Universidade Federal de São Carlos
Campus Sorocaba
Prof. Fábio Luciano Verdi

Laboratório de Redes de Computadores
Em dupla

Projeto RIP

- Projeto Algoritmo de Vetor de Distância
  - Começo dia 26/08/2015
  - Entrega e apresentação dia 23/09
  - Entregar o código fonte via Drive. Não é necessário entregar relatório para este projeto.

*** A descrição abaixo é uma adaptação do livro do Kurose que norteia a implementação do algoritmo de vetor de distância. A dupla é livre para criar novas e inteligentes soluções para resolver a atividade.

Implementando um algoritmo de vetor de distância

OBS: A tarefa abaixo está descrita considerando o uso da linguagem C. Entretanto, a dupla deverá implementar a tarefa usando outra linguagem, a critério da dupla. Entenda o código em C apenas como referência e aprendizado para compreensão da atividade.

Visão geral

Neste laboratório, você escreverá um conjunto distribuído de procedimentos que implementam um roteamento de vetor de distâncias assíncrono distribuído para a rede mostrada na Figura Lab.4-1.

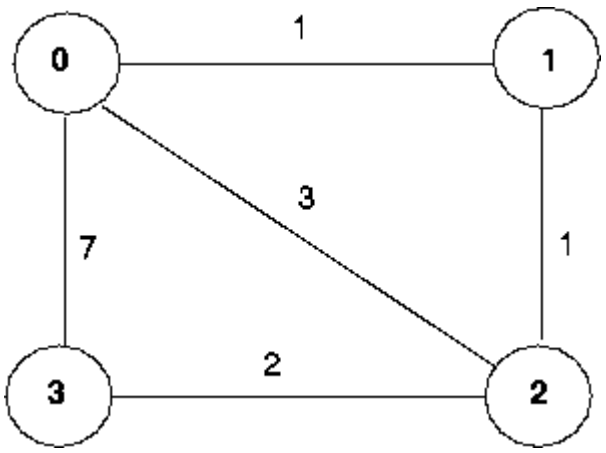


Figura Lab.4-1: Topologia de rede e custos dos enlaces para o laboratório de roteamento de vetor de distâncias.

Tarefa básica

Rotinas que você irá escrever. Para a parte básica da tarefa, você escreverá as seguintes rotinas que executarão assincronamente dentro do ambiente emulado que escrevemos para esta tarefa.

Para o nó "0", você escreverá as rotinas:

• **`rtinit0()`** Esta rotina será chamada uma vez no início da emulação. `rtinit0()` não possui argumentos. Ela deverá inicializar a tabela de distâncias no nó "0" para refletir os custos 1, 3 e 7 para os nós 1, 2 e 3, respectivamente. Na Figura 1, todos os links são bidirecionais, e os custos em ambas as direções são idênticos. Após inicializar a tabela de distância, e qualquer outra estrutura de dados necessária para sua rotina do nó "0", ela deverá então enviar aos vizinhos diretamente conectados (neste caso, 1, 2, e 3) o custo dos seus caminhos de menor custo para todos os outros nós da rede. Esta informação de custo mínimo é enviada para os nós vizinhos em um *pacote de rotina* através da chamada rotina `tolayer2()`, conforme descrita abaixo. O formato do pacote de rotina é descrito abaixo também.

• **`rtupdate0()(struct rtpkt *rcvdpkt)`**. Esta rotina será chamada quando o nó "0" receber um pacote de rotina enviado por um de seus vizinhos diretamente conectados. O parâmetro `*rcvdpkt` é um ponteiro para o pacote que foi recebido.

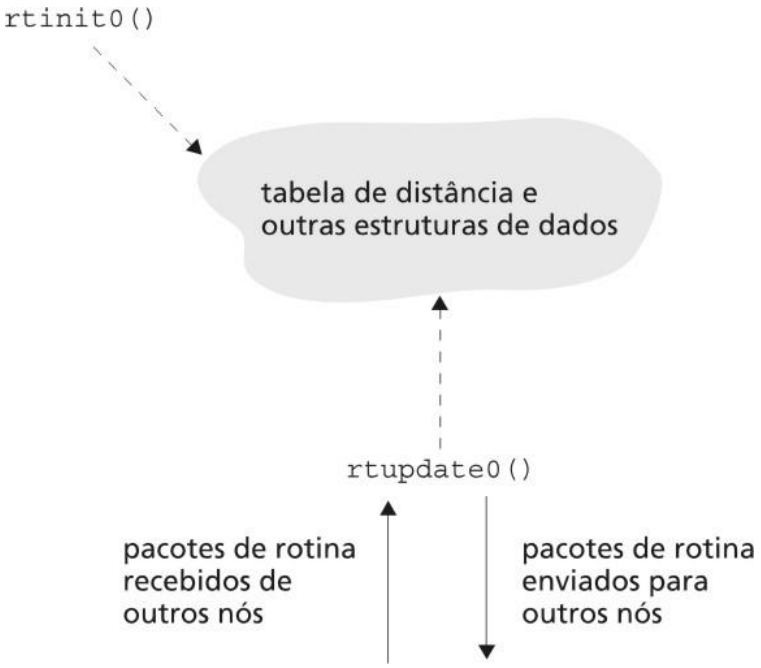
`rtupdate0()` é o "coração" do algoritmo de vetor de distâncias. O valor que ele recebe num pacote de rotina vindo de algum outro nó *i* contém o custo do caminho mais curto de *i* para todos os outros

nós da rede. `rtupdate0()` usa estes valores recebidos para atualizar sua própria tabela de distância (como especificado pelo algoritmo de vetor de distâncias). Se o seu próprio custo mínimo para outro nó mudar como resultado da atualização, o nó "0" informa a todos os vizinhos diretamente conectados sobre essa mudança em custo mínimo enviando para eles um pacote de rotina. Ressaltamos que no algoritmo de vetor de distâncias, apenas os nós diretamente conectados irão trocar pacotes de rotina. Portanto, os nós 1 e 2 se comunicam entre si, mas os nós 1 e 3 não.

Conforme vimos em aula, a tabela de distância dentro de cada nó é a principal estrutura de dados usada pelo algoritmo de vetor de distâncias. Você achará conveniente declarar a tabela de distância como uma disposição 4-por-4 de `int´`s, onde a entrada `[i,j]` na tabela de distância no nó "0" é o custo atual do nó "0" para o nó *i* pelo vizinho direto *j*. Se "0" não estiver diretamente conectado ao *j*, você pode ignorar essa entrada. Usaremos a convenção de que o valor inteiro 999 é infinito.

A Figura Lab.4-2 fornece uma visão conceitual do relacionamento dos procedimentos dentro do nó "0".

Rotinas similares são definidas para os nós 1, 2 e 3. Portanto, você escreverá 8 procedimentos ao todo: `rtinit0()`, `rtinit1()`, `rtinit2()`, `rtinit3()`, `rtupdate0()`, `rtupdate1()`, `rtupdate2()`, `rtupdate3()`



Lab.4-2: Relação entre procedimentos dentro do nó 0.

Interfaces de software

Os procedimentos descritos acima são os que você irá escrever. Escrevemos as seguintes rotinas que podem ser chamadas pelas suas rotinas:

`tolayer2(struct rtpkt pkt2send)`

onde `rtpkt` é a seguinte estrutura, que já está declarada para você. O procedimento `tolayer2()` é definido no arquivo prog3.c

```
extern struct rtpkt {
   int sourceid; /* id do nó que está enviando o pkt, 0, 1, 2 ou 3 */
   int destid /* id do roteador para o qual o pkt está sendo enviado (deve
   ser um vizinho imediato) */
   int mincost[4]; /* custo mínimo para o nó 0 ... 3 */
   };
```

Note que `tolayer2()` é passado como estrutura, não como um ponteiro para uma estrutura.

`printdt0()` imprimirá a tabela de distância para o nó "0". Ela é passada como um ponteiro para uma estrutura do tipo `distance_table`. `printdt0()` e a declaração da estrutura para a tabela de distância do nó "0" está declarada no arquivo `node0.c`. Rotinas de impressão similares estão definidas para você nos arquivos `node1.c`, `node2.c` e `node3.c`.

O ambiente de rede simulado

Seus procedimentos `rtinit0()`, `rtinit1()`, `rtinit2()`, `rtinit3()` e `rtupdate0()`, `rtupdate1()`, `rtupdate2()`, `rtupdate3()` enviam pacotes de rotina (cujo formato está descrito acima) dentro do meio. O meio entregará os pacotes em ordem e sem perda para o destinatário específico. Apenas nós diretamente conectados podem se comunicar. O atraso entre o remetente e o destinatário é variável (e desconhecido).

Quando você compilar seus procedimentos e meus procedimentos juntos e executar o programa resultante, deverá especificar apenas um valor relativo ao ambiente de rede simulado:

- **Tracing.** Ajustar um valor de tracing de 1 ou 2 imprimirá informações úteis sobre o que está acontecendo dentro da emulação (exemplo: o que ocorre com pacotes e temporizadores). Um valor de tracing de "0" desliga esta opção. Um valor maior do que 2 exibirá todos os tipos de mensagens de uso próprio para depuração do meu emulador. Um valor igual a 2 pode ser útil para você fazer o debug do seu código. Mantenha em mente que implementações reais não provêem tais informações sobre o que está acontecendo com seus pacotes.

Tarefa básica

Você escreverá os procedimentos `rtinit0()`, `rtinit1()`, `rtinit2()`, `rtinit3()` e `rtupdate0()`, `rtupdate1()`, `rtupdate2()`, `rtupdate3()` que juntos implementarão uma computação distribuída assíncrona das tabelas de distância para a topologia e os custos mostrados na Figura Lab.4-1.

Você deverá colocar seus procedimentos para os nós 0 até 3 em arquivos chamados `node0.c`, ... `node3.c`. Não é permitido declarar nenhuma variável global que seja visível fora de um dado arquivo C (exemplo: qualquer variável global que você definir em `node0.c`. pode ser acessada somente dentro do `node0.c`). Isso o forçará a cumprir convenções que deveriam ser adotadas se você fosse executar os procedimentos em quatro nós distintos. Para compilar suas rotinas: `cc prog3.c node0.c node1.c node2.c node3`. Versões de protótipos desses arquivos encontram-se em: node0.c, node1.c, node2.c, node3.c. Você pode adquirir uma cópia do arquivo prog3.c em: http://gaia.cs.umass.edu/kurose/network/prog3.c.

Para sua saída de amostra, seus procedimentos devem imprimir uma mensagem sempre que seus procedimentos `rtinit0()`, `rtinit1()`, `rtinit2()`, `rtinit3()` ou `rtupdate0()`, `rtupdate1()`, `rtupdate2()`, `rtupdate3()` forem chamados, fornecendo o tempo (disponível pela minha variável global `clocktime`). Para `rtupdate0()`, `rtupdate1()`, `rtupdate2()` e `rtupdate3()`, você deve imprimir a identidade do remetente do pacote de rotina que está sendo passado para sua rotina, se a tabela de distância for, ou não, atualizada, o conteúdo da tabela de distância (você pode usar minha rotina de impressão) e uma descrição de todas as mensagens enviadas para os nós vizinhos como resultado de cada atualização da tabela de distância.

A saída de amostra deve ser uma listagem de saída com um valor de TRACE igual a 2. Destaque a tabela de distância final produzida em cada nó. Seu programa será executado até que não haja mais nenhum pacote de rotina em trânsito na rede. Nesse ponto, o emulador terminará.