FACULDADE DE CIÊNCIAS E TECNOLOGIA
UNIVERSIDADE NOVA DE LISBOA

Departamento de Engenharia Electrotécnica

Sistemas de Telecomunicações

2013/2014

Trabalho 2:

Protocolos de nível Lógico com janela deslizante

Mestrado integrado em Engenharia Eletrotécnica e de Computadores

http://tele1.dee.fct.unl.pt/st

Índice

1. OBJETIVO

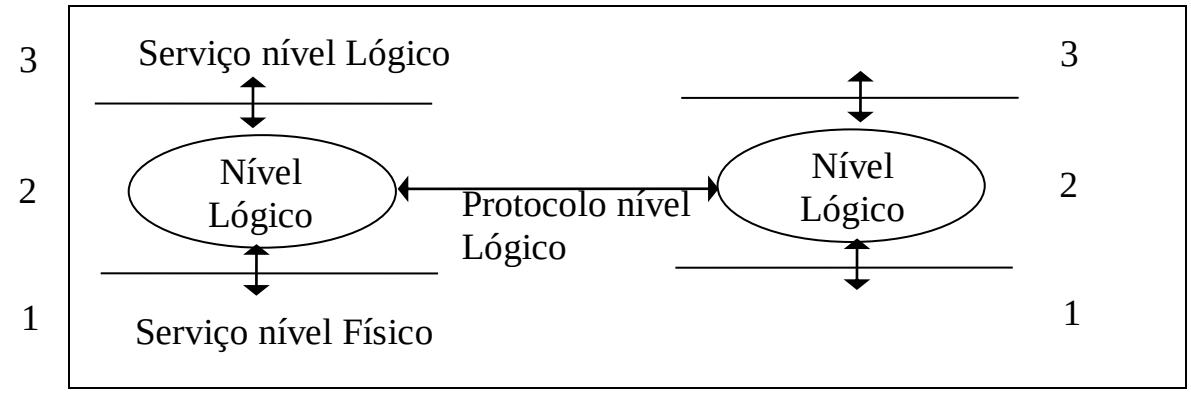
Familiarização com os protocolos de nível lógico baseados em janela deslizante do tipo Stop&Wait, Go-Back-N e Retransmissão Seletiva.

O trabalho consiste no desenvolvimento de um protocolo de nível lógico baseado em janela deslizante, do tipo Retransmissão Seletiva (*Selective Repeat*), de forma faseada. Para isso, é fornecido um simulador de sistema desenvolvido na disciplina sobre sockets TCP, que simula o funcionamento do protocolo de nível rede e de nível físico.

> **Sugestões**: Em certas partes do enunciado aparece algum texto formatado de um modo diferente que começa com a palavra "Sugestões". Não é obrigatório seguir o que lá está escrito, mas pode ser importante para os alunos ou grupos onde ainda não haja um à-vontade muito grande com programação, estruturas de dados e algoritmia.

2. ESPECIFICAÇÕES

Pretende-se desenvolver um protocolo de nível Lógico para uma ligação física ponto-a-ponto, que interage com os protocolos de nível Rede e de nível Físico através respetivamente das interfaces dos serviços de nível Lógico e de nível Físico. Desta forma vai ser simulado o sistema representado abaixo, através do conjunto de primitivas dos dois serviços.



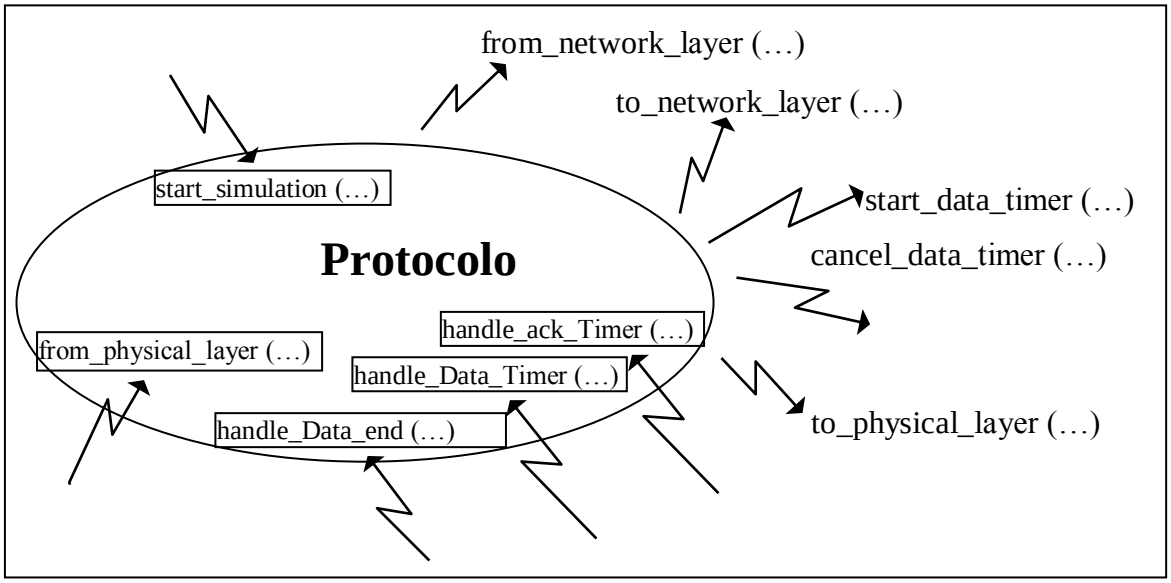
A abordagem proposta foi inspirada no simulador descrito nas secções 3.3 e 3.4 do livro recomendado (Computer Networks 5ª edição), embora faça uma simulação um pouco mais próxima da realidade (e.g. considera o tempo de transmissão das tramas de dados). O protocolo de nível lógico vai reagir a eventos e pode invocar comandos.

O nível Lógico começa com o evento que é servido pelo método `start_simulation()`. Para receber pacotes (*strings*) do nível Rede, o nível Lógico usa o método `from_network_layer()` e para enviar pacotes (*strings*) para o nível Rede invoca o método `to_network_layer()`. Os dados devem ser entregues pela mesma ordem em que foram recebidos, recuperando de erros do canal de nível Físico.

O protocolo pode enviar tramas para o nível Físico usando `to_physical_layer()` e pode receber tramas do nível Físico no seu método `from_physical_layer()`, invocado pelo protocolo de nível Físico.

Por fim, pode usar um conjunto de funcionalidades de suporte para gerir temporizadores. Usando os métodos `set_data_timer` e `cancel_data_timer`, pode armar ou cancelar um temporizador identificado por um número maior ou igual a zero (designado de *key*), embora

neste trabalho apenas se use um único temporizador com um valor de chave fixo. Quando o tempo expira, o método `handle_Data_Timer()`, é chamado. A figura em baixo ilustra o que se acabou de descrever. Os alunos devem programar o balão designado por "Protocolo".



2.1. TIPOS DE TRAMAS

O protocolo pode enviar ou receber dois tipos de tramas: DATA ou ACK.

Tramas de dados (DATA)

As tramas de dados têm os seguintes campos:

- Número de sequência (`seq`)
- Confirmação (`ack`)
- Informação (`info`)

As tramas são numeradas, com um número `seq` compreendido entre `0` e um número máximo especificado numa janela (`sim.get_max_sequence()`). O campo `ack` contém o número de sequência da última trama de dados recebida.

As tramas de dados têm um tempo de transmissão não nulo, portanto o seu envio é realizado em duas fases:

1) É iniciado o envio da trama com o método `to_physical_layer`;
2) É recebido um evento `handle_Data_end`, a informar que terminou o envio da trama de dados.

Entre o início do envio da trama de dados e a receção do evento de fim de trama, não podem ser enviadas outras tramas. Usando o método `is_sending_data()` é possível verificar se está a ser transmitida uma trama de dados.

Tramas de confirmação de receção (ACK)

As tramas de confirmação de receção (ACK) são geradas após a receção de tramas de dados, indicando o número de sequência da última trama de dados recebida com sucesso. Considera-se instantânea a trama ACK, sendo enviada numa única invocação do método `to_physical_layer`. A trama ACK contém um único campo:

- Confirmação (`ack`)

Como é mais eficiente enviar esta informação (`ack`) através de tramas de dados (por *piggybacking*), definiu-se um temporizador auxiliar (`ack_timer`) que pode ser usado para esperar por uma trama de dados, só enviando o ACK após este tempo. Desta forma, após receber uma trama de dados deve-se:

1) Armar o temporizador de ACK, usando o método `start_ack_timer()`;

2a) Se aparecer uma trama de dados para transmitir, o temporizador pode ser cancelado com o método `cancel_ack_timer()`;

2b) Se o relógio expirar, é gerado o evento `handle_ack_Timer`, devendo-se enviar a trama ACK.

2.2. PROTOCOLOS DE NÍVEL LÓGICO

Nas secções 3.3 e 3.4 do livro recomendado são descritos os quatro tipos de protocolos de nível lógico que se vão realizar neste trabalho. Esta seção reproduz resumidamente os aspetos fundamentais de cada um, mas recomenda-se uma leitura atenta ao livro para uma correta realização do trabalho.

Protocolo Utópico

O protocolo utópico está descrito na secção 3.3.1 do livro e corresponde a realizar o envio das tramas sequencialmente sem nenhum mecanismo de recuperação de erros. O recetor limita-se a receber as tramas e enviar para o nível rede quando são recebidas por ordem. O emissor envia sequencialmente todas as tramas.

É fornecido o código completo de um emissor e de um recetor deste protocolo juntamente com o enunciado.

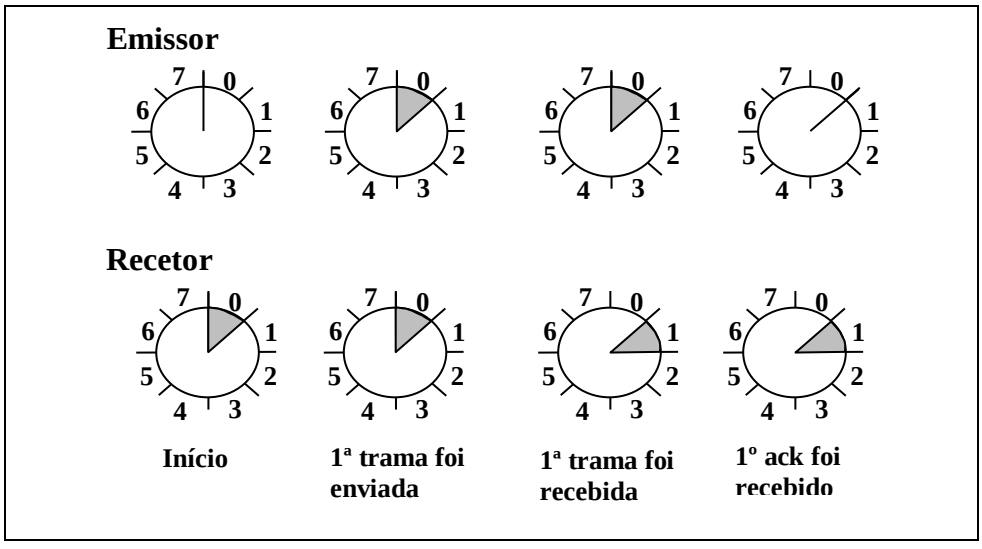
Protocolo Simplex Stop & Wait

O protocolo simplex stop & wait está descrito nas secções 3.3.2 e 3.3.3 do livro e corresponde a realizar o envio das tramas sequencialmente com mecanismos de recuperação de erros. O emissor deve armar um temporizador cada vez que envia uma trama. Caso expire, deve reenviar a trama. Quando um ACK confirma a trama, deve-se cancelar o temporizador e enviar a próxima trama.

Protocolo Stop & Wait (duplex)

O protocolo Stop & Wait está descrito na secção 3.4.1 do livro e corresponde a um protocolo de janela deslizante com janelas de transmissão e receção unitárias.

Os emissores mantêm o próximo número de sequência a transmitir e o recetor o próximo receber, de acordo com o esquema à direita.

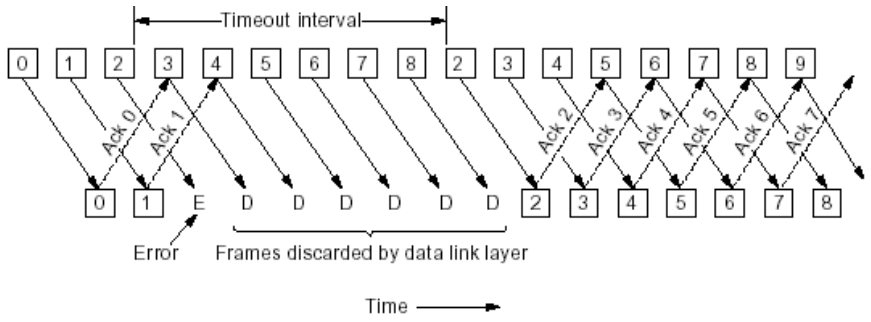


Na prática, este protocolo junta o emissor e o recetor do protocolo simplex stop&wait num único objeto que pode enviar e receber dados em paralelo. A grande diferença é que a confirmação da receção de tramas passa a poder feita pelo campo ACK da trama de dados. Recorre-se a um temporizador (*ACK_timer*) para adiar o envio da trama ACK, à espera que surja uma trama de dados para enviar antes do envio da trama ACK.

Protocolo Go-Back-N

O protocolo Go-Back-N está descrito na secção 3.4.2 do livro. Corresponde a um protocolo de janela deslizante com janela de receção unitária, onde é usado *pipelining* para melhorar o desempenho quando o produto *atraso*banda* é elevado.

Neste caso, o emissor vai necessitar de manter um array de buffers de transmissão, podendo transmitir até um máximo de `sim.get_send_window()` tramas sem receber confirmações (i.e. janela de transmissão). Quando ocorre um erro, tem de retransmitir todas as tramas de dados a partir da que não foi confirmada, como está representado na figura.



A gestão de temporizadores pode ser complexa neste protocolo. Neste trabalho **deve usar apenas um temporizador para todas as tramas**. O temporizador deve ser reiniciado cada vez que é confirmada uma nova trama de dados. Quando expira, deve ser reenviada a trama mais antiga não confirmada.

Protocolo Retransmissão Seletiva (*Selective Repeat*)

O protocolo com retransmissão seletiva está descrito na secção 3.4.3 do livro, embora não sejam usadas tramas NACK (*Negative* ACK) neste trabalho. Corresponde a um protocolo de janela deslizante com janelas de emissão e de receção de dimensão arbitrárias, onde é usado *pipelining* para melhorar o desempenho quando o produto *atraso*banda* é elevado.

A grande diferença face ao protocolo anterior é que o recetor pode receber tramas fora de ordem dentro da janela de receção, guardando-as no *buffer* de receção. No entanto, as tramas são enviadas de forma ordenada para o nível rede. Tal como anteriormente, deve usar apenas um temporizador.

2.3. CENÁRIO DE SIMULAÇÃO

Neste trabalho simula-se uma rede com tempo de propagação variável e com uma taxa de perda de tramas constante. São usadas dois programas:

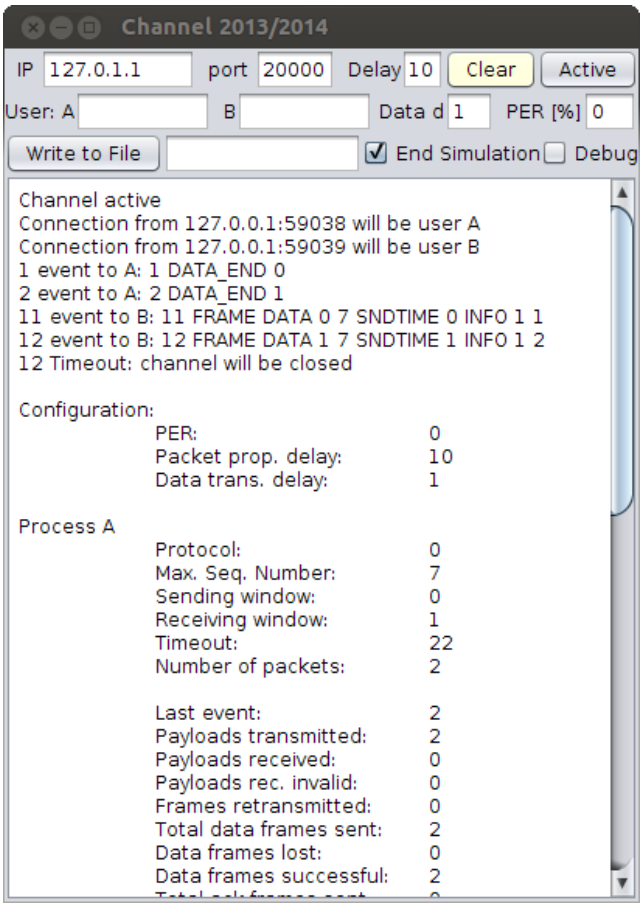
- *Protocol* (*protocolo*) – realiza o protocolo de nível lógico e emula o nível rede, controlando o envio e receção de pacotes de dados numerados sequencialmente. A parte do nível lógico será feita pelos alunos;
- *Channel* (*canal*) – liga duas instâncias de *Protocol*, emulando o tempo de propagação e perdendo tramas com uma certa probabilidade.

Após arrancar, o *canal* aceita duas ligações TCP de dois programas *Protocol*, começando a partir daí a simulação. O canal realiza um sequenciador de eventos discretos, recebendo os comandos gerados pelos protocolos e gerando os eventos relacionados com o nível físico e com os temporizadores apresentados anteriormente ordenados de acordo com o tempo de simulação. O tempo de simulação é medido em unidades de tempo virtuais, designadas de *tics*.

Tanto o *protocolo* como o *canal* fornecidos com o enunciado escrevem resumidamente (ou exaustivamente, no modo *debug*) os eventos e comandos que são gerados, e o conteúdo da fila de espera com eventos à espera de serem disparados.

3. DESENVOLVIMENTO DO PROGRAMA

3.1. APLICAÇÃO CANAL



A aplicação *canal* é fornecida totalmente realizada. Consiste numa aplicação em Java com a interface gráfica representada à direita. Quando se prime o botão "*Active*" o *canal* arranca um *ServerSocket* no IP e porto indicados, ficando preparado para receber ligações de *protocolos*, que serão designados respetivamente de *protocol* (*User*) A e B.

A aplicação *canal* permite realizar várias configurações do cenário de simulação:

- "*Delay*" – tempo de propagação de tramas no canal;
- "*Data d*" – duração de uma trama de dados (tempo entre o envio da trama *Data* e a geração do evento *Data_end*);
- "*PER [%]*" – (*Packet Error Rate*) taxa média de perda de tramas no canal, que pode afetar todas as tramas transmitidas com igual probabilidade;
- "*End Simulation*" – controla se o canal termina automaticamente uma simulação quando não recebe eventos durante um segundo, ou se é deixada a decisão ao utilizador, carregando no botão "*Active*".
- "Write to File" – controla se se escrevem as mensagens ecoadas para o ecrã num ficheiro.

No final da simulação é escrito um relatório com o valor das várias configurações usadas no *canal* e nos *protocolos*, e com várias medidas de desempenho do sistema para os *protocolos* A e B. Destacam-se entre estas medidas:

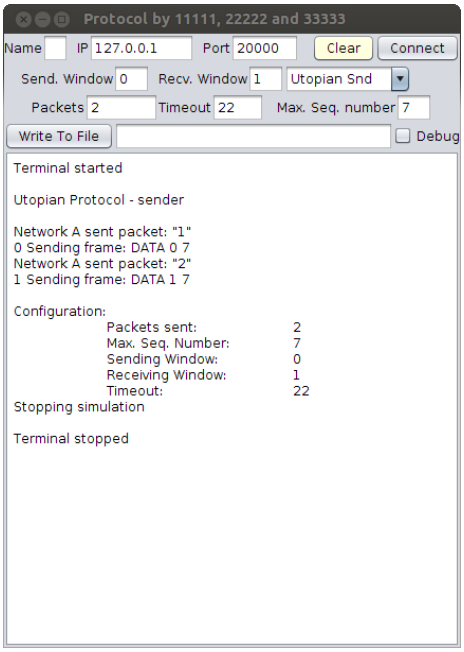
- *Last event* – tempo do último evento que o *protocolo* recebeu/originou, traduzindo-se no atraso total que ocorreu para enviar e receber todos os dados;
- *Payloads transmitted* – número de pacotes transmitidos pelo nível rede;
- *Payloads received* – número de pacotes recebidos no nível rede;
- *Payloads rec. invalid* – número de pacotes recebidos fora de ordem no nível rede;
- *Frames retransmitted* – número de tramas de dados retransmitidas;
- *Total data frames sent* – número total de tramas de dados enviadas;
- *Data frames lost* – número de tramas de dados perdidas por erros no canal;
- *Data frames successful* – número de tramas de dados recebidas com sucesso;
- *Total ack frames sent* – número total de tramas ACK enviadas;
- *Ack frames lost* – número de tramas ACK perdidas por erros no canal;
- *Ack frames successful* – número de tramas ACK recebidas com sucesso;
- *Timeouts* – número de eventos Timeout recebidos;
- *Ack timeouts* – número de eventos ACK_Timeout recebidos.

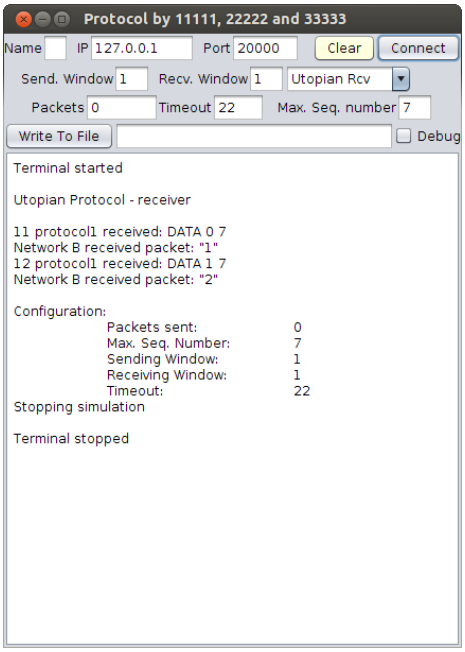
O desempenho dos protocolos realizados vai ser medido usando estas métricas, destacando-se o atraso e o rácio do número total de pacotes transmitidos por pacote de dados. Sugere-se que para as várias variantes do protocolo se meça o desempenho para: i) um cenário sem erros e com timeout ajustado (PER=0 e Timeout=22 tics); ii) cenário com erros e com timeout ajustado (PER=50% e Timeout=22 tics); e iii) cenário com erros e timeout longo (PER=50% e Timeout=44 tics).

Pode correr o *canal* a partir do terminal com o comando: `java -jar Channel.jar`

3.2. APLICAÇÃO PROTOCOLO

O trabalho consiste exclusivamente na realização dos protocolos de nível lógico. Tudo o resto é fornecido completamente realizado. As duas figuras abaixo representam os nós *protocolo* A e B com as mensagens geradas de acordo com a figura do canal apresentada anteriormente para o protocolo Utópico.





A interface gráfica permite definir a qual aplicação canal é feita a ligação, escolhendo o IP e o porto. A simulação arranca quando se prime o botão "*Connect*" e o canal gera o evento de início de simulação.

Através da interface gráfica, é possível definir:

- *Packets* – o número de pacotes que vão ser enviados durante a simulação;
- *Max. Seq. number*– o número máximo de sequência (geralmente dado por 2^n-1);
- *Send Window* – o tamanho da janela de transmissão;
- *Recv Window* – o tamanho da janela de receção;
- *Timeout* – o tempo de espera por uma confirmação antes de reenviar uma trama de dados.

Existe também uma *combobox* que permite escolher o protocolo de nível lógico: *Utopian Snd*; *Utopian Rcv*; *Simplex Snd*; *Simplex Rcv*; *Stop & Wait*; *Go-Back-N*; e *Selec. Repeat*. O objetivo do trabalho é desenvolver o código para os quatro últimos protocolos.

O programa fornecido é composto por três pacotes, cada um com as classes seguintes:

- Pacote `terminal`:
  - *Terminal,java* (completa) – Classe principal com interface gráfica, que faz a gestão de sincronismo dos vários objetos usados;
  - *Connection.java* (completa) – Thread que trata uma ligação TCP ao canal;
  - *NetworkLayer.java* (completa) – Classe que realiza a interface com o nível rede;
- Pacote `simulator`:
  - *Frame.java* (completa) – Classe que guarda e serializa tramas;
  - *Event.java* (completa) – Classe que guarda e serializa eventos;
  - *Log.java* (completa) – Interface que define a função *Log*;
- Pacote `protocol`:
  - *Simulator.java* (completa) – Interface que define todos os comandos que podem ser usados na realização de um protocolo de nível lógico;
  - *Callbacks.java* (completa) – Interface que define todos os métodos que têm de ser implementados na realização de um protocolo de nível lógico;
  - *Base_Protocol.java* (completa) – Classe de suporte que define um conjunto de funções de suporte para manipular números de sequência de pacotes, uma interface simplificada para temporizadores, e uma realização por omissão para todos os métodos que têm de ser implementados na realização de um protocolo de nível lógico. Todas as realizações dos protocolos herdam desta classe, portando podem usar os métodos definidos;
  - *Utopian_snd.java* (completa) – Realização do emissor do protocolo de nível lógico Utópico;
  - *Utopian_rcv.java* (completa) – Realização do recetor do protocolo de nível lógico Utópico;
  - *Simplex_snd.java* (**a completar**) – Realização do emissor do protocolo de nível lógico Simplex;
  - *Simplex_rcv.java* (**a completar**) – Realização do recetor do protocolo de nível lógico Simplex;
  - *StopWait.java* (**a completar**) – Realização do protocolo de nível lógico Stop & Wait;
  - *GoBackN.java* (**a completar**) – Realização do protocolo de nível lógico Go-Back-N;
  - *SelectiveRepeat.java* (**a completar**) – Realização do protocolo de nível lógico *Selective Repeat*.

Os alunos apenas vão modificar estes cinco últimos ficheiros, com a realização dos protocolos pretendidos, utilizando principalmente os métodos definidos nas interfaces `Simulator`, `Callbacks` e da classe *Base_Protocol*, descritos na secção seguinte.

3.2.1. Comandos

No código que implementa os protocolos, pode-se invocar sobre o objeto `sim` os seguintes métodos (que foram definidos na `interface Simulator`). O propósito de cada um está explicado de seguida:

- Obter a dimensão da janela de transmissão:

```
int get_send_window();
```

- Obter a dimensão da janela de receção:

```
int get_recv_window();
```

- Obter a número máximo de sequência:

```
int get_max_sequence();
```

- Obter o valor do timeout:

```
long get_timeout();
```

- Obter o tempo atual de simulação:

```
long get_time();
```

- Enviar a trama `frame` para o nível físico (i.e. para o canal):

```
void to_physical_layer(simulator.Frame frame);
```

- Armar o temporizador de ACK:

```
void start_ack_timer();
```

- Cancelar o temporizador de ACK:

```
void cancel_ack_timer();
```

- Testar se o temporizador de ACK está ativo:

```
void is_ack_timer_active();
```

- Parar a simulação:

```
void stop();
```

A `interface Simulator` também define um conjunto de métodos para armar e parar o relógio para tramas de dados, apresentadas de seguida. No entanto, neste trabalho recomenda-se que use as três funções equivalentes disponibilizadas pela classe `Basic_Protocol`, apresentadas na seção seguinte (3.2.1.1), que usam apenas uma chave.

- Armar um temporizador associado à chave `key`. Caso seja armado duas vezes com a mesma chave, cancela o relógio anterior:

```
void start_data_timer(int key);
```

- Cancelar o temporizador associado à chave `key`:

```
void cancel_data_timer(int key);
```

- Testar se o temporizador associado à chave `key` está ativo:

```
boolean isactive_data_timer (int key);
```

3.2.1.1. Temporizadores de dados

Na classe `Basic_Protocol` são definidos três métodos (herdados pelas classes que implementam os protocolos) que armam, param e testam o temporizador com a chave `TIMER_KEY=1`.

```
void start_data_timer();
void cancel_data_timer();
boolean isactive_data_timer();
```

Recomendação: Use apenas este conjunto de funções para controlar o temporizador que recupera da perda de tramas de dados.

3.2.2. Eventos

Pense nos eventos como algo que provoca a invocação dos métodos de *callback*. Estes métodos vão ser implementados pelos alunos na classe *Protocol*, e foram definidos na interface `interface Callbacks`:

- Evento de início de simulação:

```
void start_simulation(long time);
```

- Evento de fim de transmissão da trama de dados com o nº de sequência `seq`:

```
void handle_Data_end(long time, int seq);
```

- Evento de disparo do temporizador associado à chave `key` (neste trabalho usa-se apenas a chave `DATA_KEY=1`)

```
void handle_Data_Timer(long time, int key);
```

- Evento de disparo do temporizador de ACK:

```
void handle_ack_Timer(long time);
```

- Evento de receção da trama `frame` do nível físico:

```
void from_physical_layer(long time, simulator.Frame frame);
```

- Evento de fim de simulação:

```
void end_simulation(long time);
```

Em todos os eventos, é recebido o tempo de simulação atual, em `time`.

3.2.3. Nível rede

A classe `terminal.NetworkLayer`, define os métodos de troca de pacotes com o nível Rede, e está instanciada no objeto `net`:

- Obter um novo pacote do nível rede (caso já não exista mais nenhum para enviar, retorna `null`):

```
String from_network_layer();
```

- Enviar um novo pacote para o nível rede (caso exista um erro no conteúdo devolve `false`):

```
boolean to_network_layer(String packet);
```

3.2.4. Geração e leitura de Tramas

As tramas são objetos da classe `simulator.Frame (import Simulator.Frame)`. Esta classe contém os campos já explicados anteriormente (`seq`, `ack` e `info`) mais um chamado `kind`. Para além disso, tem métodos estáticos para criar novas instâncias de objetos, e métodos normais para aceder aos vários campos. O campo `kind` define o tipo de trama tendo dois valores para uma trama válida: `Frame.DATA_FRAME` ou `Frame.ACK_FRAME`; e o valor `Frame.UNDEFINED_FRAME`, quando ela não está inicializada.

Para obter o valor de `kind`, pode-se usar o método `kind()`.

Para criar uma nova trama de cada um dos dois tipos é possível usar os métodos:

```
public static Frame new_Data_Frame(int seq, int ack, String info);
public static Frame new_Ack_Frame(int ack);
```

Para aceder aos campos usam-se os métodos:

```
public int seq();       // for DATA_FRAME
public String info();   // for DATA_FRAME
```

```
public int ack();       // for DATA_FRAME or ACK_FRAME
public long snd_time(); // time when it was sent
```

Também é possível obter uma descrição textual do conteúdo da trama:

```
public String kindString();   // devolve o nome do tipo de trama
public String toString();     // devolve o nome do tipo e o resumo do conteúdo
```

Para permitir o transporte das tramas através de sockets TCP, foram definidas duas funções para converter o conteúdo para string e restaurar o conteúdo a partir de uma string (i.e. fazer a serialização do objeto). Estas funções não vão ser usadas pelos alunos, mas geram a representação que é mostrada no log da aplicação.

```
public String frame_to_str();
public boolean str_to_frame(String line, Log log);
```

3.2.1.1. Números de sequência

A classe `Basic_Protocol`, herdada por todos os protocolos, define um conjunto de funções que permite gerir os números de sequência usados nas tramas de dados e ACK, com valores entre `0` e `sim.get_max_sequence()`:

- Adicionar uma unidade ao número de sequência `n`, respeitando a ordem circular:

```
int incr_seq(int n);
```

- Adicionar `k` unidades ao número de sequência `n`, respeitando a ordem circular:

```
int add_seq(int n, int k);
```

- Decrementar uma unidade ao número de sequência `n`, respeitando a ordem circular:

```
int decr_seq(int n);
```

- Testar se o número de sequência `b` verifica a condição `a <= b < c`, tendo em conta a ordem circular:

```
boolean between(int a, int b, int c);
```

- Calcula a diferença entre dois números de sequência, tendo em conta a ordem circular:

```
int diff_seq(int a, int b);
```

3.2.5. Protocolo Utópico

As classes *Utopic_snd* e *Utopic_rcv* foram programadas inteiramente para servir de exemplo para os alunos. Elas realizam o protocolo utópico descrito anteriormente.

A classe *Utopic_snd* implementa o envio de tramas. Esta classe acrescenta (em relação à `interface Callbacks`) uma variável de estado e um método para centralizar o envio de tramas para o nível físico.

A variável de estado serve para controlar os números de sequência usados nas tramas:

```
private int next_frame_to_send; // Número da próxima trama de dados a enviar
```

O método de envio de tramas é o seguinte:

```
boolean send_next_data_packet() {
  String packet= net.from_network_layer();
  if (packet != null) {
    // The ACK field of the DATA frame is always the sequence number before 0,
    //   because no packets will be received!
    Frame frame = Frame.new_Data_Frame(next_frame_to_send, decr_seq(0), packet);
    sim.to_physical_layer(frame);
    next_frame_to_send= incr_seq(next_frame_to_send);
    return true;
  }
  return false;  // Failed; no more packets to send
}
```

Este método é chamado:

- No arranque da simulação:

```
public void start_simulation(long time) {
  sim.Log("\nUtopian Protocol - sender\n\n");
  send_next_data_packet();    // Start sending the first data frame
}
```

- Cada vez que termina o envio da trama de dados anterior:

```
public void handle_Data_end(long time, int seq) {
  send_next_data_packet();    // Send the next data frame
}
```

A classe *Utopic_rcv* implementa a receção de tramas. Ela acrescenta (em relação à `interface Callbacks`) uma variável de estado para controlar os números de sequência recebidos:

```
private int frame_expected;     // Número esperado na próxima trama a receber
```

A receção de tramas é feita no método `from_physical_layer`, que neste caso faz um pouco mais do que a versão original do livro – verifica se o número de sequência é o esperado, e avança o número esperado para a próxima trama.

```
public void from_physical_layer(long time, Frame frame) {
  sim.Log(time + " protocol1 received: " + frame.toString() +"\n");
  if (frame.kind() == Frame.DATA_FRAME) {     // Check the frame kind
    if (frame.seq() == frame_expected) {    // Check the sequence number
      net.to_network_layer(frame.info()); // Send the frame to the network layer
      frame_expected = incr_seq(frame_expected);
    }
  }
}
```

Os restantes métodos da interface `Callbacks` não são usados neste protocolo e limitam-se a escrever que foram invocados.

3.2.6. Protocolo Simplex *Stop&Wait*

As classes *Simplex_snd* e *Simplex_rcv* devem ser programadas de maneira a implementar o protocolo simplex *Stop & Wait*, partindo do código do protocolo Utópico. As grandes modificações são:

- O emissor deve criar um temporizador, e armá-lo cada vez que envia uma trama de dados, retransmitindo a trama caso não receba a confirmação;
- O receptor deve enviar um ACK por cada trama de dados recebida.

No emissor é necessário implementar o método `handle_Data_Timer`, para lidar com o expirar do timer.

3.2.7. Protocolo Duplex *Stop&Wait*

A classe *StopWait* deve ser programada para realizar o protocolo *Stop&Wait* duplex, partindo das duas classes programadas no ponto anterior. Este objeto funciona simultaneamente como emissor e recetor, reunindo todas as carateristicas dos dois objetos. Assim, a realização passa por dois passos:

1) Reunir num único ficheiro o código das classes *Simplex_snd* e *Simplex_rcv*, adaptando o código do método `from_physical_layer` para tratar os pacotes das duas classes.
2) Adicionar o temporizador *ACK_timer*, de maneira a poder enviar a confirmação de receção de tramas de dados com o campo `ack` das tramas de dados e com tramas ACK.

3.2.8. Protocolo *Go-Back-N*

A classe *GoBackN* deve ser programada a partir da classe *StopWait*, de forma a realizar o protocolo de janela deslizante Go-back-N com um único temporizador. As modificações incidem apenas no emissor, que passa a poder ter uma janela de transmissão maior do que 1.

Sugestões: Recomenda-se que declare um array de strings com a dimensão do número de identificadores de pacotes (`sim.get_max_sequence()+1`) para guardar os pacotes recebidos do nível rede, permitindo a sua retransmissão. Recomenda-se ainda que acrescente as variáveis necessárias para memorização da trama mais antiga não confirmada e da última enviada, e que as use corretamente.

3.2.9. Protocolo *Selective Repeat*

A classe *SelectiveRepeat* deve ser programada a partir da classe *GoBackN*, de forma a realizar o protocolo de janela deslizante *Selective Repeat* com um único temporizador. As modificações incidem apenas no receptor, que passa a poder ter uma janela de receção superior a um.

Sugestões: Recomenda-se que declare um array de strings com a dimensão do número de identificadores de pacotes (`sim.get_max_sequence()+1`) para guardar os pacotes recebidos do nível físico, permitindo memorizar os pacotes recebidos fora de ordem e enviá-los ao nível rede mais tarde, ordenadamente. Novamente, poderá ser necessário declarar variáveis adicionais para controlar a receção de dados.

3.3. METAS

Uma sequência para o desenvolvimento do trabalho poderá ser:

1. Programar o protocolo *Simplex* nas classes `Simplex_snd` e `Simplex_rcv`. Comece por copiar o conteúdo das classes `Utopic_snd` e `Utopic_rcv`, e perceba o que pode reutilizar;
2. Programar o protocolo *Stop&Wait* duplex na classe `StopWait`. Comece por copiar o conteúdo das classes `Simplex_snd` e `Simplex_rcv` para a classe `StopWait`;
3. Programar o protocolo *Go-Back-N* com um único temporizador na classe `GoBackN`. Comece por copiar o conteúdo da classe `StopWait` para a classe `GoBackN`;
4. Programar o protocolo de retransmissão seletiva com um único temporizador na classe `SelectiveRepeat`. Comece por copiar o conteúdo da classe `GoBackN` para a classe `SelectiveRepeat`.

Sugestões: Se algum grupo se sentir MUITO À VONTADE com o mecanismo de herança da linguagem Java, pode usá-lo para reduzir o número de linhas de código copiadas (principalmente na meta 4 do trabalho). Mas não se recomenda a sua utilização para alunos com pouca experiência em Java, como é o caso da generalidade dos alunos de ST.

TODOS os alunos devem tentar concluir **pelo menos a fase 2**. Na primeira semana do trabalho é feita uma introdução geral do trabalho, devendo-se concluir a fase 1. No fim da segunda semana deve ter iniciado a fase 3. No fim da terceira semana deve ter terminado a fase 3. No fim da quarta e última semana devem tentar realizar o máximo de fases possível, tendo sempre em conta que é preferível fazer menos e bem (a funcionar e sem erros), do que tudo e nada funcionar.

POSTURA DOS ALUNOS

Cada grupo deve ter em consideração o seguinte:

- Não perca tempo com a estética de entrada e saída de dados
- Programe de acordo com os princípios gerais de uma boa codificação (utilização de indentação, apresentação de comentários, uso de variáveis com nomes conformes às suas funções...) e
- Proceda de modo a que o trabalho a fazer fique equitativamente distribuído pelos dois ou três membros do grupo.