MAC 5701 - Tópicos em Ciência da Computação
Uma arquitetura distribuída para o problema de roteamento em grafos dinâmicos

Marcelo Luís Vinagreiro
Orientador: Prof. Alfredo Goldman vel Lejbman
Responsável: Profa. Yoshiko Wakabayashi

25 de junho de 2002

1 Introdução

Um dos principais assuntos em Teoria de Grafos [3] é o problema da busca de caminhos entre dois ou mais pontos de um grafo. Uma aplicação prática desse problema está presente nos diversos sítios de busca de trajetos em cidades espalhados pela Internet, e que estão se tornando cada vez mais populares.

No entanto, as implementações correntes dos sistemas de identificação de rotas apresentam limitações, uma vez que as arquiteturas que as compõem são, em geral, estáticas. Dessa forma, quando um usuário solicita a melhor rota entre dois pontos de um grafo, esses sistemas, em sua maioria, não levam em consideração o estado atual do fluxo (trânsito) nas vias, considerando apenas fatores estáticos, como distância.

Para resolver este problema, existem vários trabalhos que estendem o modelo de grafos estáticos, de forma a permitir mudanças no grafo, ou apenas alterações de peso nos arcos/arestas[9]. Com esta flexibilidade adicional podem ser resolvidos problemas práticos interessantes como a busca da melhor rota, segundo critérios que eventualmente podem mudar durante o processamento, como por exemplo, a quantidade de fluxo.

Um outro aspecto que pode ser observado nos sistemas de roteamento em uso é que eles apresentam uma estrutura inerentemente centralizada. Por exemplo, os trabalhos desenvolvidos por Fawcett e Robinson [4] para o problema de roteamento adaptivo, apresentam uma característica dinâmica, porém é essencialmente centralizado, inviabilizando requisitos desejáveis a sistemas computacionais como escalabilidade, tolerância a falhas, etc.

A característica centralizada desses sistemas de roteamento pode ser explicada em parte, uma vez que as estruturas de dados envolvidas (geralmente são grafos descrevendo a região de monitoramento), apresentam um alto grau de inter-dependência, dificultando a descentralização.

Distribuir um grafo em diversas máquinas permite resolver o problema de escalabilidade. Entretanto, vários outros problemas têm de ser solucionados, dentre eles: como realizar o particionamento do grafo, qual algoritmo de roteamento deve ser utilizado, além de problemas inerentes a sistemas distribuídos.

Além de aplicações imediatas como a busca de caminhos em grandes grafos dinâmicos, o modelo proposto também pode ser usado em áreas emergentes da Computação, como a Computação Móvel [7], uma vez que em sistemas móveis, além da existência de uma rede fixa, também há comunicação via rádio entre as unidades móveis [7] e bases de transmissão. Neste caso, as unidades móveis poderiam descobrir a melhor rota de forma dinâmica.

Tendo em mente as considerações anteriormente expostas, deseja-se desenvolver nesta tese de Mestrado uma arquitetura a qual seja possível a obtenção de estimativas de caminhos mínimos [3] em um grafo particionado em um conjunto de subgrafos, sendo que as partições estão distribuídas em uma coleção de servidores que se comunicam através de uma rede fixa. Admite-se ainda que os dados mudam com o decorrer do processamento, supondo-se assim que o grafo é dinâmico, isto é, seu estado muda com o tempo.

Esse texto está organizado da seguinte forma: na próxima seção é apresentado o ambiente computacional para o qual uma arquitetura como a proposta aqui poderia ser utilizada. Em seguida são analisados os pré-requisitos levantados para a construção de uma arquitetura que suporte grafos dinâmicos em ambiente distribuído, nas seções seguintes serão apresentadas as técnicas estudadas/desenvolvidas para atender tais requisitos. Posteriormente, serão vistos os resultados obtidos até o momento bem como o estado atual da implementação e, finalmente, serão explicitadas algumas considerações finais.

1.1 Um possível cenário de aplicação da infra-estrutura estudada

Conforme foi dito anteriormente, o objetivo principal desse estudo é apresentar soluções eficientes que permitam manter um modelo de monitoramento das condições de algum sistema dinâmico, como por exemplo, as condições do trânsito de uma dada região, como a cidade de São Paulo. Um exemplo de sistemas dessa natureza pode ser encontrado no sítio do projeto SIDAM[1] (Sistemas de Informações Distribuídas para Agentes Móveis).

Normalmente em uma dada região, tal como uma cidade da proporção de São Paulo, torna-se quase inviável manter apenas um único servidor (também chamado aqui de estação-base) para atender todas as consultas/atualizações ao sistema, o que certamente degradaria a performance, dada a grande quantidade de requisições a que o sistema estaria sujeito.

Uma alternativa para isso seria utilizar uma estrutura distribuída de estações, cada qual cuidando de uma sub-região (subgrafo) do domínio de aplicação, permitindo-se assim um melhor balanceamento de carga do sistema bem como o estabelecimento de políticas para tolerância a falhas [10]. Com ela, poderia ser possível que clientes do sistema se conectassem à estação-base mais próxima ou a com menor carga, a fim de pedir informações como por exemplo, qual a melhor rota em termos de tempo do ponto a até o destino b.

2 Análise de Requisitos

Conforme foi explicado na seção anterior, o principal objetivo dessa tese de Mestrado é desenvolver uma estrutura distribuída que permita armazenar partições disjuntas de um grafo qualquer (subgrafos) em um grupo de servidores conectados via uma rede de comunicação. Além disso, considera-se que os dados presentes no grafo podem ser alterados durante o transcorrer

[1] http://www.ime.usp.br/~sidam/

do tempo, por exemplo, novos arcos podem ser adicionados ou removidos e, principalmente, os pesos relacionados podem ser modificados.

Um sistema dessa natureza é particularmente interessante para os diversos sítios de busca de trajetos em cidades, muito comuns na Internet. Entretanto, conforme foi explicado na introdução, tais sítios não levam em consideração características como a dinamicidade da informação, ou seja, normalmente os trajetos gerados entre dois pontos de uma cidade levam em consideração fatores estáticos como a distância. Com um arquitetura mais ampla, tais sistemas poderiam informar trajetos tendo como base informações que se alteram com o tempo, tais como, o fluxo nas vias, etc.

Também conforme já foi apresentado anteriormente, atualmente existem alguns trabalhos que permitem a busca de rotas em um sistema dinâmico, como por exemplo, os trabalhos de Fawcett e Robinson [4]. Entretanto, esses sistemas são em geral centralizados, ou seja, o grafo dinâmico está todo em apenas uma máquina, que recebe todas as requisições de consulta/atualização do sistema. Assim sendo, deseja-se desenvolver um sistema mais geral que permita a um grafo dinâmico ser particionado e apresentar uma arquitetura distribuída.

Durante os últimos meses, a principal atividade foi o levantamento de bibliografia e identificação dos requisitos essenciais para o desenvolvimento das idéias contidas nesse texto. Definiu-se, além disso, detalhes como qual linguagem de programação utilizar, qual a arquitetura de computação distribuída a ser considerada, configuração de ambientes e instalação de softwares para iniciar a etapa de desenvolvimento.

Dessa forma, verificou-se a necessidade do estudo e pesquisa para o desenvolvimento dos seguintes componentes:

1. Desenvolvimento de uma biblioteca de suporte à estrutura de dados que representa os grafos;
2. Implementação de uma camada intermediária (*middleware*) para suporte ao modelo de computação distribuída, permitindo que se realizem as ações de busca/atualização no grafo particionado em um conjunto de servidores, que se comunicam via uma rede de interconexão.
3. Algoritmos eficientes de busca de caminhos em grafos estáticos ou dinâmicos;
4. Generalização de algoritmos de busca de caminhos para sistemas distribuídos, possivelmente seguindo as idéias presentes em algoritmos de roteamento em redes de computadores.

Definiu-se a linguagem de programação Java[2] como a linguagem de desenvolvimento padrão, dada sua forte capacidade de implementação orientada a objetos. Além disso, durante todo o desenvolvimento/especificação deseja-se aplicar, sempre que possível, padrões comuns de orientação a objetos (*Design Patterns*), conforme descrito em [2].

A seguir, serão apresentados os resultados obtidos até o momento, sobretudo durante o procedimento de pesquisa à literatura existente, visando avaliar as tecnologias e algoritmos que existem nessa área de conhecimento.

Na próxima seção, serão apresentados as idéias gerais absorvidas durante o processo de levantamento bibliográfico, mostrando as metodologias já existentes sobretudo no que concerne aos modelos de computação distribuída, grafos dinâmicos, algoritmos de caminhos mínimos e algoritmos de roteamento. Na seção 3 é proposto um algoritmo de roteamento por hierarquias mistas, como uma possível solução para o problema de busca de rotas em grafos distribuídos.

[2] http://www.javasoft.com/

Na seção 4 são apresentados os resultados até agora obtidos com a implementação atual bem como são fornecidos detalhes dos próximos passos da tese. E, finalmente, a última seção tece algumas conclusões finais.

2.1 Levantamento de metodologias usadas

Nesta seção serão apresentadas algumas metodologias já existentes utilizadas para o desenvolvimento dos requisitos apresentados na seção anterior.

2.2 Uma estrutura de dados para grafos

Como foi explicado anteriormente, para o desenvolvimento de qualquer algoritmo ou idéia envolvendo teoria de grafos, o primeiro requisito é ter alguma estrutura de dados que descreva a implementação de um grafo.

Para tanto, um dos primeiros trabalhos dessa tese foi especificar um pacote para armazenar a estrutura de grafos em Java. Definiu-se uma biblioteca que segue a mesma linha das estruturas de dados existentes para grafos, como por exemplo, a biblioteca Java Graph Base[3]. No entanto, uma das peculiaridades primordiais da biblioteca especificada é o suporte a concorrência, ou seja, permitir que vários processos (*threads*) possam realizar simultaneamente buscas/atualizações no grafo de forma a impedir estados inconsistentes [10].

A biblioteca especificada apresenta essencialmente as seguintes classes:

1. AbstractGraph: classe abstrata com métodos para as funcionalidades gerais a grafos.
2. Graph: classe que estende AbstractGraph para grafos.
3. Vertex: classe que representa um vértice do grafo.
4. Edge: representa um arco de um dado grafo.
5. AbstractData: interface genérica contendo os métodos essenciais para manipulação dos dados presentes em arcos/vértices.
6. Data: implementação da interface AbstractData.
7. Path: classe responsável por armazenar um caminho sendo calculado por um dado algoritmo, por exemplo o algoritmo de Dijkstra [3].
8. Algorithms: classe contendo métodos estáticos para os algoritmos de busca, como por exemplo, o algoritmo de Dijkstra.

Basicamente, cada processo que deseja processar um dado algoritmo (por exemplo, a busca do caminho mínimo pelo algoritmo de Dijkstra), deve passar seu identificador único e global no sistema *threadId* de forma que o pacote possa lidar com múltiplos processos executando algoritmos simultanemente num dado grafo.

As funcionalidades essenciais da biblioteca especificada já foram implementadas e estão sendo utilizadas para realizar os testes iniciais.

[3] http://www.cs.rpi.edu/projects/pb/jgb/

2.3 Arquiteturas de computação distribuída

Como um dos principais objetivos desse trabalho é proporcionar que um grafo possa ser particionado em subgrafos disjuntos, os quais possam estar localizados possivelmente em máquinas diferentes.

Para isso, torna-se necessário adotar algum modelo de computação distribuída que permita a descentralização dos subgrafos de um grafo.

Assim, decidiu-se seguir um modelo para a criação de um grafo distribuído usando CORBA (Common Object Request Broker Architecture)[4]. CORBA é uma arquitetura para desenvolvimento em computação distribuída baseada no modelo RPC (Remote Procedure Call) [10]. Através do modelo RPC é possível que um cliente consiga executar um procedimento de um programa que está rodando em uma máquina remota e obter os dados de resposta, sendo que todo o processamento referente à chamada do procedimento é feito na remotamente. Para chamar o método remoto, o cliente dispõe de um interface (*proxy*), contendo as assinaturas dos métodos do programa, que está executando na máquina remota. O CORBA funciona como uma camada intermediária (*middleware*) responsável pela comunicação cliente/servidor.

Uma das grandes vantagens de CORBA é sua característica multi-plataforma e multi-linguagem, ou seja, um cliente escrito em Delphi rodando em uma estação Windows pode perfeitamente se comunicar com um programa escrito em Java rodando em UNIX, por exemplo. Isso é possível, pois para o desenvolvimento de sistemas em CORBA é necessário que se especifique um arquivo de interface escrito em uma linguagem própria conhecida como *Interface Definition Language* (IDL). Esse arquivo de interface é posteriormente compilado por um compilador presente na biblioteca CORBA, que gera o código para a linguagem de programação a que ele foi projetado. Por exemplo, um compilador para a linguagem Java gera as classes Java que descrevem os tipos descritos no arquivo de interface IDL.

Existem outras arquiteturas de computação distribuída, dentre elas destacam-se a arquitetura *Java Remote Method Invocation*[5] da Sun MicroSystems[6] e DCOM[7] da Microsoft[8] que não apresentam características como multi-plataforma/multi-linguagem, as quais caracterizam a arquitetura CORBA e por isso, decidiu-se usar CORBA para o suporte à computação distribuída.

2.4 Algoritmos em Grafos

2.4.1 Algoritmos para caminho mínimo

Um dos principais assuntos em teoria de grafos é o problema do caminho mínimo *Shortest Path Problem* [3]. Nesse problema, deseja-se encontrar uma rota mais curta (se alguma existir) entre dois vértices em um grafo. Ou seja, dado um grafo $G = (V, A)$, sendo V o conjunto de vértices e A o conjunto de arcos, associa-se uma função $w : A \rightarrow \Re$ mapeando os pesos de cada arco, onde, cada peso pode representar possíveis critérios de minimização. Assim, o peso de um caminho $p = (v_0, v_1, ..., v_k)$ é a soma dos pesos de seus arcos: $w(p) = \sum_{i=1}^{k} w(v_{i-1}, v_i)$. Um

[4]http://www.omg.org/
[5]http://java.sun.com/products/jdk/rmi/
[6]http://www.sun.com/
[7]http://www.microsoft.com/com/tech/DCOM.asp
[8]http://www.microsoft.com/

caminho de peso mínimo de um vértice u a um vértice v é $min(w(p))$ se existe ao menos um caminho, e ∞, caso contrário.

Para resolver esse problema, existem vários algoritmos cada qual com peculiaridades próprias, por exemplo, o algoritmo de Bellman-Ford descrito na seção 24.1 de [3]) resolve o problema para grafos com pesos quaisquer, com valores negativos ou não. Entretanto, se existem ciclos com peso negativo alcancáveis a partir do vértice original, o algoritmo devolve que não há um caminho mínimo. Esse algoritmo é baseado no roteamento utilizando o esquema de vetores de distância (*distance vector*), e que foi implementado para as primeiras versões dos roteadores da rede Arpanet e implementações do protocolo RIP (Routing Information Protocol [10]). Nesse algoritmo, cada nó mantém uma tabela que lista a rota mais barata para cada outro nó da rede. Em intervalos regulares de tempo, cada nó envia as informações presentes na sua tabela para os nós vizinhos (*broadcast*). O cálculo de uma rota entre dois nós tem custo de $O(V * A)$, considerando-se um grafo com V vértices e A arcos. Usando estruturas de dados mais eficientes para as buscas, tais como, heaps, árvores B ou *hashtables* [3], esse custo pode ser reduzido. Por outro lado, o algoritmo proposto por Dijkstra (descrito na seção 24.3 de [3]) admite grafos com pesos não-negativos. O custo do algoritmo é $O(V^2)$, considerando-se um grafo de V vértices.

O algoritmo de Dijkstra é um dos mais empregados para resolver o problema de caminho mínimo e as suas melhores implementações apresentam um tempo de execução inferior ao algoritmo de Bellman-Ford.

2.4.2 Algoritmos para grafos dinâmicos

Tradicionalmente algoritmos para grafos operam em estruturas estáticas. Normalmente, um grafo é dado e um problema é resolvido sobre ele. Entretanto, grafos dinâmicos não são fixos o tempo todo, uma vez que podem ocorrer mudanças nos seus estados internos[9]. Em geral, muitos problemas que envolvem grafos apresentam uma natureza dinâmica, tal como simular o estado de uma rede em um grafo, onde constantes atualizações dos valores dos pesos bem como a inserção/remoção de arcos ocorrem frequentemente. Dessa forma, deseja-se ter algoritmos que realizem buscas de rotas em grafos dinâmicos de uma forma eficiente. Para tanto algoritmos propostos por Ramalingan e Reps (RR) [9] e Frigioni, Marchetti e Nanni (FMN) [9] têm gerado bons resultados em termos práticos, comparando-se com as melhores implementações de algoritmos de caminhos mínimos em grafos estáticos, como o algoritmo de Dijkstra implementado com filas de prioridades. A seguir, serão apresentados os dois principais algoritmos utilizados para busca de caminhos mínimos em grafos dinâmicos: RR e FMN.

2.4.3 Algoritmo de Ramalingam e Reps

Nesta seção é descrita a primeira solução completamente dinâmica para o problema de busca de caminhos mínimos (*single source shortest path*) em digrafos com arcos contendo pesos positivos. Essa solução foi proposta por Ramalingam e Reps em [9], cuja performance é avaliada em termos de um parâmetro $|\delta|$, o qual é o número de vértices afetados por uma mudança δ, adicionado do número de arcos com ao menos um nó final[10] afetado. Ele exige $O(|\delta|.log(|\delta|))$ no pior caso para realizar inserções ou remoções.

[9]Neste trabalho somente serão permitidas mudanças dinâmicas nos pesos dos arcos

[10]Seja um arco arco (a, b), ligando o nó a em b. O nó final é b.

É passado ao algoritmo um grafo dirigido $G = (V, A)$ que teve um arco modificado. Considere $v \to w$ o arco modificado. O algoritmo precisa reestabelecer os caminhos mínimos em todas as árvores de caminhos mínimos geradas a partir do grafo. Nós chamaremos um vértice y de *afetado* pela modificação se há um outro vértice x tal que o comprimento do caminho mínimo $d(x, y)$ é mudado no novo estado do grafo G.

O algoritmo primeiramente encontra todos os vértices afetados, e então incrementalmente atualiza a árvore de caminhos mínimos para cada vértice afetado, com respeito ao arco modificado $v \to w$.

Na verdade, o algoritmo mantém um grafo dirigido acíclico (DAG) de caminhos mínimos para cada vértice, ao invés de uma árvore. Os autores usam o seguinte predicado para testar se um arco $a \to b$ está em algum dos caminhos mínimos presentes no DAG para um determinado vértice afetado:

$$SP(a, b, s) \equiv ((d(a, s) = w(a, b) + d(b, s)) \bigwedge (d(a, s) \neq \infty))$$

Onde $w(.,.)$ representa o peso de um arco e $d(.,.)$ representa a distância mínima obtida entre dois vértices.

Se $d(x, y)$ muda com o resultado da remoção do arco $v \to w$, então todos os caminhos mínimos de x até y no grafo original precisam ter passado por $v \to w$. Dessa forma, todos os caminhos mínimos de v até y precisam também ter passado por $v \to w$, e $d(v, y)$ é também mudado. Brevemente, v é a fonte de todas as mudanças para os vértices que são afetados. Para encontrar todos os vértices afetados, uma algoritmo incremental de caminhos mínimos com raiz em v é executado.

O código a seguir é adaptado de [9]. Nele, $Succ(v)$ retorna o conjunto de vértices vizinhos w para os quais há um arco $v \to w$ no grafo e, $Pred(v)$ retorna o conjunto de vizinhos de u para os quais há um arco $u \to v$. Considere o símbolo : como sendo *tal que*. No algoritmo, toma-se a liberdade de assumir que os conjuntos são implementados como pilhas, quando necessário. A variável formal dir, indica se os arcos devem ser considerados estarem na orientação usual(FORWARD) ou, em orientação invertida(REVERSE). Finalmente, assuma a existência de uma estrutura de dados de um heap [3], com as funções de inserção, remoção, comuns.

O procedimento DeleteEdge a seguir, realiza a remoção de um arco, computa o conjunto de arcos afetados e, então propaga mudanças para esses vértices. O procedimento DeleteUpdate, é um algoritmo para incrementalmente manter uma estrutura de caminhos mínimos.

```
procedure DeleteEdge(v → w)
    Remove v → w do grafo G
    dir ← REVERSE
    AffectedVertices ← DeleteUpdate(v → w, v)
    dir ← FORWARD
    for x ∈ AffectedVertices do DeleteUpdate(v → w, x)
return

procedure DeleteUpdate(v → w, z)
    AffectedVertices ← ∅
    if ¬∃x ∈ Succ(v) : SP(v, x, z) then
    /* Fase 1: Identifica vértices afetados */
```

```
    WorkSet ← {v}
    while WorkSet ≠ ∅ do
      u ← pop(WorkSet)
      push(u, AffectedVertices)
      for x ∈ Pred(u) : SP(x,u,z) do
        if ∃ y ∈ Succ(x)-AffectedVertices : SP(x,y,z)
          then push(x, WorkSet)
    /* Fase 2: Determina novas distâncias de z para todos vértices afetados */
    PQ ← ∅
    for a ∈ AffectedVertices do
      d(a,z) ← min({w(a,b)+d(b,z)):w(a,b) < ∞
        ∧ b ∉ AffectedVertices} ⋃ {∞})
    if d(a,z) ≠ ∞ then HeapInsert(PQ, a, d(a,z))
    while PQ ≠ ∅ do
      a ← HeapDeleteMin(PQ)
      for c ∈ Pred(a): w(c,a) + d(a,z) < d(c,z) do
        d(c,z) ← w(c,a) + d(a,z)
        HeapAdjust(PQ, c, d(c,z))
  return AffectedVertices
```

A seguir é apresentado o procedimento InsertEdge, para inserir um arco no grafo, usando o procedimento InsertUpdate, que incrementalmente mantém a estrutura de caminhos mínimos.

```
  procedure InsertEdge(v → w)
    Adicione v → w ao grafo G
    dir ← REVERSE
    AffectedVertices ← InsertUpdate(v → w, v)
    dir ← FORWARD
    for x ∈ AffectedVertices do InsertUpdate(v → w, x)
  return

  procedure InsertUpdate(v → w, z)
    WorkSet ← {v → w}
    VisitedVertices ← {v}
    AffectedVertices ← ∅
    while WorkSet ≠ ∅ do
      e ← pop(WorkSet)
      if w(x,u) + d(u,z) < d(x,z) then
        push(x, AffectedVertices)
        d(x,z) ← w(x,u) + d(u,z)
        for y ∈ Pred(x) do
          if SP(y,x,v) ∧ y ∉ VisitedVertices then
            push (y → x, WorkSet)
            push (y, VisitedVertices)
  return AffectedVertices
```

2.4.4 O Algoritmo de FMN

O algoritmo FMN foi proposto por três autores: Frigioni, Marchetti e Nanni [1]. Nesta seção, será descrito o algoritmo proposto para encontrar caminhos mínimos em digrafos com arcos contendo pesos positivos. Considere OUT(x) como sendo o conjunto de arcos (x,y) saindo de um dado vértice x e IN(x) o conjunto (y,x) de arcos chegando ao vértice x. E, $W_{x,y}$ denota o peso de um arco (x,y).

A estratégia proposta em [1] é baseada no uso de dois ingredientes principais: a noção de nível de um arco e a noção de *ownership* (domínio) de um vértice, que pode ser descrito conforme a seguir:

1. Seja $G = (V, E)$ um digrafo e z um vértice em G. O nível anterior (*backward_level*) do arco $(z, q) \in OUT(z)$ e de vértice q, relativo ao vértice z, é a quantidade $b_level_z(q) = d(q) - W_{z,q}$. O nível posterior (*forward_level*) do arco $(v, z) \in IN(z)$ e de vértice de v, relativo ao vértice z, é a quantidade $f_level_z(v) = d(v) + W_{v,z}$. Percebe-se que os níveis de um arco (z, q) provêm informação sobre o caminho mínimo disponível de s a q passando por z. Por exemplo, suponha que, enquanto ocorrer o processamento de uma inserção de arco, o algoritmo computa uma nova distância $d_1(z)$ do vértice z, e que existe um arco $(z, q) \in OUT(z)$ tal que $b_level_z(q) - d_1(z) = d(q) - W_{q,z} - d_1(z) > 0$, i.e., $d(q) > d_1(z) + W_{q,z}$. Isso significa que após uma inserção ao caminho compreendido do vértice início s até q que passa através de z, torna esse novo caminho menor do que o caminho mínimo de s até q no grafo anterior à inserção. No caso de uma remoção de arcos, suponha que a operação é realizada no arco que está no corrente cálculo de caminho mínimo de s até o vértice z. Seja l o comprimento do caminho mínimo de s até z encontrado pelo algoritmo. Suponha agora que há um arco $(q, z) \in IN(z)$, tal que $f_level_z(q) - l = d(q) + W_{q,z} - l < 0$, i.e., $l > d(q) + W_{q,z}$. Isso significa que o caminho de s até z que passa por q é menor do que o caminho corrente de comprimento l.

2. De forma a limitar o número de arcos visitados pelos algoritmos cada vez que um vértice muda sua distância do início s, para cada vértice $z \in V$, cada um dos conjuntos $IN(z)$ e $OUT(z)$ é particionado em dois subconjuntos como mostrado a seguir: cada arco $(x, y) \in E$ tem seu dono (*owner*) que precisa ser ou x ou y. Para cada vértice $x \in V$, $IN - OWN(x)$ denota o subconjunto de arcos em $IN(x)$ contendo os arcos cujo dono é x e, $\overline{IN - OWN}(x)$ denota o conjunto de arcos em $IN(x)$ que não tem x como dono. Analogamente, $OUT - OWN(x)$ e $\overline{OUT - OWN}(x)$ representam os arcos em $OUT(x)$ que tem x com dono e que não tem x como dono respectivamente. A partição acima satisfaz às seguintes propriedades:

I. $(z, q) \in OUT - OWN(z)$ se e somente se $(z, q) \in \overline{IN - OWN}(q)$;

II. $(z, q) \in \overline{OUT - OWN}(z)$ se e somente se $(z, q) \in IN - OWN(q)$. O digrafo G tem um *ownership* limitado a k se ambos os conjuntos $IN - OWN(x)$ e $OUT - OWN(x)$ contêm no máximo k arcos. A noção de *ownership* k-limitado é requerido somente para limitar o tempo de execução do algoritmo.

Para cada vértice, os arcos em $\overline{IN - OWN}(x)$ e em $\overline{OUT - OWN}(x)$ são armazenados em duas filas de prioridades denotadas F_x e B_x, com prioridades dadas por seus f_level e b_level, respectivamente. Em qualquer momento que um vértice z muda sua distância do vértice de início s, os algoritmos visitam os arcos que têm z como dono (ao máximo k), e somente os arcos que não têm z como dono, conforme apresentado na intuição descrita acima.

No caso da inserção de um arco, o número de *output updates* é dado pelo número de vértices

que mudam as suas distâncias a partir do início s, como consequência da modificação. Se a inserção do arco (x, y) diminui a distância do vértice y ao início s, uma fila de prioridades global C é usada, como no algoritmo de Dijkstra; quando um vértice z é retirado da fila C e sua nova distância a partir de s $d_1(z) < d(z)$ é computada, não todos os arcos saindo de z são visitados de forma a propagar a nova distância $d_1(z)$. Em particular, a informação armazenada no heap B_z é usada, para selecionar arcos (z, q) em $IN - \overline{OWN}(z)$ para serem visitados. Eles são os arcos cuja prioridade em B_z é maior do que $d_1(z)$, i.e., os arcos que conduzem ao vértice q diminuiram as suas distâncias a partir de s, se escolher z como novo pai na árvore de caminho mínimo.

No caso de remoção de arcos, o número de *output updates* é dado pelo número de vértices que, ou mudam as suas distâncias a partir do vértice inicial s, como consequência da modificação, ou precisam mudar o pai em $T(s)$, tanto que o novo caminho mínimo a partir de s é tão bom quanto o anterior. O algoritmo proposto em [1] para remoção de arcos é realizado em 2 fases: primeiro, encontre todos os vértices que precisam ser atualizados, e então compute as novas distâncias a partir de s, bem como a nova árvore de caminho mínimo. Em particular, se z é um vértice que tem sua distância a partir de s incrementada, devido a uma remoção de arco, o algoritmo precisa realizar duas tarefas:

1. encontre o novo caminho mínimo de s a z, escolhendo um novo pai para z em $T(s)$.

2. descubra os vizinhos de z que também tiveram suas distâncias incrementadas. O número de arcos em $\overline{IN - OWN}(z)$ visitados pelo algoritmo na primeira fase, a fim de encontrar o melhor caminho possível de s a z é limitado como segue: Somente arcos (q, z) em $IN - OWN(z)$ com $f_level_z(q) - d(z) = d(q) + W_{z,q} - d(z) < 0$ são considerados, usando a informação armazenada no heap F_x. A segunda fase realiza uma computação similar ao algoritmo de Dijkstra, no subgrafo de G induzido pelos vértices que mudam as suas distâncias de s, como consequência da remoção Nessa fase, somente os arcos entre os vértices afetados são visitados.

Se o grafo G tem um *ownership* k limitado, então o algoritmo exige $O(klogn)$.

2.5 Algoritmos para roteamento em redes

Os problemas de roteamento têm sido amplamente estudados para a transmissão de datagramas[11] através de redes de computadores. Dentre os diversos algoritmos de roteamento em rede, destaca-se a família de algoritmos baseados nos estados dos *links*, *Link State*, dos quais o mais famoso é o *New Arpanet Routing Algorithm* proposto por Mcquillan [8].

Os detalhes desses algoritmos podem variar mas, em geral, eles seguem a mesma idéia: cada nó mantém um banco de dados topológico representando o estado da rede. Usualmente a base topológica é considerada como um grafo formado pelos nós da rede e os arcos representam as conexões entre os nós. Cada arco mantém a informação do seu peso que pode ser, por exemplo, o número de pacotes por unidade de tempo trafegando por aquele arco. Assim, cada nó calcula a rota mais barata partindo de si próprio para cada um dos outros nós da rede (utilizando algoritmos clássicos para encontrar caminhos de peso mínimo em grafos [3]). A partir daí, cada membro da rede constrói uma tabela de roteamento mostrando, para cada destino, qual vizinho é o primeiro ponto para a rota de menor custo. Quando as condições da rede mudam, as informações são repassadas por meio de *broadcast* e as bases topológicas/tabelas de roteamento

[11] As informações que trafegam pelas redes são particionadas em unidades menores (pacotes) chamados de datagramas. O datagrama possui informações codificadas sobre o seu destino.

precisam ser atualizadas.

Apesar desses algoritmos resolverem o problema de roteamento, eles apresentam um sério inconveniente, uma vez que se faz necessário manter a informação de todos os nós da rede e, no caso de uma rede complexa e densa como por exemplo, a Internet ou uma WAN, essas técnicas tornam-se inviáveis.

Visando solucionar o principal problema relacionado aos primeiros algoritmos de roteamento, surgiram as idéias de roteamento hierárquico para transmissão de pacotes, ou seja, permitir que um vértice tome decisões de roteamento baseando-se em informações do estado global de um agrupamento de nós vizinhos e não no estado local de cada nó.

Dessa forma, para o problema aqui exposto propõe-se uma alternativa de roteamento hierárquico[12] baseado na estrutura distribuída da informação. Por ele, as estações-base formariam uma estrutura de hierarquias de forma tal que a troca de informações seja feita no nível das hierarquias às quais a estação-base pertence, diminuindo-se consideravelmente o tamanho das mensagens.

3 Roteamento Hierárquico em Sistemas Distribuídos

A partir deste momento, serão apresentadas as idéias gerais do modelo de hierarquias proposto para generalizar algoritmos de roteamento para um ambiente distribuído, conforme apresentado no item 4 da seção Análise de Requisitos.

3.1 Modelagem do problema

Conforme foi discutido na seção 1, o modelo proposto procura resolver o problema de se encontrar um caminho que minimize o custo de um determinado recurso, como distância, tempo, etc.

Para tanto, considera-se que o sistema é constituído por um conjunto de estações-base, cada qual mantendo informação das condições dos links que estão sob a sua área de cobertura. Assume-se que a informação armazenada numa estação (E_i) é representada por um grafo $G_i = (V_i, A_i)$, com a estrutura das vias de tráfego, bem como os estados de cada via. Nesse sentido, pode-se considerar que cada estação armazena um subgrafo G_i de um grafo maior (G), representando o domínio da aplicação, ou seja, $G = \bigcup_{i=1}^{N} G_i$, onde N é o número de estações servindo o domínio.

De acordo com o que já foi explicado anteriormente, os vértices de um grafo G_i representam cruzamentos entre vias e os arcos podem ser interpretados como a junção (link) entre dois cruzamentos. São nos links, que os estados são guardados. Na ilustração a seguir é apresentada uma região contendo duas ruas com 5 cruzamentos, representados por V_i, com $1 \leq i \leq 5$. A rua que apresenta os vértices V_1,V_3 e V_5 apresenta um único sentido, enquanto a rua que mantém V_2, V_3 e V_4 tem duplo sentido. Para essa região, é apresentado o grafo ao lado, com as respectivas arestas indicando os sentidos.

[12]Modelo de hierarquias mistas.

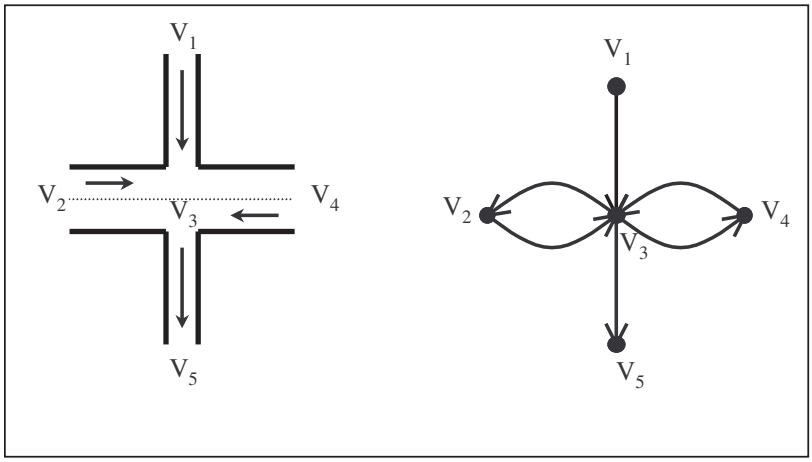

Figura 1. Representação de um domínio em estrutura de grafos.

Assume-se que a região de fronteira da área de cobertura de E_i é representada por um conjunto de vértices especiais, que formam a borda ($B(G_i)$) de E_i. As bordas são nós de junção inter-estações, ou seja, o grafo G, representante do domínio completo, é formado pelo conjunto dos subgrafos G_i colapsados pelas bordas $B(G_i)$. Considera-se também que a intersecção $G_i \bigcap G_j = B(G_i) \bigcap B(G_j)$, ou seja, a intersecção dos vértices entre estações é dada pela intersecção dos conjuntos de nós das respectivas bordas, sem a existência de nós internos.

Na figura a seguir, duas estações E_A e E_B são responsáveis pela cobertura de todo um domínio de aplicação (G). Nesse caso:

$B(G_A) = \{1, 2, 5, 6, 8, 10, 11, 12\}$;
$B(G_B) = \{10, 11, 12, 13, 14, 16, 18, 19\}$;
$B(G_A) \bigcap B(G_B) = \{10, 11, 12\}$.

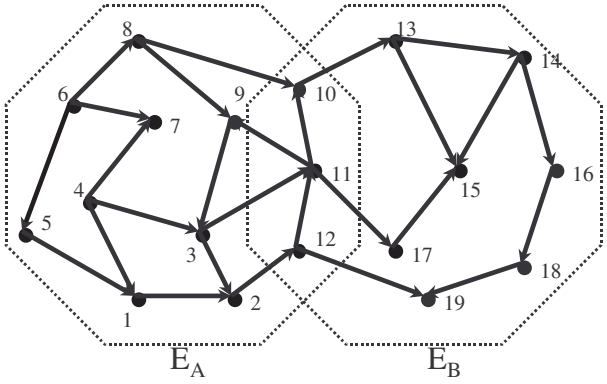

Figura 2. Topologia de um domínio de aplicação sob a cobertura de duas ERBs.

Um usuário U_i requisita informação de roteamento entre dois pontos quaisquer do domínio de aplicação à estação-base E_i responsável pela área na qual U_i está localizada no momento. E_i

então solicita à estação que armazena o ponto de origem (E_j, por exemplo) para que se obtenha o roteamento. Ao fim do processo, E_j envia o resultado para a estação onde U_i está localizada, possivelmente a mesma estação E_i ou uma nova estação, E_k.

3.2 Estrutura de Hierarquias

Conforme foi apresentado na seção 2, um dos maiores problemas relacionados com os primeiros algoritmos de roteamento de pacotes em redes é que eles necessitavam manter em sua base topológica, a informação de todos os nós da rede, o que tornou a utilização dessas classes de algoritmos inviável para redes de grande porte, em termos práticos.

Por conta disso, começaram a ser investigadas novas alternativas para o uso efetivo dos algoritmos de roteamento, dentre elas, as técnicas de roteamento hierárquico têm obtido bons resultados principalmente em redes de grande complexidade [6].

Para o problema que está sendo estudado, propõe-se um modelo de roteamento por hierarquias mistas a fim de se obter uma melhor performance do sistema, sobretudo no que diz respeito à troca de mensagens entre estações. Entretanto, devido ao fato de se utilizar uma estrutura de hieraquias de informações, claramente, haverá uma perda na precisão dos dados, a qual, acredita-se não ser muito significativa pois, como será apresentado no algoritmo a seguir, serão privilegiadas informações mais precisas nas regiões próximas aos pontos de origem/destino e na medida em que os vértices vão se tornando mais distantes dos pontos considerados, informações de hierarquias mais altas (menos precisas) serão consideradas nos cálculos.

Dessa forma, assume-se a existência de uma estrutura de hierarquias envolvendo as estações que cobrem o domínio de aplicação. Para simplificar o entendimento, será adotada a nomenclatura $H_i(j)$ para representar a hierarquia no nível i à qual a estação E_j está relacionada.

A hierarquia base de cada estação é formada pela própria estação, ou seja, $H_0(j) = Ej$. Os próximos níveis de hierarquias nos quais E_j está envolvida, são dados por $H_i(j) = V_{i-1}(j) \bigcup H_{i-1}(j)$. Onde $V_i(j)$ representa um subconjunto das hierarquias vizinhas de $H_i(j)$ e geralmente é determinado à priori. Na ilustração a seguir, são apresentadas 45 estações cobrindo uma determinada região de um domínio de aplicação. Através da estrutura de hierarquias apresentada, verifica-se que, por exemplo, $H_0(36) = \{E_{36}\}$, $V_0(36) = \{E_{37}, E_{46}, E_{47}\}$ e $V_1(36) = \{H_1(16), H_1(23), H_1(38), H_1(55), H_1(58)\}$.

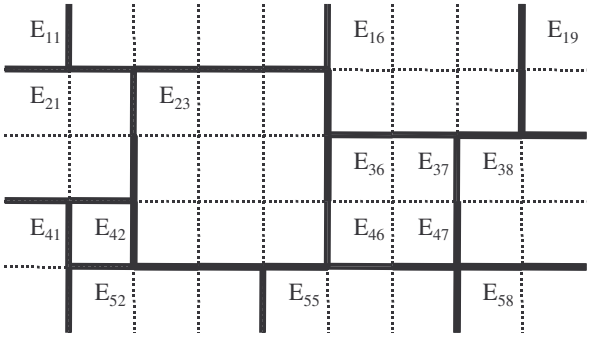

Figura 3. Estrutura de hierarquias.

Dentro de cada hierarquia $H_i(j)$, necessita-se eleger uma ERB que execute o papel de líder, a fim de receber as informações dos estados das sub-hierarquias[13] de $H_i(j)$, $Sub(H_i(j))$, e montar a informação do estado geral da própria hierarquia que, eventualmente, poderá ser passada para o líder das super-hierarquias[14] de $H_i(j)$, $Sup(H_i(j))$.

Para a eleição de líderes, pode-se adotar algoritmos de eleição, tais como os algoritmos descritos em Tanenbaum e van Steen [10]. Para tanto, poderia ser considerado algum critério como a escolha da estação com menor carga, privilegiando-se o balanceamento de carga do sistema. Além disso, possivelmente um líder em $Sub(H_i(j))$ poderia certamente ser também eleito para liderar $H_i(j)$.

Dentre os papéis da estação eleita líder de uma hierarquia estão a recepção dos estados das sub-hierarquias, a atualização de uma base de dados (base topológica) da hierarquia, elaboração de uma tabela de estado (detalhes de como será calculada a base topológica e a tabela de estado serão apresentados no próximo item) e principalmente, enviar informações sobre o estado da hierarquia, através da tabela construída, para outras hierarquias.

Devido à vasta gama de responsabilidades atribuídas aos líderes de hierarquias, a performance do sistema poderia ficar comprometida, sobretudo nas camadas de hierarquias superiores. Por conta disso, um sistema dessa natureza deveria prever a existência de co-líderes, que poderiam ser nomeados, caso o líder começasse a passar por sérios problemas de desempenho. Os co-líderes poderiam trabalhar como réplicas do líder, a fim de distribuir a carga. Porém, os co-líderes somente executariam o papel de envio de informações sobre o estado da hierarquia, não cabendo outros papéis como de cálculo das tabelas de estado ou atualização da base topológica.

Convém destacar ainda que a hierarquia mais elevada, ou seja, a hieraquia que contém o conjunto de todas estações do domínio não precisaria, em princípio, escolher líderes, nem tão pouco calcular um base topológica e a respectiva tabela de estados.

3.3 Cálculo do Estado das Hierarquias

Após cada hierarquia $H_i(j)$ ter um líder $L_i(j)$ eleito, cabem aos sub-líderes[15] enviar as tabelas de estados das respectivas sub-hierarquias, a fim de que $L_i(j)$ calcule a base topológica bem como a tabela de estados de $H_i(j)$.

A maneira pela qual a base topológica bem como a respectiva tabela de estados de uma dada hierarquia $H_i(j)$ são obtidas será descrita a seguir. No caso da hierarquia base $H_0(j)$ de uma estação E_j (onde o único representante é a própria estação E_j), a estação executa um algoritmo para encontrar o caminho de custo mínimo para todas as combinações possíveis dos vértices na borda de E_j ($B(G_j)$). A partir daí, pode ser montada a tabela de estados, guardando para cada par de vértices (b_w, b_k), com b_w, b_k em $B(E_j)$, o custo mínimo calculado para sair de b_w e chegar em b_k. A base topológica, no caso da hierarquia base $H_0(j)$, é o próprio subgrafo G_j.

No caso das hierarquias superiores de E_j, assim que o líder recebe as tabelas de estados atualizadas de cada uma das suas sub-hierarquias, ele trata de montar um grafo a partir das tabelas recebidas e obtêm os melhores caminhos para todos os pares de vértices na borda. Esse grafo representa a base topológica da hierarquia e a partir daí, pode-se gerar a tabela de estados para a hierarquia conforme descrito anteriormente.

[13] Conjunto de hierarquias em $H_i(j)$

[14] Conjunto de hierarquias em $H_{i+1}(j)$

[15] Líderes de sub-hierarquias.

Na ilustração a seguir é mostrada a construção do grafo de melhores rotas entre as bordas da hierarquia de nível 1 das estações E_1 e E_2.

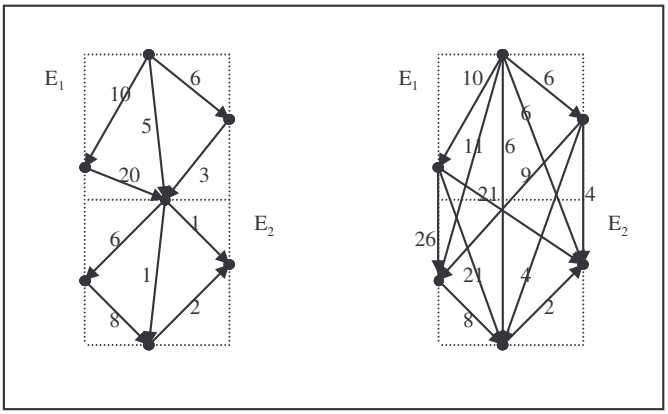

Figura 4. Grafo contendo os melhores caminhos calculados por $L_1(1) = L_1(2)$.

3.4 Roteamento por Hierarquias Mistas

A partir deste momento, será apresentado um algoritmo de roteamento através do qual clientes poderiam se apoiar para obter rotas de peso mínimo, utilizando a estrutura exposta anteriormente.

O algoritmo se baseia no roteamento por hierarquias mistas, ou seja, partindo do ponto de origem até o destino, o sistema constrói um grafo com as tabelas de estados de variados níveis de hierarquias que compõem o domínio de aplicação, levando em consideração informações mais precisas (presentes em hierarquias dos níveis mais baixos) para as regiões mais próximas dos pontos de origem/destino e, à medida que as regiões se afastam dos pontos de origem/destino, utilizam-se as informações contidas em hierarquias mais altas (porém menos precisas), visando-se diminuir a troca de mensagens entre as estações.

Para tanto, considere um cliente solicitando um roteamento do vértice de origem u até o vértice v. Nesse caso, E_u e E_v serão as estações responsáveis pelas áreas de cobertura das regiões onde se encontram os vértices u e v, respectivamente; G_u o grafo representante de E_u e G_v descrevendo E_v. Finalmente, seja W um inteiro representando o número de níveis de hierarquias do sistema.

Inicialmente, E_u calcula os caminhos mínimos de u até cada cada um dos vértices de $B(G_u)$ que são atingíveis a partir de u. Com essas informações, E_u inicia a construção de um grafo de roteamento R_u, tendo inicialmente o nó u ligado por um arco a cada um dos vértices no conjunto $B(G_u)$, que foram alcançadas a partir de u. Os arcos armazenam o custo total mínimo que foi calculado anteriormente para sair de u e chegar aos vértices na borda de G_u.

A partir daí, E_u requisita para cada um dos líderes das hierarquias diretamente superiores, as tabelas de estados de cada uma das hierarquias em $V_i(u)$, com $0 \leq i < W - 1$. Ao receber as tabelas de estados, E_u atualiza o grafo R_u, inserindo novos arcos e nós. Essas inserções

obedecem a seguinte regra: procura-se na tabela de estados recebida os pares de vértices (T_k, U_p) e custo $C_{k,p}$ tais que T_k (ou U_p) está em R_x. Dessa forma, adiciona-se um nó U_p (ou T_k) e um arco (T_k, U_p) com o custo $C_{k,p}$ em R_x.

Em E_y também se inicia a construção de um grafo de roteamento R_y seguindo um procedimento análogo ao descrito para a construção de R_x. No caso da construção de R_y, E_y busca as tabelas de estados de cada uma das hierarquias em $V_i(y)$, com $0 \leq i \leq Z$, onde Z é o nível de hierarquia onde a Z-ésima hierarquia de E_x e a Z-ésima hierarquia de E_y são vizinhas. Por exemplo, se E_x e E_y pertencem à mesma hierarquia no nível 3, E_y calculará R_y até o nível de hierarquia 2. O resultado disso é que R_y conterá estimativas das melhores rotas partindo dos vértices pertencentes à borda da hierarquia de nível 2 que contém y ($H_2(y)$) até o ponto y. E, ao final disso, E_y deveria construir a tabela de estados com essas informações e, de alguma forma, enviar a E_x, para que esse último atualize o grafo de roteamentos R_x.

Como conseqüência da construção de R_y, mantendo rotas de y aos vértices da borda de $H_Z(y)$, deve-se explicar que E_x não precisará solicitar a tabela de estados ao líder da hierarquia $H_Z(y)$.

No caso específico quando x e y pertencem à mesma área de cobertura, isto é, $E_x = E_y$, a construção de R_x procede normalmente, porém R_y não precisa ser obtido. Basta apenas se calcular os caminhos mínimos de y aos vértices da borda de E_y ($= E_x$), alcancáveis de y, bem como as melhores rotas entre x e y, adicionando os resultados obtidos em R_x.

Com o grafo de R_x formado, E_x pode executar um algoritmo para encontra um caminho mínimo de x até y, como o algoritmo de Dijkstra [3].

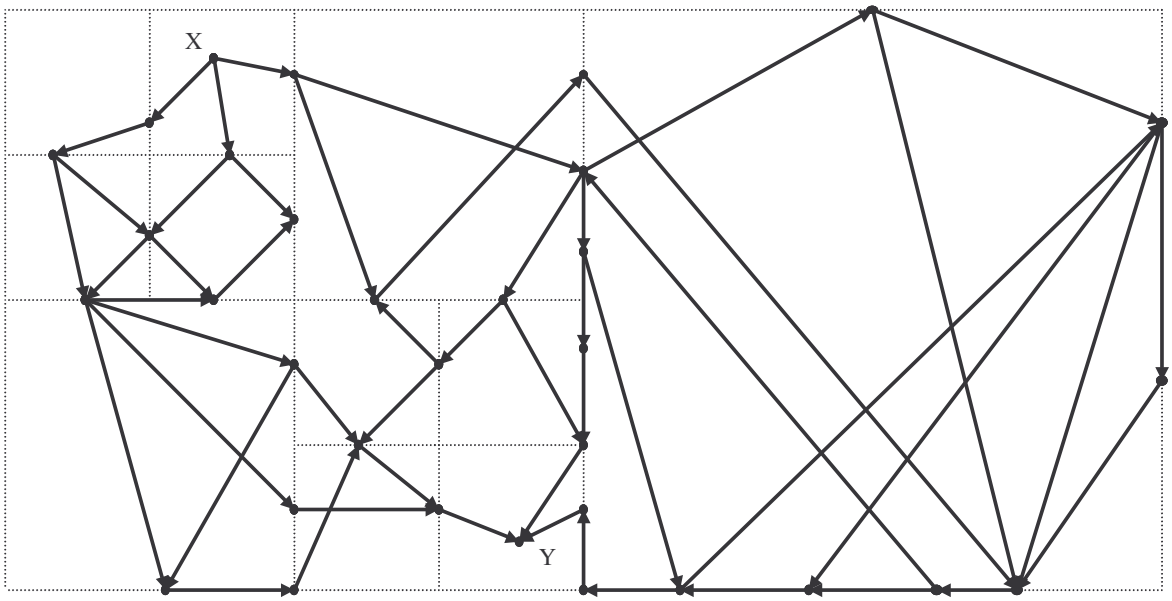

Figura 4. Grafo R_x construído por E_x para obter uma estimativa de caminho de peso mínimo entre x e y num domínio de aplicação G.

4 Estado atual da implementação

A partir deste momento serão apresentados os resultados obtidos até o momento com a implementação atual do sistema. O objetivo principal dessa fase inicial de implementação é procur avaliar o limite máximo de conexões suportado em modelos usando grafos estáticos em ambiente centralizado.

Conforme foi dito anteriormente já foi desenvolvido um pacote para armazenar a estrutura de um grafo em Java. O pacote implementado apresenta suporte à concorrência, uma vez que um sistema como o que está sendo desenvolvido necessita ter a possibilidade de que vários processos simultâneos possam estar realizando consultas/atualizações no grafo.

Com a biblioteca de grafo desenvolvida, resolvemos realizar alguns testes, para avaliar o desempenho de uma arquitetura centralizada contendo um grafo estático, ou seja, um servidor apenas armazenando um grafo suficientemente grande para receber pedidos de clientes, requisitando trajetos de custo mínimo entre dois pontos do grafo. O nosso objetivo com isso foi tentar mostrar através de simulações que um sistema centralizado, conforme o descrito em [4] representa um ponto de gargalo, uma vez que ele é o responsável por todo o processamento do sistema.

Para tanto, utilizamos CORBA como o ambiente distribuído, em particular, o pacote JaCORB 1.1[16], que se trata de uma implementação de CORBA em Java.

Foi desenvolvido um servidor em JaCORB implementado através da classe GraphServerImpl o qual é o responsável por receber solicitações de trajetos de peso mínimo para serem calculados a partir de um grafo localizado na mesma máquina. O grafo utilizado é bastante simples: trata-se de um grafo em forma de grade de dimensões 100x100, com arcos contedo pesos aleatórios. Conforme foi explicado, nesse primeiro momento não estavámos interessados em avaliar grafos dinâmicos, ou seja, cujo valor dos pesos variam em função do tempo.

O algoritmo para caminhos mínimos utilizado foi o algoritmo de Dijkstra [3], que como já foi explicado, apresenta um bom desempenho em relação a outros algoritmos de busca de caminhos mínimos, em grafos estáticos.

Os testes foram realizados nas estações de trabalho do IME-USP, utilizando o pacote Java JDK 1.3.1[17].

Com esse ambiente computacional, alguns testes foram realizados através da criação de threads disparados simultaneamente requisitando trajetos mínimos para o mesmo par de pontos: o vértice de início localizado no nó {0,0} da grade e o vértice de destino, na posição {99,99}. Os resultados obtidos são apresentados na tabela a seguir:

[16] http://www.jacorb.org/
[17] http://www.javasoft.com/

Threads	Tempo(seg)
1	3.2
2	6.6
3	10.5
4	13.4
5	16.1
10	28.1
20	58.1
30	61.0
40	98.3

Na prática, o número máximo de acessos simultâneos ao servidor foi da ordem de 40 requisições para o ambiente computacional testado, uma vez que para um número maior de threads simultâneas acessando o sistema, a máquina virtual apresentou exceção de falta de memória (OutOfMemoryError). Acredita-se que em máquinas com arquiteturas cliente-servidor, mais modernas que as estações da trabalho, esses valores podem ser superados, no entanto, avaliando-se comparativamente os valores obtidos, percebe-se que a partir de 20 conexões simultâneas, o sistema apresenta um atraso bastante considerável, mostrando que um sistema centralizado apresenta sérios problemas de performance, possivelmente justificando a próxima etapa a ser investigada nessa tese de Mestrado, avaliar a performance de um sistema distribuído.

5 Conclusão

Este estudo permitiu verificar os pré-requisitos básicos para o desenvolvimento da tese de Mestrado. A partir de então foi possível realizar o levantamento bibliográfico de pesquisas já realizadas na área de sistemas distribuídos, grafos dinâmicos e algoritmos de roteamento em redes. Também foi possível implementar a infra-estrutura básica para todos os algoritmos e modelos que possam vir a serem implementados.

Com a arquitetura inicialmente implementada, foi possível realizar testes com sistemas centralizados (em um único servidor), armazenando grafos estáticos.

O objetivo dessa fase inicial foi o de justamente verificar que, para um sistema com muitas requisições de consulta/atualização, a utilização de apenas um servidor para atender às solicitações apresenta sérios problemas de performance, conforme foi visto na seção anterior.

Também foi modelada a idéia de um algoritmo de busca de rotas para sistemas distribuídos, baseado nos conhecidos algoritmos de roteamento hierárquico em redes de computadores. O algoritmo aqui exposto se baseia em uma estrutura de hierarquias mistas, conforme explicado na seção 3.

Esse trabalho permitiu realizar grande parte do trabalho inicial da tese, além de dar um melhor entendimento sobre os próximos passos a serem investigados: implementar a generalização do sistema de busca de caminhos para uma arquitetura distribuída, através da implementação do algoritmo de roteamento por hierarquias mistas proposto e, com isso, tentar avaliar as vantagens ou desvantagens em relação a sistemas centralizados. Após essa etapa, procurar-se-á avaliar o modelo obtido em sistemas contendo grafos dinâmicos, nesse caso, podendo utilizar esquemas

adaptativos como os algoritmos de busca de caminhos em grafos dinâmicos apresentado na seção 2.4 ou implementar uma solução mais apropriada para o problema.

Referências

[1] D. Alberts, G. Cattaneo, G. Italiano, U. Nanni, and C. Zaroliagis. A software library of dynamic graph algorithms. *Proceedings of Algorithms and Experiments (ALEX98) Trento, Italy*, pages 129–136, February 9–11 1998.

[2] J. Cooper. *Design Patterns Java Companion.* 1998.

[3] T. Cormen, C. Leiserson, R. Rivest, and C. Stein. *Introduction to Algorithms.* MIT Press, 2a. edition, 2001.

[4] J. Fawcett and P. Robinson. Adaptive routing for road traffic. *IEEE Computer Graphics and Applications*, 20(3):46–53, May/June 2000.

[5] L. Kleinrock and F. Kamoun. Hierachical routing for large networks: Performance evalution and optimization. *Computer Networks*, 1(2):155–174, 1977.

[6] P. Lauder, R. Kummerfeld, and A. Fekete. Hierachical network routing. *Proceedings of TriComm'91, IEEE Conference on Communications Software, Chapel Hill, North Carolina, USA*, pages 105–114, April 1991.

[7] A. Loureiro and R. Mateus. *VIII Escola de Computação - Introdução à Computação Móvel.*

[8] J. McQuillian. Adaptive routing algorithms for distributed computer networks. *BBN Rep. 2831, Bolt Beranek and Newman Inc., Cambridge MA*, May 1974.

[9] R. Ramanlingam and T. Reps. An incremental algorithm for a generalization of the shortest path problem. *Computer Science Department Technical Report 1087*, May 1992.

[10] A. Tanenbaum and M. Steen. *Distributed Systems - Principles and Paradigms.* Prentice Hall, 2002.