

Uma Arquitetura XML para Computação Colaborativa P2P

Felipe C. Pereira¹, Fábio C. Lourenço¹, Éber A. Schmitz¹, Felipe M. G. França²
NCE/IM – Pós-Graduação em Ciência da Computação¹ e PESC – COPPE²

Universidade Federal do Rio de Janeiro

felipep@ufrj.br, fabiocl@ufrj.br, eber@nce.ufrj.br, felipe@cos.ufrj.br

Resumo

Este artigo introduz a IeC (Infraestrutura Colaborativa): uma arquitetura para computação colaborativa peer-to-peer — P2P — implementada sobre XML, que é de fácil implantação por usuários não-especialistas e de simples utilização por desenvolvedores de aplicações para ambientes colaborativos. O uso de um mecanismo simples de escalonamento distribuído para o balanceamento de carga nos nós computacionais participantes é avaliado em termos da escalabilidade da arquitetura proposta e da qualidade do balanceamento. Tal foi obtido através de simulações usando-se como benchmark um problema do tipo RCPS — Resource Constrained Project Scheduling.

1. Introdução

Novas tecnologias possibilitam o agrupamento de uma variada quantidade de recursos computacionais distribuídos geograficamente, tais como supercomputadores, sistemas de armazenamento, fontes de dados, dispositivos especiais e serviços, que podem ser utilizados como um único recurso [1]. Esse novo paradigma está sendo chamado de *grid computing* [2] [3] [4]. O *grid* tem a principal função de coordenar esse conjunto virtual de recursos em larga escala de forma segura e eficiente [3].

Luis Sarmenta introduziu, em sua tese de doutorado [5], a idéia de *computação voluntária*, também chamada de *computação colaborativa*, onde também se permite a construção de plataformas paralelas de alta performance de uma forma simples, rápida e barata. Tanto *grid computing*, como *computação colaborativa*, são formas de *metacomputação*, que procuram tornar fácil a utilização de diversos recursos computacionais espalhados geograficamente. Entretanto, configurar um *grid* não é uma coisa fácil para não-especialistas.

Um princípio básico da computação colaborativa é permitir que qualquer internauta, sem experiência no assunto, consiga realizar a configuração para participar de uma rede colaborativa. A plataforma de computação colaborativa proposta por Sarmenta foi desenvolvida em Java. Também no caso da computação colaborativa, estações pessoais de trabalho, ociosas na maior parte do tempo, podem ser utilizadas para a realização de tarefas complexas que, de outra forma, requereriam máquinas de grande poder computacional e, portanto, de elevado custo.

Arquiteturas de *grid* e de computação colaborativa tais como SETI@Home [6], *Bayanihan Web-based Volunteer Computing System* [5] [7] e *XtremWeb* [8], adotam o estilo de controle centralizado. Em tal abordagem, uma única máquina, ou um conjunto designado, é responsável por fazer a distribuição das tarefas e a coleta de resultados no sistema. Se por um lado tal abordagem tem a qualidade de ser simples, por outro lado, pode não prover a escalabilidade desejada para usufruir plenamente da natureza dos sistemas colaborativos, que podem ter o número de máquinas que integram o sistema reduzido ou ampliado a qualquer momento. Além disso, a diferenciação de classes entre as máquinas participantes torna maior a complexidade do setup.

A computação *peer-to-peer* — P2P — pode ser definida como uma classe de aplicações que, baseadas em um controle descentralizado, i.e., distribuído entre todos as máquinas participantes, usufruem de recursos de armazenamento, processamento, conteúdo, e presença humana disponíveis nas fronteiras da Internet [9]. Segundo o estudo de taxonomia de Krauter [10], P2P pode ser classificada como uma forma de organização plana das máquinas, em que todas se comunicam com qualquer outra (*overlay*) sem a necessidade de uma máquina servidor intermediária. Aplicações como o *Gnutella* [11] popularizaram o P2P na WWW onde, dado que operar em um ambiente de conectividade instável e endereços IPs

imprevisíveis, projetos P2P são, geralmente, independentes de DNS e de servidores centrais.

Este artigo introduz a **IeC** — Infraestrutura Colaborativa — uma arquitetura colaborativa com uma organização descentralizada tipo P2P [9] [11] [12]. Com a adoção de XML [13] para o formato das mensagens trocadas entre os *peers*, tal arquitetura possui o objetivo de facilitar a criação de aplicações colaborativas, possibilitando que os desenvolvedores destas aplicações concentrem-se nas particularidades de seus aplicativos, deixando com a **IeC** os serviços de troca de mensagens entre os *peers*. A escolha de XML como padrão para a formatação das mensagens deve-se ao fato deste ser um padrão de mercado, que permite interoperabilidade entre diferentes plataformas.

O restante do texto está organizado da seguinte forma. Na próxima seção é introduzida a proposta de arquitetura colaborativa incluindo detalhes de seus principais aspectos. A Seção 3 apresenta a definição dos experimentos idealizados para uma avaliação preliminar da **IeC**, apresentando e discutindo seus resultados. Nossas conclusões são apresentadas na Seção 4.

2. IeC: uma infraestrutura colaborativa

A arquitetura proposta aqui provê um ambiente colaborativo transparente para as aplicações que o utilizam, ou seja, uma aplicação construída para funcionar nesta arquitetura não precisa saber dos detalhes de como, ou por quem, uma requisição de trabalho computacional gerada será atendida e nem como a resposta será entregue, cabendo tal à arquitetura **IeC**. Manter a carga balanceada entre os nós presentes, garantir a capacidade da rede se recuperar a falhas de nós, se adaptando a novas topologias, também são atribuições da **IeC**.

A Figura 1 mostra os fluxos de mensagens que circulam no sistema colaborativo. Uma aplicação produtora envia pacotes de requisição para o módulo da infra-estrutura e recebe pacotes de resposta da infra-estrutura, a qual envia pacotes de comunicado para outros nós da rede. Aplicações consumidoras recebem pacotes de requisições da arquitetura **IeC** e enviam os pacotes de resposta de volta.

Na construção da arquitetura **IeC**, os seguintes objetivos foram perseguidos:

- Facilidade de integração com aplicações colaborativas;
- Facilidade de configuração;
- Possibilidade de interoperabilidade entre diversas plataformas;

- Possibilidade de ser estendida.

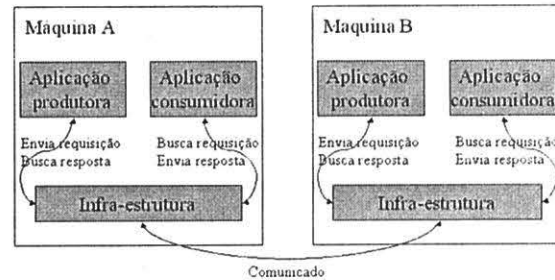


Figura 1. Relacionamento das aplicações com a infraestrutura IeC e entre os nós computacionais.

2.1. API

Para a arquitetura **IeC** operar, cada nó computacional que participa do sistema colaborativo deve instalar um módulo do sistema, que fornece uma API padronizada para as aplicações produtoras e consumidoras de trabalhos. Este módulo é responsável por:

- escalonar as requisições recebidas de aplicações produtoras entre os diversos nós que possuem aplicações registradas para consumi-las;
- receber as respostas das requisições feitas por aplicações registradas no nó, entregando-as para estas;
- fornecer informação sobre a distribuição de carga para as aplicações, permitindo que elas controlem o tamanho do grão (Seção 2.7);
- re-enviar requisições não atendidas no tempo máximo configurado (*time-out*).

A API foi escrita no modelo COM da Microsoft, que é um modelo de componentes padrão para o Windows, possibilitando que aplicações desenvolvidas, em praticamente todas as linguagens disponíveis para este ambiente, possam entrar em contato com o sistema. O fato de a arquitetura utilizar mensagens em XML e trocá-las via comunicação TCP/IP via sockets, possibilita que a própria infra-estrutura seja re-escrita para outras plataformas, permitindo inclusive que estratégias diferentes de escalonamento, descoberta e disseminação de recursos e controle de carga, sejam utilizadas na mesma rede.

Os seguintes métodos são disponibilizados pela API do sistema (a Figura 2 ilustra uma dinâmica desses métodos):

- **Registro** (*DoRegister*) – Através desse método, uma aplicação se habilita a utilizar os serviços oferecidos. Caso forneça algum serviço, o tipo de requisição que ela atende é indicado. A aplicação

recebe um identificador de aplicação (*AppID*) que deverá ser utilizado para identificá-la no relacionamento com a **IeC**;

- **Requisitar serviço (GetRequest)** – Este método permite a uma aplicação produtora (ou cliente) requisitar um trabalho. A aplicação recebe um identificador para o trabalho solicitado (*ReqID*);
- **Buscar resposta (GetResponse)** – Esse método permite que uma aplicação verifique se alguma requisição feita anteriormente foi respondida, sendo entregue caso exista. O *ReqID* identifica a requisição que foi respondida;
- **Buscar requisição de serviço (SendRequest)** – Através deste método, uma aplicação consumidora (ou servidora) busca uma requisição para resolver. O retorno será vazio se não existir uma requisição pendente;
- **Enviar resposta (SendResponse)** – Uma aplicação servidora, utiliza este método para enviar a resposta para um trabalho feito, atendendo a uma requisição recebida;
- **Buscar distribuição (GetDistribution)** – Verifica como está a distribuição das requisições entre o conjunto de máquinas que fornece um serviço.

2.2. Componentes da arquitetura IeC

O módulo da **IeC** foi construído com uma arquitetura de componentes, sendo cada um responsável por atender uma funcionalidade específica. Assim, podemos dividir o módulo nos seguintes componentes principais:

- **Gerente de resposta** – É o guardião das respostas recebidas para as solicitações feitas. As respostas vão sendo removidas à medida que as aplicações que fizeram as requisições respondidas as buscam;
- **Gerente de requisição** – Mantém uma fila para cada tipo de requisição recebida, sendo a fonte de requisições para as aplicações consumidoras;
- **Gerente de envio** – Mantém o controle das requisições enviadas pelas aplicações produtoras registradas. Para cada requisição pendente, são mantidos os IPs das máquinas que a receberam, e o horário do último envio. Este módulo informa para o escalonador a necessidade de envio redundante da requisição, conforme a política configurada;
- **Escalonador** – Componente chave do sistema. Este módulo é responsável por distribuir as requisições recebidas entre os nós computacionais. Cabe a ele decidir qual nó

existente é o mais adequado para receber uma requisição, a fim de manter o sistema com carga equilibrada;

- **Gerente de conectividade** – Sua função é manter o nó conectado ao sistema da melhor forma para este. Esse componente é responsável em descobrir máquinas no sistema e iniciar a comunicação com estas, fornecendo para o escalonador a lista das máquinas conhecidas;
- **Receptor de comunicados** – Recebe os comunicados dos diversos nós, sendo responsável por distribuir as informações aos outros componentes do sistema. Respostas são repassadas ao gerente de resposta, dando baixa no gerente de envio (fim da pendência da requisição); requisições são repassadas para o gerente de requisição, informações de carga são repassadas ao escalonador.

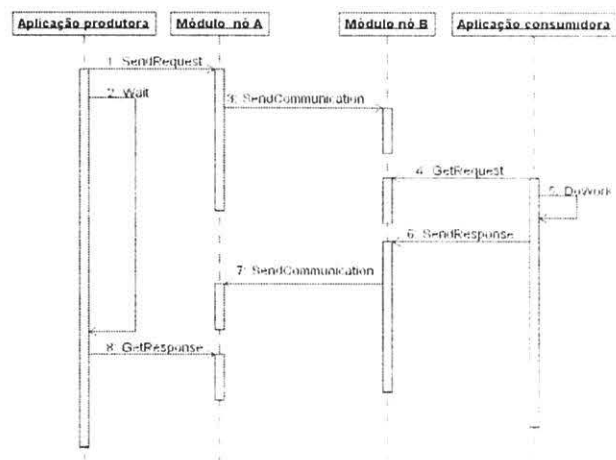


Figura 2. Ciclo de vida de uma requisição.

2.3. Pacotes de informação

Os tipos de requisição são nomeados de forma única, permitindo que diversas aplicações possam conviver juntas na rede sem que haja conflito. Para cada requisição feita, é gerado um pacote XML contendo o descritor do solicitante e o pacote da aplicação. O descritor contém o IP do nó e o *AppID* da aplicação que gerou a requisição. O pacote da aplicação é um nó do pacote XML que recebe o nome do tipo da requisição. Este nó XML pode conter diversos sub-nós, compostos ou não, de acordo com a necessidade da aplicação representar os dados da requisição. Da mesma forma, uma resposta a uma requisição é representada por um pacote XML contendo o destino da resposta, ou seja, IP do nó e

AppID, e um nó XML com o nome do tipo da requisição, organizado da maneira que a aplicação julgar necessário.

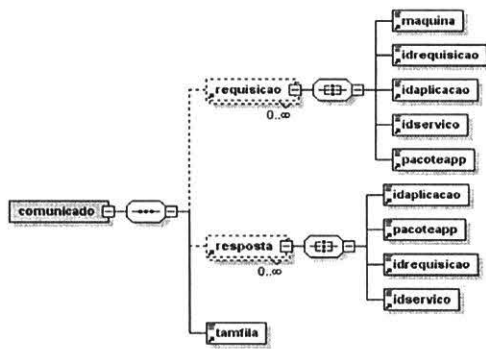


Figura 3. Estrutura de um pacote de comunicado (XML).

A troca de mensagens entre os diversos nós presentes na rede é feita por pacotes maiores, chamados de *pacotes de comunicado*, os quais agregam diversos pacotes de respostas e requisições, além de dados sobre a situação no nó, que serão utilizados para a gerência da rede. A Figura 3 mostra a estrutura de um pacote de comunicado. A estrutura dos pacotes são descritas e validadas através de *XML Schemas* [14].

2.4. Escalonamento

O escalonamento adotado ocorre de forma totalmente descentralizada, ou seja, cada nó toma sua própria decisão sem apoio de um servidor central que possua uma visão completa da rede. Entretanto, ainda é desejado que uma requisição deva ser enviada para um nó que seja a melhor alternativa para o desempenho global do sistema.

Neste ambiente, a estratégia de escalonamento busca uma computação de alta vazão [10], o que significa a realização da maior quantidade de trabalhos possíveis em um determinado intervalo de tempo, visando maximizar o desempenho do sistema como um todo e não de uma aplicação específica.

A abordagem de escalonamento desejada deve atender os requisitos acima, porém, sem introduzir *overheads* e retardos significativos no sistema.

Uma primeira estratégia utilizada em nossos testes foi uma abordagem *round-robin*, na qual cada nó envia, para os nós conhecidos que possuem consumidores para o tipo de requisição, suas requisições, escolhendo o nó destino a partir de uma lista circular. Tal abordagem poderia funcionar adequadamente em um ambiente controlado, em que as máquinas, com uma mesma capacidade de processamento, estariam sendo utilizadas

exclusivamente para o ambiente colaborativo. Entretanto, em ambientes reais, esta abordagem pode sobrecarregar de requisições máquinas lentas ou não ociosas, deixando máquinas mais poderosas desocupadas. Até mesmo em um ambiente controlado, conforme mencionado, estes problemas podem ocorrer, visto que se as requisições possuem tamanho variado, é possível que algumas máquinas fiquem sobrecarregadas com requisições mais demoradas.

A solução adotada para o componente escalonador, foi a de cada nó manter a informação do tamanho da fila de requisições, mantida pelo componente “Gerente de Requisição” de cada máquina conectada ao nó em questão. Assim, a máquina escolhida para enviar uma requisição é sempre a com menor fila que possua uma aplicação consumidora registrada para a requisição. Para manter a informação das filas das máquinas, incluímos o dado tamanho de fila no pacote de comunicado trocado em todas as comunicações (nó **tamfila** da estrutura mostrada na Figura 3). O *overhead* de comunicação é aceitável dado o pequeno aumento induzido no tamanho do pacote.

Apesar do grande dinamismo do sistema, em que diversos nós fazem requisições simultâneas, essa abordagem adapta-se às mudanças de forma rápida a diversas situações, tais como:

- casos em que a visão de um nó sobre outro está defasada em relação ao estado atual. Exemplo: dado uma situação em que o nó A possui a informação que o nó mais desocupado é o nó D; porém, o nó B fez diversas solicitações, tornando o nó D carregado com requisições. Na primeira requisição do nó A para o nó D, este responde com o tamanho de sua fila, permitindo que o nó A reavalie se o nó D continua, ou não, sendo a melhor opção, permitindo uma rápida adaptação a esta nova situação;
- existência de máquinas mais lentas. Neste caso, todas as máquinas do sistema começam a receber requisições na mesma velocidade que outras mais rápidas. Como as mais rápidas consomem suas requisições de forma mais acelerada, as filas das mais lentas ficam maiores, e o sistema direciona os novos pacotes somente para as máquinas mais rápidas, até que as filas voltem ao equilíbrio.

2.5. Redundância

A política de envio de requisições pode ser configurada para gerar redundância de pacotes. Isto é feito por dois principais motivos: (i) tornar o ambiente tolerante a falhas; (ii) aumentar a eficiência do sistema

quando existem máquinas ociosas. Como a arquitetura **IeC** é descentralizada, cada nó pode escolher utilizar a configuração mais adequada às necessidades das suas aplicações registradas.

Para atender o primeiro motivo apresentado, o sistema permite a configuração de um *time-out* para as requisições enviadas, logo, caso o nó que recebeu uma requisição não responda no tempo programado pelo nó que a enviou, uma nova é gerada para um outro nó. O componente “Gerente de Envio” é o responsável por avisar o escalonador que o pacote deve ser re-enviado, indicando as máquinas que não devem recebê-lo. Para o segundo, foi adotada uma política de “acionamento” para requisições, que visa melhorar o desempenho em aplicações que despejam um lote de requisições iniciais e depois ficam aguardando as diversas respostas. Aplicações no estilo *Master-Worker* [1] [5] [15] podem se beneficiar desta política.

A política de “acionamento” consiste em definir o número máximo de requisições de um determinado tipo que uma máquina pode receber. Desta forma, se uma aplicação despejar um número de requisições maior que o número máximo definido para o tipo de requisição, multiplicado pelo número de máquinas, as requisições serão represadas no nó origem até que alguma máquina entregue alguma resposta. O objetivo é evitar uma má distribuição inicial devido, por exemplo, a uma máquina ser mais lenta que outras. Os pacotes vão sendo liberados conforme as máquinas vão atendendo os pacotes enviados anteriormente. A redundância é gerada quando o número máximo definido para o tipo de requisição, multiplicado pelo número de máquinas, é maior do que o número de pacotes. Neste caso, são enviadas requisições redundantes em um esquema de lista circular, tal como na abordagem de *eager scheduling* [7], até que todas as máquinas recebam a quota estipulada.

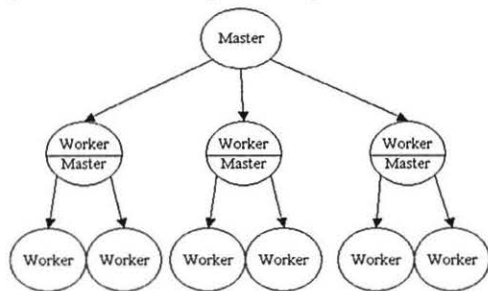


Figura 4. Master-Worker com hierarquia.

O uso desta política deve ser estudado cuidadosamente. O número máximo estipulado deve ser de um tamanho suficiente para que os nós consumidores não fiquem parados esperando por

pacotes, porém, quanto maior o tamanho, mais redundância e *overhead* serão gerados ao sistema. Outro problema é a utilização em aplicações que usem abordagens *Master-Worker* com hierarquia (Figura 4), em que os trabalhadores re-dividem o trabalho, tornando-se mestres. Nestas aplicações, é provável a ocorrência de *deadlocks*, pois, como existe dependência entre as requisições, se requisições que precisam ser resolvidas para liberar seus pais, na árvore de dependência, ficarem represadas no nó de origem, o sistema todo fica parado até que novos nós entrem no sistema, o que pode nunca acontecer.

2.6. Conectividade

O objetivo do sistema é que cada nó mantenha um conjunto de conexões suficientes para:

- garantir a resiliência do sistema, mantendo todos os nós do sistema conectados entre si diretamente, ou indiretamente, mesmo após a saída ou quedas de nós;
- garantir acesso aos nós que atendem as requisições solicitadas.

Na versão atual da arquitetura, o gerenciamento de conexões está estático, feito através de um arquivo XML, carregado por cada nó presente na rede. Tal arquivo contém a relação dos nós presentes e os tipos de requisições atendidas por cada nó.

A meta é que a arquitetura consiga resolver suas conexões de forma dinâmica, o que pode ser conseguido, por exemplo, através da estratégia utilizada pelo *Gnutella* [11]. Para um nó entrar na rede é necessário um ponto de partida, ou seja, o endereço de pelo menos um nó que faça parte da rede. Isto é feito mantendo-se uma lista dos nós que estão quase sempre conectados. Depois de conectado, o nó utiliza o mecanismo de PING e PONG para descobrir outros nós. Neste mecanismo, o nó que está se juntando a rede envia uma mensagem broadcast PING para anunciar sua presença. Os nós que recebem uma mensagem PING respondem com uma mensagem PONG, a qual contém informações sobre o nó.

2.7. Controle da granularidade

Um dos serviços fornecidos pela arquitetura às aplicações colaborativas é o fornecimento da situação da distribuição de requisições entre as máquinas que consomem um dado tipo de requisição. O objetivo desta comunicação, é permitir que as aplicações não fiquem totalmente cegas em relação ao ambiente em que elas estão trabalhando, permitindo-as tomar decisões que possam melhorar o desempenho delas. A informação passada para as aplicações é o nível da

dispersão. Para isso, utilizamos o *coeficiente de variação*, cálculo estatístico para medir dispersão relativa, definido como $\frac{\delta^2}{x}$, onde δ é o desvio padrão do tamanho das filas e x a média. A arquitetura converte o resultado em três valores discretos, conforme mostra a Tabela 1.

Tabela 1. Níveis de dispersão informados.

Dispersão calculada (x)	Nível informado
$0\% \leq x < 30\%$	0 – Baixa dispersão.
$30\% \leq x < 50\%$	1 – Média dispersão.
$x \geq 50\%$	2 – Alta dispersão.

Se o ambiente está com baixa dispersão, isto indica distribuição de carga está boa, logo, uma aplicação pode utilizar essa informação para aumentar o grão de suas requisições, porque não existe nó ocioso, e um grão maior significa menor *overhead* de comunicação. De forma contrária, uma dispersão maior indica que existem máquinas trabalhando mais do que outras. A aplicação pode optar em diminuir o grão nesta situação para aproveitar o processamento ocioso.

3. Avaliação da arquitetura

3.1. Base Experimental

Com fins a avaliar preliminarmente a arquitetura construída, foi implementado um algoritmo [16] para busca de soluções ótimas de problemas do tipo RCPS (*Resource-Constrained Project Scheduling*) [17] através de um esquema de enumeração implícita com *branch-and-bound* [18]. Nestes problemas, o objetivo é descobrir como escalonar as tarefas de um projeto tal que este tenha o menor tempo de duração possível. As tarefas podem possuir uma relação de precedência uma com as outras, e necessitam de recursos, os quais, em nosso experimento, são finitos, porém, renováveis.

O algoritmo consiste em enumerar todos as possibilidades de escalonamento e achar a melhor delas. Isto significa percorrer uma árvore para achar a folha que represente o melhor resultado. Como é possível avaliar que alguns ramos da árvore jamais serão a solução, estes são podados, diminuindo o espaço de busca. A escolha de problemas RCPS para avaliar a arquitetura *IeC*, como veremos adiante, se justifica pela dinâmica da carga imposta ao conjunto de máquinas participantes.

A estratégia *Master-Worker* (Figura 4) com hierarquia é utilizada para se distribuir os trabalhos entre os diversos *trabalhadores*. Cada máquina (nó),

recebe um ramo para resolver, descobre os ramos do próximo nível (sub-ramos) e, se estiver em um estado de produtor, solicita a outros trabalhadores para resolvê-los, ou, caso esteja no estado consumidor, adiciona a uma pilha interna para buscá-los posteriormente. Cada trabalhador responde para seu respectivo *mestre* o melhor escalonamento conseguido a partir da situação recebida. Isto é feito combinando-se o próprio trabalho com o melhor resultado informado por seus trabalhadores, se houver.

Dois estratégias de mudança entre os dois estados (consumidor e produtor) foram utilizadas. A primeira, estática, baseada num parâmetro inicial, e a segunda, automática, utilizando o conceito descrito na Seção 2.7. Na estratégia estática, todos os nós iniciam no estado produtor, permanecendo neste estado até que receba o número de pacotes configurado para o parâmetro *espalhamento*. Nesse estado, os nós exploram a árvore utilizando busca em largura, procurando abrir ramos longes da folha para tentar manter uma uniformidade no tamanho do grão. Após a fase de espalhamento, os nós passam ao estado de consumidor, utilizando uma pilha de tamanho conceitualmente infinito para o problema. Devido à utilização de pilha, a busca na árvore passa ser em profundidade, o que economiza recursos de memória dos nós. Na estratégia dinâmica, todos os nós iniciam com suas pilhas internas com valor igual a zero. A aplicação de cada nó pergunta ao *IeC* a situação da dispersão, aumentando o tamanho da pilha nas situações de baixa dispersão, e diminuindo-a nas situações de média e alta (com velocidade maior quando alta). Para evitar que a pilha alcance um tamanho maior do que a realidade necessária para o problema, a pilha cresce somente quando sua ocupação está igual, ou maior, do que noventa por cento. A mudança entre os estados consumidor e produtor acontece de acordo com a disponibilidade da pilha. Se a pilha interna não possuir capacidade para receber um sub-ramo, o nó será produtor, caso contrário, será consumidor.

Nossos experimentos estão baseados na medição dos tempos de solução de um projeto com 15 atividades. Os seguintes parâmetros serão medidos: (i) tempos de execução real e serial em função do número de máquinas utilizadas tanto para a estratégia dinâmica, como para a estática, sendo a última com o parâmetro de espalhamento fixo igual a 100; (ii) distribuição de carga nas máquinas durante o tempo de execução.

O tempo de execução *real* é o tempo do início da execução real do problema até o final, medido pela máquina que iniciou o problema. Dado que cada máquina i gasta um tempo x_i processando dados para

aplicação, o tempo serial é o somatório dos tempos x_i de todas as máquinas, ou seja, o tempo que seria gasto se nenhuma máquina trabalhasse em paralelo. O tempo ideal consiste na divisão do tempo serial pelo número de máquinas utilizadas, que seria o tempo decorrido se todas as máquinas permanecessem trabalhando ao mesmo tempo durante todo o experimento. O experimento foi realizado utilizando-se 10 máquinas AMD Athlon 1,1 Ghz, com 256 Mbytes de memória e conectadas em uma rede Ethernet a 10 Mbps.

3.2. Resultados

Tabela 2. Medições feitas para as duas estratégias, variando-se o número de máquinas.

# Máq.	Tempo Real (s) estática	Tempo Serial (s) estática	Aceleração estática	Tempo Real (s) dinâm.	Tempo Serial (s) dinâm.	Aceleração dinâm.
2	1745,31	2993,37	1,72	1493,36	2976,58	1,99
4	857,19	2983,63	3,48	774,57	3046,37	3,93
6	726,04	3158,81	4,35	541,56	3120,02	5,76
8	681,52	3035,58	4,45	451,88	3255,18	7,20
10	658,96	3083,66	4,68	430,74	3840,09	8,92

A Tabela 2 apresenta os tempos medidos e a Aceleração obtida em função destes para os diversos números de máquinas. As Figuras 5 e 7 apresentam de forma gráfica os resultados encontrados para as duas estratégias de espalhamento utilizadas, uma estática e outra dinâmica. Em tempo, na contabilidade do tempo serial, o tempo gasto em comunicação via TCP/IP não está incluído, somente os tempos de execução locais às máquinas participantes o são. No entanto, quando medido o tempo real, esses tempos de comunicação estão sendo contabilizados. Ou seja, as Figuras 5 e 7 são ligeiramente pessimistas com relação ao desempenho real da arquitetura IeC.

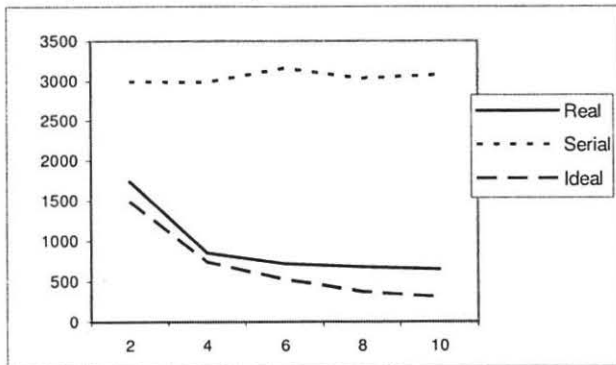


Figura 5. Gráfico do tempo x número de máquinas para a estratégia estática de espalhamento.

Os resultados apresentados nas Figuras 5 e 7 sugerem um resultado interessante quanto a

escalabilidade. Nas duas estratégias de espalhamento, o tempo medido tende a se aproximar do ideal comportamento, porém, a estratégia estática indica um crescente distanciamento de acordo com o aumento de número de máquinas, podendo-se observar na Tabela 2 uma queda na aceleração com relação ao número de máquinas. Já a estratégia dinâmica, visualizada na Figura 6, o distanciamento não ocorre, porém, aponta para um maior aumento do tempo serial. Este comportamento pode ser explicado porque o tempo serial contém além do processamento útil da aplicação, o tempo gasto para se empacotar e desempacotar pacotes para o IeC. No caso da estratégia dinâmica, quanto mais máquinas, maior será o número de pacotes gerados para manter o sistema estabilizado, justificando um aumento do tempo serial.

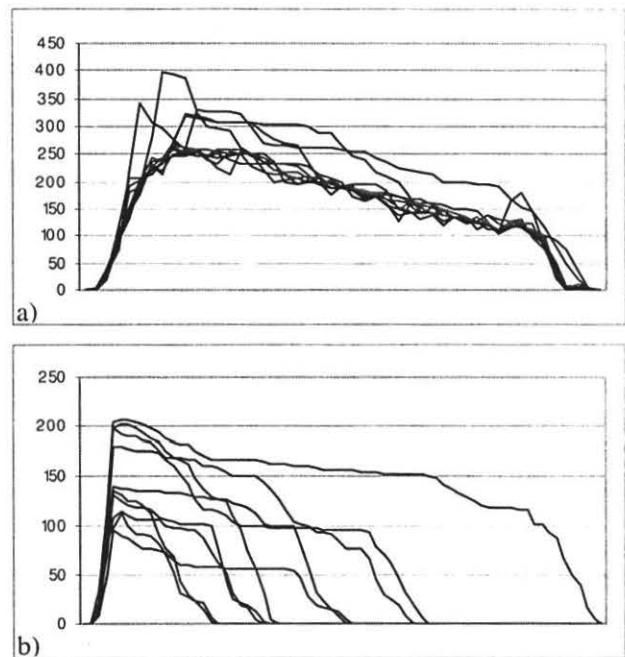


Figura 6. Distribuição de carga com 10 máquinas para as estratégias dinâmica (a) e estática (b).

A Figura 6 apresenta a distribuição de carga para o experimento com 10 máquinas, tanto para a estratégia dinâmica (a) e estática (b). Podemos observar que a estratégia dinâmica consegue manter todas as máquinas com carga durante toda a execução do experimento enquanto que, na estratégia estática, as máquinas recebem uma carga inicial que vai diminuindo com taxas diferenciadas, acarretando em ociosidade dos nós ao longo do tempo, chegando a ponto de somente existir uma máquina trabalhando. Este fato justifica a maior aceleração e escalabilidade da estratégia dinâmica em relação à estática, observado na Tabela 2 e nas Figuras 5 e 7.

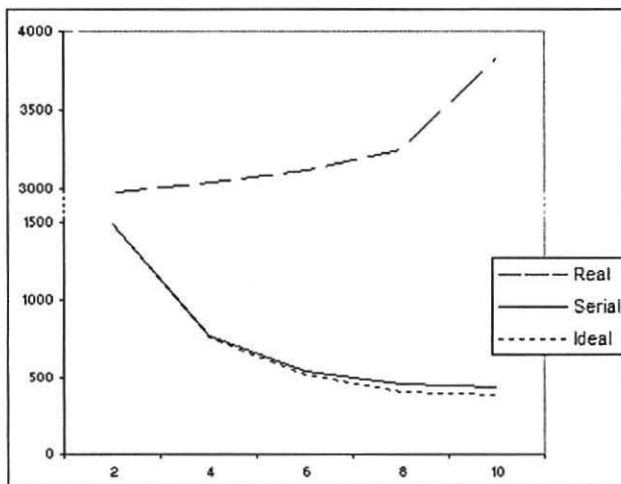


Figura 7. Gráfico do tempo x número de máquinas para a estratégia dinâmica de espalhamento.

4. Conclusão

Neste artigo propomos uma arquitetura XML colaborativa P2P, o IeC, que tem a ambição de tornar fácil a construção e a implantação de aplicações colaborativas P2P. Foi dada ênfase a um modelo dinâmico de escalonamento totalmente distribuído, e no controle da granularidade dos pacotes gerados com base na informação da carga no sistema. É importante notar que a resolução de problemas RCPS via *branch-and-bound* se encontraria como alvo de uma exploração de paralelismo com granularidade comparável às abordagens consolidadas por soluções do tipo *cluster computing*. Os resultados do experimento realizado indicam que a plataforma é escalável, e mostra o benefício que uma aplicação pode obter ao controlar o tamanho do grão de forma dinâmica.

Referências

[1] Chunlin, L., Layuan, L. – “Agent framework to support the computational grid” – *The Journal of Systems and Software*, 2002.

[2] Foster, I., Kesselman, C. - “The GRID: Blueprint for a Future Computing Infrastructure”. *Computational Grids*, Capítulo 2, 1998.

[3] N’emeth, Z., Sunderam, V. – “A Formal Framework for Defining Grid Systems” – *2nd IEEE/ACM International Symposium on Cluster Computing and the Grid (CCGRID’02)*, p. 202, 2002.

[4] Foster, I., Kesselman, C., Tuecke, S. – “The anatomy of the grid: Enabling scalable virtual organizations” – *Journal of Supercomputer Applications*, 15(3):200--222, 2001.

[5] Sarmenta, L. F. G. – *Volunteer Computing* – Tese de doutorado Massachusetts Institute of Technology, 2001.

[6] SETI@home, 2003, “Search for Extraterrestrial Intelligence at home”. Em: <http://setiathome.ssl.berkeley.edu/>, Acessado em 06/12/2003.

[7] Sarmenta, L. F. G., Hirano, S. – “Bayanihan: Building and studying web-based volunteer computing systems using Java” – *Future Generation Computer Systems*, 15(5-6):675--686, 1999. Special Issue on Metacomputing.

[8] Fedak, G., Germain C., Néri, V., Cappello, F. – “XtremWeb: A Generic Global Computing System” – *Workshop on Global Computing on Personal Devices (CCGrid2001)*, Berlim, Alemanha, IEEE Press, 2001.

[9] Foster, I., Iamnitchi, A. – “On Death, Taxes, and Convergence of Peer-to-Peer and Grid Computing” – *2nd International Workshop on Peer-to-Peer Systems (IPTPS’03)*, February 2003, Berkeley, CA.

[10] Krauter, K. – “A Taxonomy and Survey of Grid Resource Management Systems for Distributed Computing” – *Software - Practice and Experience*, 2001.

[11] Ripeanu, M. – “Peer-to-Peer Architecture Case Study: Gnutella Network” – *International Conference on Peer-to-peer Computing (P2P2001)*, Linkoping, Sweden, August 2001.

[12] Schintke, F., Schütt, T., Reinefeld, A. – “A Framework for Self-Optimizing Grids Using P2P Components” – *DEXA Workshops*, 2003: 689-693.

[13] The World Wide Web Consortium (W3C), “Extensible Markup Language (XML)”, Em: <http://www.w3c.org/XML/>, Acessado em 12/05/2004.

[14] The World Wide Web Consortium (W3C), “XML Schema”, Em: <http://www.w3c.org/XML/Schema>, Acessado em 12/05/2004.

[15] Heymann, E., Senar, M., Luque, E., Liviny, M. – “Adaptative Scheduling for Master-Worker Applications on the Computational Grid” – *Proceedings of the First International Workshop on Grid Computing (GRID 2000)*, 2000.

[16] Lourenço, F. C. - *Uma abordagem branch and bound para o RCPSP em um ambiente de computação colaborativa peer to peer* – Tese de mestrado NCE/IM – Pós-Graduação em Ciência da Computação, em andamento.

[17] W. Herroelen, E. Demeulemeester, B. De Reyck – “A classification scheme for project scheduling” – in J. Węglarz (Ed.), *Project Scheduling: Recent models, algorithms and applications*, Kluwer, Dordrecht, 1999.

[18] E.L. Demeulemeester, W.S. Herroelen, S. E. Elmaghraby – “Optimal procedures for the discrete time/cost trade-off problem in project networks” – *European Journal of Operational Research*, 88: 50-68, 1996.