O protocolo de enlace

Site: 2024/2

Curso: 2024.2 - PROJETO DE PROTOCOLOS - Turma 01

Livro: O protocolo de enlace

Impresso por: WAGNER FLORES DOS SANTOS

Data: Friday, 7 Feb 2025, 10:09

Descrição

Este livro contém a descrição do protocolo de enlace, e dos mecanismos que podem ser usados em seu desenvolvimento. Ele lhe ajudará com os conhecimentos necessários durante o desenvolvimento do seu protocolo.

Índice

1. Introdução

Imagine que um equipamento configurável seja instalado remotamente, e precise ser monitorado ou mesmo acessado e controlado à distância. Uma situação dessa pode existir com equipamentos de comunicação, câmeras e sistemas de vigilância eletrônica, por exemplo. No nascente mundo de IoT (Internet das Coisas), algo assim pode também ser uma necessidade relacionada a dispositivos quaisquer, como geradores em PCH (*Pequenas Centrais Hidrelétricas*), sistemas de refrigeração, sistemas de distribuição de energia elétrica, fazendas eólicas, e mesmo boias marítimas para sensoriamento ambiental. Sendo comum equipamentos possuirem algum sistema embarcado com razoável poder de processamento, e serem configuráveis e monitoráveis, o acesso remoto traz uma facilidade para sua operação, monitoramento e manutenção.

Uma boia marítima sendo acessada remotamente a partir de uma embarcação .. ver também este video

O acesso a um equipamento assim, à distância, pode ser feito por um enlace sem-fio. Supondo que o acesso ao equipamento seja feito para transferir pequenas quantidades de dados, o uso de uma tecnologia de comunicação sem-fio com baixa taxa de dados pode ser apropriada. Isso é razoável, pois é comum esses equipamentos apresentarem interfaces de gerenciamento orientadas a terminal, com comandos textuais (ex: CLI - Command Line Interface), as quais são práticas e exigem poucos recursos de memória, processamento e comunicação. Por isso pode ser mais relevante uma tecnologia de comunicação com maior alcance do que maior taxa de dados.

Nosso estudo nesta unidade se guiará por um cenário desse tipo. O objetivo é desenvolvermos um protocolo de enlace para acesso remoto a equipamentos, usando um canal sem-fio. Nosso protocolo deve servir como camada de enlace em uma pilha de protocolos TCP/IP. Isso significa que a implementação do protocolo deve ser capaz de se integrar à pilha de protocolos mantida pelo sistema operacional. Assim, se o protocolo for desenvolvido em um sistema Linux, ele deve ser acessado por meio de uma interface de rede. Isso torna possível o uso de aplicações típicas TCP/IP com o protocolo, tais como telnet e ssh (úteis para acesso a CLI).

Vamos então especificar nosso protocolo, e para isso precisamos definir precisamente suas características. Isso envolve também revisar mecanismos típicos de protocolos, para identificarmos quais deles serão aplicados no protocolo.

1.1. O serviço do protocolo de enlace

O protocolo de enlace a ser desenvolvido se destina a implantar enlaces ponto-a-ponto usando um canal sem-fio. A camada física a ser utilizada tem certas particularidades, como baixa taxa de dados, médio alcance, e ausência de detecção de acesso ao meio.

Serviço:

- protocolo de enlace ponto-a-ponto, para camada física do tipo UART
- encapsulamento de mensagens com até 1024 bytes
- recepção de mensagens livres de erros
- garantia de entrega
- controle de acesso ao meio
- conectado (estabelecimento de sessão)

2. A camada física

Antes de especificarmos o protocolo, precisamos primeiro identificar algumas opções de transceivers RF a serem usadas como referência para sua camada física. Existem algumas opções no mercado, voltadas em especial para IoT.

RF APC 220

O transceiver RF APC 220 oferece taxas configuráveis de 1200 a 19200 bps, com alcance de até 1000 m em campo aberto. No caso de distâncias como essa, a taxa de transmissão possível de ser obtida é de 2400 bps, porém distâncias menores possibilitam taxas maiores, até o máximo de 19200 bps. Ele opera na banda de 418 a 455 MHz, com canais que podem ser selecionados. Ele pode ser usado com um adaptador USB, o que facilita sua integração a um computador ou RaspberryPi. Também é possível usá-lo diretamente com uma interface TTL, o que simplifica sua integração com Arduino. Portanto, com ele podem-se criar enlaces de média distância e baixas taxas de transmissão.

Um par de transceivers APC 220, com o adaptador USB destacado à direita

Esse transceiver é acessado programaticamente como se fosse uma interface serial. Assim, o envio e recepção de dados é feito como escritas e leituras em uma interface serial do tipo UART. Nenhuma facilidade para delimitação de mensagens, endereçamento, sincronização e tratamento de erros, ou controle de acesso ao meio (MAC) é fornecida. De fato, tais serviços devem ser implementados em um protocolo de enlace que use esse transceiver como camada física.

- Datasheet do APC 220
- Manual do APC 220
- Um tutorial sobre APC 220

NRF24L01

O transceiver NRF24L01, fabricado pela Nordic, é uma opção de comunicação sem-fio entre dispositivos como Arduino, PIC, Raspberry, BeagleBone, dentre possivelmente outros. Ele apresenta alcance de 50 metros em campo aberto, que pode ser melhorado se for usada uma antena apropriada. Seu canal de comunicação opera na banda de 2,4 GHz, e proporciona taxas de até 2 Mbps. Ao contrário do APC 220, o NRF24L01 implementa uma forma de enquadramento, com quadros de até 31 octetos e detecção de erros por CRC. O módulo NRF24L01 possui um firmware que possibilita a comunicação multiponto com até 6 nodos, implementando um esquema de endereçamento. Ele consegue retransmitir quadros em caso de falha, o que simplifica a implementação de protocolos de enlace.

Os módulos NRF24L01 disponíveis em mercado for projetados para uso em sistemas embarcados, e não como interfaces de comunicação de computadores. No entanto, é possível usar um adaptador USB para conectá-lo a um computador. O uso dele por meio da interface USB tem limitações, em especial na seleção de modos de operação (ver datasheet) mas é possível transmitir quadros.

- Datasheet NRF24L01 (e um datasheet resumido)

2.1. Acesso programático à interface serial

Uma interface serial é vista no sistema operacional Linux como um dispositivo de entrada e saída orientado a caractere. Isso significa que leituras e escritas nesse dispositivo são feitos como fluxos de dados, sem possibilidade de endereçar um dado específico. Dito de outra forma, não se consegue fazer acesso randômico a dados, ao contrário de discos rígidos ou SSD, em que se pode acessar um bloco específico qualquer ali armazenado.

No sistema operacional, o acesso a um dispositivo de entrada e saída se faz com chamadas típicas de arquivo. Isso é possível porque tais dispositivos são representados dentro do sistema operacional com uma interface de arquivos (uma abstração existente em sistemas da família Unix). As operações básicas de arquivos são:

- *open* e *close*: para iniciar e encerrar o acesso ao dispositivo
- *read* e *write*: para ler e escrever dados dos dispositivo
- *ioctl*: para configurar parâmetros operacionais, ou enviar comandos ao dispositivo

Existem alguns detalhes sobre como acessar e usar interfaces seriais em programas. Há muitos parâmetros e modos de operação que podem ser selecionados nessas interfaces. Para isso, existem APIs elementares disponíveis no sistema operacional, como *termios*, que auxiliam a escrita de programas. Dois bons tutoriais sobre a API *termios* podem ser vistos nestas páginas:

- Serial Programming HOWTO
- Serial Programming Linux

A API *termios* oferece todos os detalhes para acesso a interfaces seriais, porém tem um baixo nível de abstração. Por isso existem outras APIs mais simplificadas, que facilitam a escrita de programas que acessam seriais. Por outro lado, tais APIs não oferecem a versatilidade de *termios*.

Uma API simples para C++

Em C++, uma API razoavelmente simples foi desenvolvida para uso na disciplina de Projeto de Protocolos. Essa API possibilita acessar portas seriais por meio de instâncias da classe Serial. Essa classe implementa operações de envio e recepção de octetos, além de possibilitar acesso direto ao descritor da porta serial. Além disso, as operações de recepção podem ser feitas em modo bloqueante ou não-bloqueante.

- SerialPP: repositório no github com arquivos de código-fonte dessa API

A declaração da classe Serial pode ser vista a seguir:

```
class Serial {
public:
    Serial();

    // cria um objeto Serial associado a uma porta serial.
    // "path": path do dispositivo serial (ex: /dev/ttyS0)
    // "rate": taxa de transmissão (ex: B9600 ... ver termios.h)
    Serial(const string & path, int rate);

    // cria um objeto serial associado a um arquivo para fins de debug
    Serial(const string & path); // abre um arquivo ao invés da serial

    // construtor de cópia
    Serial(const Serial& orig);

    virtual ~Serial();

    // obtém o descritor de arquivo da porta serial
    int get() const;

    // envia dados pela serial
    int write(const char * buffer, int len);

    // lê até "len" dados da serial. Versão não-bloqueante
    int read(char * buffer, int len);

    // lê até "len" dados da serial.
    // "e block for true: bloqueia à espera de receber pelo menos um byte. Esse bloqueio
    // é limitado por um timeout, caso tenha sido especificado.
    // se block false: não-bloqueante
    int read(char * buffer, int len, bool block);

    // lê um único byte em modo bloqueante (também limitado por timeout)
    char read_byte();

    // define o valor de timeout. Se for "0" (zero), desativa o timeout
    // Valor inicial de timeout é "0"
    void set_timeout(int to); // timeout em milissegundos
};
```

Com essa API, podem-se escrever programas que se comunicam pela porta serial. Veja uma demonstração de uso com estes dois programas:

- O primeiro deles envia uma única mensagem de texto por uma porta serial. O nome da porta serial deve ser informado pelo primeiro argumento de linha de comando (ex: /dev/ttyUSB0 ou /dev/pts/1)
- O segundo recebe uma mensagem de texto, e apresenta-a na tela

```
#include <iostream>
#include "Serial.h"

using namespace std;

int main(int argc, char * argv[]) {
  // cria um objeto Serial para acessar a porta serial indicada em argv[1],
  // com taxa de 9600 bps
  Serial rf(argv[1], B9600);

  // esta é a mensagem a ser transmitida
  string msg = "um teste ...\r\n";
  vector<char> buffer(msg.begin(), msg.end());

  // envia a mensagem
  int n = rf.write(buffer);

  cout << "Enviou " << n << " bytes" << endl;

  cout << "Tecle ENTER para encerrar o programa:";
  getline(cin, msg);

  cout << endl;
}
```

O programa transmissor.

```
#include <iostream>
#include "Serial.h"

using namespace std;

int main(int argc, char * argv[]) {
  // cria um objeto Serial para acessar a porta serial indicada em argv[1],
  // com taxa de 9600 bps
  Serial rf(argv[1], B9600);

  // recebe até 32 octetos
  auto buffer = rf.read(32);

  cout << "Recebeu " << buffer.size() << " bytes: ";

  // mostra na tela os caracteres recebidos
  cout.write(buffer.data(), buffer.size());

  cout << endl;
}
```

O programa receptor.

Uma API Python

Existe uma API Python para acesso a portas seriais, chamada pyserial. Ela possibilita também o acesso programático de forma simplificada a portas seriais. Para usá-la, use pip3 para instalá-la:

```
sudo pip3 install pyserial
```

Veja uma tradução do transmissor e receptor mostrados como exemplo, no caso da API Serial para C++.

```
#!/usr/bin/python3

from serial import Serial
import sys

try:
  porta = sys.argv[1]
except:
  print('Uso: %s porta_serial' % sys.argv[0])
  sys.exit(0)

try:
  p = Serial(porta, 9600, timeout=10)
except Exception as e:
  print('Não conseguiu acessar a porta serial', e)
  sys.exit(0)

msg = 'um teste ...\r\n'

n = p.write(msg.encode('ascii'))
print('Enviou %d bytes' % n)

input('Digite ENTER para terminar:')

sys.exit(0)
```

O transmissor em Python.

```
#!/usr/bin/python3

from serial import Serial
import sys

try:
  porta = sys.argv[1]
except:
  print('Uso: %s porta_serial' % sys.argv[0])
  sys.exit(0)

try:
  p = Serial(porta, 9600, timeout=10)
except Exception as e:
  print('Não conseguiu acessar a porta serial', e)
  sys.exit(0)

# recebe até 128 caracteres
msg = p.read(128)
print('Recebeu: ', msg)

sys.exit(0)
```

O receptor em Python

2.2. Emulador de link serial

O uso de transceivers pode dificultar o início do desenvolvimento do protocolo. Um emulador de link serial facilita o desenvolvimento, por abstrair particularidades dos transceivers. O programa **serialemu** emula um link serial com determinada taxa de bits, BER e atraso de propagação. Para usá-lo, deve-se fazer o seguinte:

1. Obtenha o código-fonte do serialemu neste repositório no github. Ele foi publicado a partir do CLion, que usa o cmake.
2. Compile-o desta forma usando o programa cmake:

   ```
   cd SerialEmu
   cmake CMakeLists.txt
   make
   ```

   1. Caso o comando acima não funcione (porque o cmake não está instalado, por exemplo), você pode compilá-lo manualmente:

      ```
      g++ -o serialemu *.cpp -lpthread -lutil -std=c++11
      ```

3. Copie o programa compilado para algum subdiretório mais conveniente ... por exemplo, */home/aluno*, e depois mude para esse subdiretório:

   ```
   aluno@M2:~/CLionProjects/Serialemu$ cp -a serialemu /home/aluno/
   aluno@M2:~/CLionProjects/Serialemu$ cd /home/aluno
   aluno@M2:~$
   ```

4. Execute-o de forma que ele apresente suas opções de execução:

   ```
   aluno@M2:~$ ./serialemu -h
   Uso: ./serialemu [-b BER][-a atraso][-f][-B taxa_bits] | -h

   BER: taxa de erro de bit, que deve estar no intervalo  [0,1]
   atraso: atraso de propagação, em milissegundos.
   taxa_bits: taxa de bits em bits/segundo
   -f: executa em primeiro plano (nao faz fork)
   ```

5. Execute-o então da forma desejada, selecionando a taxa de bits (default: ilimitada), BER (default: 0) e atraso de propagação (default: 0). O serialemu automaticamente vai para segundo plano (*daemonize*), liberando o terminal. Ex:

   ```
   aluno@M2:~$ ./serialemu -B 9600
   /dev/pts/17 /dev/pts/2
   aluno@M2:~$
   ```

   ... e anote os dois caminhos informados pelo *serialemu*: eles são as duas portas seriais que correspondem às pontas do link serial emulado.
6. Execute seu protocolo usando essas portas seriais virtuais.

OBS: você pode testar o serialemu usando os programas Python de exemplo sobre envio e recepção pela serial. Supondo que o serialemu criou as portas seriais */dev/pts/1* e */dev/pts/2*, e que o transmissor seja *tx.py*, e o receptor seja *rx.py*, você pode fazer assim:

1. Em um terminal execute o receptor:

   ```
   python3 rx.py /dev/pts/1
   ```

2. Em outro terminal execute o transmissor

```
python3 tx.py /dev/pts/2
```

Mesma que se opte pelo uso do emulador de serial, deve-se notar que, ao final, **o protocolo deve ser usado na plataforma Linux com o transceiver RF**. Assim, o uso do emulador de serial tem por finalidade somente facilitar o desenvolvimento dos mecanismos básicos do protocolo.

3. Mecanismos de protocolos

Um protocolo aplica alguns mecanismos para oferecer o serviço especificado. Por mecanismos entendem-se funcionalidades relacionadas a como as comunicações acontecem no protocolo.

3.1. Enquadramento

Uma das funções mais elementares de um protocolo de enlace é encapsular dados em mensagens, transmitindo-as e recebendo-as pela camada física. O protocolo de camada superior, que usa o serviço definido pelo protocolo de enlace, usa como unidade de comunicação o conceito de mensagem (cujo nome depende da terminologia adotada pelo protocolo em questão, tal como pacote ou datagrama). Uma mensagem é formada por uma certa quantidade de octetos, os quais podem estar estruturados de alguma forma para representar informações do protocolo. Do ponto de vista do protocolo de camada superior, mensagens são enviadas e recebidas como um todo. Assim, o serviço do protocolo de enlace deve, no mínimo, ser capaz de enviar e receber unidades de dados formadas por tais sequências de octetos.

O **enquadramento** (*framing*) é uma função do protocolo de enlace responsável por delimitar quadros na interface com a camada física. Deve-se ter em mente que, dependendo da camada física, o envio e recepção de sequências de octetos pode ou não ser estruturada. No caso de não estruturada, cada octeto é enviado ou recebido de forma independente. Se for estruturada, octetos são enviados e recebidos em blocos, que não necessariamente são convenientes para a camada de enlace (ex: tamanhos dos blocos). Por isso cabe à camada de enlace delimitar as unidades de dados de protocolo (PDU) dentro das sequências de octetos.

Existe mais de uma abordagem para delimitar quadros, como pode ser lido em mais detalhes no capítulo 11 de *Data Communications and Computer Networks*, de Behoruz Forouzan, e capítulo 5 de *Redes de Computadores e a Internet*, de James Kurose e Keith Ross. A tabela a seguir resume quatro abordagens clássicas para enquadramento.

Abordagem	Descrição	Exemplos
Quadros de tamanho fixo / duração definida	Quadros têm sempre mesmo comprimento ou duração	ATM, TDMA-based
Sentinela	Padrão de bits/bytes delimita quadros	PPP, HDLC
Contador / duração	Cabeçalho contém duração ou comprimento do quadro	IEEE 802.11
Presença/ausência de portadora	Ausência de portadora delimita quadros	IEEE 802.11, IEEE 802.3

A maioria dessas abordagens exploram ou dependem de características específicas da camada física. Por exemplo, *presença/ausência de portadora* significa que a existência de um sinal decodificável corresponde a um quadro em transmissão. A detecção de portadora e decodificação de sinal, com a recepção dos octetos em bloco, é feita pela camada física. Nesse caso, a camada de enlace não precisa ter a funcionalidade de enquadramento.

Os casos citados não são exaustivos. Novas tecnologias de camada física podem apresentar outros requisitos para a camada de enlace. Por exemplo, a camada física em redes GPON estrutura as transmissões em janelas de 125 us, cabendo aos equipamentos (OLT e ONU) usarem esse tempo para transmitirem dados. Com uma taxa de 2.5 Gbps, é possível transmitir pouco mais de 300 kB. A camada de enlace nesse tipo de rede envia múltiplos quadros em cada janela concedida, delimitando-os com uma abordagem de contador. Assim, cabe à camada de enlace transmitir quantos quadros forem possíveis dentro da janela de tempo disponível.

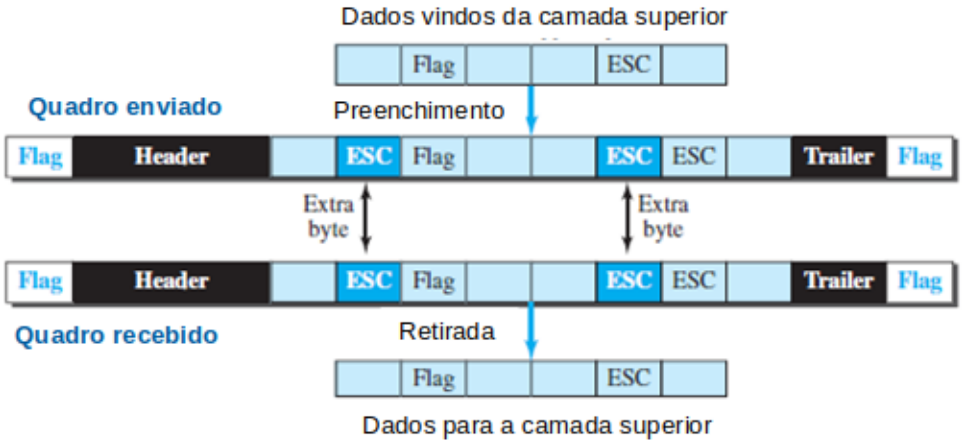
Diferente de GPON, outras tecnologias podem realizar transmissões em unidades de tamanho fixo, contendo pequenas quantidades de octetos. Por exemplo, em redes ATM as transmissões ocorrem em unidades de 48 octetos. O transceiver RF NRF24L01, mostrado no capítulo sobre camada física, transmite *frames* de 31 octetos.

Uma camada de enlace cujos quadros tenham tamanhos na casa de centenas de octetos, ou mais, terá que fragmentar seus quadros para enviá-los e recebê-los usando tais tecnologias. Isso implica a necessidade de identificar o primeiro e último *frame* de um quadro, para que possa ser remontado corretamente pelo receptor.

De forma geral, o enquadramento envolve a adaptação entre a estruturação dos dados transmitidos apresentada pela camada física, e a estrutura dos quadros conforme definido no serviço do protocolo de enlace.

Sentinela (ou preenchimento de byte)

Dentre as abordagens, apenas *sentinela* depende exclusivamente da camada de enlace. Essa abordagem, que usa padrões de bits ou valores especiais de octetos, historicamente é usada em comunicação do tipo serial síncrona ou assíncrona. Ela usa um octeto especial, chamado de *Flag*, para demarcar o início e fim de quadro. O procedimento desta técnica pode ser resumido na figura a seguir. Ela mostra as etapas desde a transmissão até a recepção.



A sequência de transmissão e recepção de um quadro pode ser descrita da seguinte forma, de acordo com a figura:

1. A camada superior envia uma PDU (*Packet Data Unit*) para a camada de enlace. Essa PDU, formada por múltiplos octetos, pode ter um conteúdo arbitrário.
2. A camada de enlace encapsula essa PDU em um quadro (*frame*), Isso envolve adicionar um cabeçalho (header) e possivelmente um sufixo (trailer) contendo informações necessárias ao protocolo de enlace.
3. A camada de enlace analisa o conteúdo do quadro em busca de octetos cujos valores tenham significado especial para a função de enquadramento. Em específico, octetos com valores *Flag* e *ESC* não podem aparecer dentro do quadro nesta etapa, pois eles têm um significado especial para que o receptor consiga delimitar o quadro e recebê-lo corretamente. O transmissor realiza o procedimento de **preenchimento de byte** para cada octeto especial encontrado. Esse procedimento envolve inserir um octeto com valor ESC antes do octeto especial, e então mudar o valor desse octeto. A ideia é esconder o octeto especial, para que não seja interpretado de forma indevida pelo receptor.
4. Após o preenchimento de byte, a camada de enlace insere octetos Flag no início e no final do quadro, e então o transmite (na prática, a camada de enlace transmite a *Flag*, seguida dos octetos do quadro, e da Flag *final*).
5. O receptor identifica o início de um quadro ao receber uma *Flag*. Ele passa a então a receber os octetos do quadro.
6. O receptor, durante a recepção do conteúdo do quadro, faz o processo reverso ao preenchimento. Sempre que encontrar um octeto com valor *ESC*, ele o descarta e então recupera o valor original do octeto seguinte.
7. O receptor termina a recepção do quadro ao receber uma nova *Flag*. Ele então interpreta o cabeçalho e o sufixo e, se o quadro for válido, desencapsula seu conteúdo e entrega à camada superior.

3.2. Detecção de erros

Comunicações estão sujeitas a erros, tais como mensagens corrompidas ou mesmo perdidas. Isso se deve a diferentes fatores, como características do tipo de canal de comunicação utilizado e limitações do hardware ou software (ver uma boa discussão no capítulo 3 do livro de Georg Holzmann). Em um protocolo de enlace, que se comunica diretamente por meio da camada física, a ocorrência de erros deve ser prevista e devidamente tratada.

Em um protocolo, o controle de erros envolve:

1. **Detecção de erros:** cada mensagem recebida deve ter verificada a integridade de seu conteúdo. Mensagens corrompidas devem ser rejeitadas (descartadas). Com isso, o protocolo aceita somente mensagens cujos conteúdos na recepção sejam idênticos àqueles transmitidos (dentro de certos limites).
2. **Recuperação de erros:** mensagens descartadas devido a erros podem ser recuperadas. Existem duas abordagens:
   - *Correção de erros:* mensagens podem incluir bits de redundância que possibilitam, dentro de certos limites, corrigir bits corrompidos. Esse tipo de técnica não costuma ser usada em nível de enlace, devido ao seu custo dado pela quantidade de bits de redundância para que sejam efetivas.
   - *Retransmissão:* mensagens descartadas são simplesmente retransmitidas. Isso é justificado pela (suposta) baixa frequência de erros de transmissão, o que torna menos custoso retransmitir mensagens que, eventualmente, sofram algum tipo de erro.

O protocolo de enlace a ser desenvolvido deve receber apenas mensagens corretas. Para isso, pelo menos deve ser implementada a funcionalidade de detecção de erros.

Detecção de erros

Durante uma transmissão, uma mensagem pode ser corrompida, pois bits podem ter seus valores invertidos ou serem mesmo suprimidos. Para ter uma ideia da chance disso ocorrer, deve-se tomar como base a taxa de erro de bit do canal de comunicação, apelidada de BER (*Bit Error Rate*). Assim, se BER é a probabilidade de um bit sofrer um erro, a probabilidade de um quadro de tamanho F bits não sofrer erros é:

$$P_F = (1 - BER)^F$$

Por exemplo se BER for 10^{-6}, e *F* for 1500 bytes (ou 12000 bits), a probabilidade de um quadro não ter erros é:

$$P_F = (1 - 10^{-6})^{12000} = 0.9881$$

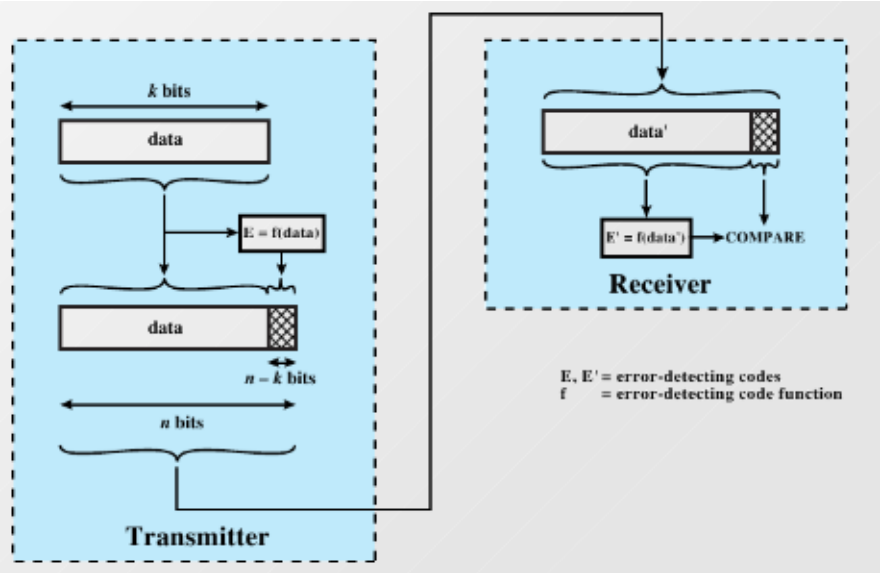
... o que não é muito bom: entre 1 e 2% dos quadros, em média, terão erros, uma taxa razoavelmente alta. Porém, se BER for 10^{-10}:

$$P_F = (1 - 10^{-10})^{12000} = 0.9999988$$

... 0,012% dos quadros, em média, sofrerão erros, o que é bem melhor.

Essa análise mostra que erros de transmissão são possíveis. O protocolo de enlace deve ser capaz de detectá-los, para evitar desencapsular o conteúdo de um quadro corrompido. Assim, ele facilita o trabalho de protocolos de camadas superiores.

A detecção de erros implica adicionar à mensagem transmitida uma certa quantidade de bits de redundância. Os valores desses bits são determinados com base no conteúdo da mensagem. Na recepção, calcula-se novamente o valor dos bits de redundância e compara-se com os bits de redundância incluídos na mensagem. Se forem iguais, presume-se que a mensagem esteja correta. A figura a seguir ilustra o processo desde a geração dos bits de redundância, sua inclusão na mensagem, e sua verificação na recepção.



Bits de redundância para detecção de erros (FONTE: William Stallings. Data Communications, 5th ed.)

A figura mostra que, em uma mensagem com k bits, são anexados alguns bits de redundância. Esses bits adicionais são gerados por uma função, que os calcula a partir da mensagem original. A mensagem transmitida, assim, possui *n* bits, o que corresponde aos *k* bits dos dados originais acrescidos dos bits de redundância. No receptor, cada mensagem recebida é verificada quanto à sua integridade. A mesma função usada pelo transmissor é aplicada sobre os dados da mensagem recebida, e seu resultado é comparado com os bits de redundância nela contidos. Se forem iguais, a mensagem é considerada correta e, portanto, aceita.

Uma técnica de detecção de erros, não importa o quão boa seja, está sujeita a aceitar mensagens corrompidas como se fossem corretas. Por outro lado, o contrário jamais deve ocorrer: recusar uma mensagem correta. Mensagens com erros podem passar incólumes, o que se chama *falso positivo*, porque a quantidade de bits de redundância é menor que a quantidade de bits de dados (com raras exceções). Por exemplo, usando uma técnica hipotética, uma mensagem pode ter 128 bits de dados, e 16 de redundância. Isso significa que existem 2^{16} possíveis valores de redundância que podem ser obtidos, porém a mensagem pode ter 2^{128} diferentes valores. Isso significa que cada valor de redundância pode corresponder a 2^{112} diferentes sequências de bits de dados. Se uma mensagem, ao ser transmitida, tiver alguns bits invertidos de forma que a mensagem recebida tenha uma sequência de bits que gere o mesmo valor de redundância, o erro passará despercebido. Sendo assim, as técnicas de detecção de erros devem ser projetadas de forma que seja muito improvável que isso aconteça. Para isso, devem-se identificar as características de erros de transmissão.

Erros podem acontecer aleatoriamente, porém não afetam bits individuais de uma mensagem de forma independente. Erros costumam afetar conjuntos de bits próximos, o que se chama de erro em rajada. Isso faz sentido, pois uma causa relevante de erros é interferência por outros sinais, ou mesmo ruídos, que possuem alguma duração. Assim, uma técnica de detecção de erros eficiente deve levar em conta não só os valores dos bits a serem transmitidos, mas também suas posições. Por exemplo, se uma mensagem transmitir a sequência A0 A1, mas a mensagem recebida contiver A1 A0, a função de detecção de erro deve ser capaz de identificar que houve um erro.

A detecção de erros em um protocolo envolve basicamente a escolha de uma dentre duas técnicas:

- **Checksum (soma de verificação):** os bits de redundância são calculados como uma soma de todos os octetos da mensagem. Essa técnica é simples e fácil de implementar, porém não leva em conta a posição dos octetos. Usando o exemplo em que se transmite a sequência A0 A1, e se recebe A1 A0, uma técnica do tipo *checksum* não conseguiria identificar o erro. Algumas variações buscam reduzir essa deficiência, realizando somas de grupos de octetos, de forma a reduzir a probabilidade de que sequências com posições trocadas passem sem serem detectadas.
- **CRC (verificação de redundância cíclica):** os bits de redundância são calculados usando uma forma de álgebra polinomial. Essa técnica é largamente usada em protocolos de enlace, pois leva em conta não só o valor, mas também a posição dos bits de dados. Em contrapartida, ela depende de um parâmetro, chamado de polinômio gerador, cuja definição é complexa (ver uma discussão no livro *Redes de Computadores*, de Andrew Tanenbaun).

CRC

A detecção de erros com CRC envolve realizar uma divisão entre os bits da mensagem e um certo padrão de bits. Essa divisão é feita em módulo 2, com operações XOR para subtração durante a divisão. Para fins de demonstração da validade dessa técnica, define-se:

- **M**: mensagem com *n* bits
- **F**: *k* bits de redundância a serem anexados à mensagem
- **T**: quadro a ser transmitido, contendo n+k bits
- **P**: padrão com *k+1* bits a serem usados na divisão

Para começar, deseja-se que, na recepção, T/P não tenha resto, sendo que T é definido por:

$$T = 2^k M + F$$

Para testar essa condição, primeiro faz-se a divisão de $2^k M$ por P:

$$2^k M / P = Q + R/P$$

... sendo Q o quociente e R o resto da divisão. Sendo uma divisão binária, R tem no mínimo 1 bit a menos que P. Usa-se R como os bits de CRC:

$$T = 2^k M + R$$

Agora deve-se testar se usar R como bits de CRC satisfaz a condição de T/P ter resto zero:

$$T/P = (2^k M + R)/P = Q + R/P + R/P$$

Como qualquer número somado a si próprio em aritmética de módulo 2 é zero, o resultado pode ser escrito assim:

$$T/P = (2^k M + R)/P = Q + (R + R)/P = Q$$

... e portanto não há resto !

DICA: CRC16 em Python

Existem módulos Python para cálculo de CRC, tais como PyCRC e crcmod. No entanto, é difícil acertar os parâmetros para que eles calculem os mesmos valores de CRC que a versão do algoritmo da RFC 1662 (CRC do PPP). Sendo assim, foi implementada uma classe para calcular CRC16 que segue fielmente o algoritmo dessa RFC. Essa classe está contida no módulo *crc*:

- Módulo crc para Python3

Abaixo segue a descrição dessa classe gerada com pydoc:

```
    class CRC16(builtins.object)
     |  Classe CRC16: calcula e verifica FCS com base no algoritmo
     |  de CRC descrito na RFC 1662 (PPP)
     |
     |  Methods defined here:
     |
     |  __convert__(self, data)
     |      Converte os dados para um objeto bytes
     |
     |  __init__(self, data=b'')
     |      data: contém os dados para calcular o FCS. Armazena esses
     |      dados em um bufer interno. Deve ser um objeto str, bytes ou
     |      bytearray
     |
     |  calculate(self)
     |      Calcula o valor do FCS (sem o complemento de 1 ao final)
     |
     |  check_crc(self)
     |      Verifica o valor de FCS contido nos dados armazenados no buffer interno
     |
     |  gen_crc(self)
     |      Gera o valor de FCS (com complemento de 1). Retorna um objeto
     |      bytes com os dados seguidos do valor de FCS (LSB e depois MSB)
     |
     |  update(self, data)
     |      Acrescenta mais dados ao buffer interno
     |
     |  clear(self)
     |      Limpa o buffer interno
     |
```

Um exemplo de uso dessa classe pode ser visto a seguir:

```
#!/usr/bin/python3

import crc

fcs = crc.CRC16('Mensagem Transmitida')
msg = fcs.gen_crc()
print('Mensagem com FCS:', msg)

fcs.clear()
fcs.update(msg)
print('Resultado da verificação da mensagem com FCS:', fcs.check_crc())

msg=msg[:-1]
fcs.clear()
fcs.update(msg)
print('Resultado da verificação da mensagem com FCS após modificá-la:', fcs.check_crc())
```

DICA: CRC16 em C++

- Classe CRC16

Um exemplo de uso dessa classe pode ser visto a seguir:

```
#include <iostream>
#include <vector>
#include <iomanip>
#include "CRC16.h"

void dump(const std::vector<uint8_t> & buffer, std::ostream & out) {
    int n = 0;

    out << std::hex << std::setprecision(2);
    for (auto & c: buffer) {
        int x = (unsigned char)c;
        out << x << " ";
        n++;
        if ((n % 20) == 0) out << std::endl;
    }
}

int main() {
    // um buffer com dados
    std::vector<uint8_t> buffer = {0x88, 0x89, 0xaa, 0x0, 0xf1};

    // cria um gerador/verificador de CRC16
    auto crc = make_crc16(buffer);

    // mostra o conteúdo do buffer antes de gerar o CRC
    dump(buffer, std::cout);
    std::cout << std::endl;

    // gera o valor de CRC16 para o buffer, anexando-o ao final desse buffer
    crc.generate_into(buffer);

    // mostra o conteúdo do buffer após gerar o CRC
    dump(buffer, std::cout);
    std::cout << std::endl;

    // cria novo gerador/verificador de CRC
    auto checker = make_crc16(buffer);

    // verifica se o buffer está correto, com base em seu valor de CRC
    std::cout << std::boolalpha;
    std::cout << "Verificação do buffer: " << checker.check() << std::endl;

}
```

Referências adicionais

Alguns textos apresentam maiores detalhes sobre códigos de detecção de erros, e como implementá-los.

- Implementação de checksum
- Implementação de CRC em software
- *A Painless Guide to CRC Error Detection Algorithms, 3rd Edition* por Ross Williams
- Implementação do CRC-16 para PPP
- Tabela de polinômios geradores para CRC

3.3. Garantia de entrega e mecanismos ARQ

A garantia de entrega pode ser definida como um serviço do protocolo que possibilita que o transmissor se certifique de que uma mensagem foi entregue ou não ao destino. Enquanto uma mensagem não tiver sua entrega assegurada, ela permanece na fila de saída mantida no transmissor pelo protocolo. Esse serviço tipicamente é implementado usando algum mecanismo ARQ.

A família de mecanismos ARQ (*Automatic Repeat reQuest*) têm como finalidade garantir a entrega de mensagens, preservando a ordem de envio e buscando eficiência no uso do canal. Tais mecanismos se baseiam em alguns elementos:

- Dois tipos de mensagens: dados e confirmação
- Mensagens de confirmação positiva (ACK) e, opcionalmente, negativa (NAK)
- Mensagens são numeradas de acordo com uma sequência
- Retransmissão de mensagens perdidas ou recusadas

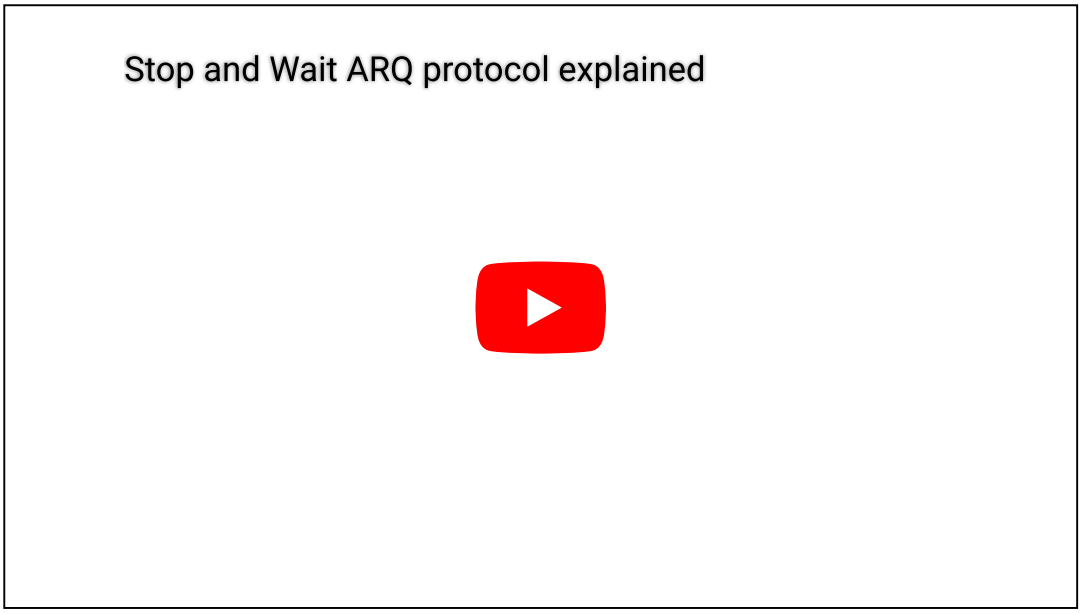
Existe a RFC 3366, que discute recomendações sobre mecanismos ARQ em protocolos de enlace.

Esta seção inclui uma breve revisão sobre os tipos de mecanismos ARQ, e suas eficiências quanto à utilização do canal de comunicação.

Mecanismos ARQ e seus desempenhos esperados

Mecanismos ARQ (Automatic Repeat Request) servem para implementar **Controle de erros** em protocolos de comunicação. Esse serviço tem por objetivo garantir a entrega de PDUs no destino. Os mecanismos ARQ se baseiam na identificação de PDUs por números de sequência, no envio de confirmações de PDUs recebidas, na definição de tempos máximos de espera por confirmação (timeout) e por fim em retransmissões de PDUs extraviadas ou corrompidas. Os mecanismos ARQ conhecidos são:

- **Stop-and-Wait (Pare-e-Espere):** só transmite a próxima PDU quando receber uma confirmação (ACK) da última PDU enviada. Retransmite a última PDU enviada se um tempo máximo de espera pelo ACK (ou timeout) for excedido.



- **Go-Back-N:** transmite até N PDUs sem receber confirmação, quando então espera os ACK antes de enviar mais PDUs. Caso exceda o timeout de uma das PDUs enviadas, retransmite todas as PDUs a partir da PDU

não confirmada. Esse mecanismo usa janela deslizante para controlar o envio das PDUs.

- **Selective Repeat:** transmite até N PDUs sem receber confirmação, quando então espera os ACK antes de enviar mais PDUs. Caso exceda o timeout de uma das PDUs enviadas, retransmite somente essa PDU.

Na discussão a seguir presume-se que os mecanismos ARQ sejam utilizados em protocolos de enlace. Assim, daqui em diante, as PDUs serão denominadas *quadros*.

Um mecanismo ARQ deve tratar corretamente diferentes tipos de erros, tais como:

- Perda de quadro de dados (inclui quadro de dados recebido com erros, e assim descartado).
- Perda de quadro ACK
- Quadro ACK atrasado, fazendo com que ocorra um *timeout* de espera por confirmação e consequente retransmissão

A rigor, Stop-and-Wait já é efetivo em realizar controle de erros, porém tem o potencial de causar uma baixa utilização do meio de transmissão. Os outros mecanismos, Go-Back-N e Selective Repeat, buscam melhorar o aproveitamento do meio, e assim estão também relacionados com o **controle de fluxo**. O serviço de controle de fluxo tem duas motivações principais:

1. Evitar que um transmissor mais rápido sobrecarregue um receptor
2. Melhorar o aproveitamento do meio de transmissão, quando há necessidade de confirmação de quadros e o meio apresentar um atraso de propagação significativo.

Desempenho do Stop-and-wait

O mecanismo Stop-and-wait apresenta desempenho que depende de seus parâmetros e de características físicas do enlace. Para avaliar seu desempenho, deve-se determinar qual a taxa efetiva de transmissão que pode ser obtida. A taxa efetiva é definida com a razão entre quantidade de bits transmitidos (contidos em um ou mais quadros) e o tempo que foi necessário para que sejam entregues no destino. Quadros são considerados entregues somente quando o transmissor recebe uma confirmação do receptor. Isto não é a mesma coisa que a taxa de bits nominal do enlace, pois essa taxa informa o tempo que o transmissor leva para transmitir cada bit pelo meio de transmissão (ou melhor, o inverso de quanto tempo dura cada bit no meio). A taxa efetiva pode ser calculada descobrindo-se qual é a utilização do meio de transmissão. A utilização é um valor entre 0 e 1 que informa a razão entre o tempo em que o meio foi de fato utilizado (i.e. quanto tempo o transmissor de fato precisou para transmitir um ou mais quadros), e o tempo total necessário para fazer a entrega desses quadros. Por exemplo, se o transmissor levou 1 segundo para transmitir um quadro, e depois ficou 1 segundo esperando até receber a confirmação do receptor, a utilização será de 0.5 (o transmissor usou o meio durante 1 segundo, mas o tempo total para entregar o quadro foi de 2 segundos). Assim, as informações necessárias para calcular o desempenho do Stop-and-wait são:

- **Taxa nominal de transmissão, ou taxa de bits (B):** razão entre quantidade de bits que saem do transmissor e o tempo que leva para que saiam.
- **Tempo de transmissão (A_t):** tempo gasto pelo transmissor para transmitir um quadro. É o tempo gasto desde quando o primeiro bit do quadro começa a sair do transmissor, até quando o último bit termina de sair. Esse tempo depende da quantidade de bits transmitidos (*F*) e da taxa de bits nominal do meio (*B*): $A_t = F/B$
- **Atraso de propagação (A_p):** tempo gasto pelo sinal para se propagar desde o transmissor até o receptor. Independe da quantidade de bits transmitidos, pois é uma característica física do meio.
- **Outros atrasos (A_o):** outros atrasos envolvidos durante a entrega do quadro
- **Tempo de envio (A_e):** tempo necessário para que um quadro seja totalmente transmitido do transmissor até o receptor: $A_e = A_t + A_p$
- **Tempo de entrega (*A*):** tempo necessário para que um quadro seja entregue no receptor, i.e. desde o instante em que o quadro começa a ser transmitido até quando o reconhecimento chega ao transmissor: $A = A_t + 2A_p + A_{t_{ACK}} + A_o$
- **Utilização do meio (*U*)**: razão entre o tempo em que o meio foi de fato usado pelo transmissor para transmitir o quadro, e o tempo total necessário para que o quadro seja considerado entregue: $U = \frac{A_t}{A} = \frac{A_t}{A_t + 2A_p + A_{t_{ACK}} + A_o}$
- **Taxa efetiva de transmissão (*E*):** a taxa de bits percebida pelo transmissor para fazer a entrega de quadros: $E = B \cdot U$
- **Timeout (T_o):** tempo máximo de espera por um reconhecimento.

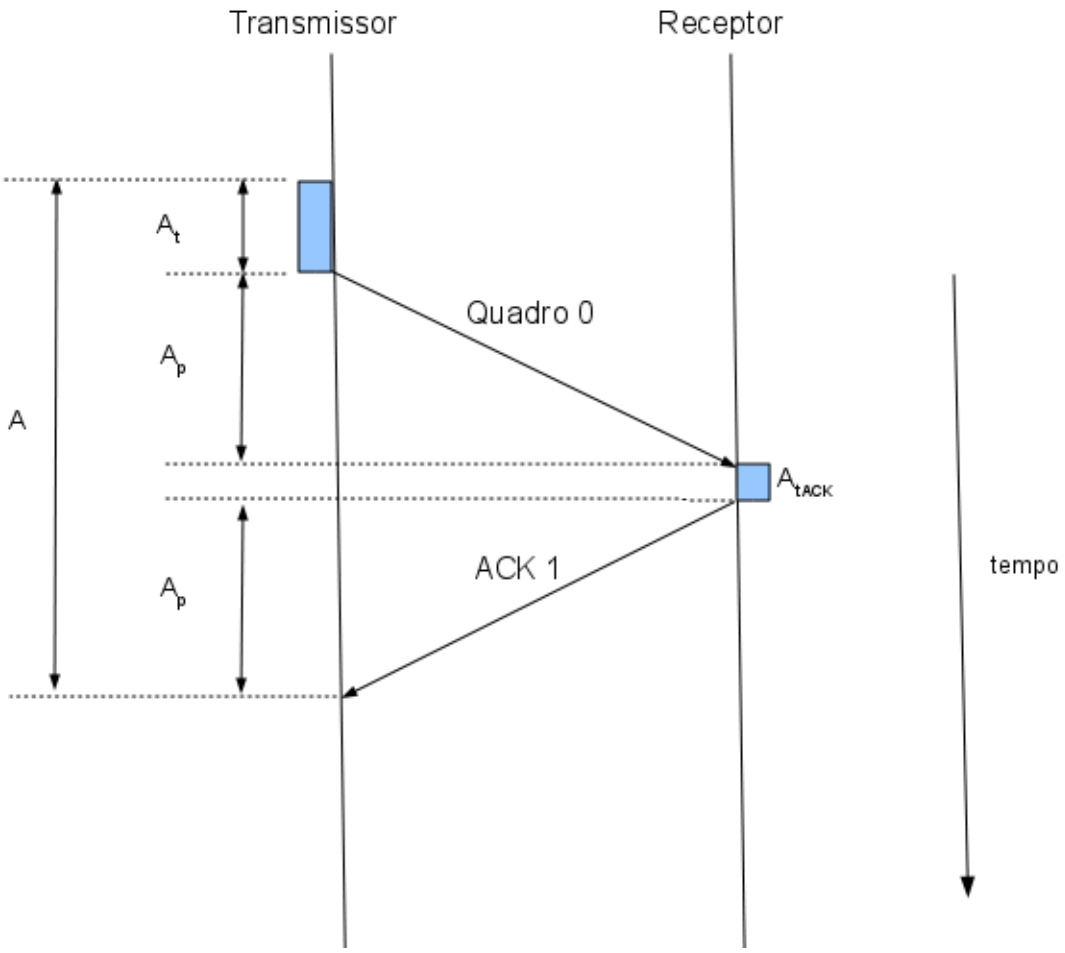


Figura: Um exemplo de envio de quadro e sua confirmação, com os tempos envolvidos

O mecanismo ARQ Stop-and-wait é eficaz no controle de erros. Isto significa que ele garante que um quadro seja entregue no destino. Ser eficaz no controle de erros não significa ser **eficiente**. Se o atraso de propagação for da ordem de grandeza do atraso de transmissão, a utilização do meio se reduz significativamente. Por exemplo, se o atraso de propagação for metade do de transmissão:

$$U = \frac{A_t}{A_t + 2A_p + A_{t_{ACK}}} = \frac{A_t}{A_t + 2A_t/2 + A_{t_{ACK}}} = \frac{A_t}{2A_t + A_{t_{ACK}}} < \frac{1}{2}$$

Já se esses dois atrasos forem iguais:

$$U = \frac{A_t}{A_t + 2A_p + A_{t_{ACK}}} = \frac{A_t}{A_t + 2A_t + A_{t_{ACK}}} = \frac{A_t}{3A_t + A_{t_{ACK}}} < \frac{1}{3}$$

E assim, quanto maior for o atraso de propagação, pior será a utilização do meio de transmissão. A raiz do problema está no tempo em que o meio deixa de ser usado por causa do atraso de propagação. Isto também quer dizer que uma certa quantidade de quadros poderia ser transmitida enquanto não chega a confirmação do primeiro quadro. Essa melhoria no ARQ se chama controle de fluxo, pois visa regular a quantidade de quadros que podem ser enviados de acordo com a capacidade do meio de transmissão e do sistema destino. O ARQ mais simples que implementa uma forma de controle de fluxo se chama **Go-back-N**.

Mecanismo ARQ Go-Back-N

Go-back-N transmite até uma certa quantidade de quadros, sem ainda ter recebido uma confirmação do primeiro quadro enviado. À medida que as confirmações forem recebidas, novos quadros podem ser enviados. Esse mecanismo está ilustrado na figura abaixo:

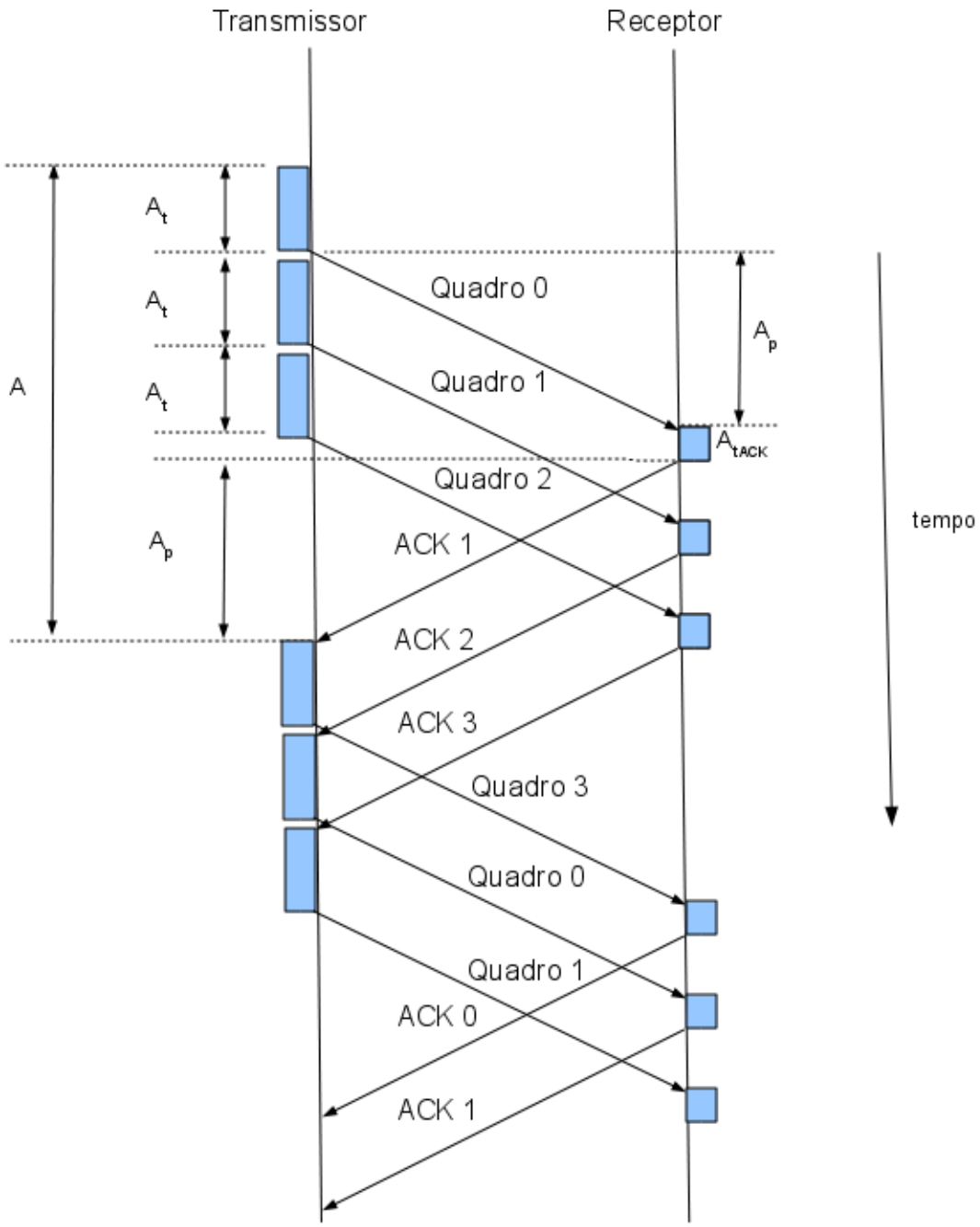
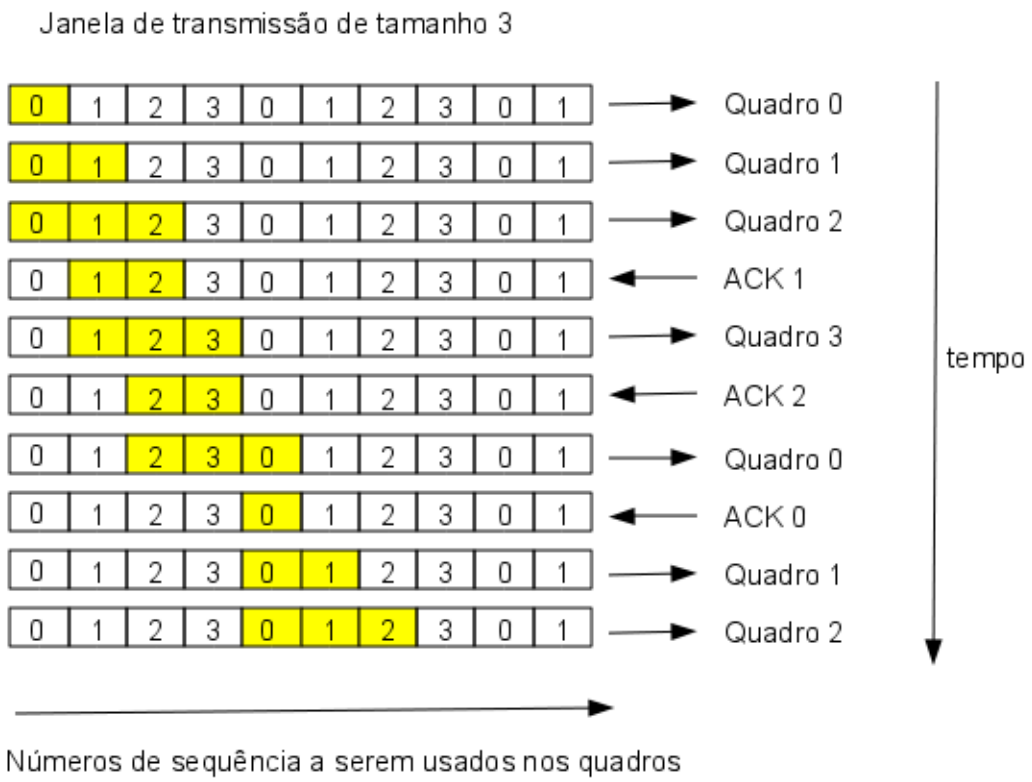


Figura 1: uma possível sequência de transmissão com Go-back-N

Para controlar a quantidade de quadros enviados usa-se a técnica de janela deslizante. A janela é o conjunto de quadros transmitidos e ainda não confirmados, e que precisam ser lembrados para caso seja necessário retransmiti-los. Assim, sempre que um novo quadro é transmitido aumenta-se o tamanho da janela em uma unidade, e quando um quadro é confirmado diminui-se o tamanho da janela em ao menos uma unidade. Há um tamanho máximo para a janela, que corresponde à quantidade máxima de quadros que podem ser transmitidos sem ter ainda uma confirmação de entrega. Esse tamanho máximo de janela é de grande importância para a eficiência do Go-back-N, pois tem influência direta na utilização máxima que pode ser obtida do meio de transmissão.

Um controle com janela deslizante precisa que se numerem os quadros sequencialmente, de forma a representar a ordem em que foram enviados. Assim, quadros fora de ordem não são aceitos no destino (ao menos no caso do Go-back-N). Por razões de praticidade, a quantidade de números de sequência necessária para o Go-back-N é igual ao tamanho máximo da janela + 1. Por exemplo, se o tamanho máximo da janela for 3, são necessários no mínimo 4 números de sequência (ex: 0 a 3). Se a quantidade de números de sequência for igual ao tamanho da janela, pode ocorrer um erro no controle de erros do mecanismo (tente descobrir que erro

seria esse ...). A figura abaixo ilustra um exemplo em que se fazem transmissões de quadros com janela de tamanho máximo 3 (a janela a cada instante é destacada em amarelo, e os números correspondem aos números de sequência dos quadros):



Note que, assim como no Stop-and-wait, os quadros de reconhecimentos indicam qual o próximo quadro a ser aceito pelo destino. Além disso, o Go-back-N aceita reconhecimentos cumulativos. Quer dizer, se houver três quadros na janela de transmissão, e for recebido um reconhecimento relativo ao segundo quadro, o transmissor considera que tanto o primeiro quanto o segundo quadro foram recebidos com sucesso. Isto é possível pois nesse ARQ o receptor aceita apenas quadros em ordem.

Desempenho do Go-back-N

Mas qual deve ser a utilização do meio de transmissão com o Go-back-N ? Para responder a essa questão devem-se considerar duas situações:

1. **O atraso para receber o primeiro ACK é maior ou igual ao tempo de transmissão para enviar todos os quadros permitidos pelo tamanho máximo de janela:** essa situação está mostrada na figura 1, que mostra uma sequência de transmissão. A utilização nesse caso será:
$$U = \frac{N \cdot A_t}{A_t + 2A_p + A_{t_{ACK}}}$$
2. **O atraso para receber o primeiro ACK é menor que o tempo de transmissão de uma janela de tamanho máximo:** nesse caso, a utilização é máxima (= 1), pois todo o atraso de propagação é aproveitado para enviar quadros.

Utilização do meio com Go-Back-N sujeito a erros

Quando ocorrem erros, o Go-Back-N precisa reenviar todos os quadros da janela de transmissão. Um erro pode acontecer em qualquer quadro da janela de transmissão, sendo que a partir do quadro perdido deve-se esperar um tempo equivalente ao *timeout* de espera por reconhecimento. Somente após esse *timeout* o transmissor voltará a transmitir, retransmitindo quadros a partir do quadro perdido. Assim a transmissão de quadros se compõe de conjuntos de quadros transmitidos com sucesso, intercalados por tempos de espera *timeout* devido a quadro perdido (i.e. não reconhecido). Para calcular a utilização do Go-Back-N sujeito a erros, algumas simplificações serão assumidas:

1. A sequência de quadros transmitidos para fins de análise será longa (idealmente infinita).

2. A sequência de transmissão será modelada como a transmissão sucessiva de conjuntos de quadros, que correspondem às janelas de transmissão.
3. Cada conjunto de quadros transmitidos será entendido como até N-1 quadros enviados com sucesso seguidos por um quadro perdido, ou N quadros enviados com sucesso. Quando acontece um erro, o transmissor fica bloqueado até que aconteça um *timeout* (prazo de espera pela chegada de reconhecimento).

Cada conjunto de quadros apresenta uma certa utilização, que depende do quadro que sofreu erro. A utilização de cada conjunto pode ser calculada da seguinte forma:

1. *A utilização na ausência de erros:* é a utilização já definida anteriormente, que será chamada de U_N, sendo calculada com a equação:
   $$U_N = min(1, \frac{NA_t}{A})$$
2. *A utilização no caso da ocorrência de um erro:* é a utilização resultante de uma sequência de transmissão em que após um certo número de quadros enviados com sucesso acontece um erro, e será chamada de U_j. O índice *j* se refere a quantidade de quadros enviados com sucesso antes do erro, e varia de 0 a N-1. Essa utilização pode ser calculada com a equação:
   $$U_j = min(1, \frac{jA_t}{T_o + jA_t})$$

   ... sendo que o parâmetro *A* corresponde ao atraso para recepção de um reconhecimento (i.e. $A = A_t + 2A_p + A_{t_{ACK}}$), A_t é o atraso de transmissão, A_p é o atraso de propagação e T_o é o timeout de espera por reconhecimento.

A utilização total depende da probabilidade de acontecer cada uma das possíveis utilizações, que variam de U_0 a U_{N-1}. Assumindo-se que a probabilidade de um quadro ser entregue sem erro seja dada pela variável P_q, as probabilidades para cada caso são (obs: janela de tamanho N):

Posição do erro	Probabilidade
1	$1 - P_q$
2	$P_q(1 - P_q)$
3	$P_q^2(1 - P_q)$
N	$P_q^{N-1}(1 - P_q)$

O caso especial em que não ocorrem erros tem probabilidade P_q^N.

Obs: a variável P_q corresponde à probabilidade de que um quadro seja enviado com sucesso, e seu ACK seja recebido pelo transmissor. Ela pode ser calculada assim:

$$P_q = (1 - BER)^{L + L_{Ack}}$$

... sendo *L* o tamanho do quadro em bits, e L_{Ack} o tamanho do quadro ACK também em bits.

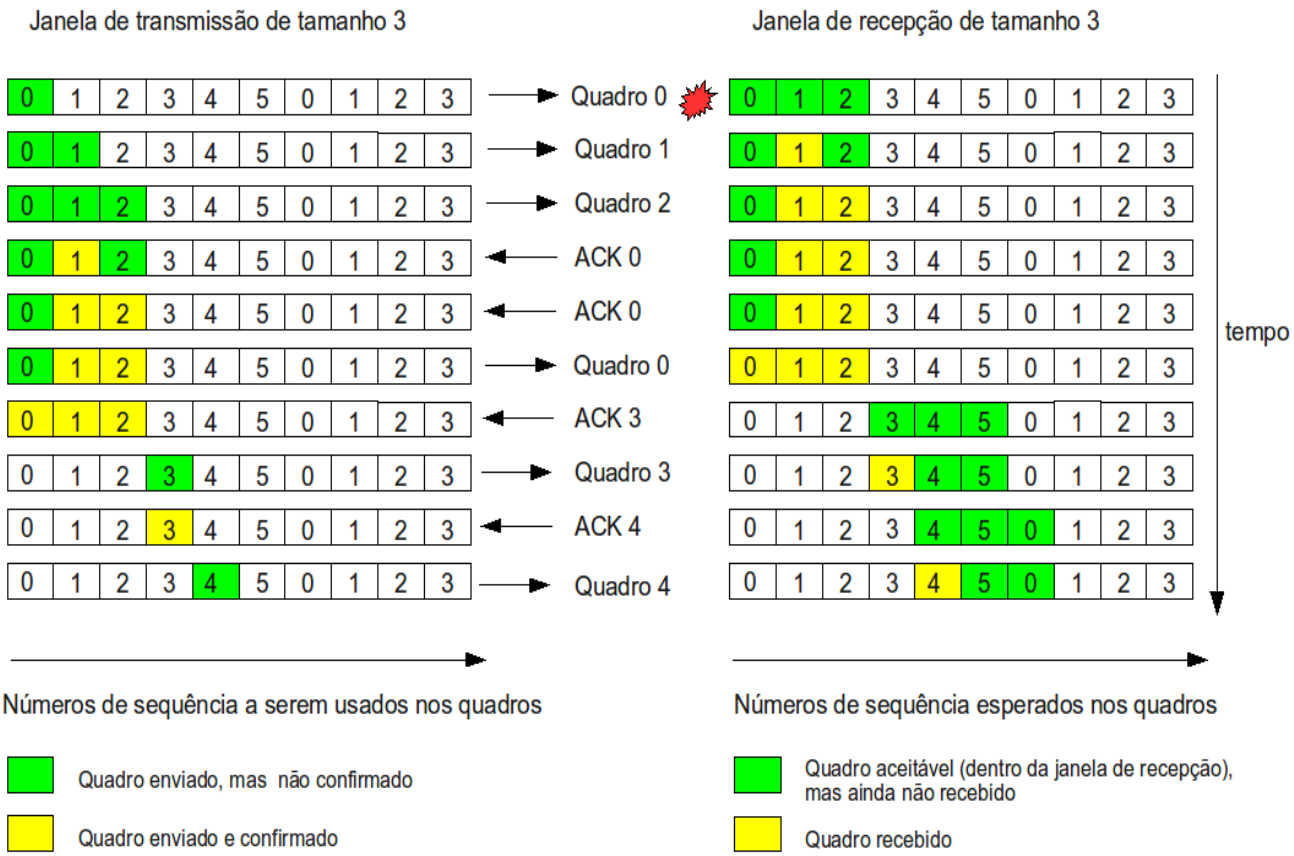
A utilização total pode ser calculada fazendo-se uma média ponderada das utilizações U_j correspondentes a cada caso. Os pesos a serem usados são as probabilidades definidas acima. A expressão resultante para a utilização pode ser escrita da seguinte forma:

$$U = P_q^N \cdot U_N + \sum_{j=0}^{N-1} P_q^j (1 - P_q) \cdot U_j$$

A expressão acima se aplica tanto a casos em que a utilização na ausência de erros for 100% (i.e. U = 1), quanto menor que 100% (U < 1).

Selective Repeat

O ARQ Selective Repeat aceita receber quadros fora de ordem, pois o receptor mantém uma janela de recepção com o mesmo tamanho que a janela de transmissão. Assim, o receptor aceita um quadro se seu número de sequência estiver contido na janela de recepção atual. O limite inferior da janela de recepção avança quando chega um quadro com o primeiro número de sequência da janela. Quando o limite inferior da janela avança, os quadros correspondentes são passados para a camada superior. O receptor sempre envia ACKs com o número do quadro que está no início da janela de recepção.



O desempenho do Selective Repeat, na ausência de erros, é o mesmo do Go-Back-N.

Utilização do meio com Selective Repeat sujeito a erros

Na presença de erros, o ARQ Selective Repeat proporciona uma utilização melhor que Go-Back-N. Abaixo segue a utilização obtida com Selective Repeat para os possíveis cenários de erros:

QUANDO A UTILIZAÇÃO NA AUSÊNCIA DE ERROS FOR < 1

Quer dizer, $N \cdot A_t < A_t + 2 \cdot A_p + A_{t_{ACK}}$. Nesse caso a utilização é a mesma do Go-Back-N:

$$U = N \cdot A_t \cdot \left(\frac{P_F}{A_t + 2 \cdot A_p + A_{t_{ACK}}} + \frac{1 - P_F}{T_o + A_t + 2 \cdot A_p + A_{t_{ACK}}}\right)$$

QUANDO A UTILIZAÇÃO NA AUSÊNCIA DE ERROS FOR = 1

Quer dizer, $N \cdot A_t \geq A_t + 2 \cdot A_p + A_{t_{ACK}}$

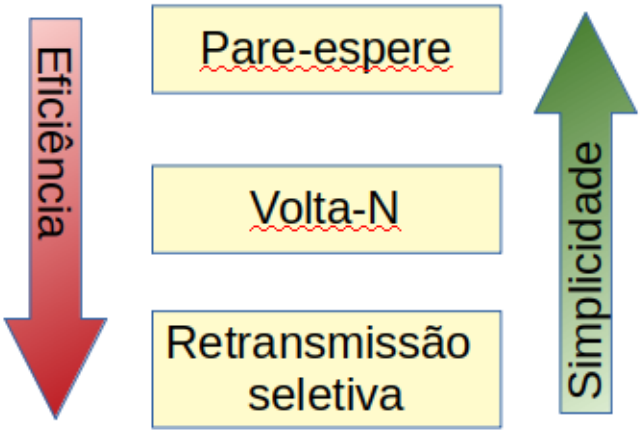
Há dois casos a analisar:

1. **Timeout maior que o tempo para enviar uma janela completa:**
$$U = P_F + \frac{N \cdot A_t \cdot (1 - P_F)}{T_o + A_t + 2 \cdot A_p + A_{t_{ACK}}}$$
2. **Timeout menor que o tempo para enviar uma janela completa:**
$$U = P_F + \frac{N \cdot (1 - P_F)}{N + 1}$$

3.4. A Garantia de entrega no protocolo de enlace

O protocolo de enlace deve ter o mecanismo de entrega, mas qual das opções de mecanismos ARQ deve ser utilizada ? Qual o critério para fazer uma boa escolha ?

Como visto na seção anterior, os três mecanismos ARQ clássicos são eficazes quanto à entrega confiável de mensagens. O que os diferencia são suas eficiências na utilização do canal de comunicação e suas complexidades. A eficiência é influenciada pelo atraso de propagação característico do canal de comunicação, e pela taxa de erros desse canal. A comparação dos mecanismos pode ser vista neste figura.



Comparação entre mecanismos ARQ com respeito à eficiência e simplicidade

O critério de escolha deve se guiar pela pergunta*: qual o mecanismo mais simples que apresenta uma eficiência aceitável ?* A resposta deve se fundamentar em uma análise dos mecanismos no cenário de utilização do protocolo de enlace.

Escolha do mecanismo ARQ a ser utilizado no protocolo de enlace

- ARQ Híbrido

A análise inicia com o mecanismo ARQ mais simples. Para a análise de utilização do canal usando *pare-e-espere,* define-se *B* a taxa de bits, *F* o tamanho de quadro, e *L* a distância a ser percorrida pelo sinal no enlace. No canal de comunicação considerado, B é de 9600 bps, F é 1024 octetos, e L tem valor de 1000 m. Com isso, a utilização do canal pode ser estimada desta forma:

$$B = 9600bps$$
$$F = 1024bytes$$
$$F_{ack} = 4bytes$$
$$L = 1000m$$
$$A_t = \frac{F}{B} = \frac{1024 * 8}{9600} = 853ms$$

$$A_{tack} = \frac{F_{ack}}{B} = \frac{4 * 8}{9600} = 3.33ms$$

$$A_p = \frac{L}{c} = \frac{1000}{3 \cdot 10^8} = 3.33us$$

$$U = \frac{A_t}{A_t + A_{tack} + 2 \cdot A_p} = 0.996$$

A análise mostra que a eficiência do pare-espere está em torno de 99.6%. Isso é consistente com a relação entre atraso de propagação do canal e o tempo de transmissão de um quadro. O atraso de propagação é muito menor do que o tempo de transmissão, então ele pode ser considerado desprezível. Porém esse tempo de transmissão é alto porque a taxa de dados B é pequena ... e se ela fosse maior ?
Supondo que a taxa de dados B fosse de 2 Mbps (correspondente ao transceiver NRF24L01, por exemplo), a utilização do canal seria:

$$A_t = \frac{F}{B} = \frac{1024 * 8}{2x10^6} = 4096us$$

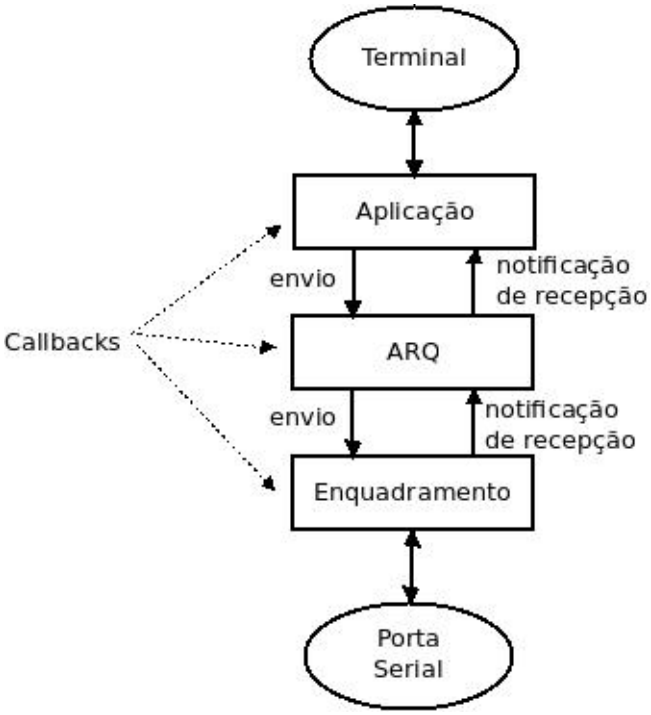
$$U = \frac{A_t}{A_t + A_{tack} + 2 \cdot A_p} = 0.994$$

Mesmo que a taxa de dados B fosse 2 Mbps, a eficiência seria de 99.4%.

Essa análise mostra que o pare-espere é plenamente satisfatório para o protocolo de enlace em desenvolvimento.

Modelagem do mecanismo ARQ

O mecanismo ARQ *pare-e-espere* deve ser modelado antes de ser implementado no protocolo. Sua disposição na estrutura do protocolo deve ser esta:



A estrutura do protocolo com a garantia de entrega representada pela subcamada ARQ

Assim, a subcamada *ARQ* envia e recebe quadros de dados e de confirmação da subcamada inferior (detecção de erros + enquadramento). O comportamento do protocolo, na subcamada ARQ, deve ser modelado apropriadamente com uma MEF comunicante, para que possa ser em seguida implementado.

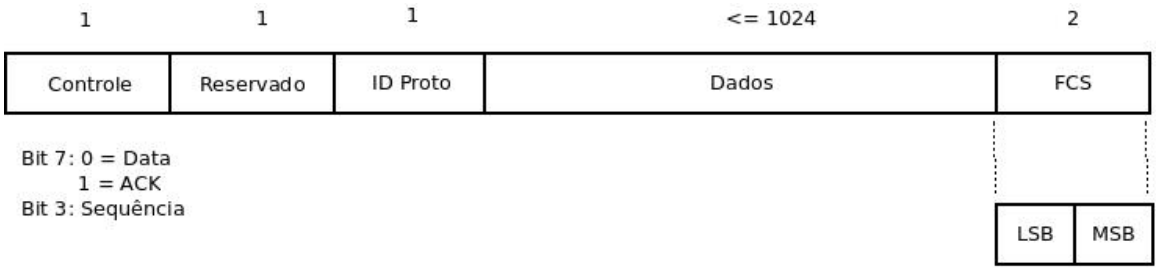
3.5. Formato de quadro do protocolo

O momento em que se planeja a incorporação do mecanismo ARQ ao protocolo é propício para que se defina qual exatamente deve ser seu formato de quadro. Por simplicidade, independente da subcamada do protocolo, o formato do quadro deve ser o mesmo. Porém, informações específicas de cada subcamada podem ser embutidas em cada quadro, na forma de um cabeçalho. Assim, cada subcamada interpreta ou modifica os campos do cabeçalho que lhe dizem respeito.

O formato proposto para o quadro possui um cabeçalho com três campos:

- **Controle**: contém tipo de quadro (ACK ou DATA) e número de sequência (0 ou 1)
- **Reservado**: este campo fica reservado para uso futuro
- **Proto:** contém o tipo de conteúdo transportado pelo quadro (**OBS**: este campo existe somente em quadros do tipo DATA)

O campo FCS, ao final do quadro, contém o valor do código CRC-16-CCITT. Esse campo ocupa 2 bytes, sendo o primeiro o LSB (*Least Significant Byte*) do código CRC, e o último o MSB. Esse campo é anexado pelo enquadramento na transmissão, e removido na recepção.



O formato de um quadro

3.6. O controle de acesso ao meio

O enlace a ser estabelecido usa um canal compartilhado. Somente um dos lados do enlace pode transmitir em um determinado momento. Por isso é necessário que o protocolo inclua um mecanismo de controle de acesso ao meio (MAC) para arbitrar as transmissões no enlace.

Três abordagens para acesso ao meio podem ser identificadas (maiores detalhes estão no capítulo 5, seção 5.3, do livro *Redes de Computadores e a Internet, 5a. edição*, de James Kurose e Keith Ross):

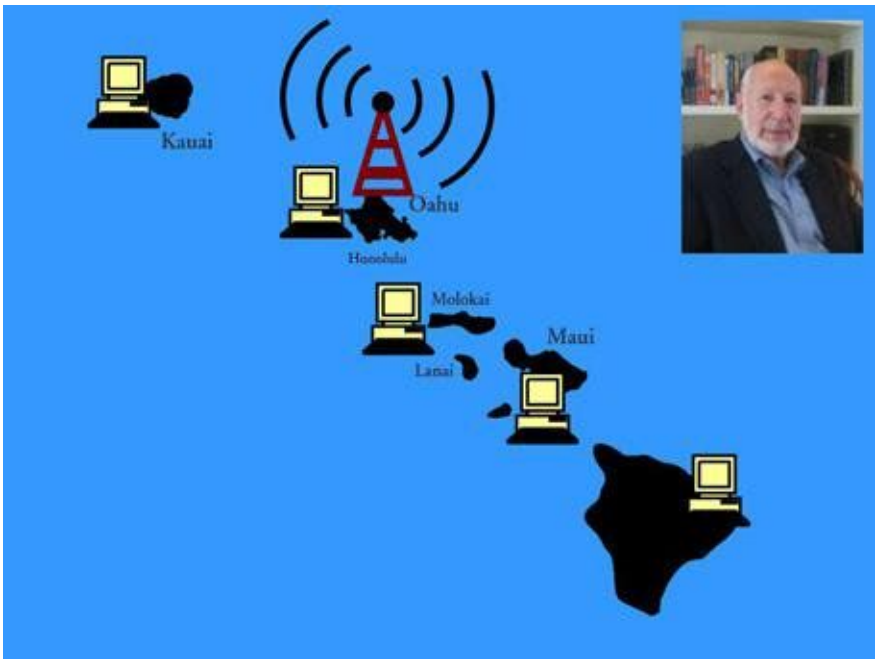
1. **Por revezamento:** usando mensagens especiais, o direito de acesso ao meio é definido inequivocamente para cada um dos lados alternadamente. Esse acesso ao meio é livre de disputa (*contention free*), o que significa que colisões não ocorrem.
2. **Por divisão de tempo:** ambos os lados se mantêm sincronizados, e usam intervalos de tempo cíclicos predeterminados para suas transmissões (isso é uma forma de TDMA). Essa abordagem também é livre de disputa.
3. **Aleatória:** cada um dos lados pode tentar transmitir em instantes arbitrários. Se transmissões se sobrepuserem (*colisões*), as mensagens envolvidas são perdidas e devem ser retransmitidas. Usam-se esperas de tempo aleatórias para evitar que as retransmissões causem novas colisões. Essa abordagem envolve disputa por acesso ao meio (*contention based*).

Para o protocolo de enlace, as abordagens 1 e 3 são viáveis. Para a abordagem 1, pode-se imaginar um acesso ao meio do tipo mestre-escravo. Para a abordagem 3, o acesso ao meio clássico Aloha poderia ser implementado.

Independente da escolha do tipo de acesso ao meio, ele deve estar integrado ao mecanismo ARQ. A detecção de perda de quadro feita pelo ARQ, por meio de mensagens de confirmação, é um evento comum ao ALOHA, por exemplo.

Aloha

A rede Aloha foi um projeto pioneiro de uma rede sem-fio da Universidade do Hawaii, e foi implantada em 1971. Essa rede interligava as ilhas do arquipélago com um serviço de dados baseado em links de rádio, com taxas de dados de 9600 bps. A forma com que a comunicação se dava usava um método acesso ao meio inovador chamado ALOHA, que se baseava em acesso ao meio aleatório.



A rede ALOHA no arquipélago do Hawaii, com a foto de Norman Abramson, um de seus criadores

Na versão original do ALOHA, um nó da rede poderia transmitir uma nova mensagem de forma imediata e incondicional. Cada mensagem recebida era confirmada por uma mensagem curta. Se o transmissor não recebesse essa mensagem dentro de um certo prazo, ele esperava um tempo aleatório e então tentava novamente. Esse mecanismo de confirmação servia para detectar e corrigir erros devido a colisões, que ocorriam quando as mensagens transmitidas por dois nós se sobrepunham (interferiam) no receptor.

Em resumo, o acesso ao meio do ALOHA pode ser descrito desta forma:

1. Transmissor envia imediatamente uma mensagem
2. Transmissor espera uma mensagem de confirmação em sentido contrário
3. Se confirmação não for recebida dentro de um certo prazo:
    i. Espera um tempo aleatório, sorteando um número inteiro entre 0 e N e multiplicando-o por uma constante de tempo chamada de *slot*.
    ii. Retransmite a mensagem, e volta ao passo 2

Mestre-escravo

Em um acesso ao meio do tipo mestre-escravo, também chamada de *polling*, apenas um nó pode transmitir a cada momento. O direito de transmissão é concedido por meio da posse de uma mensagem de controle, chamada usualmente de *token* (ou *ficha*). Como seu nome indica, um dos nós é responsável por enviar o *token* para cada um dos demais nós, escalonando-os para transmissão. A posse do *token* é limitada a um certo tempo, quando então ele perde a validade e o direito de transmitir é revogado. Nesse acesso ao meio, não há chance de haver colisão, pois somente o nó possuidor do *token* pode transmitir.

Existem variações para o mestre-escravo, tais como:

1. Cada escravo deve devolver o token ao mestre, quando tiver encerrada sua transmissão, ou seu tempo de posse de token esgotar
2. O mestre automaticamente envia o token a um novo escravo quando o tempo de concessão do escravo atual terminar, sem precisar receber o token de volta desse escravo.

A abordagem 1 tem a vantagem de melhorar o aproveitamento do tempo disponível, pois o escravo informa quando não precisar mais transmitir. O overhead da transmissão final do token pelo escravo pode ser considerado aceitável, pois é compensado pela fatia do tempo de concessão que foi poupada. Esse é exatamente o aspecto negativo da abordagem 2, que poupa a transmissão final de *token* pelos escravos, mas não aproveita o tempo concedido e não utilizado. Por outro lado, em momentos de alta ocupação do canal, a abordagem 2 pode apresentar ligeira vantagem ao evitar a transmissão do token dos escravos para o mestre.

3.7. Gerenciamento de sessão

A próxima função do protocolo de enlace envolve o estabelecimento, manutenção e terminação de conexão. Sendo um protocolo ponto-a-ponto, deve-se estabelecer um enlace entre as duas pontas participantes antes de poder transferir dados. Isso evita que as transmissões entre um par de participantes sejam confundidas com transmissões de outros pares.

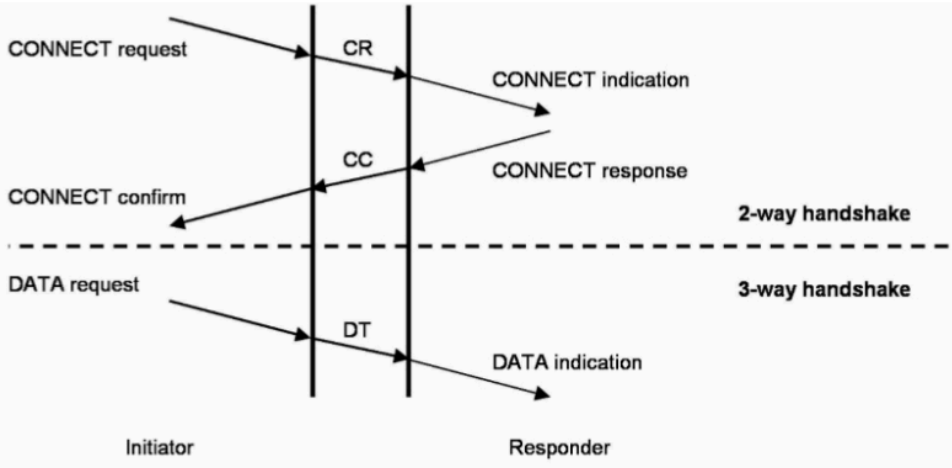
Nas seções 5.2 e 5.3 do capítulo 5 do livro *Protocol Engineering, 2nd ed*, de Hartmut Konig, há uma explicação sobre o gerenciamento de conexões e diversas formas de implementá-la. No caso do protocolo de enlace, serão usados estabelecimento e terminação explícitos de conexões, e manutenção de conexão.

Estabelecimento de conexão

Para criar uma conexão deve-se:

- **estabelecer a conexão:** ambos participantes devem se sincronizar para aceitar ou recusar a conexão
- **negociar parâmetros de conexão:** a conexão pode envolver parâmetros operacionais do protocolo que devem ser comuns a ambos participantes. Como exemplo, citam-se tamanho máximo de PDU e identificador de conexão.

A sincronização entre os participantes implica a troca de mensagens, a qual pode ser feita com duas (*2-way handshaking*) ou 3 (*3-way handshaking*) mensagens:



A sincronização do tipo *2-way* é adequada em comunicações unidirecionais, mas não para comunicações bidirecionais. Isso se deve ao fato de que a primeira mensagem (*CR*) contém parâmetros de conexão do participante que iniciou o processo, e a segunda mensagem (*CC*) contém parâmetros do outro participante. Se a mensagem *CC* for perdida, o participante iniciador da conexão não tem como saber se a conexão foi aceita e quais os parâmetros definidos pelo outro participante. Assim, uma terceira mensagem enviada pelo iniciador serve para que o outro participante saiba que a conexão foi estabelecida, e que assim ele pode usá-la para enviar dados. De forma resumida:

1. Ao receber a mensagem *CC*, o iniciador já pode enviar dados pela conexão
2. Ao receber a terceira mensagem, o outro participante também pode enviar dados pela conexão. OBS: essa terceira mensagem pode ser uma mensagem de dados comum, pois ela serve para que o outro participante saiba que a conexão foi estabelecida com os parâmetros negociados.

Manutenção de conexão

Um enlace pode ter momentos de ociosidade, quando não há mensagens de dados para serem transmitidas. Isso pode ser confundido com casos em que o enlace se rompe (ex: um dos participantes é desligado, ou o meio de comunicação é seccionado). Para evitar que um enlace rompido seja interpretado com um enlace ocioso (e vice-versa), o protocolo deve fazer a manutenção de conexão.

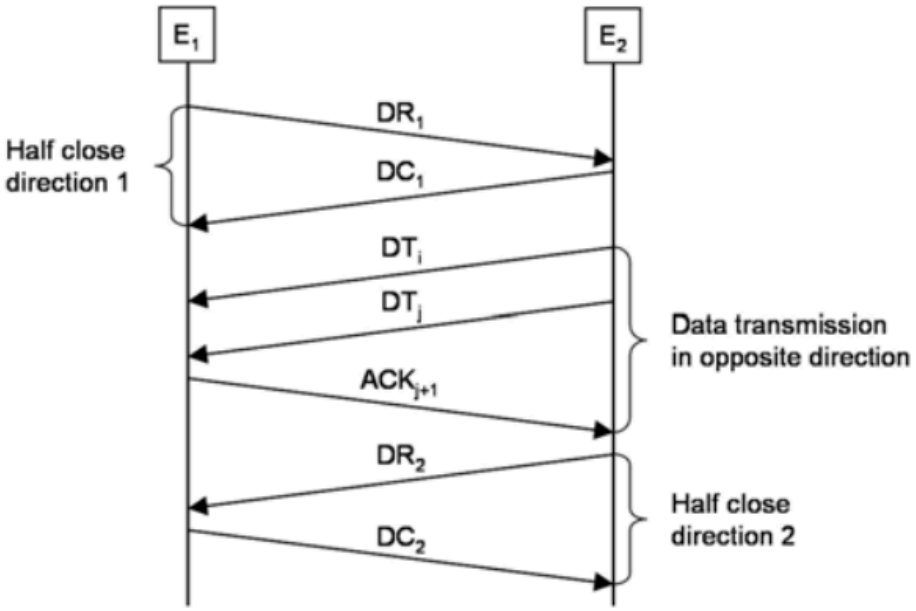
A manutenção de conexão pode ser feita com o envio periódico de mensagens de verificação. Por exemplo, o protocolo PPP usa mensagens *Keep-Alive* enviadas a cada 10 segundos para monitorar o estado do enlace. Se três mensagens *Keep-Alive* consecutivas forem perdidas, o enlace é terminado. Nesse caso, após um certo tempo (ex: 30 segundos) o protocolo PPP tenta restabelecer o enlace. O protocolo TCP também possui um mecanismo opcional para manutenção de conexão chamado de *Keep Alive*. Uma abordagem como essa poderia ser incluída no protocolo de enlace em desenvolvimento.

Terminação de conexão

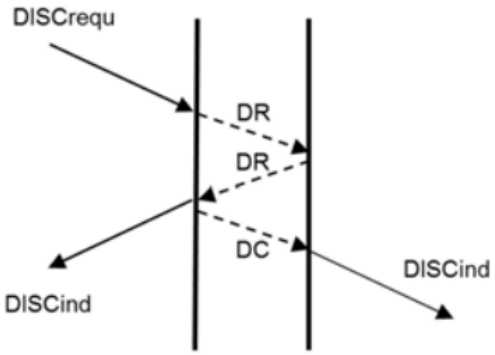
A terminação de conexão implica duas necessidades:

1. A sincronização entre os participantes quanto ao término da conexão (ambos participantes devem fechar a conexão)
2. A garantia de que todas as mensagens de dados pendentes sejam entregues

A solução não é tão simples quanto parece. Uma discussão detalhada pode ser lida na seção 5.3.3 do livro Protocol Engineering, 2nd ed, de Hartmut Konig. Com base nessa explicação e no fato de que o protocolo de enlace em desenvolvimento é bidirecional, deve-se usar a terminação de conexão feita por ambos participantes do enlace. Assim um participante envia uma mensagem para terminação de conexão, e após sua confirmação entra-se em estado de conexão parcialmente fechada (*half-close connection*). O outro participante envia suas mensagens pendentes, e em seguida envia sua mensagem de terminação de conexão. Após a confirmação dessa última mensagem de terminação, a conexão é considerada terminada. A figura a seguir exemplifica esse procedimento.



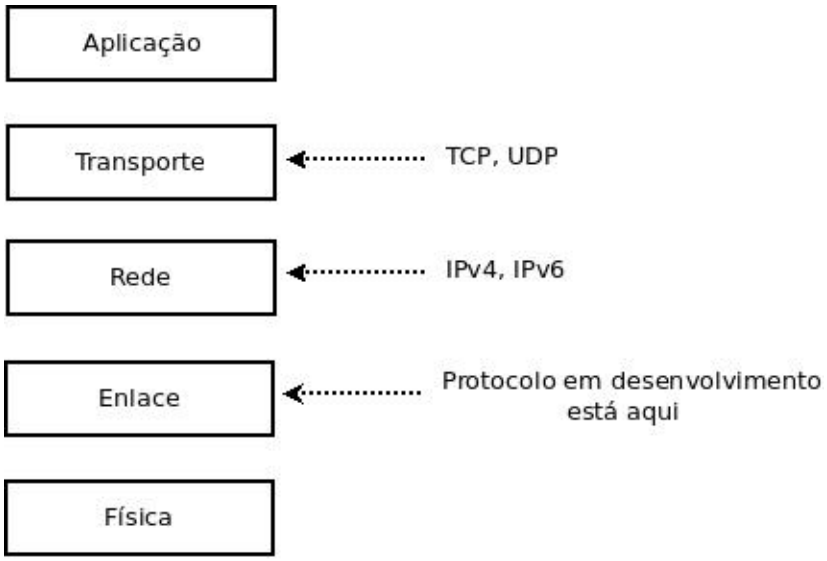
Uma simplificação pode ser feita se ambos participantes encerrarem a conexão simultaneamente. Nesse caso, a terminação pode ser sincronizada com três mensagens (*3-way handshake*):



Por fim, pode acontecer de mensagens de terminação de conexão serem perdidas. Isso manteria um ou mesmo ambos participantes esperando indefinidamente pelo término de conexão. Uma solução para evitar essa situação é usar *timeout* para a espera de confirmação da mensagem de término de conexão.

4. Integração com subsistema de rede do Linux

O protocolo em desenvolvimento se situa na camada de enlace. Sendo assim, ele se situa entre as camadas de rede e física, como ilustrado nesta figura.



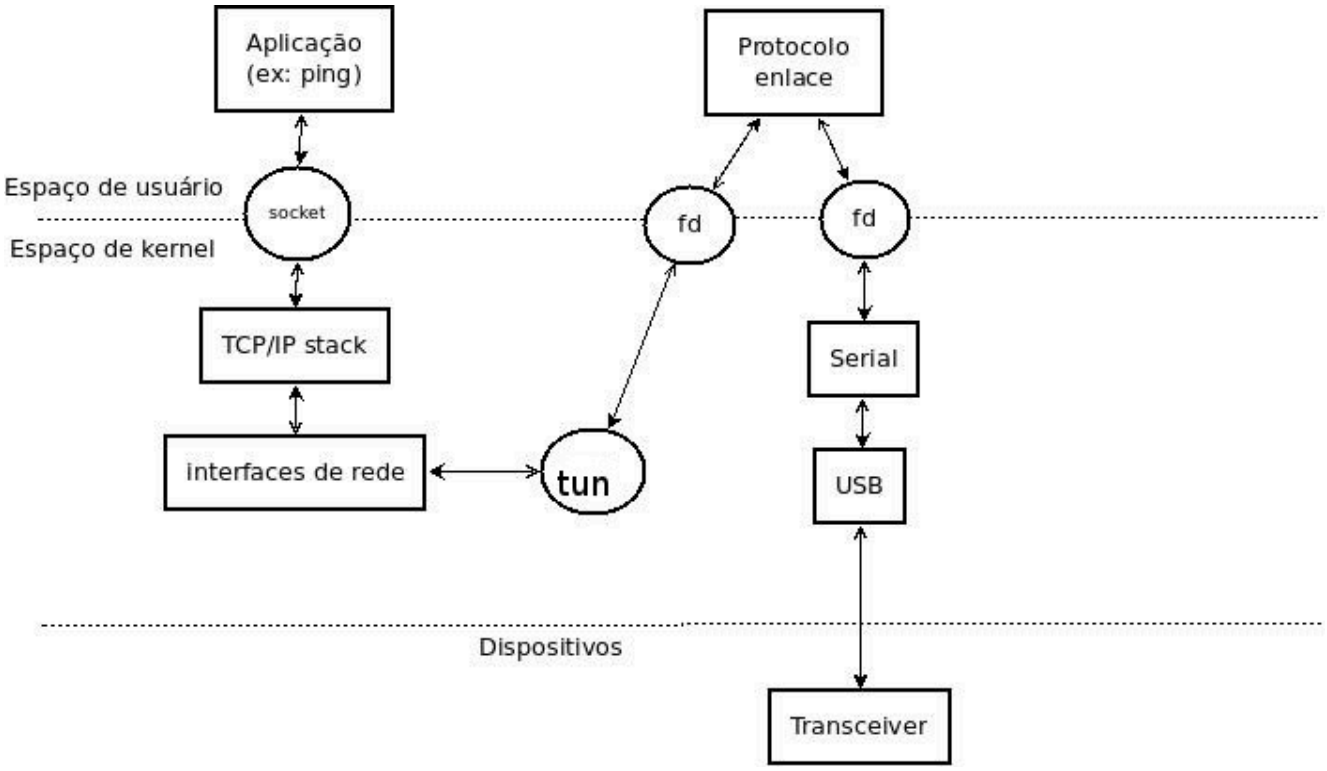
As camadas de rede e a localização do protocolo

A integração do protocolo de enlace com o subsistema de rede do Linux pode ser feita de duas formas:

1. **Criação de um device driver no kernel Linux:** uma tarefa árdua, e que envolve um bom conhecimento sobre o kernel Linux e desenvolvimento de device drivers. As bibliotecas de programação usuais disponíveis em espaço de usuário não podem ser usadas. A depuração é difícil. O protocolo fica bem integrado ao sistema operacional.
2. **Implementação do protocolo em espaço de usuário:** um protótipo em espaço de usuário é mais fácil de criar, pois pode usar as APIs existentes para a linguagem de programação escolhida. A depuração fica facilitada. Porém a integração com o sistema operacional apresenta um certo overhead devido ao fluxo de processamento transitar frequentemente entre espaços de sistema e de usuário.

Para o objetivo do projeto 1, a segunda opção é a mais adequada. A implementação em espaço de usuário deve usar uma interface de rede do tipo *tun* para fazer a integração com o subsistema de rede. Com isso, existe uma interface de rede associada ao enlace estabelecido pelo protocolo, a qual possui parâmetros de rede IP (endereço, máscara), e rotas podem ser definidas para destinos alcançáveis através dela. Com isso, qualquer aplicação TCP/IP pode se comunicar usando o protocolo desenvolvido.

A figura a seguir mostra um diagrama que esquematiza a integração do protótipo com o Linux, evidenciando a interface de rede do tipo *tun*. Note que a interação entre o protocolo e o kernel é feita por meio de descritores de arquivo, identificados pela sigla *fd* no diagrama. Esses descritores implementam uma interface de arquivo, e por isso suportam suas operações *read* e *write*.



A criação e configuração de uma interface *tun* pode ser feita de forma programática, usando chamadas de sistema do Linux. Após criar a interface, pode-se testar a comunicação com um simples *ping*:

```
ping 10.0.0.2
```

Para o protocolo PTP, pode ser mais conveniente pensar na interface *tun* como um objeto. Sua implementação como uma classe pode ser feita como mostrado a seguir:

- Uma classe C++ para a interface tun
- Uma classe Python para a interface tun (ver documentação)

Uma demonstração sobre a interface tun

A criação de uma interface tun implica instanciar a classe Tun, definindo apropriadamente seus parâmetros (nome da interface, configuração IP). Em seguida, deve-se ativar a interface para que seja efetivamente criada no sistema operacional. Feita essa preparação, podem-se enviar e receber quadros do sistema operacional. Veja este exemplo, que mostra quadros a serem enviados por meio da interface tun. Note que quadros recebidos dessa interface estão descendo a pilha de protocolos, e portanto são quadros a serem enviados.

```
import sys,time
from tun import Tun

# instancia a classe Tun
tun = Tun("tun0","10.0.0.1","10.0.0.2",mask="255.255.255.252")
# ativa a interface
tun.start()

while True:
  # obtém um quadro a ser enviado ... ele vem do subsistema de rede
  proto,frame = tun.get_frame()
  print('Protocolo: %d, dados: %s' % (proto, frame)
```

Na interface tun, um quadro é formado pelo seu identificador de protocolo e seu conteúdo. O identificador de protocolo indica qual protocolo de camada três gerou o quadro a ser enviado, ou deve processar um quadro recebido. Se for 0800_H, o quadro contém um datagrama IPv4, e, se for $86DD_H$, contém um datagrama IPv6. Ele tem correspondência com o campo ID_PROTO de nosso protocolo de enlace.

A recepção de um quadro é feita pelo envio desse quadro para a interface tun. Assim, ele pode ser entregue ao subsistema de rede, e ser processado pela pilha de protocolos. A entrega de um quadro para a interface tun deve incluir seu identificador de protocolo, além de seu conteúdo. Veja este exemplo, que implementa um loopback.

```
import sys,time
from tun import Tun


tun = Tun("tun0","10.0.0.1","10.0.0.2",mask="255.255.255.252",mtu=1500,qlen=4)
tun.start()

while True:
  proto,frame = tun.get_frame()
  # envia o quadro de volta para o Linux !
  tun.send_frame(frame, proto)
```

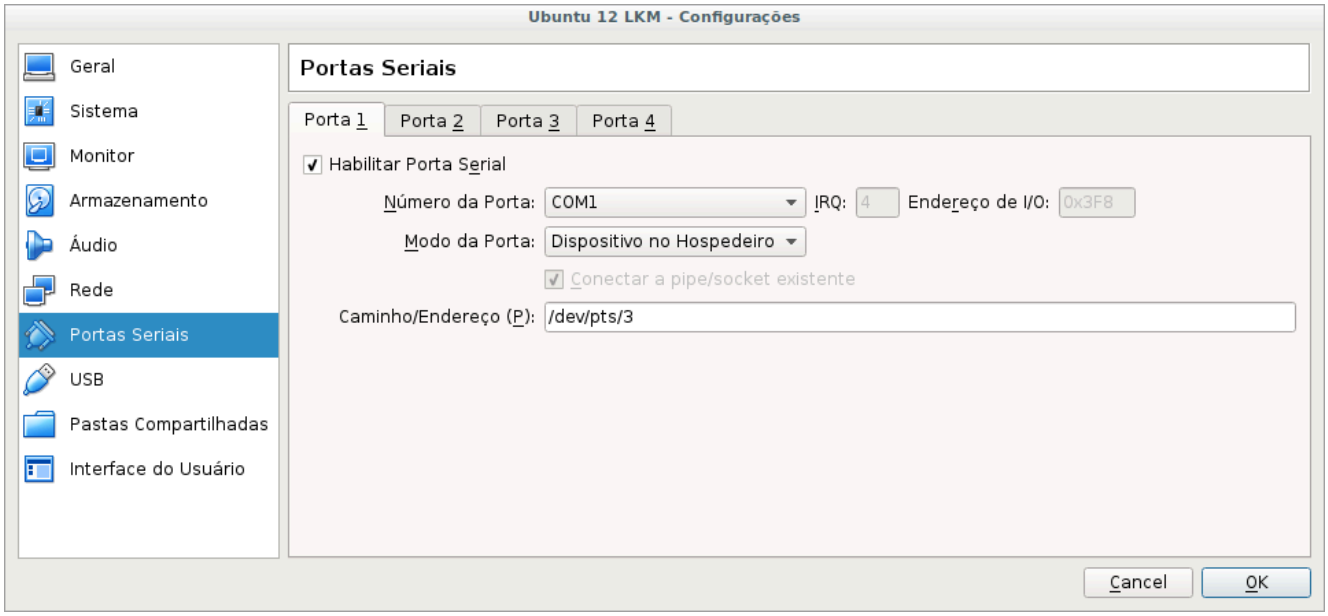
Por fim, objetos de classe Tun se integram perfeitamente com o poller. Pode-se definir um Callback que monitore um objeto Tun, para que possam ser detectados os quadros a serem enviados. Este último exemplo mostra um Callback feito para demonstrar como integrar a interface tun com o poller.

```
import poller
import sys,time
from tun import Tun

class CallbackTun(poller.Callback):

    def __init__(self, tun):
        poller.Callback.__init__(self, tun.fd, 1000)
        self._tun = tun

    def handle(self):
        l = self._tun.get_frame()
        print('Lido:', l)

    def handle_timeout(self):
        print('Timeout !')

tun = Tun("tun0","10.0.0.1","10.0.0.2",mask="255.255.255.252",mtu=1500,qlen=4)
tun.start()

cb = CallbackTun(tun)
sched = poller.Poller()

sched.adiciona(cb)

sched.despache()
```

4.1. Plataforma de desenvolvimento e teste com Virtualbox

O teste de comunicação através de um enlace entre duas interfaces *tun* demanda que essas interfaces residam em hosts diferentes. Se ambas interfaces estiverem no mesmo host, a comunicação se dará na realidade através da interface *loopback*. Uma plataforma de teste simplificada pode ser implantada usando o Virtualbox:

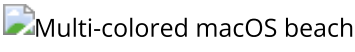
1. **Na máquina real execute o serialemu:** uma das pontas da serial emulada deve ser usada pelo protocolo na máquina real.
2. **Porta serial na máquina virtual**: antes de executar a máquina virtual, habilite sua primeira porta serial e associe-a à outra serial do serialemu (ver figura a seguir).
3. **Porta serial dentro da máquina virtual**: uma vez iniciada a máquina virtual, deve-se nela executar o protocolo de forma que use a porta serial */dev/ttyS0*.



Configuração da porta serial na máquina virtual: clique na máquina virtual com o botão da direita, e selecione o menu Configurações (ou Settings)

5. Programação assíncrona

Normalmente, o código de um programa é executado de forma direta, com uma coisa acontecendo por vez. Se uma função depende do resultado de outra função, ela tem que esperar o retorno do resultado. Até que isso aconteça, o programa inteiro praticamente para de funcionar, visto da perspectiva do usuário.
Usuários do Mac e do Ubuntu Linux, por exemplo, conseguem ver isso como o cursor giratório em arco-íris (ou "*beachball*", como normalmente é chamado). Este cursor é o jeito do sistema operacional dizer: "o programa atual que você está usando teve que parar e esperar algo terminar de ser executado, e estava demorando tanto que fiquei preocupado se você estava pensando no que aconteceu."


Multi-colored macOS beachball busy spinner

Essa é uma situação que não faz bom uso do poder de processamento do computador — especialmente em uma era em que computadores tem múltiplos núcleos de processamento disponíveis. Não há sentido em ficar esperando por algo quando outra tarefa pode ser executada em um núcleo de processador diferente, e receber depois uma notificação quando terminar. Isso possibilita fazer mais coisas durante esse tempo, o que é a base da **programação assíncrona**. Dependendo do ambiente de programação se usa, existem APIs que possibilitam executar essas tarefas de forma assíncrona.

Um programa orientado-a-eventos (*event-driven*) é escrito de forma a reagir a eventos. Um evento diz respeito a qualquer acontecimento que pode desencadear ações em um programa, tais como a recepção de uma mensagem ou a ocorrência de um *timeout*. O conceito de programação orientada-a-eventos, que aqui será chamada de programação assíncrona, é importante para o desenvolvimento de certos tipos de aplicações, em que o processamento acontece comandado por eventos.

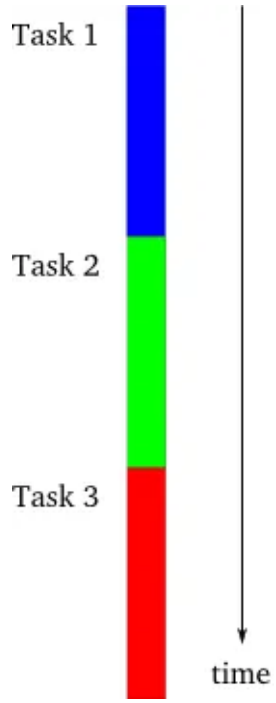
O termo programação assíncrona se refere à forma com que tais programas são escritos. Em especial, isso diz respeito ao comportamento do programa ao realizar uma chamada, ou iniciar uma operação, cujo término depende de algo ocorrer em algum instante futuro. Por exemplo, a leitura de um comando pelo terminal depende de o usuário digitar algo para que se complete. Se a forma de o programa realizar esse tipo de operação implicar um bloqueio, em que se aguarda até que a operação se complete, então o tipo de chamada é denominado síncrono. Em uma chamada síncrona, o programa fica ocioso enquanto espera pelo término da operação, sem realizar qualquer processamento. Porém, se o programa não bloqueia ao iniciar uma operação, podendo executar outras ações enquanto o término da operação original não acontece, tal forma de realizar uma chamada é dita assíncrona. Quando essa chamada concluir, isso é informado ao programa de alguma maneira, e assim ele pode dar a operação por completada. Tomando o exemplo da leitura pelo teclado, o programa ativa a espera por algo ser digitado e, quando isso acontecer, alguma rotina do programa é invocada para tratar os dados lidos. A programação assíncrona é portanto um termo usado para designar métodos e técnicas para a modelagem e escrita de programas cujas interações com seu ambiente computacional se valem de chamadas assíncronas.
Em geral, programas são escritos para responder a ações de usuários e do ambiente em que se encontram. No caso de software de rede esse aspecto é proeminente, pois esse tipo de software reage a recepções e envios de mensagens. Por isso a identificação de aspectos relacionados a eventos nesse tipo de software pode ser de grande ajuda em sua análise e projeto.

5.1. Modelos de execução

Existem dois modelos de execução de tarefas mais conhecidos, e a revisão desses modelos, que são síncronos, pode ajudar a entender o que significa um modelo de execução assíncrona. Para descrevê-los, tome-se como exemplo um programa que consiste de três tarefas distintas que devem ser realizadas para que o programa conclua. O termo tarefa aqui se refere *algo que deve ser feito*. Em seguida, esses modelos serão comparados com um modelo assíncrono.

Modelos de execução síncrona

O primeiro modelo é um modelo síncrono com uma *thread* (*single-threaded*), mostrado na figura a seguir:



O modelo síncrono

Esse é o estilo mais simples de programação. As tarefas são realizadas uma por vez, o que significa que uma tarefa só é iniciada quando outra é terminada. Se as tarefas são realizadas em uma ordem definida, a execução de uma tarefa pode supor que as tarefas anteriores terminaram sem erros, e que seus resultados estão disponíveis para serem utilizados - o que simplifica sobremaneira a lógica do programa.

Em contraste ao modelo com uma tarefa, existe o modelo com múltiplas *threads* (*multi-threaded*), mostrado na próxima figura:

O modelo com múltiplas threads

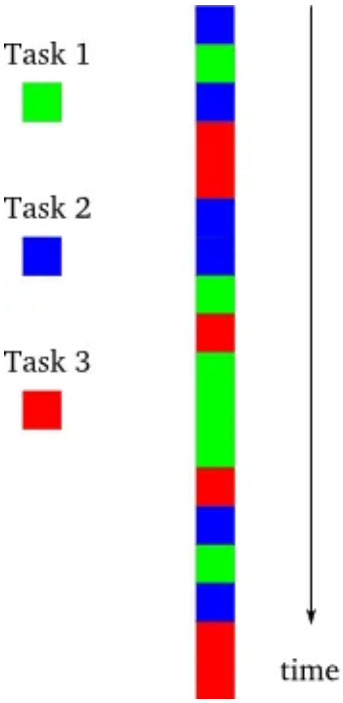
Nesse modelo, cada tarefa é executada em uma *thread* em separado. As *threads* são gerenciadas pelo sistema operacional e, em um sistema com múltiplos processadores ou núcleos, podem ser executadas de forma realmente concorrente. Em um sistema com um processador com um único núcleo, elas podem ser executadas

de forma entrelaçada, revezando o uso do processador a critério do sistema operacional. O importante é que, nesse modelo com múltiplas *threads*, os detalhes da execução são tratados pelo sistema operacional e o programador pode simplesmente pensar que suas tarefas são executadas de forma independente e simultaneamente. Apesar de a situação parecer simples, como mostrado na figura acima, na prática programas com múltiplas *threads* podem ser bastante complexos devido à necessidade de coordenar as *threads* entre si. A comunicação entre *threads*, e sua coordenação, é um tópico avançado de programação e pode ser difícil de escrever um programa correto.

Alguns programas implementam paralelismo usando múltiplos processos ao invés de múltiplas *threads*. Apesar de os detalhes de programação serem diferentes, isso pode ser considerado o mesmo modelo de programa.

O modelo de execução assíncrona

Um modelo de execução assíncrona pode ser resumido na figura a seguir:



Um modelo assíncrono

Nesse modelo, as tarefas são intercaladas, porém em uma única *thread* de execução. Cada tarefa é representada por uma **corotina**, que pode ter sua execução interrompida (ser pausada), e futuramente voltar a executar do ponto onde parou. Isso é mais simples que o modelo com múltiplas *threads*, porque o programador sempre sabe que, quando uma tarefa executa, as outras outras estão sem executar. Note-se a diferença em relação a um programa com múltiplas *threads* em um sistema com único processador ou núcleo, em que as *threads* também são intercaladas porém sob comando do sistema operacional. Assim, se o programador supor que apenas uma tarefa está ativa por vez, porque o programa está sendo executado em um sistema mono-processado, seu programa irá funcionar incorretamente se for executado em um sistema multi-processado. No caso de um programa assíncrono, mesmo em um sistema multi-processado apenas uma tarefa está em execução por vez.

Existe outra diferença entre esses dois modelos de execução. Em um programa com múltiplas *threads*, a decisão de suspender uma *thread* e executar outra está em larga medida fora do controle do programador. Quem controla esse chaveamento é o sistema operacional, e o programador deve presumir que uma *thread* pode ser suspensa e substituída por outra em qualquer instante. Porém no modelo assíncrono, uma tarefa continua a executar até que explicitamente libere o controle para outra tarefa. Essa é uma simplificação adicional em relação ao caso com múltiplas *threads*.

Referências

- Este texto foi adaptado deste artigo: In Which We Begin at the Beginning.
- Este outro texto apresenta uma boa introdução sobre execução assíncrona, incluindo o conceito de corotinas.

5.2. Modelo de execução assíncrona para o protocolo

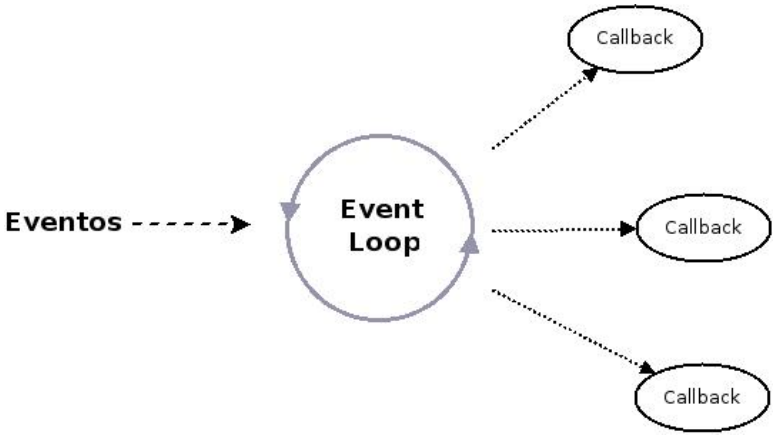
Em um primeiro momento, pode parecer que um protocolo pode ser implementado segundo um modelo de execução síncrono. Porém isso traz grandes dificuldades para execução das ações do protocolo, as quais dependem de eventos aos quais o protocolo deve reagir. Por exemplo:

- Um protocolo deve ser capaz de receber uma mensagem a qualquer momento, e notificá-la para a aplicação que a consumirá
- Comunicações bidirecionais dependem de mensagens poderem ser enviadas e recebidas de forma independente
- Certas funcionalidades do protocolo podem se tornar complexas, ou mesmo inviáveis, com um modelo síncrono (ex: mecanismos de manutenção de conexão, como a troca de mensagens que informam o status das entidades conectadas)

O modelo de comportamento de protocolos, como abordado neste estudo, se baseia em máquinas de estados finitos comunicantes. Em cada estado, o protocolo deve reagir a eventos e realizar ações, possivelmente mudando de estado. O modelo assíncrono (ou orientado a eventos) está naturalmente representado nessa forma de descrever o comportamento de um protocolo. Por isso, a escrita do software do protocolo se torna mais natural se for adotado um modelo assíncrono.

Modelo assíncrono em software

Seguindo um paradigma orientado-a-eventos (*event-driven*), o protocolo fica no controle das ações, detectando eventos para tratá-los. Cada evento detectado assincronamente é encaminhado para o bloco funcional a que se destina. Nenhuma ação do protocolo é bloqueante, pois eventos devem ser tratados o mais rápido possível. Um protocolo que funcione dessa forma deve portanto possuir algum mecanismo de detecção e encaminhamento de eventos.



*Um loop de eventos: eventos são detectados e encaminhados para tratadores de eventos (**callbacks)** previamente definidos*

Existem APIs para programação assíncrona para algumas linguagens de programação, com as quais se pode escrever um programa que segue esse modelo. Em geral, essas APIs possuem um mecanismo interno que implementa um loop de eventos. Os detalhes de como os eventos são detectados, e como eles desencadeiam a execução de tratadores de eventos, são abstraídos.

- **C++**: Asio
- **Python**: asyncio, twisted
- **Java**: algumas APIs ...

Para nosso estudo, usaremos uma API própria, e que foi escrita para fins puramente didáticos. Dentro do possível, ela procura simplificar a escrita de programas orientados a eventos.

A API Poller

- Código-fonte do Poller (C++)

A API Poller foi criada inicialmente para a linguagem C++, e ela oferece uma abstração simplificada para a escrita de programas orientados a eventos, com uma abordagem similar ao *design pattern* Reactor. O *Poller* monitora um conjunto de descritores de arquivos (ou sockets) para fins de leitura. Cada descritor é associado a algum bloco funcional do protocolo por meio de um Callback, o qual inclui também um tempo máximo de espera por dados nesse descritor. Quando um descritor tem dados para serem lidos, o *Poller* executa o *Callback* correspondente, que tem a responsabilidade de realizar a leitura do descritor e processar os valores lidos. Se um timeout ocorrer, o *Poller* executa o *Callback* correspondente, para que possa tratar o timeout. Por fim, se um descritor for um número negativo, o *Poller* o considera um *timer* (uma vez que somente seu timeout deve ser levado em conta). Esse *Poller* está declarado como mostrado a seguir:

```
// Poller: um despachador de eventos
// Um objeto poller é capaz de monitorar um conjunto de descritores de arquivos
// e executar um callback para cada desccritor pronto para acesso
// Cada descritor pode especificar um timeout próprio
class Poller {
 public:
  Poller();
  ~Poller();

  // adiciona um evento a ser vigiado, representado por um Callback
  void adiciona(Callback * cb);

  // remove callback associado ao descritor de arquivo fd
  void remove(int fd);
  void remove(Callback * cb);

  // remove todos callbacks
  void limpa();

  // vigia os descritores cadastrados e despacha os eventos (chama os callbacks)
  // para ser lido, ou até que expire o timeout (em milissegundos)
  void despache_simples();
  void despache();
};
```

Um *Callback*, por sua vez, é definido pela classe abstrata *Callback*. Cada tipo de evento deve ter seu *Callback* específico definido como uma especialização dessa classe.

```
// classe abstrata para os callbacks do poller
class Callback {
public:
    // fd: descritor de arquivo a ser monitorado. Se < 0, este callback é um timer
    // tout: timeout em milissegundos. Se < 0, este callback não tem timeout
    Callback(int fd, long tout);

    // cria um callback para um timer (equivalente ao construtor anterior com fd=-1)
    // out: timeout
    Callback(long tout);

    // ao especializar esta classe, devem-se implementar estes dois métodos !
    // handle: trata o evento representado neste callback
    // handle_timeout: trata o timeout associado a este callback
    virtual void handle() = 0;
    virtual void handle_timeout() = 0;

    // operator==: compara dois objetos callback
    // necessário para poder diferenciar callbacks ...
    virtual bool operator==(const Callback & o) const;

    // getter para o descritor de arquivo a ser monitorado
    int filedesc() const;

    // getter do valor de timeout remanescente
    int timeout() const;

    // ajusta timeout restante
    void update(long dt);

    // recarrega valor de timeout inicial
    void reload_timeout();

    // desativa timeout
    void disable_timeout();

    // reativa timeout
    void enable_timeout();

    // desativa ou ativa o monitoramento de dados no descritor de arquivos deste callback
    // o timeout não é afetado
    virtual void disable();
    virtual void enable();

    // informa se o monitoramento de dados no descritor de arquivos deste callback está ativado
    bool is_enabled() const;

    // informa se o timeout deste callback está ativado
    bool timeout_enabled() const;
};
```

Um exemplo para uso do Poller mostra um programa que reage a eventos do tipo digitação no teclado pelo usuário. Esse programa precisa de um único *Callback*, chamado de *CallbackStdin*. Sempre que o usuário digitar

alguma coisa e pressionar ENTER, executa-se o método *handle* de *CallbackStdin*, que lê o texto digitado e apresenta-o na tela. Se o usuário ficar mais que alguns segundos sem nada digitar, o tratador *handle_timeout* é executado, e apresenta uma mensagem de *timeout* na tela. Veja o código desse exemplo a seguir.

```
#include <iostream>
#include <string>
#include "poller.h"

using namespace std;

class CallbackStdin: public Callback {
 public:
  CallbackStdin(long tout): Callback(0, tout) {}

  void handle() {
    string w;

    getline(cin, w);
    cout << "Lido: " << w << endl;
  }

  void handle_timeout() {
     cout << "Timeout !!!" << endl;
  }

};

int main() {
  // cria o objeto CallbackStdin, com timeout de 5000ms
  CallbackStdin cb(5000);
  //cria um poller: ele contém um loop de eventos !
  Poller sched;

  // registra o callback no poller
  sched.adiciona(&cb);

  // entrega o controle ao poller.
  // Ele irá esperar pelos eventos e executará
  // os respectivos callbacks
  sched.despache();
}
```

A API Poller em Python

As classes Poller e Callback estão implementadas em Python, porém explorando algumas facilidades existentes nessa linguagem. Ambas estão disponíveis no módulo *poller*:

- Módulo poller
- Documentação sobre módulo poller *(obtida com pydoc)*

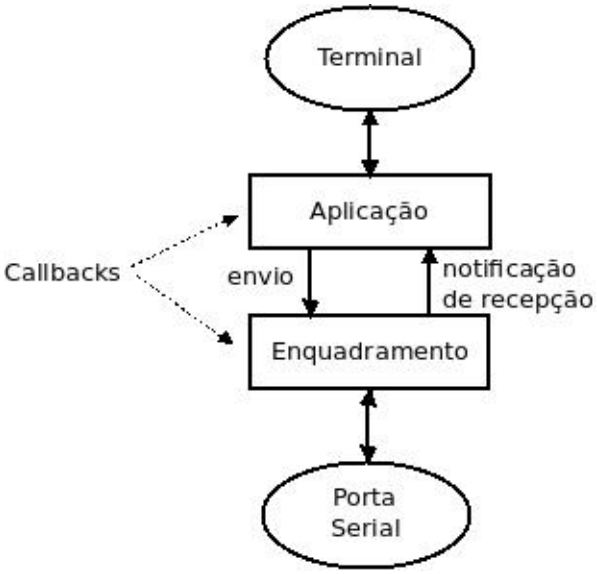
Um programa de demonstração parecido com aquele em C++ foi criado para mostrar o uso da classe Poller:

```
#!/usr/bin/env python3

import poller
import sys,time

# declara um Callback capaz de fazer leitura do teclado, e com tempo limite
class CallbackStdin(poller.Callback):

    def __init__(self, tout:float):
        'tout: tempo limite de espera, em segundos'
        poller.Callback.__init__(self, sys.stdin, tout)
        self._cnt = 0

    def handle(self):
        'O tratador do evento leitura'
        l = sys.stdin.readline()
        print('Lido:', l)
        self._cnt = 0

    def handle_timeout(self):
        'O tratador de evento timeout'
        self._cnt += 1
        print(f'Timeout: cnt={self._cnt}')
        if self._cnt == 3:
          # desativa o timeout deste callback, e também o evento leitura !
          self.disable_timeout()
          self.disable()

####################################

# instancia um callback
cb = CallbackStdin(3)

# cria o poller (event loop)
sched = poller.Poller()

# registra o callback no poller
sched.adiciona(cb)

# entrega o controle pro loop de eventos
sched.despache()
```

5.3. Uma arquitetura para o protocolo

Um modelo de execução assíncrona traz benefícios para a escrita do protocolo, como discutido na seção anterior. Para fins do projeto do protocolo de enlace, considera-se o uso da API Poller, apesar de outra API para programação assíncrona ser possível (ex: asyncio ou twisted, em Python, asio em C++ ou tokio em Rust). A API Poller tem projeto simplificado, sendo destinada a uso didático, e por isso foi a escolhida para ilustrar a explicação sobre o projeto do protocolo.

Antes de prosseguir na escrita do protocolo, torna-se necessário pensar em como seu software será construído. Em outras palavras, qual a arquitetura do protocolo, a qual descreve os elementos de software a serem utilizados, incluindo seus relacionamentos. Para esclarecer que arquitetura seria apropriada, inicia-se com a versão inicial do protocolo. Ela ainda não tem uma estrutura, pois se resume ao enquadramento. Ela pode ser representada pela figura a seguir.



A versão inicial do protocolo, que contém somente o enquadramento

Segundo esse diagrama, o protocolo até o momento pode ser entendido como um bloco funcional que implementa o enquadramento, e outro faz o papel de aplicação. O enquadramento envia e recebe quadros pela porta serial, e a aplicação lê e apresenta sequências de caracteres por meio do terminal. Assim, a aplicação lê alguns caracteres do terminal, e os envia para o enquadramento, que os encapsula em um quadro e envia pela porta serial. Em sentido contrário, o enquadramento recebe octetos da porta serial de forma a reconhecer um quadro, quando então desencapsula os dados nele contidos e os entrega para a aplicação. Essa, por sua vez, ao receber dados vindos do enquadramento os apresenta no terminal.

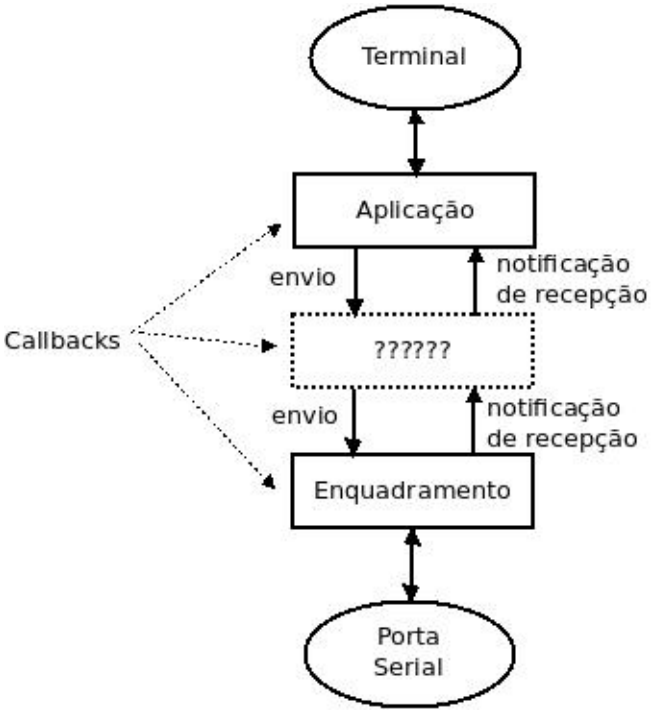
Uma implementação como essa para o protocolo, usando a API Poller, implica dois Callbacks:

- *Enquadramento*: trata eventos do tipo octeto disponível na porta serial, e timeouts
- *Aplicação*: trata eventos do tipo caracteres disponíveis no terminal (entrada padrão)

Observando o diagrama, ambos Callbacks devem estar relacionados. A *Aplicação* deve enviar sequências de caracteres para *Enquadramento*, e o *Enquadramento* deve notificar a *Aplicação* sobre a recepção de dados. Do ponto de vista de software, bastaria associar *Aplicação* a *Enquadramento*, e vice-versa (por exemplo, com uma relação de agregação). Isso poderia resolver este problema imediato, mas e se o protocolo precisar ser estendido ?

O protocolo de enlace deve ter outras funcionalidades além de fazer enquadramento. Por exemplo, seu serviço pode incluir detecção de erros, garantia de entrega e controle de sessão (pelo menos). A extensão do protocolo para conter esses mecanismos deve implicar mínimas alterações em seu software ... idealmente, nenhuma modificação deveria ser feita nas funcionalidades já implementadas. Isso demanda refletir um pouco mais sobre a arquitetura do protocolo.

O enquadramento implementa a funcionalidade mais elementar do protocolo, que é a comunicação usando quadros. Outros mecanismos a serem incorporados devem usar o serviço oferecido pelo enquadramento. A garantia de entrega, por exemplo, implica a confirmação de quadros enviados, e a retransmissão de quadros perdidos. Esse outro mecanismo é orientado a quadros, seja de dados ou confirmação, e por isso depende do enquadramento. O controle de sessão, que corresponde à capacidade do protocolo estabelecer uma conexão, também implica trocas de mensagens de controle, e por isso depende da capacidade de comunicação com quadros. Então é razoável imaginar que o enquadramento é a base para os demais mecanismos. A figura a seguir ilustra o protocolo sendo estendido para acomodar algum novo mecanismo, seguindo essa conclusão de que, provavelmente, dependerá do enquadramento.



O protocolo com algum novo mecanismo

De acordo com esse diagrama, um novo mecanismo poderia se inserir entre o enquadramento e a aplicação. Com isso, a aplicação enviaria seus dados para esse mecanismo, que faria algum processamento e os enviaria para o enquadramento. O fluxo de dados para envio, assim, passaria por aplicação, seguida do novo mecanismo, e então o enquadramento. Em sentido oposto, a recepção passaria por enquadramento, o novo mecanismo, e então a aplicação. Se mais algum mecanismo for acrescentado ao protocolo, pode-se imaginar que um novo bloco deve se inserir entre enquadramento e aplicação, e os fluxos de envio e recepção terão mais um estágio de processamento. Isso sugere que o software do protocolo poderia ser modelado com uma arquitetura em camadas.

O protocolo em camadas

Um modelo em camadas para o protocolo implica algumas considerações:

- Cada camada deve ser um *Callback*, para que possa interagir com a API Poller
- Cada camada deve definir uma operação para enviar dados vindos da camada superior
- Cada camada deve definir uma operação para receber dados vindos da camada inferior
- Cada camada deve possuir uma referência à sua camada inferior, e também à superior

Ao invés de escrever cada camada para que atenda esses requisitos, uma saída mais simples seria definir o que seria uma camada genérica, e então especializá-la para cada camada concreta a ser usada no protocolo. Isso simplificaria a escrita do protocolo, e sua extensão para acomodar novos mecanismos. Essa camada genérica poderia ser modelada como a classe Subcamada, no diagrama de classes a seguir.

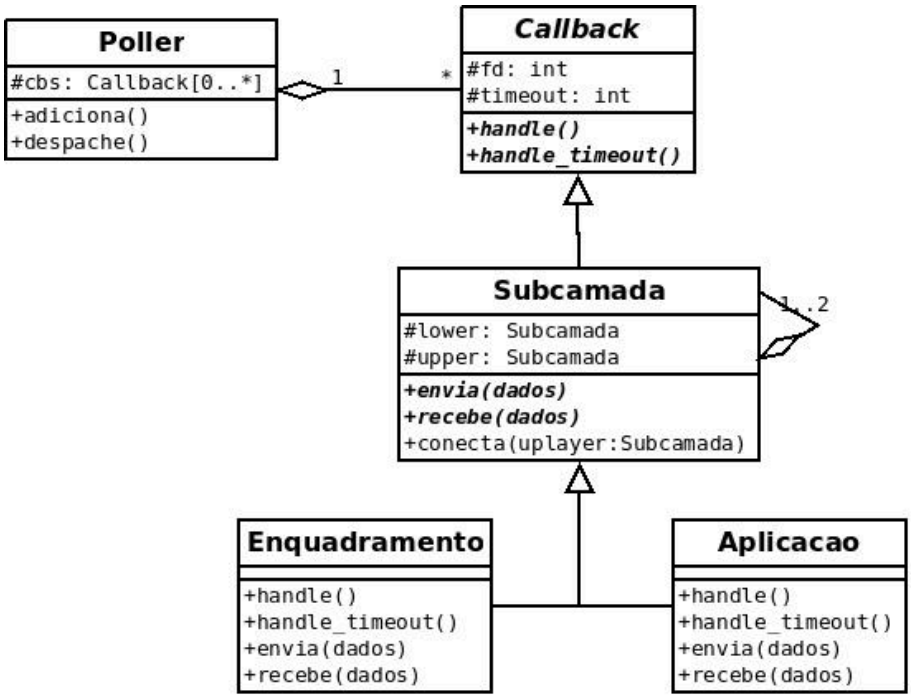


Diagrama de classes com a classe Subcamada, e classes Enquadramento e Aplicacao como suas especializações

Supondo que a classe *Subcamada* já exista, e esteja definida como no diagrama de classes acima, um esboço da classe *Enquadramento* poderia ser escrito da seguinte forma:

```
class Enquadramento(Subcamada):

    def __init__(self, porta_serial:Serial, tout:float):
      Subcamada.__init__(self, porta_serial, tout)
      # dev: este atributo mantém uma referência à porta serial
      self.dev = porta_serial
      # buffer: este atributo armazena octetos recebidos
      self.buffer = bytearray()

    def handle(self):
      # lê um octeto da serial, e o armazena no buffer
      # Encaminha o buffer para camada superior, se ele tiver 8 octetos
      octeto = self.dev.read(1)
      self.buffer += octeto
      if len(self.buffer) == 8:
        # envia o conteúdo do buffer para a camada superior (self.upper)
        self.upper.recebe(bytes(self.buffer))
        self.buffer.clear()

    def handle_timeout(self):
      # Limpa o buffer se ocorrer timeout
      self.buffer.clear()
      print('timeout no enquadramento em', time.time())

    def envia(self, dados:bytes):
      # Apenas envia os dados pela serial
      # Este método é chamado pela subcamada superior
      quadro = bytearray()
      quadro.append(0x7e)
      quadro += dados
      quadro.append(0x7e)
      self.dev.write(bytes(quadro))
```

A classe *Aplicacao*, por sua vez, poderia ser escrita assim:

```
class Aplicacao(Subcamada):

    def __init__(self):
        Subcamada.__init__(self, sys.stdin)

    def recebe(self, dados:bytes):
      # mostra na tela os dados recebidos da subcamada inferior
      print('RX:', dados)

    def handle(self):
      # lê uma linha do teclado
      dados = sys.stdin.readline()

      # converte para bytes ... necessário somente
      # nesta aplicação de teste, que lê do terminal
      dados = dados.encode('utf8')

      # envia os dados para a subcamada inferior (self.lower)
      self.lower.envia(dados)
```

Juntando tudo, um programa que executasse a aplicação e o enquadramento poderia ser este:

```
import poller
import sys
from proto1 import Subcamada
from enquadramento import Enquadramento
from aplicacao import Aplicacao
from serial import Serial

Timeout = 15 # 15 segundos

# nome da porta serial informada como primeiro argumento
# de linha de comando
porta = Serial(sys.argv[1])

# cria objeto Enquadramento
enq = Enquadramento(porta, Timeout)

# Cria objeto Aplicacao
app = Aplicacao()

# Conecta as subcamadas
# Deve ser feito a partir da subcamada inferior
enq.conecta(app)

# cria o Poller e registra os callbacks
sched = poller.Poller()
sched.adiciona(enq)
sched.adiciona(app)

# entrega o controle ao Poller
sched.despache()
```

Veja estes arquivos para uma versão desse exemplo em C++:

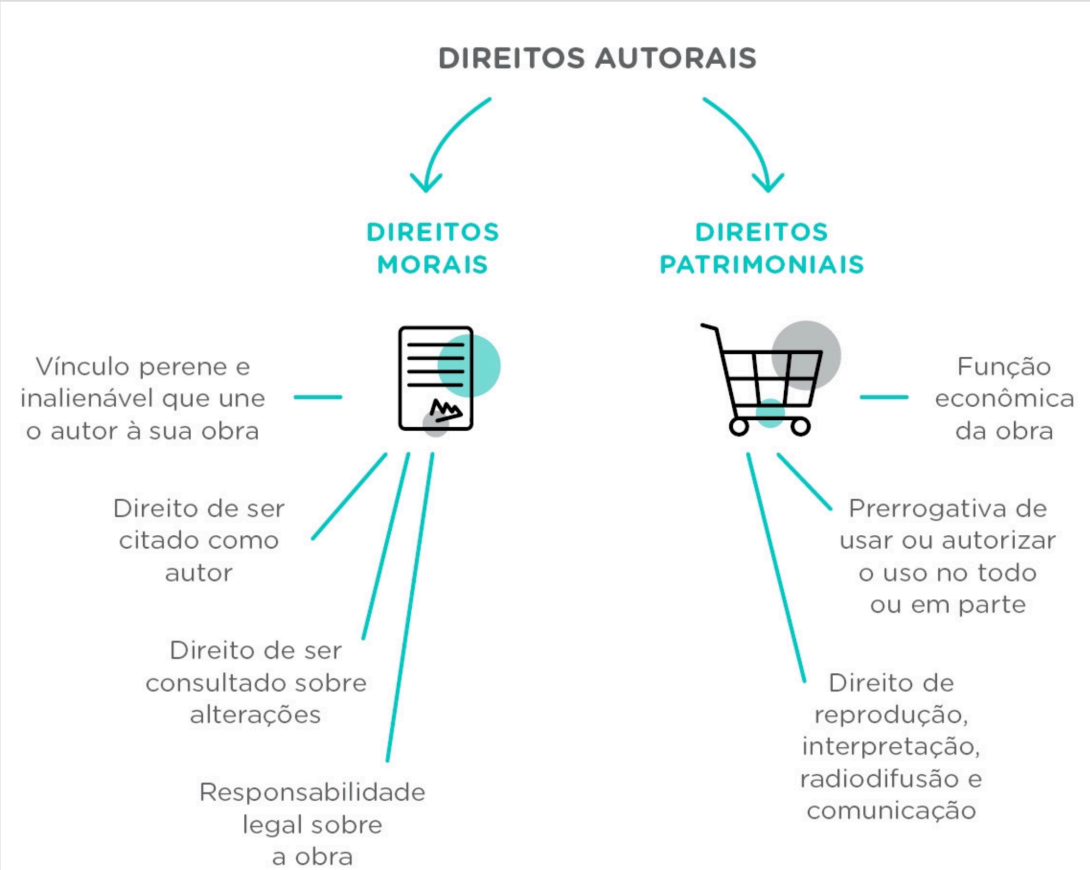
- main_demo.cpp
- aplicacao.h
- enquadramento.h

Essa estrutura para o software do protocolo pode parece um pouco demais à primeira vista. Talvez houvesse uma forma mais simples de implementá-lo, sem necessidade de amarrar o código a essas classes. Porém o protocolo possui vários mecanismos para que seu serviço seja efetivamente oferecido, e eles devem ser desenvolvidos gradualmente como extensões do protocolo. O modelo de software proposto prevê essa necessidade, e direciona a escrita do código para que os mecanismos sejam feitos como peças que são encaixadas no protocolo. Em outras palavras, esses outros mecanismos a serem futuramente incluídos no protocolo serão implementados de forma modular, apresentando coesão e baixo acoplamento. Com isso, a implementação se torna ao final mais simples.

6. Direitos autorais

Ao desenvolver os materiais para sua unidade curricular, fique atento(a) ao que preza a Lei de Direitos Autorais (Lei Nº 9.610/1998).

No caso de uso de materiais de terceiros é importante observar se você está citando pequenos trechos ou utilizando pedaços de obra o que pode ferir o direito patrimonial - isto é, de quem detém os direitos de reprodução da obra.



Legenda: Aspectos dos direitos morais e patrimoniais na preparação de conteúdos para EaD

Fonte: FILATRO (2018)

É preciso analisar caso a caso quando o assunto é utilização de materiais de terceiros. Nem sempre, citar a fonte garante que que você esteja cumprindo a lei.

E os seus direitos?

Quando se trata dos seus direitos autorais, tudo vai depender do grau de autoria do recurso que você está produzindo. Ela pode ser uma obra autoral ou uma obra derivada de outras, por exemplo. Em todo caso, mesmo que você faça um *mash-up* de recursos de terceiros disponíveis para uso, mesclados com textos seus, procura-se

garantir que o seu nome apareça na ficha técnica como autora do objeto de aprendizagem (no caso o livro digital). Veja nas páginas a seguir um exemplo de ficha técnica e mais detalhes sobre o tipo de licença que o IFSC adota em seus materiais didáticos: a **Creative Commons**.

6.1. Modelo de ficha técnica

Ficha técnica de primeira versão

A ficha abaixo é utilizada em obras aqui no Moodle que estejam na sua primeira versão, ou seja, que não são derivadas de outras.

Título: Inserir um título para o livro	
VERSÃO ORIGINAL - 2019.1	
Autoria	Nome e Sobrenome
Design educacional	Nome e Sobrenome
Design multimídia	Nome e Sobrenome
Revisão textual	Nome e Sobrenome

Ficha técnica para versões derivadas

A ficha abaixo é utilizada em obras derivadas de outras. Nesse caso, indica-se a equipe que trabalhou na versão atual. Mantém-se a indicação de quem fez a versão anterior (de onde o livro foi replicado) e a indicação da versão original (a primeira versão). Não é preciso indicar todo o histórico de versões, apenas a versão imediatamente anterior e a versão original. Nas versões anteriores, deve-se inserir o link dos respectivos livros no Moodle. Os alunos não terão acesso às versões anteriores, mas o link é necessário para fins de comparação legal. Caso tenhamos que diferenciar o que é obra do autor original e o que foi adaptado, ao comparar as edições, podemos identificar tais mudanças. Esta indicação atende ao requisito de obras derivadas da licença Creative Commons. Caso a sua obra seja a primeira derivação da original, basta excluir a "versão anterior".

Título: Inserir um título para o livro	
VERSÃO ATUAL - 2019.1	
Adaptação e atualização	Nome e Sobrenome
Design educacional	Nome e Sobrenome
Design multimídia	Nome e Sobrenome Nome e Sobrenome
Revisão textual	Nome e Sobrenome

VERSÃO ANTERIOR - 2018.1 (link)	
Adaptação e atualização	Nome e Sobrenome

VERSÃO ORIGINAL - 2016.1 (link)	
Autoria	Nome e Sobrenome Nome e Sobrenome

6.2. Tipos de licenciamento Creative Commons

A Coordenadoria de Materiais Didáticos adota as recomendações da Capes e de outros órgãos vinculados ao Ministério da Educação que tratam do uso do conjunto de licenças chamado **Creative Commons**.

Licenças Creative Commons

Há diferentes tipos de licença, das mais restritivas até o domínio público.

Clique aqui e conheça cada um dos tipos de licença

Nossa sugestão é que você utilize a licença **Atribuição-NãoComercial-CompartilhaIgual - CC BY-NC-SA.** Essa licença permite que outros remixem, adaptem e criem a partir do seu trabalho para fins não comerciais, desde que atribuam a você o devido crédito e que licenciem as novas criações sob termos idênticos. Isto significa que eu posso ampliar e adaptar o material que você produziu para o meu contexto, mas você continuará sendo citado(a) como autor da versão original (lembra da ficha técnica que vimos antes?). O material que eu criar (derivado), por sua vez, deverá manter o mesmo tipo de licença, permitindo que outras pessoas adaptem o que eu criei.

> **É essencial que todos os materiais que você produzir estejam com a ficha técnica indicando a equipe que trabalhou na elaboração e com a informação do tipo de licença daquele material.**

7. Ficha técnica

Título: Um livro sobre a ferramenta livro	
VERSÃO ORIGINAL - 2019.1	
Autoria	Luís Henrique Lindner