* criado na máquina submissora quando se submete uma aplicação ao Condor. Esse processo atua como um agente para a aplicação executada remotamente realizar chamadas ao sistema assim como ilustra a figura 5.

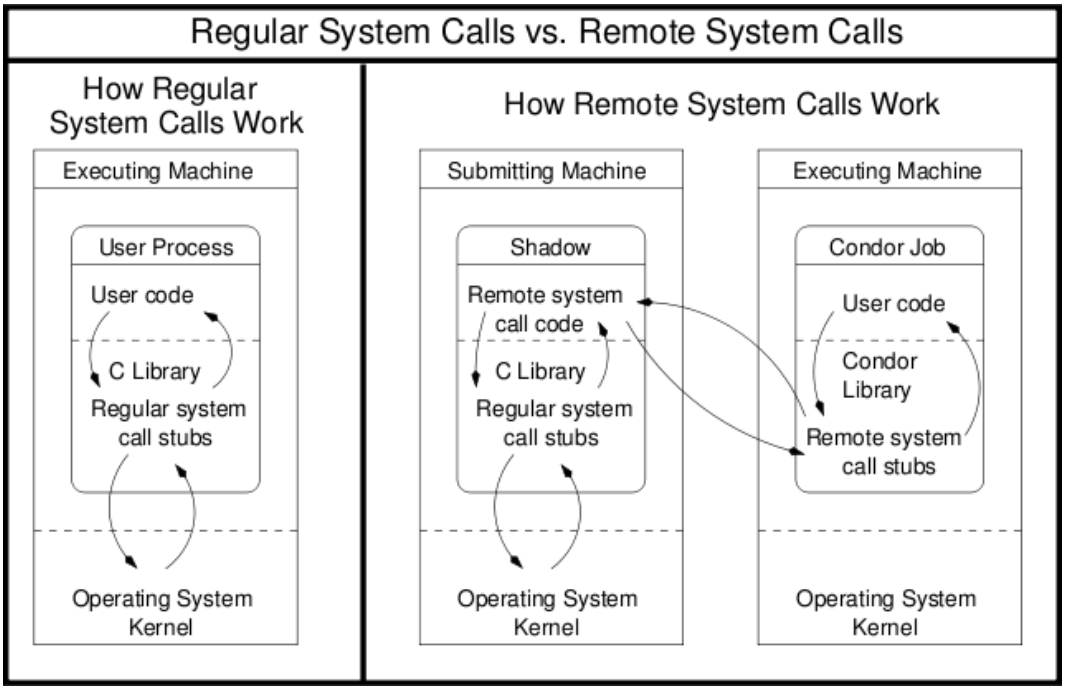

Figura 5: Chamadas remotas ao sistema executadas no universo Standard [13].

O processo *condor_ shadow* então realiza a chamada ao sistema em nome da aplicação remota. Os resultados dessa chamada são empacotados em uma mensagem enviada ao *stub* da biblioteca Condor na máquina remota. Dessa forma, garante-se que a aplicação a ser submetida não precisa tomar conhecimento das chamadas remotas. Para ela, todos os procedimentos parecem ser realizados localmente.

O Condor realiza todo o processo de *checkpoint* e migração de processos em modo de

usuário, o que gera algumas limitações no tipo de processo que pode sofrer *checkpoint*. Esses são os casos de processos que utilizam as chamadas *fork(), exec()* e *system()*, além de processos que realizam comunicação via *pipes*, semáforos e memória compartilhada. Da mesma maneira, são proibitivos o uso de *threads* no nível de *kernel* e o acesso a arquivos com permissões conjuntas de leitura e escrita.

Os universos MPI e PVM devem ser utilizados para aplicações paralelas que utilizem essas bibliotecas de desenvolvimento. O universo Globus é direcionado a prover a interface padrão Condor a usuários que desejam submeter suas aplicações à máquinas gerenciadas pelo Globus [15]. Por fim, o universo Scheduler é utilizado para submeter aplicações as quais devem ser imediatamente postas em execução nas máquinas originárias da submissão. O objetivo desse universo é prover meios que facilitem a construção de meta-escalonadores que gerenciem a submissão e a remoção de aplicações em uma fila Condor. Um exemplo de meta-escalonador que utiliza esse ambiente é o DAGMan o qual é descrito na Seção 5.3

5.3 DAGMan

O DAGMan (*Directed Acyclic Graph Manager*) consiste em um escalonador de grafos acíclicos direcionados. Dessa maneira, ele possibilita a especificação de dependências entre um conjunto de programas, onde a saída de um programa pode ser a entrada de outro. A figura 6 apresenta um exemplo simples de um DAG onde os programas são representados pelos vértices do grafo e as dependências entre eles pelas arestas.

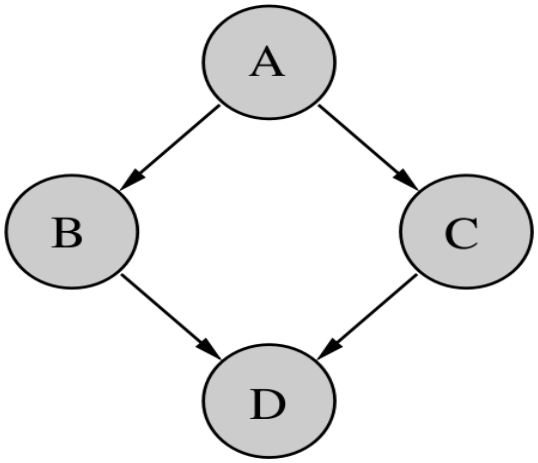

Figura 6: Um grafo acíclico direcionado com quatro nós [13].

No caso da figura 6, o programa A será executado primeiro. Após o seu término, os programas B e C serão postos para execução em paralelo. Somente após a finalização desses processos é que o processo C será iniciado.

5.4 Arquitetura

Uma infra-estrutura Condor é formada por um nó gerenciador central e um número arbitrário de outros nós divididos entre nós executores (doadores de recursos) e nós submissores, assim como ilustrado na figura 7. Cada componente da figura apresenta funcionalidades

bem definidas, implementadas através de *daemons* (representados dentro de retângulos de cantos arredondados).

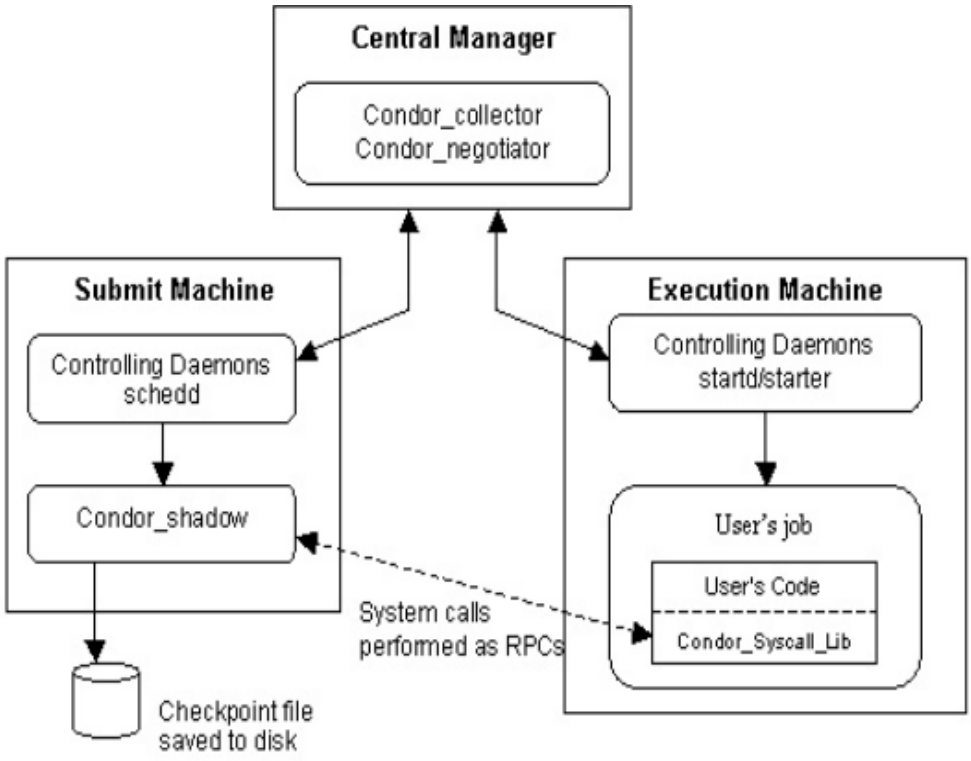

Figura 7: Arquitetura básica de um *pool* Condor [8].

O papel do gerenciador central é coletar informações a respeito dos estados dos recursos das máquinas executoras e dos pedidos de execução das máquinas submissoras. Dessa maneira, ele apresenta um *daemon condor_collector* responsável por receber notificações regulares de *classAds* e armazená-las para posteriores consultas. Dado que o gerenciador central apresenta todas as ofertas e procuras de recursos, ele é também o responsável pela realização dos *matchmakings*.

Os nós submissores apresentam dois tipos de *daemons*: o *condor_schedd* e o *condor_shadow*. O primeiro dos *daemons* possibilita ao nó submeter processos e inseri-los em uma fila. As diversas ferramentas para ver e manipular a fila de processos se conectam ao *condor_schedd* para realizar os seus trabalhos. O segundo *daemon* serve à requisições de transferência de arquivos e ao armazenamento de informações de progresso de execução de aplicações. No caso de aplicações ligadas ao universo padrão do Condor, é papel do *condor_shadow* intermediar as chamadas remotas ao sistema, executando a chamada ao sistema na máquina submissora e retornando o resultado via rede à aplicação sendo executada.

Todo nó executor deve apresentar um *daemon condor_startd* ativo. Esse *daemon* permite a uma máquina executar aplicações, garantindo a política na qual aplicações remotas serão iniciadas, suspensas, reiniciadas, interrompidas ou finalizadas. Quando o *daemon condor_startd* está pronto para executar uma aplicação, ele dispara um *daemon condor_starter*. Esse *daemon condor_starter*, por sua vez, é o programa que dispara a

aplicação na máquina remota. Ele configura o ambiente remoto de execução e monitora a aplicação durante o seu tempo de vida. Também é papel do *condor_ starter* detectar o término da aplicação e enviar informações de estado de volta ao nó submissor. Ao completar o seu trabalho, o *daemon condor_ starter* é finalizado.

Todos os *daemons* descritos são monitorados pelo *daemon condor_ master*. Esse *daemon* é responsável por disparar e manter a execução dos demais, além de informar ao sistema administrador sobre possíveis falhas nesses programas.

5.5 Regras de Escalonamento

As filas de aplicações no Condor não são apenas do tipo FIFO (*First in First Out*). Há também a implementação de filas de prioridade, onde é possível favorecer determinados usuários em detrimento de outros. Pode-se, por exemplo, habilitar a opção de *preempção*. Nesse caso, ao chegar um processo de prioridade X em uma fila contendo um processo de prioridade Y sendo executado, onde X tem maior prioridade que Y, o processo Y é interrompido para que X execute. A inanição de execução de processos de baixa prioridade é evitada através da implementação de algoritmos de compartilhamento justo e da priorização baseada em taxas, onde um usuário de prioridade 2 recebe, aproximadamente, o dobro de recursos de um usuário de prioridade 1, assumindo que quanto maior o valor da prioridade, mais relevante ela é.

6 PBS

Desenvolvido pela NASA nos anos 90, o PBS consiste em um sistema de gerenciamento de recursos baseado em filas e capaz de atuar em redes Unix multiplataforma [2]. Ele recebe requisições de aplicações, as coloca em execução, realiza o monitoramento das mesmas e entrega os resultados ao submissor.

Algumas das principais características do PBS são:

- **Balanceamento de Carga** - o escalonador provê diversas maneiras de distribuir a carga de trabalho entre as máquinas disponíveis baseado seja na configuração de *hardware*, na disponibilidade de recursos ou mesmo na utilização de dispositivos de entrada como, por exemplo, o teclado;
- **Transferência Automática de Arquivos** - usuários PBS podem especificar os arquivos necessários para uma execução remota. O PBS transfere esses arquivos para a máquina executora e, somente após o término da transferência, é que a aplicação tem início;
- **Interdependência entre Comandos** - o PBS permite a definição de dependência entre comandos, incluindo ordenação de execuções, sincronização e execução condicionada a sucesso ou falha de um comando anterior;
- **Autorização de Acesso** - é possível permitir ou negar acesso a determinadas máquinas ou usuários;
- **Mapeamento de Usuários** - permite que usuários apresentem diferentes nomes de acesso em diferentes máquinas;

- **Suporte a Aplicações Paralelas** - aplicações desenvolvidas com as bibliotecas de programação paralela MPI, MPL, PVM e HPF podem ser escalonadas para serem executadas em uma máquina multiprocessada ou mesmo em máquinas distintas;
- **Contabilidade** - o PBS mantém arquivos detalhados de descrição das atividades realizadas por usuários;
- **Ambiente Gráfico** - interface gráfica para a submissão, consulta e enfileiramento de comandos;
- **Interface de Programação** - disponibilização de uma API para escrever novos comandos, integrar as funcionalidades do PBS nos códigos das aplicações ou implementar novas políticas de escalonamento.

O PBS segue o padrão POSIX, incluindo o POSIX 1003.2d referente ao padrão POSIX de sistema de gerenciamento de processos em lote.

6.1 Arquitetura

A arquitetura do PBS consiste basicamente de quatro componentes: os comandos, o servidor, os nós executores e o escalonador [16].

- **Comandos** - comandos podem ser submetidos via linha de comando ou via interface gráfica. Eles são utilizados para submeter, monitorar, modificar ou remover processos em execução. Os comandos podem ser instalados em qualquer tipo de plataforma suportada pelo PBS e podem ser disparados sem que haja outros componentes PBS presentes. Os comandos PBS podem ser de três tipos: usuário, do qual qualquer usuário autorizado pode fazer uso; operador e gerenciador. Nos dois últimos casos, os privilégios de acesso diferem entre si. Exemplos de comandos de usuário, de operador e de gerenciador, respectivamente, são o *qsub* o qual cria um processo para ser submetido ao servidor; o *qrun* o qual executa um processo; e o *qmgr* o qual provê uma interface de administração do sistema.
- **Servidor** - a função do servidor é prover serviços básicos para o gerenciamento de processos, sendo o responsável por tratar uma ou mais filas de processos. Dessa maneira, ele deve conhecer *a priori* a lista de nós que podem executar aplicações. Todos os comandos e outros componentes se comunicam com o servidor através de uma rede IP. Clientes acessam as informações a respeito das filas por meio de consultas ao servidor. Os processos residentes nas filas de roteamento são candidatos a migrar para uma nova fila possivelmente gerenciada por outro servidor;
- **Executores** - todo nó executor deve apresentar um processo (*daemon*) que efetivamente coloca as aplicações para executar. Esse *daemon*, chamado MOM (*Machine Oriented Miniserver*), é o criador de todos os processos em execução. É papel do MOM colocar um processo em execução ao receber uma cópia desse processo vinda do servidor e então criar uma sessão idêntica à do *login* do usuário quando possível. Também é responsabilidade do MOM redirecionar os arquivos de saída gerados durante a execução das aplicações;

- **Escalonador** - consiste em um *daemon* que aplica a política de escalonamento ativa localmente em cada *cluster*. Isso porque cada *cluster* pode desejar criar as suas próprias regras de escalonamento. Ao ser executado, o escalonador é capaz de se comunicar com diversos MOMs, para coletar informações a respeito dos recursos, e com o servidor para saber quais processos estão em espera por execução.

A figura 8 ilustra a interação entre os componentes da arquitetura PBS. Na primeira fase (1), um evento disparado pelo cliente requisita ao servidor o início de um novo ciclo de escalonamento. Em seguida (2), o servidor envia um comando de escalonamento ao escalonador. Esse, por sua vez, (3) requere informações a respeito dos estados dos recursos aos MOMs ativos. Os MOMs retornam as informações desejadas ao escalonador (4). O escalonador então requere ao servidor informações a respeito da aplicação a ser executada (5). O servidor retorna as informações ao escalonador (6) que, nesse momento, está pronto para realizar a escolha da melhor máquina para execução. O escalonador envia ao servidor o endereço da máquina que executará a aplicação (7). O servidor, enfim, envia a aplicação ao MOM da máquina escolhida (8).

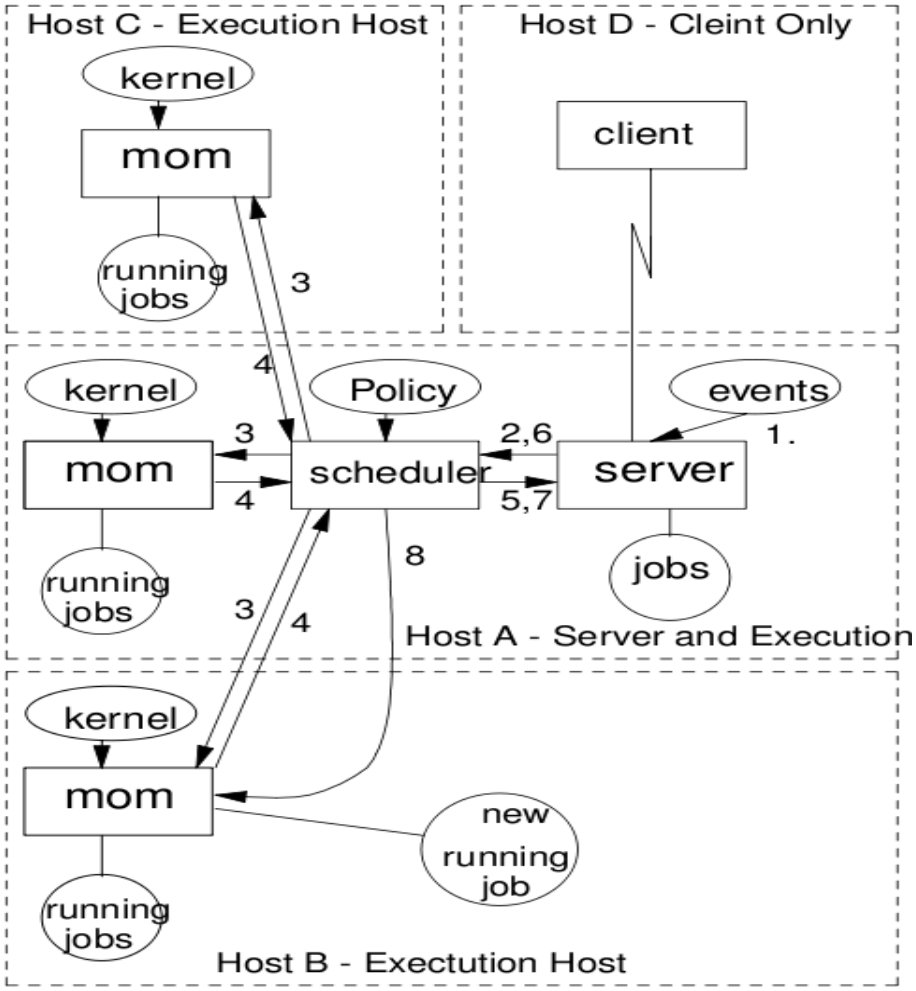

Figura 8: Interações realizadas em um ciclo de escalonamento [17].

6.2 Regras de Escalonamento

Toda aplicação submetida a execução deve definir uma lista de recursos necessários para a sua execução (quantidade de processamento, montante de memória, número de nós, entre outros), a sua prioridade, o tempo de execução desejado e as suas dependências. Essas informações guiam o escalonador na escolha de quais aplicações devem executar em quais recursos.

No PBS, a política padrão adotada é baseada em uma FIFO[2], mas é possível definir novas políticas de escalonamento. O servidor e os atributos das filas utilizadas para gerenciar o escalonamento de processos podem ser ajustados por um cliente com privilégios de administrador para refletir novos interesses na alocação de recursos. Porém, é preciso atentar que os mecanismos de controle residem no componente escalonador e não no servidor como já descrito na Seção 6.1.

Todo ciclo de escalonamento é iniciado pelo servidor. Esses ciclos acontecem quando: uma nova aplicação chega para ser executada ou é finalizada; um período especificado de tempo desde o último ciclo expirou; ou houve um comando forçado para o servidor escalonar (vindo de algum administrador do sistema). Ao ser contactado para um novo ciclo, o escalonador se torna um cliente privilegiado do servidor. Dessa maneira, ele é capaz de comandar o servidor em qualquer ação de gerenciamento necessária.

As políticas de escalonamento podem ser implementadas em três tipos de linguagem: C, TCL e BaSL. BaSL é uma linguagem baseada em C, procedural, desenvolvida para o PBS e convertida para C através do compilador *basl2c*. Essa linguagem apresenta predefinições e construtores que são utilizados para obter informações do servidor, das filas e dos nós de execução. Escalonadores baseados em TCL utilizam o interpretador TCL e comandos extras ao PBS para a comunicação com o servidor e com o MOM. Os escalonadores escritos em C são similares aos TCL.

6.3 Torque

Em 2004, surgiu o Torque (*Terascale Open-Source Resource and Queue Manager*), um gerenciador de recursos de código aberto baseado previamente na versão 2.3.12 do OpenPBS [19, 20, 21, 22]. Com mais de 1200 pontos de modificação no código do OpenPBS, o Torque incorporou significativas melhorias ao PBS com contribuições de importantes organizações da área de Computação de Alto Desempenho tais como NCSA (*National Center for Supercomputing Applications*), OSC (*Ohio Supercomputer Center*), USC (*University of Southern California*) e TeraGrid.

Algumas das características inseridas ao OpenPBS pelo Torque estão nas áreas de:

- **Tolerância a Falhas** - foram adicionadas novas condições, detecções e tratamento de falhas;
- **Interface de Escalonamento** - adição de novas e mais acuradas informações para consulta a respeito dos recursos. Também foram adicionadas interfaces para um maior controle do comportamento das aplicações. Além disso, houve a inclusão de permissão para a coleta de estatísticas a respeito das aplicações já executadas;

[2]O escalonador percorre a lista ordenada de aplicações e escolhe aquela que possa ser executada nos recursos disponíveis. Essa prática aumenta a taxa de utilização do sistema. Para que não aconteça inanição dos processos um limite máximo de espera pode ser definido [18].

- **Escalabilidade** - melhorias significativas no servidor para a implementação do modelo MOM (*Message-Oriented Middleware*) de comunicação. O gerenciador tornou-se escalável para *clusters* de até 2500 processadores, podendo servir aplicações maiores (que utilizem até 2000 processadores) e suportando mensagens maiores do servidor;

- **Usabilidade** - adição de novas funcionalidades de *logging* de maneira mais legível ao homem.

Em sua versão 2.1.2, Torque passou a oferecer um mecanismo de redirecionamento do ambiente gráfico X11 e a suportar comandos clientes na plataforma Windows via Cygwin.

6.4 Maui

O Maui surgiu com o objetivo de suprir algumas carências no desempenho das políticas de escalonamento implementadas no sistema IBM LoadLeveler tais como a alta taxa de ociosidade de nós de uma máquina paralela à espera de trabalho [18]. Trata-se de um gerenciador de processos de código aberto para *clusters* e supercomputadores. Por não ser um gerenciador de recursos, o Maui atua como um componente adicional de sistemas dessa categoria tais como o PBS e o Torque [23]. Nesses casos, ele estende as capacidades de gerenciamento de recursos inserindo as seguintes características:

- **Prioridade de Aplicações** - é possível atribuir pesos aos vários objetivos de um escalonamento;

- **Reserva Antecipada de Recursos** - toda reserva apresenta três componentes: a lista de recursos, o tempo de reserva e a lista de controle de acesso. A reserva antecipada permite a implementação de características tais como escalonamento do tipo *backfill* e baseada em prazos, suporte a qualidade de serviço e meta-escalonamento.

- **Suporte a Qos** - administradores podem configurar diversos tipos de qualidade de serviço, cada tipo de qualidade apresentando um conjunto próprio de prioridades, políticas de isenção e configuração de acesso aos recursos. Os administradores podem então configurar usuários, grupos, contas e classes como beneficiários dos privilégios de QoS;

- **Políticas de Doação de Recursos** - permitem a especificação das condições nas quais os recursos são disponibilizados para serem alocados às aplicações;

- **Políticas de Backfill** - essas políticas permitem ao escalonador realizar melhores usos dos recursos disponíveis ao executar aplicações fora da ordem estabelecida em determinada fila. No caso da aplicação de maior prioridade da fila não poder ser executada por falta de recursos, uma aplicação de menor prioridade e com requisições mais simples de recursos pode ser executada no lugar da primeira [24];

- **Modo de Teste** - nesse modo, o escalonador funciona como se estivesse executando normalmente ou em modo de produção, mas o Maui torna-se incapaz de iniciar, interromper, cancelar ou modificar atributos das aplicações ou dos recursos. Usando esse modo, é possível validar a interface escalonador-gerenciador de recursos e testar comandos Maui sem que as operações do sistema sejam comprometidas;

- **Simulador de Desempenho** - trata-se de um simulador para analisar o desempenho dos escalonadores. Ele permite analisar diferentes configurações, muitas vezes perigosas, sem que esses experimentos afetem a execução das aplicações.

7 OAR

O OAR consiste em um gerenciador de recursos para sistemas altamente distribuídos desenvolvido no Instituto Politécnico Nacional de Grenoble na França. Ele segue uma filosofia de que é possível construir um sistema complexo de gerenciamento de recursos, com componentes de alto nível, sem que a eficiência e a escalabilidade do mesmo seja comprometida [25].

Os desenvolvedores do OAR acreditam que um sistema de banco de dados de propósito geral é a melhor escolha para garantir uma interface amigável e poderosa para a extração e análise de dados a respeito dos estados dos recursos e das execuções das aplicações. Esse pensamento motivou o desenvolvimento do OAR no topo de um sistema de banco de dados relacional, o MySQL. Conseqüentemente, não há nenhuma API para uso do OAR. O sistema é todo definido a partir de esquemas do banco de dados e assim, cada módulo, tal como o escalonador, pode ser desenvolvido em qualquer linguagem que apresente uma biblioteca de acesso a banco de dados [26].

Perl foi escolhida como linguagem de desenvolvimento porque linguagens de *script* tendem a se adequar bem a tarefas de sistemas de baixo nível tal como a execução remota de processos. O desenvolvimento de aplicações com Perl é bastante simples uma vez que a linguagem provê estruturas de dados de alto nível.

Algumas das principais características do OAR são:

- Execução de aplicações interativas ou de lote;
- Gerenciamento de filas com prioridade;
- Controle de admissão;
- Suporte para aplicações multi-paramétricas;
- Tempo de parede gerenciável;
- Suporte para a reserva de nós;
- Casamento de oferta-procura de recursos;
- *Logging* de informações com interface amigável;
- Interrupção de execução quando os recursos são requeridos;
- *Checkpoint* de processos e nova submissão dos mesmos;
- Ausência de *daemons* nos nós executores;
- *Rsh* e *ssh* como protocolos de execução;
- Mecanismos para políticas de *backfilling*.

Em [25], Capit *et. al.* realizaram experimentos para validar o OAR. Eles concluíram que mesmo esse sistema sendo composto por um gerenciador de banco de dados e uma linguagem interpretada, ele atinge um nível de desempenho aceitável quando comparado com outros sistemas de gerenciamento de recursos como o caso do Maui acoplado ao Torque e do SGE.

7.1 Arquitetura

A arquitetura do servidor OAR é formada por dois componentes: o banco de dados relacional e os *scripts* Perl assim como ilustra a figura 9.

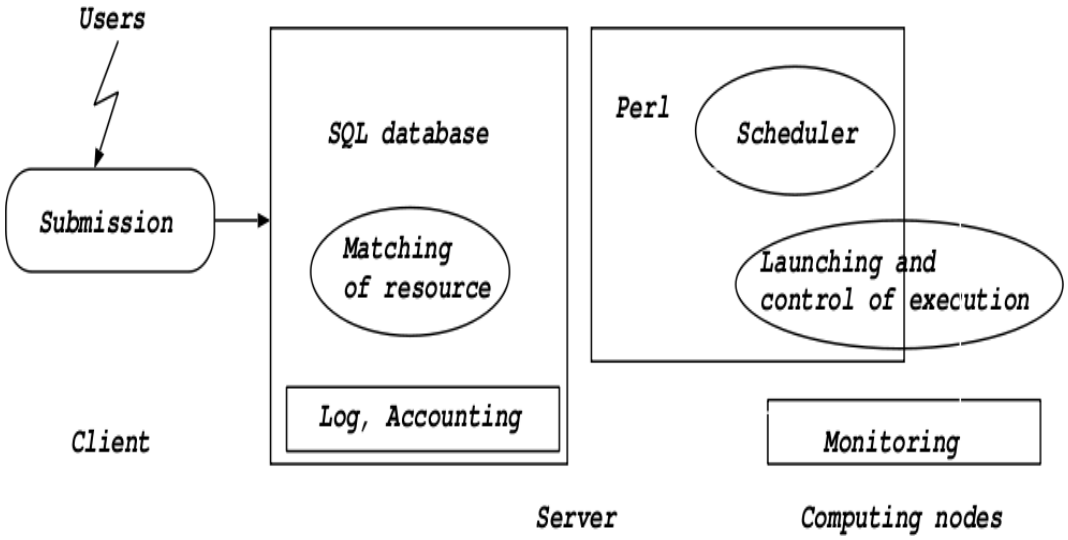

Figura 9: Arquitetura global do OAR [25].

O banco de dados é utilizado como único meio de troca de informações entre os módulos. Ele é o responsável por casar as ofertas e procuras de recursos utilizando consultas SQL. Essa prática de armazenar informações em banco de dados atribui características de robustez e eficiência ao OAR, visto que bancos de dados têm poucas chances de se tornar um gargalo do sistema por serem capazes de processar de forma eficiente milhares de consultas simultaneamente. A robustez, por sua vez, é dependente dos módulos que precisam deixar o sistema em um estado coerente.

No OAR, cada aplicação deve ser representada por uma tabela SQL. Um exemplo de tabela é ilustrada na figura 10, onde a coluna da direita apresenta uma breve descrição de cada campo.

A parte de execução do servidor é composta por uma coleção de módulos independentes implementados como *scripts* Perl. Cada módulo é responsável por tarefas bem específicas tais como iniciar e controlar a execução de aplicações. Para a realização dessas tarefas, os módulos interagem com o sistema através do banco de dados SQL.

As tarefas relacionadas ao monitoramento de recursos, assim como outras tarefas administrativas, são realizadas com a ferramenta Taktuk [27] a qual é chamada a partir do OAR e apresenta uma interface de acesso ao banco de dados. Para escalar bem em sistemas gerenciados pelo OAR, o Taktuk foi projetado para ser altamente paralelizado e distribuído.

fi elds	comment
idJob	numeric identifi er of the job
jobType	either INTERACTIVE or PASSIVE
infoType	machine to contact for interactive jobs
state	either 'Waiting', 'Hold', 'toLaunch', 'to-Error', 'toAckReservation', 'Launching', 'Running', 'Terminated' or 'Error'
reservation	either 'None' (general case), 'toSchedule' or 'Scheduled' (reservation of a precise time slot)
message	additional information (warnings, reason for termination, ...)
user	user that request the job execution
nbNodes	number of nodes required
weight	correspond to the number of processors required on each node (if several are available)
command	command to execute (the job itself)
bpid	PID used to kill the job when needed
queueName	queue in which the job is waiting for scheduling
maxTime	maximal execution time of the job
properties	sql expression used to match ressources compatible with the job
launchingDirectory	directory in which the command has to be executed
submissionTime	date of submission
startTime	date of beginning of the execution
stopTime	date of termination of the execution

Figura 10: Tabela SQL representando uma aplicação no OAR [25].

As aplicações são submetidas para execução através de comandos independentes os quais enviam e recebem informações diretamente do banco de dados. Os comandos interagem com o módulo central do OAR por meio de notificações. Uma submissão é iniciada com uma conexão ao banco de dados para obter as regras de admissão apropriadas. Essas regras são usadas para configurar valores de parâmetros não providos pelos usuários e para a validação da submissão. As regras são armazenadas no banco em código Perl.

Uma vez submetido a uma fila de execução, um processo recebe um identificador, é incluído no banco de dados e retorna uma mensagem ao usuário informando que a aplicação submetida está em espera por execução. Em seguida, uma mensagem é enviada ao módulo central para que ele escalone o processo tão logo for possível.

7.2 Regras de Escalonamento

Apesar da arquitetura de escalonamento do OAR ser rica em funcionalidades, apresentar mecanismos de prioridade de aplicações, reserva de recursos e *backfilling*, os algoritmos utilizados por ela são bastante simples. Cada aplicação é submetida para uma fila a qual implementa as suas próprias regras de admissão e escalonamento.

A política padrão implementada no OAR consiste na FCFS (*First Come First Served*)

com *backfilling* conservativo. Nessa política, todas as aplicações são ordenadas de acordo com o seu tempo de chegada na fila. Gradativamente, cada aplicação é escalonada o mais breve possível desde que não atrase aplicações posicionadas à sua frente na fila.

8 Mosix

Mosix é um sistema de gerenciamento para alto desempenho direcionado a *clusters* e grades organizacionais[3] de plataforma Linux. Ele consiste de diversos algoritmos adaptativos para o compartilhamento de recursos. O núcleo da tecnologia Mosix é a capacidade de múltiplos nós trabalharem de forma cooperativa como se fossem um único sistema [28].

Os algoritmos do Mosix são projetados para reagir à variação na utilização de recursos através da migração de processos de um nó para outro de maneira transparente, realizando assim o balanceamento de carga do sistema [29, 30].

O Mosix é implementado como uma camada de *software* que provê um ambiente de execução Linux não-modificado. Dessa maneira, aplicações podem ser executadas sem a necessidade de alteração de código ou mesmo ligação com alguma biblioteca. Essas aplicações podem ser executadas remotamente como se estivessem na máquina local [31, 10].

O sistema Mosix pode ser executado em modo nativo ou em uma máquina virtual. No modo nativo, o desempenho é maior, mas requer modificações ao núcleo do Linux, enquanto as máquinas virtuais podem ser executadas em qualquer Sistema Operacional que ofereça suporte a virtualização tais como Windows, Linux e OS-X.

8.1 Características

O Mosix é implementado como uma camada de *software* que permite a aplicações serem executadas remotamente como se estivessem sendo executadas no nó local. Essa característica é alcançada pela interceptação de chamadas ao sistema. Se um processo é migrado, então a grande maioria de suas chamadas ao sistema são redirecionadas para o nó de origem. Dessa maneira, a consistência de dados, assim como mecanismos de comunicação inter-processos tais como sinais, semáforos e identificação de processos são mantidos intactos. Uma outra vantagem da transparência de execução é que os usuários do sistema Mosix não precisam saber onde seus programas estão sendo executados, não necessitam alterar suas aplicações e nem ligá-las a nenhuma biblioteca adicional. Além disso, também não há necessidade de autenticação e nem da cópia de arquivos para nós remotos.

O sistema Mosix suporta dois tipos de processos: os processos Linux e os processos Mosix. No primeiro tipo, os processos executam no modo nativo do Linux sem serem afetados pelas modificações impostas ao Sistema Operacional pelo Mosix. Nesse tipo de processo estão as tarefas administrativas e aquelas que não são indicadas a sofrer migração. Já no segundo tipo de processos, há o usufruto das funcionalidades providas pelo Mosix, entre elas a migração.

Outra classe de processos Linux são aqueles criados com o comando *mosrun -E*. Esses processos não podem ser migrados, mas podem ser atribuídos aos nós com menor carga de trabalho do *cluster* através do comando *mosrun -b*.

[3] Grades formadas por múltiplos *clusters*.

Cada processo no Mosix apresenta um identificador de sua origem, chamado UHN (*Unique Home-Node*), onde o mesmo foi criado. Processos migrados utilizam recursos locais sempre que possível, mas interagem com o ambiente original do usuário via UHN. Por exemplo, se um processo migrado invoca a chamada *gettimeofday()*, ele irá obter a hora atual de sua UHN.

No Mosix, não há um controle central ou uma relação mestre-escravo entre os nós administrados, ou seja, cada nó funciona como um sistema autônomo realizando suas próprias decisões de maneira independente. Esse modelo permite a configuração dinâmica do sistema uma vez que nós podem entrar/sair da infra-estrutura com o mínimo de impacto. A escalabilidade é atingida incorporando aleatoriedade nos algoritmos de controle, onde cada nó realiza suas decisões baseado em um conhecimento parcial do estados dos demais nós.

Algumas das principais características presentes no Mosix serão apresentadas nas seções seguintes.

8.1.1 Criação de Clusters Lógicos

Na configuração Mosix, os nós podem ser divididos em *clusters* lógicos os quais são alocados a usuários ou objetivos específicos. *Clusters* lógicos podem abranger nós de diferentes *clusters*, mas não correspondem necessariamente a *clusters* físicos.

8.1.2 Descoberta Automática de Recursos

O algoritmo para descoberta de recursos é baseado em uma disseminação aleatória. Nesse algoritmo cada nó regularmente monitora os estados de seus recursos e repassa essa informação, juntamente com informações similares recebidas de outros nós, a outros nós escolhidos aleatoriamente, sendo que nós do mesmo *cluster* lógico têm maiores chances de serem escolhidos.

Informações a respeito de novos recursos na estrutura de um *cluster* são disseminadas gradativamente, enquanto informações sobre a desconexão de nós devem ser rapidamente repassadas.

8.1.3 Migração de Processos

A migração de processos pode ser invocada automaticamente ou de forma manual. No primeiro caso, a migração é supervisada por algoritmos que tentam aumentar o desempenho do sistema migrando processos que requisitam mais memória que o disponível no nó local ou apenas migrando um processo de um nó lento para um mais veloz. As decisões da migração automática são baseadas em uma caracterização do processo sendo executado e nas últimas informações a respeito da disponibilidade de recursos no *cluster*. A caracterização de processos é realizada pela coleta contínua de informações a respeito de traços de execução desses processos tal como taxas de chamadas ao sistema, volume de comunicação e entrada/saída de dados. Essa caracterização alimenta algoritmos que determinam qual é o melhor nó para executar o processo. Esses algoritmos consideram também as respectivas velocidades, cargas de trabalho e memória disponível dos nós juntamente com o tamanho do processo a ser migrado.

Se um processo é migrado, então a grande maioria de suas chamadas ao sistema são redirecionadas para o nó de origem. Uma desvantagem dessa prática consiste no aumento

da sobrecarga para o gerenciamento dos processos migrados e da rede de comunicação. Visando reduzir essa sobrecarga, o Mosix implementa um mecanismo de *sockets* migráveis o qual permite aos processos trocar mensagens através de uma comunicação direta, sem a necessidade de passar por seus respectivos nós de origem. Nos *sockets* migráveis, cada processo tem uma caixa postal para onde outros processos podem enviar mensagens. Dessa maneira, os processos remetentes não necessitam conhecer os nós onde os processos destinos estão sendo executados, somente identificá-los por seus *pids* e UHNs.

Após a migração de processos, o Mosix garante que as chamadas ao sistema que não são direcionadas ao nó origem não são capazes de modificar ou acessar recursos do nó receptor. Aos processos migrados só é permitido o acesso à CPU e à memória dos nós para que a segurança do ambiente seja preservada.

8.1.4 Suporte a Esquemas de Prioridade de Processos

O esquema de prioridades permite que processos de mais alta prioridade podem sempre ser migrados para uma máquina mesmo que para isso seja preciso transferir um processo de menor prioridade sendo executado localmente para outro destino. Por padrão, processos vindos de outro nó devem sempre ser transferidos na chegada de processos criados pelo doador de recursos local ou de processos de mais alto privilégio.

Administradores de *clusters* podem especificar prioridades a processos habilitados a serem executados em seus domínios. Dessa maneira, dois ou mais *clusters* podem ser compartilhados de maneira simétrica ou assimétrica, dependendo se ambos oferecem as mesmas ou diferentes prioridades a processos um do outro.

8.1.5 Controle de fluxo

Quando um usuário cria um grande número de processos ou quando um grande número de processos migram de volta a um *cluster*, pode ocorrer uma sobrecarga de trabalho. Para evitar a sobrecarga, cada nó pode configurar um limite no número de processos locais de certas classes. Quando esse limite é alcançado, outros processos dessa classe são automaticamente congelados e têm suas imagens de memória armazenadas em arquivos.

Processos congelados são reativados de modo circular de maneira a permitir a execução de todos os processos sem que a máquina local seja sobrecarregada. Quando mais recursos se tornam disponíveis no *cluster*, o algoritmo de balanceamento de carga pode migrar processos em execução para outros destinos.

8.1.6 Suporte a Desconexão de Nós

No Mosix, após um pedido de desconexão de um nó, todos os processos submetidos a partir de outra máquina sendo executados nele são migrados e todos os processos locais que foram migrados para outra máquina são transferidos de volta.

O suporte a desconexão de nós, a migração de processos e o controle de fluxos favorecem a execução de processos longos que podem ser executados remotamente, interrompidos quando há limitação de recursos e reiniciados quando os recursos voltarem a ficar disponíveis.

8.1.7 Suporte a Checkpoint

Quando um processo sofre *checkpoint*, sua imagem é salva em um arquivo. Mais tarde, esse arquivo pode ser lido e o processo reiniciado com o estado igual ao que ele mantinha quando foi interrompido. *Checkpoints* podem ser disparados manualmente, por temporizadores ou pelo programa em si.

Alguns tipos de programas não são capazes de sofrer *checkpoint*. Por motivos de segurança, esse é o caso de processos com privilégios de administrador do tipo *setuid*. Outros programas não são capazes de executar corretamente após sofrer um *checkpoint* como, por exemplo, processos que fazem extenso uso de *pipes* e *sockets* abertos. Nesse último tipo de programa estão também aqueles que utilizam identificadores de processos em seus códigos.

8.1.8 Execução de Aplicações em Lote

O Mosix é capaz de executar aplicações em lote de duas maneiras: enfileirando a aplicação até que recursos estejam disponíveis e selecionando a melhor destino inicial para a aplicação. Em ambos os casos, as aplicações são iniciadas em nós remotos a partir de binários e preservam somente algumas características do ambiente que invocou a execução. São mantidas, por exemplo, as variáveis de ambiente, as habilidades de leitura da saída padrão e escrita na saída padrão e de erros e a capacidade de receber sinais, mas não de enviá-los.

Aplicações em lote apresentam a vantagem de poupar tempo ao não precisar referenciar ao nó despachante para a realização de chamadas ao sistema. Dessa maneira, esse tipo de aplicação é recomendada para programas que realizam uma quantidade significativa de entrada/saída de dados.

9 Conclusão

Dado o objetivo de Sistemas Computacionais Distribuídos de compartilhar recursos, o papel de gerenciadores de recursos nesses sistemas é de fundamental importância. Ao gerenciar recursos, é preciso aplicar regras para a melhor utilização dos mesmos. Nesse aspecto, diferentes objetivos podem ser desejados e cabe ao escalonador executar políticas que aloquem de forma mais adequada os recursos aos processos em espera por execução.

Neste trabalho, quatro diferentes gerenciadores de recursos foram descritos com ênfase nas políticas de escalonamentos implementadas nesses sistemas. No primeiro deles, o Condor, as aplicações são submetidas e têm as suas requisições casadas com os recursos disponíveis no sistema. A prioridade de casamento é estabelecida pelo administrador ao submeter a aplicação por meio do atributo *rank* presente no *ClassAd*. Com seis diferentes ambientes de execução, o Condor provê no universo *Standard* o seu mais completo sistema de gerenciamento com mecanismos de *checkpoint* e migração de processos. Nesse modo de execução porém, não há o tratamento de canais de comunicação entre processos tais como *pipes* e *sockets* o que inviabiliza a execução de aplicações que realizam comunicação entre processos.

De maneira diferente, o Mosix implementa um mecanismo de *sockets migráveis* que permite a um processo ser migrado e ainda assim utilizar os canais de comunicação anteriormente abertos, inclusive podendo realizar chamadas ao sistema que são, na verdade, invocadas nos nós de origem das aplicações, ou seja, onde elas foram submetidas em primeira

instância. Essa característica do Mosix, apesar de gerar uma sobrecarga para o gerenciamento da migração, o faz ideal para gerenciar aplicações de longa execução em grades organizacionais, onde os recursos são freqüentemente indisponibilizados.

No Mosix, as políticas de balanceamento de carga podem ser realizadas dinamicamente de forma automática (se uma máquina passou a ter grande taxa de ocupação, os processos sendo executados nela são migrados para nós menos sobrecarregados) ao contrário do que ocorre no PBS onde o balanceamento é, *a priori*, objetivado somente no momento de submissão de um comando. É no Mosix também onde há a política de atribuição de recursos mais complexa, baseada em um histórico de execuções. Nos demais sistemas, as políticas de atribuição inicial consistem basicamente de FIFOs. Porém, em todos os sistemas de gerenciamento estudados, há a possibilidade de extensão das políticas de escalonamento, sendo que no PBS existem três opções de linguagem de programação para a implementação dos algoritmos de escalonamento.

Apesar do Mosix apresentar avançados algoritmos para o gerenciamento de recursos, ele foi o único sistema para o qual não se encontrou registro de provisão de qualidade de serviço. No momento da submissão de um processo no Mosix, os algoritmos de escalonamento tentam atribuir o processo ao melhor nó em um modo de melhor esforço, onde as especificações das requisições para execução do processo são estimadas a partir de um histórico de execuções anteriores.

Os sistemas descritos, com exceção do PBS o qual é basicamente voltado para infraestrutura de um único *cluster*, apresentam grande escalabilidade com capacidade de administrar sistemas de grandes dimensões tais como uma grade computacional. Destaque especial deve ser dado ao OAR que com uma iniciativa ousada de criar um gerenciador de recursos a partir de componentes de alto nível, conseguiu atingir um bom desempenho com a utilização de um banco de dados relacional e uma linguagem de *script*.

A reserva antecipada de recursos presente no Maui e no OAR permitem a implementação de políticas de *backfilling* as quais tendem a prover maiores taxas de ocupação dos recursos. A reserva desses sistemas é baseada em fatias de tempo dedicadas a determinada aplicação. Não é possível, por exemplo, fazer uma reserva proporcional de recurso do tipo, a cada 100 ms, reserve 50 ms do recurso A à aplicação X, funcionalidade bastante desejada de se ter em ambientes compartilhados.

De maneira geral, pelas características levantadas de cada sistema estudado, concluise que há uma convergência das funcionalidades básicas providas pelos gerenciadores de recurso, apesar de essas, muitas vezes, serem alcançadas por meio de implementações e técnicas diferentes. O que se nota é que os mecanismos para o gerenciamento de recursos em ambientes distribuídos existem, mas, muitas vezes, carecem de políticas de escalonamento mais objetivas. Essa carência é justificada pelo fato de que cada tipo de aplicação apresenta diferentes comportamentos e, assim, cada tipo de aplicação deve ter uma política de escalonamento voltada a satisfazer, da melhor forma possível, as suas necessidades. Nesse aspecto, é de fundamental importância que um gerenciador de recursos apresente facilidades para a inserção de novas políticas de escalonamento, característica encontrada de maneira mais consistente no PBS.

Referências

[1] **Condor project homepage**, August 2008. Disponível em

http://www.cs.wisc.edu/condor. Acesso em setembro de 2008.

[2] **Pbs gridworks: Openpbs**, September 2008. Disponível em http://www.openpbs.org. Acesso em setembro de 2008.

[3] **Tivoli workload scheduler loadleveler**. Disponível em http://www.ibm.com/systems/clusters/software/loadleveler.html. Acesso em setembro de 2008.

[4] **Platform lsf family – platform computing**, August 2008. Disponível em http://www.platform.com/Products/Platform.LSF.Family. Acesso em setembro de 2008.

[5] **Gridengine: Home**, August 2008. Disponível em http://gridengine.sunsource.net. Acesso em setembro de 2008.

[6] BAKER, M.; FOX, G. ; YAU, H.. **Cluster computing review**. Technical report, Center for Research on Parallel Computation - Rice University, November 1995.

[7] A., K. J.; L., N. M.. **A comparison of queueing, cluster and distributed computing systems**. Technical report, NASA Langley Research Center, 1994.

[8] CHAPMAN, C.; WILSON, P.; TANNENBAUM, T.; FARRELLEE, M.; LIVNY, M.; BRODHOLT, J. ; EMMERICH, W.. **Condor services for the global grid: interoperability between Condor and OGSA**. In: PROCEEDINGS OF 2004 UK E-SCIENCE ALL HANDS MEETING, p. 870–877, Nottingham, UK, August 2004.

[9] FREY, J.; TANNENBAUM, T.; FOSTER, I.; LIVNY, M. ; TUECKE, S.. **Condor-G: A computation management agent for multi-institutional grids**. Cluster Computing, 5:237–246, 2002.

[10] BARAK, A.; SHILOH, A.. **The mosix2 management system for linux clusters and organizational grids**. Technical report, Department of Computer Science - The Hebrew University of Jerusalem, August 2007. Disponível em http://www.mosix.org/pub/MOSIX2_wp.pdf. Acesso em setembro de 2008.

[11] KRAUTER, K.; BUYYA, R. ; MAHESWARAN, M.. **A taxonomy and survey of grid resource management systems for distributed computing**. Software Practice and Experience, 32(2):135–164, 2002.

[12] SCHOPF, J. M.. **Grid resource management: state of the art and future trends**, chapter Ten actions when Grid scheduling: the user as a Grid scheduler, p. 15–23. Kluwer Academic Publishers, Norwell, MA, USA, 2004.

[13] TANNENBAUM, T.; WRIGHT, D.; MILLER, K. ; LIVNY, M.. **Beowulf cluster computing with Linux**, chapter Condor: a distributed job scheduler, p. 307–350. MIT Press, Cambridge, MA, USA, 2202.

[14] THAIN, D.; TANNENBAUM, T. ; LIVNY, M.. **Distributed computing in practice: the condor experience.** Concurrency - Practice and Experience, 17(2-4):323–356, 2005.

[15] **The globus alliance**, aug 2008. Disponível em http://www.globus.org. Acesso em setembro de 2008.

[16] BAYUCAN, A.; HENDERSON, R.; JASINSKYJ, L.; LESIAK, C.; MANN, B.; PROETT, T. ; TWETEN, D.. **Portable Batch System Administrator Guide**. Numerical Aerospace Simulation System Division - NASA Ames Research Center, August 1998. Disponível em http://www.clusterresources.com/products/torque/docs/admin_guide.ps. Acesso em setembro de 2008.

[17] JONES, J. P.. **Pbs pro external reference specification 5.3**. Technical report, Altair Grid Technologies, March 2003. Disponível em http://www.mta.ca/torch/pdf/pbspro54/pbsproers.pdf. Acesso em setembro de 2008.

[18] BODE, B.; HALSTEAD, D. M.; KENDALL, R.; LEI, Z. ; JACKSON, D.. **The portable batch scheduler and the maui scheduler on linux clusters**. In: ALS'00: PROCEEDINGS OF THE 4TH CONFERENCE ON 4TH ANNUAL LINUX SHOWCASE & CONFERENCE, ATLANTA, p. 27–27, Berkeley, CA, USA, 2000. USENIX Association.

[19] **Torque resource manager**. Disponível em http://www.clusterresources.com/pages/products/torque-resource-manager.php. Acesso em setembro de 2008.

[20] STAPLES, G.. **Torque resource manager**. In: SC '06: PROCEEDINGS OF THE 2006 ACM/IEEE CONFERENCE ON SUPERCOMPUTING, p. 8, New York, NY, USA, 2006. ACM.

[21] JACKSON, D.. **Torque resource manager & moabr-new capabilities and roadmap forum by cluster resources, inc.** In: SC '07: PROCEEDINGS OF THE 2006 ACM/IEEE CONFERENCE ON SUPERCOMPUTING, New York, NY, USA, 2007. ACM.

[22] ROSMANITH, H.; PRAXMARER, P.; KRANZLMÜLLER, D. ; VOLKERT1, J.. **High Performance Computing and Communications**, volumen 4208/2006 de **Lecture Notes in Computer Science**, chapter Towards Job Accounting in Existing Resource Schedulers: Weaknesses and Improvements, p. 719–726. Springer Berlin / Heidelberg, September 2006.

[23] **Maui cluster scheduler**. Disponível em http://www.clusterresources.com/pages/products/maui-cluster-scheduler.php. Acesso em setembro de 2008.

[24] JACKSON, D. B.; SNELL, Q. ; CLEMENT, M. J.. **Core algorithms of the maui scheduler**. In: JSSPP '01: REVISED PAPERS FROM THE 7TH INTERNATIONAL WORKSHOP ON JOB SCHEDULING STRATEGIES FOR PARALLEL PROCESSING, p. 87–102, London, UK, 2001. Springer-Verlag.

[25] CAPIT, N.; COSTA, G. D.; GEORGIOU, Y.; HUARD, G.; MARTIN, C.; MOUNIE, G.; NEYRON, P. ; RICHARD, O.. **A batch scheduler with high level components**. In: CCGRID '05: PROCEEDINGS OF THE FIFTH IEEE INTERNATIONAL SYMPOSIUM ON CLUSTER COMPUTING AND THE GRID (CCGRID'05) - VOLUME 2, p. 776–783, Washington, DC, USA, 2005. IEEE Computer Society.

[26] **Oar**, July 2008. Disponível em http://oar.imag.fr. Acesso em setembro de 2008.

[27] **Taktuk**, November 2007. Disponível em http://taktuk.gforge.inria.fr. Acesso em setembro de 2008.

[28] BARAK, A.; LA'ADAN, O. ; SHILOH, A.. **Scalable cluster computing with mosix for linux**. In: PROC. LINUX EXPO '99, p. 95–100, Raleigh, N.C., May 1999.

[29] AMIR, Y.; AWERBUCH, B.; BARAK, A.; BORGSTROM, R. S. ; KEREN, A.. **An opportunity cost approach for job assignment in a scalable computing cluster**. IEEE Trans. Parallel Distrib. Syst., 11(7):760–768, 2000.

[30] AMAR, L.; BARAK, A.; LEVY, E. ; OKUN, M.. **An on-line algorithm for fair-share node allocations in a cluster**. In: CCGRID '07: PROCEEDINGS OF THE SEVENTH IEEE INTERNATIONAL SYMPOSIUM ON CLUSTER COMPUTING AND THE GRID, p. 83–91, Washington, DC, USA, 2007. IEEE Computer Society.

[31] **Mosix home**, 2008. Disponível em http://www.mosix.org. Acesso em setembro de 2008.