Tarefa de Programação: Implementando um Protocolo de Transporte Confiável

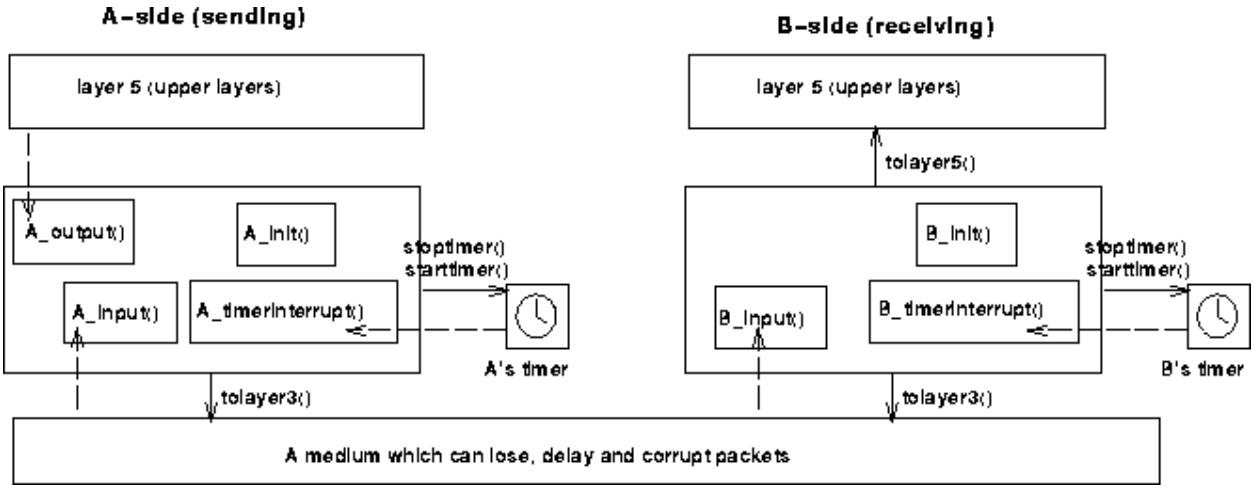
Visão Geral

Nesta tarefa de programação de laboratório, você escreverá o código de nível de transporte de envio e recebimento para implementar um protocolo de transferência de dados confiável e simples. Há duas versões deste laboratório: a versão com Protocolo de Bits Alternados e a versão com Go-Back-N. Este laboratório deve ser divertido, pois sua implementação diferirá muito pouco do que seria necessário em uma situação real.

Como você provavelmente não possui máquinas autônomas (com um sistema operacional que você possa modificar), seu código terá que ser executado em um ambiente de hardware/software simulado. No entanto, a interface de programação fornecida às suas rotinas, ou seja, o código que chamaria suas entidades de cima para baixo, é muito próxima do que é feito em um ambiente UNIX real. (De fato, as interfaces de software descritas nesta tarefa de programação são muito mais realistas do que os remetentes e destinatários de loop infinito descritos em muitos textos). A parada/inicialização de temporizadores também é simulada, e interrupções de temporizadores farão com que sua rotina de tratamento de temporizadores seja ativada.

As rotinas que você escreverá

Os procedimentos que você escreverá são para a entidade emissora (A) e a entidade receptora (B). Apenas a transferência unidirecional de dados (de A para B) é necessária. Obviamente, o lado B terá que enviar pacotes para A para confirmar (positiva ou negativamente) o recebimento dos dados. Suas rotinas devem ser implementadas na forma dos procedimentos descritos abaixo. Esses procedimentos serão chamados por (e chamarão) procedimentos implementados para emular um ambiente de rede. A estrutura geral do ambiente é mostrada na figura abaixo:



A unidade de dados transmitida entre as camadas superiores e seus protocolos é uma *mensagem*, que é declarada como:

```
struct msg {
      char data[20];
};
```

Esta declaração e todas as outras rotinas de estrutura de dados e emulador, bem como rotinas de stub (ou seja, aquelas que você deve completar) estão no arquivo prog2.c, descrito posteriormente. Sua entidade emissora receberá dados em blocos de 20 bytes da camada 5; sua entidade receptora deverá entregar blocos de 20 bytes de dados recebidos corretamente para a camada 5 no lado receptor.

A unidade de dados passada entre suas rotinas e a camada de rede é o pacote, que é declarado como:

```
struct pkt {
      int seqnum;
      int acknum;
      int checksum;
      char payload[20];
};
```

Suas rotinas preencherão o campo payload com os dados da mensagem passados da camada 5. Os outros campos do pacote serão usados por seus protocolos para garantir uma entrega confiável, como vimos em aula.

As rotinas que você escreverá são detalhadas abaixo. Como observado acima, tais procedimentos na vida real fariam parte do sistema operacional e seriam chamados por outros procedimentos no sistema operacional.

- **A_output(message)**, onde message é uma estrutura do tipo msg, contendo dados a serem enviados para o lado B. Esta rotina será chamada sempre que a camada superior no lado emissor (A) tiver uma mensagem para enviar. É função do seu protocolo garantir que os dados dessa mensagem sejam entregues em ordem e corretamente para a camada superior do lado receptor.

- **A_input(packet)**, onde packet é uma estrutura do tipo `pkt`. Esta rotina será chamada sempre que um pacote enviado do lado B (ou seja, como resultado de uma `tolayer3()` executada por um procedimento do lado B) chegar ao lado A. `packet` é o pacote (possivelmente corrompido) enviado do lado B.

- **A_timerinterrupt()** Esta rotina será chamada quando o temporizador de A expirar (gerando assim uma interrupção de temporizador). Você provavelmente desejará usar esta rotina para controlar a retransmissão de pacotes. Veja `starttimer()` e `stoptimer()` abaixo para saber como o temporizador é iniciado e parado.

- **A_init()** Esta rotina será chamada uma vez, antes de qualquer uma das suas outras rotinas do lado A ser chamada. Ela pode ser usada para realizar qualquer inicialização necessária.

- **B_input(packet)**, onde `packet` é uma estrutura do tipo `pkt`. Esta rotina será chamada sempre que um pacote enviado do lado A (ou seja, como resultado de uma `tolayer3()` executada por um procedimento do lado A) chegar ao lado B. `packet` é o pacote (possivelmente corrompido) enviado do lado A.

- **B_init()** Esta rotina será chamada uma vez, antes de qualquer uma das suas outras rotinas do lado B ser chamada. Ela pode ser usada para realizar qualquer inicialização necessária.

Interfaces de Software

Os procedimentos descritos acima são os que você escreverá. Escrevi as seguintes rotinas que podem ser chamadas pelas suas rotinas:

- **starttimer(calling_entity,increment)**, onde `calling_entity` é 0 (para iniciar o temporizador do lado A) ou 1 (para iniciar o temporizador do lado B), e increment é um valor ***float*** que indica o tempo que passará antes que o temporizador seja interrompido. O temporizador de A só deve ser iniciado (ou parado) por rotinas do lado A, e o mesmo vale para o temporizador do lado B. Para lhe dar uma ideia do valor de incremento apropriado a ser usado: um pacote enviado para a rede leva em média 5 unidades de tempo para chegar ao outro lado quando não há outras mensagens no meio.

- **stoptimer(calling_entity)**, onde `calling_entity` é 0 (para interromper o temporizador do lado A) ou 1 (para interromper o temporizador do lado B).

- **tolayer3(calling_entity, packet)**, onde `calling_entity` é 0 (para o envio do lado A) ou 1 (para o envio do lado B), e pacote é uma estrutura do tipo `pkt`. Chamar esta rotina fará com que o pacote seja enviado para a rede, com destino à outra entidade.

- **tolayer5(calling_entity, message)**, onde `calling_entity` é 0 (para entrega do lado A para a camada 5) ou 1 (para entrega do lado B para a camada 5), e mensagem é uma estrutura do tipo mensagem. Com transferência de dados unidirecional, você só chamaria esta rotina com `calling_entity` igual a 1 (entrega para o lado B). Chamar esta rotina fará com que os dados sejam passados para a camada 5.

O ambiente de rede simulado

Uma chamada ao procedimento `tolayer3()` envia pacotes para o meio (ou seja, para a camada de rede). Seus procedimentos `A_input()` e `B_input()` são chamados quando um pacote deve ser entregue do meio para a sua camada de protocolo.

O meio é capaz de corromper e perder pacotes. Ele não reordenará os pacotes. Ao compilar seus procedimentos e os meus procedimentos juntos e executar o programa resultante, você será solicitado a especificar valores referentes ao ambiente de rede simulado:

Número de mensagens a serem simuladas. Meu emulador (e suas rotinas) serão interrompidos assim que esse número de mensagens for passado da camada 5, independentemente de todas as mensagens terem sido entregues corretamente ou não. Portanto, você **não** precisa se preocupar com mensagens não entregues ou não confirmadas ainda no seu remetente quando o emulador for interrompido. Observe que, se você definir esse valor como 1, seu programa será encerrado imediatamente, antes que a mensagem seja entregue ao outro lado. Portanto, este valor deve ser sempre maior que 1.

Perda. Você deve especificar uma probabilidade de perda de pacotes. Um valor de 0,1 significa que um em cada dez pacotes (em média) são perdidos.

Corrupção. Você deve especificar uma probabilidade de perda de pacotes. Um valor de 0,2 significa que um em cada cinco pacotes (em média) são corrompidos. Observe que o conteúdo dos campos payload, sequence, ack ou checksum pode ser corrompido. Seu checksum deve, portanto, incluir os campos data, sequence e ack.

Rastreamento. Definir um valor de rastreamento de 1 ou 2 imprimirá informações úteis sobre o que está acontecendo dentro da emulação (por exemplo, o que está acontecendo com pacotes e temporizadores). Um valor de rastreamento de 0 desativará isso. Um valor de rastreamento maior que 2 exibirá todos os tipos de mensagens estranhas que são para meus próprios propósitos de depuração do emulador. Um valor de rastreamento de 2 pode ser útil para você na depuração do seu código. Você deve ter em mente que implementadores ***reais*** não possuem redes subjacentes que forneçam informações tão precisas sobre o que acontecerá com seus pacotes!

Tempo médio entre mensagens da camada 5 do remetente. Você pode definir esse valor como qualquer valor positivo diferente de zero. Observe que quanto menor o valor escolhido, mais rápido os pacotes chegarão ao remetente.

A versão do Protocolo de Bits Alternados deste laboratório.

Você deve escrever os procedimentos `A_output()`, `A_input()`, `A_timerinterrupt()`, `A_init()`, `B_input()` e `B_init()`, que juntos implementarão uma transferência unidirecional de dados do tipo stop-and-wait (ou seja, o protocolo de bits alternados, que chamamos de rdt3.0 no texto) do lado A para o lado B. **Seu protocolo deve usar mensagens ACK e NACK.**

Você deve escolher um valor muito alto para o tempo médio entre mensagens da camada 5 do remetente, para que seu remetente nunca seja chamado enquanto ainda tiver uma mensagem pendente e não confirmada que esteja tentando enviar ao destinatário. Sugiro que você escolha um valor de 1000. Você também deve realizar uma verificação em seu remetente para garantir que, quando `A_output()` for chamado, não haja nenhuma mensagem em trânsito. Se houver, você pode simplesmente ignorar (descartar) os dados que estão sendo passados para a rotina `A_output()`.

Você deve colocar seus procedimentos em um arquivo chamado prog2.c. Você precisará da versão inicial deste arquivo, contendo as rotinas de emulação que escrevemos para você e os stubs para seus procedimentos. Você pode obter este programa em http://gaia.cs.umass.edu/kurose/transport/prog2.c.

Este laboratório pode ser concluído em qualquer máquina com suporte a C. Ele não utiliza recursos do UNIX. (Você pode simplesmente copiar o arquivo prog2.c para qualquer máquina e sistema operacional de sua escolha).

Leia as "dicas úteis" deste laboratório após a descrição da versão Go_Back-N deste laboratório.

A versão Go-Back-N deste laboratório.

Você deve escrever os procedimentos `A_output()`, `A_input()`, `A_timerinterrupt()`, `A_init()`, `B_input()` e `B_init()`, que juntos implementarão uma transferência unidirecional de dados Go-Back-N do lado A para o lado B, com um tamanho de janela de 8. Seu protocolo deve usar mensagens ACK e NACK. Consulte a versão do protocolo de bits alternados deste laboratório acima para obter informações sobre como obter o emulador de rede.

Recomendamos **FORTEMENTE** que você implemente primeiro o laboratório mais fácil (Bit Alternado) e, em seguida, estenda seu código para implementar o laboratório mais difícil (Go-Back-N). Acredite em mim - não será perda de tempo! No entanto, algumas novas considerações para o seu código Go-Back-N (que não se aplicam ao protocolo de Bit Alternado) são:

- **A_output(mensagem)**, onde mensagem é uma estrutura do tipo `msg`, contendo dados a serem enviados para o lado B.

  Sua rotina `A_output()` agora será chamada às vezes quando houver mensagens pendentes e não confirmadas no meio — o que implica que você terá que armazenar em buffer várias mensagens no seu remetente. Além disso, você também precisará armazenar em buffer no seu remetente devido à natureza do Go-Back-N: às vezes, o remetente será chamado, mas não conseguirá enviar a nova mensagem porque ela está fora da janela.

  Em vez de se preocupar em armazenar em buffer um número arbitrário de mensagens, será aceitável ter um número máximo e finito de buffers disponíveis para o seu remetente (digamos, para 50 mensagens) e fazer com que o remetente simplesmente aborte (desista e saia) caso todos os 50 buffers estejam em uso em algum momento (Observação: usando os valores fornecidos abaixo, isso nunca deveria acontecer!). No ``mundo real'', é claro, seria necessário encontrar uma solução mais elegante para o problema do buffer finito!

- **A_timerinterrupt()** Esta rotina será chamada quando o temporizador de A expirar (gerando, assim, uma interrupção de temporizador). Lembre-se de que você tem apenas um temporizador e pode ter muitos pacotes pendentes e não confirmados no meio, então você terá que pensar um pouco sobre como usar este único temporizador.

Consulte a versão do protocolo de bits alternados deste laboratório acima para obter uma descrição geral do que você pode querer entregar. Você pode querer entregar a saída para uma execução longa o suficiente para que pelo menos 20 mensagens fossem transferidas com sucesso do remetente para o destinatário (ou seja, o remetente recebe ACK para essas mensagens), transferências, uma probabilidade de perda de 0,2 e uma probabilidade de corrupção de 0,2, um nível de rastreamento de 2 e um tempo médio entre chegadas de 10. Você pode anotar partes da sua impressão com uma caneta colorida mostrando como seu protocolo se recuperou corretamente da perda e corrupção de pacotes.

Dicas úteis e similares

- **Soma de verificação.** Você pode usar qualquer abordagem de soma de verificação que desejar. Lembre-se de que o número de sequência e o campo ack também podem ser corrompidos. Sugerimos uma soma de verificação semelhante à do TCP, que consiste na soma dos valores da sequência (inteiro) e do campo ack, somados

a uma soma caractere por caractere do campo payload do pacote (ou seja, trate cada caractere como se fosse um inteiro de 8 bits e simplesmente some-os).

- Observe que qualquer "estado" compartilhado entre suas rotinas precisa estar na forma de variáveis globais. Observe também que qualquer informação que seus procedimentos precisem salvar de uma invocação para a próxima também deve ser uma variável global (ou estática). Por exemplo, suas rotinas precisarão manter uma cópia de um pacote para possível retransmissão. Provavelmente seria uma boa ideia que essa estrutura de dados fosse uma variável global no seu código. Observe, no entanto, que se uma das suas variáveis globais for usada pelo remetente, essa variável **NÃO** deverá ser acessada pela entidade do destinatário, pois, na vida real, entidades comunicantes conectadas apenas por um canal de comunicação não podem compartilhar variáveis globais.

- Há uma variável global ***float*** chamada ***time*** que você pode acessar de dentro do seu código para ajudá-lo com suas mensagens de diagnóstico.

- **COMECE SIMPLES**. Defina as probabilidades de perda e corrupção como zero e teste suas rotinas. Melhor ainda, projete e implemente seus procedimentos para o caso de nenhuma perda e nenhuma corrupção e faça-os funcionar primeiro. Em seguida, trate o caso de uma dessas probabilidades ser diferente de zero e, finalmente, ambas serem diferentes de zero.

- **Depuração**. Recomendamos que você defina o nível de rastreamento como 2 e coloque MUITOS printfs no seu código enquanto depura seus procedimentos.

- **Números Aleatórios.** O emulador gera perdas de pacotes e erros usando um gerador de números aleatórios. Nossa experiência anterior mostra que os geradores de números aleatórios podem variar bastante de uma máquina para outra. Pode ser necessário modificar o código de geração de números aleatórios no emulador que fornecemos. Nossas rotinas de emulação têm um teste para verificar se o gerador de números aleatórios da sua máquina funcionará com o nosso código. Se você receber uma mensagem de erro:

  > *It is likely that random number generation on your machine is different from what this emulator expects. Please take a look at the routine jimsrand() in the emulator code. Sorry.*

  Então você saberá que precisará observar como os números aleatórios são gerados na rotina `jimsrand()`; veja os comentários nessa rotina.

Perguntas e Respostas

Confira o fórum em caso de dúvidas:

http://gaia.cs.umass.edu/kurose/transport/programming_assignment_QA.htm