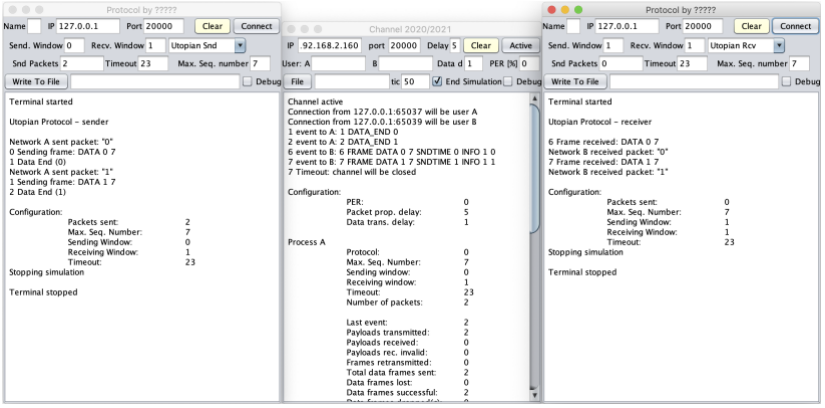
NOVA SCHOOL OF SCIENCE & TECHNOLOGY

DEPARTMENT OF ELECTRICAL AND COMPUTER ENGINEERING



Sistemas de Telecomunicações

2020/2021

Trabalho 6:
Protocolos de nível Lógico com janela deslizante

Aulas 7 a 11

Mestrado integrado em Engenharia Eletrotécnica e de Computadores

http://tele1.dee.fct.unl.pt **V1.1**

Luis Bernardo
Paulo da Fonseca Pinto

Índice

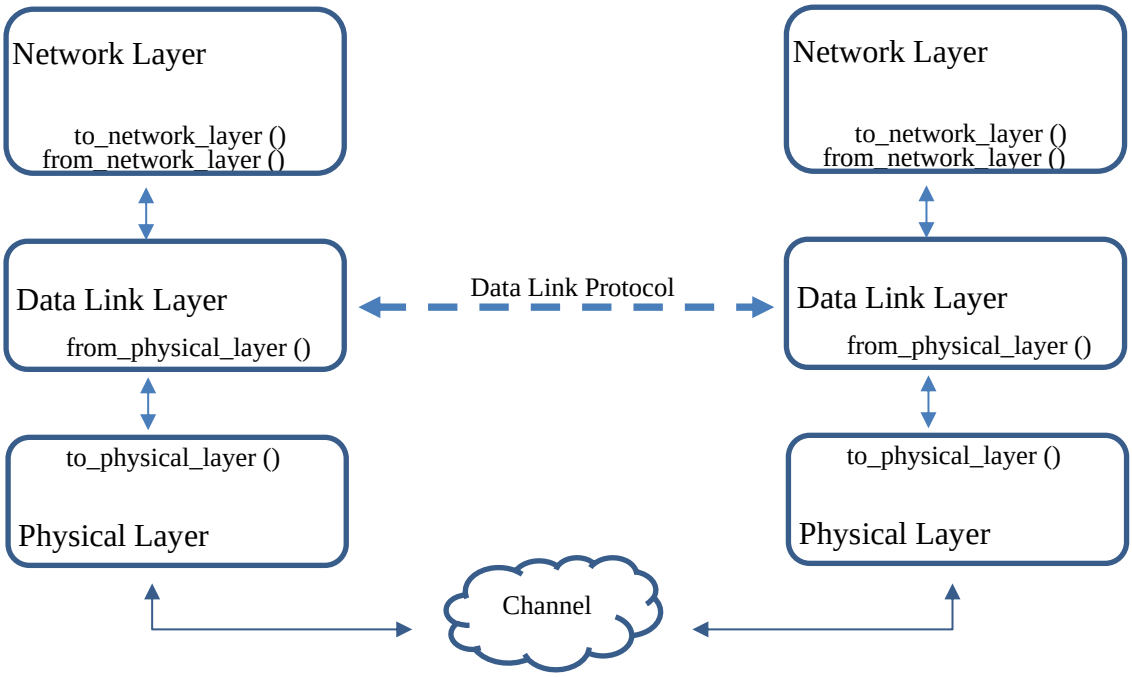
1. OBJETIVO

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 vários protocolos de nível lógico baseados em janela deslizante, de forma faseada, usando uma ferramenta construída para Sistemas de Telecomunicações.

2. ESPECIFICAÇÕES GERAIS

Pretende-se desenvolver um protocolo de nível Lógico para uma ligação física ponto-a-ponto. Vai existir um pequeno nível Rede que o vai usar e, por outro lado, ele usa um nível Físico. A interação entre todos estes níveis é feita por interfaces de serviço usando primitivas de serviço. A figura em baixo mostra o sistema e os alunos têm de implementar o nível Lógico (Data Link).



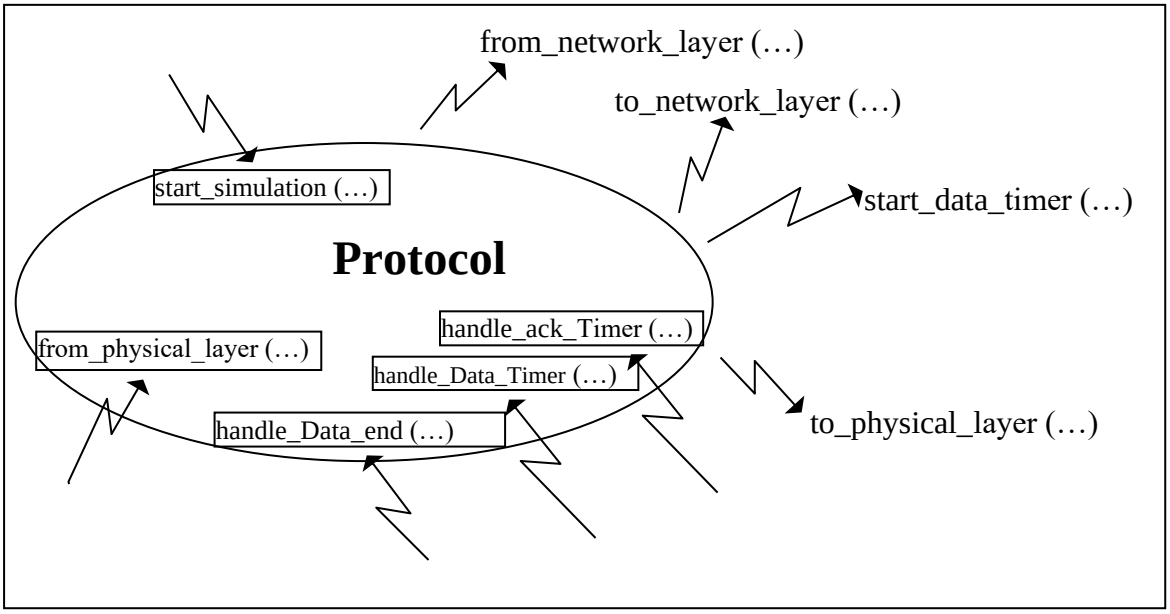
A abordagem proposta foi inspirada no simulador descrito nas secções 3.3 e 3.4 do livro recomendado (Computer Networks 5ª edição), mas esta simulação vai ser um pouco mais próxima da realidade (e.g. considera o tempo de transmissão das tramas de dados). O protocolo de nível lógico vai reagir a eventos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

A simulação começa realmente quando o nível Lógico recebe um evento do *Channel* que provoca a invocação do método `start_simulation()`.

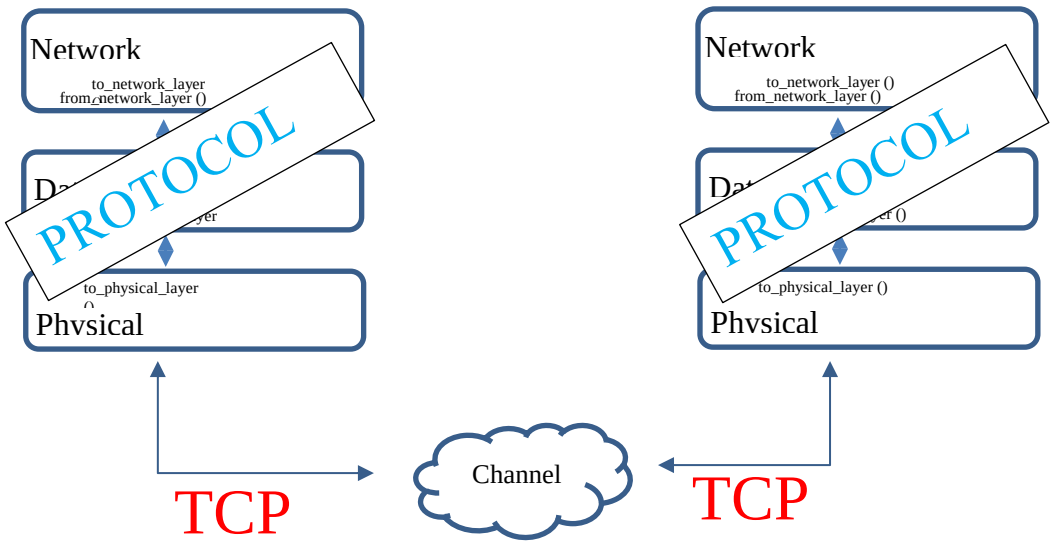
A troca de informação com o nível Rede baseia-se em Strings. As Strings recebidas pelo nível Lógico representam pacotes que têm de ser enviados em tramas pelo nível Lógico. As Strings enviadas para o nível Rede representam a parte de dados das tramas recebidas. Para receber Strings do nível Rede usa o método `from_network_layer()` e para enviar Strings invoca o método `to_network_layer()`. Os dados têm de ser entregues pela mesma ordem em que foram recebidos do nível Rede do outro lado, recuperando de erros que possam existir no canal.

Na interface com o nível Físico existem as primitivas de serviço `to_physical_layer()` para enviar tramas e `from_physical_layer()` para receber tramas.

Internamente ao nível Lógico existe um conjunto de funcionalidades já programadas de suporte para gerir temporizadores. Usando os métodos `start_data_timer` e `cancel_data_timer`, pode armar ou cancelar um temporizador para lidar com a perda de tramas de dados. Quando o tempo expira, é chamado o método `handle_Data_Timer()`. Certos eventos têm o resultado direto de chamarem métodos. Na descrição neste enunciado por vezes usa-se indistintamente métodos e eventos e pode aparecer o nome do método a designar o evento. A figura em baixo ilustra alguns dos eventos, invocações de métodos e métodos. Os alunos devem programar o balão designado por *Protocol*.



A figura seguinte mostra como se vai simular todo o sistema: a aplicação desenvolvida pelos alunos, o *Protocol*, vai correr de um lado e de outro de outra aplicação chamada de *Channel*. Para a ligação entre todos os componentes vai ser usado o TCP (a simular uma linha física). O *Channel* vai introduzir tempos de propagação variáveis e vai perder tramas de acordo com uma taxa.



São usados então dois programas:

- *Protocol* – realiza o protocolo de nível Lógico e emula o nível Rede, controlando o envio e receção de pacotes. A parte do nível Lógico será feita pelos alunos;

- *Channel* – liga duas instâncias de *Protocol*, emulando o tempo de propagação e perdendo tramas com uma certa probabilidade.

Ao arrancar, o *Channel* aceita duas ligações TCP dos dois programas *Protocol*, ficando pronto para a simulação. Na ativação envia eventos de início para os *Protocol* e começa a simulação. O *Channel* realiza um sequenciador de eventos discretos, recebendo os eventos gerados pelos *Protocols* e gerando os eventos relacionados com o nível Físico e com os temporizadores, ordenados de acordo com o tempo de simulação. O tempo de simulação é medido em unidades de tempo virtuais, designadas de *tics*, que têm uma correspondência com o tempo real – o *Channel* gere o tempo de *tic* durante cada simulação.

O *Channel* controla o tempo de transmissão das tramas de dados. Isto é, controla o tempo que demora entre o envio do primeiro bit e o envio do último bit de uma trama. Se for transmitida alguma outra trama enquanto o tempo de transmissão de uma trama está a decorrer, o canal interrompe essa transmissão, que não é recebida no protocolo de destino. As tramas ACK são muito curtas e têm um tempo de transmissão que não é considerado na simulação.

Tanto o *Protocol* como o *Channel* 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 tratados.

Vai existir um programa *Protocol* nos computadores do laboratório para os alunos poderem testar os seus programas.

2.1. FUNCIONAMENTO DO NÍVEL REDE

Vai-se assumir que o nível Rede tem um número ilimitado de buffers para receber pacotes do nível Lógico, tal como está descrito no livro.

Quando a simulação arranca, o nível Rede tem um número limitado de mensagens a enviar, que são compostas por Strings com uma sequência de números inteiros iniciada em "0". As mensagens são devolvidas a cada invocação do método `from_network_layer()`, que passa a devolver `null` quando se esgotam as mensagens.

2.2. PROTOCOLOS DE NÍVEL LÓGICO

Nas secções 3.3 e 3.4 do livro recomendado são descritas versões dos quatro tipos de protocolos de nível lógico que se vão realizar neste trabalho. A secção "Especificações Detalhadas" contém resumos dos aspetos mais fundamentais de cada um, mas recomenda-se uma leitura atenta do livro para uma correta realização do trabalho. Na realidade, vão existir cinco, e não quatro, tipos de protocolos:

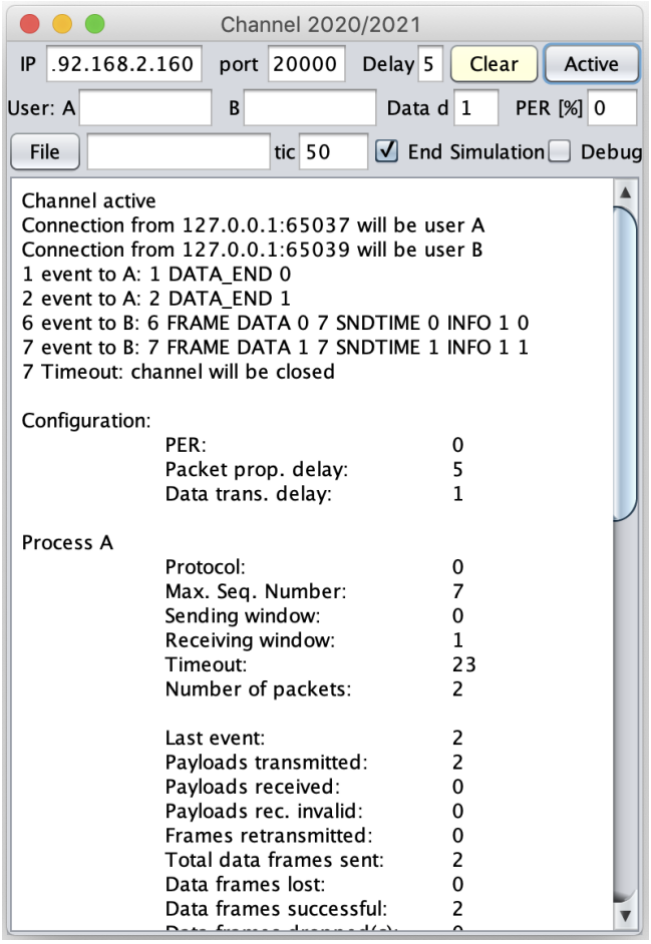
1. Protocolo Utópico – É fornecido o código completo de um emissor e de um recetor deste protocolo juntamente com o enunciado.
2. Protocolo Simplex *Stop&Wait* – Para ser executado pelos alunos.
3. Protocolo *Stop&Wait* (duplex) – Para ser executado pelos alunos.
4. Protocolo Go-Back-N – Para ser executado pelos alunos.
5. Protocolo Retransmissão Seletiva (*Selective Repeat*) – Para ser executado pelos alunos.

2.3. FUNCIONAMENTO DO NÍVEL FÍSICO

O nível Físico usa sockets TCP. Todos os dados a enviar são passados para String e depois escritos no socket como uma linha de texto. A leitura do socket é feita por uma instrução de leitura de uma linha de texto recebendo uma String.

3. APLICAÇÃO CHANNEL

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



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

- "*Delay*" – tempo de propagação de tramas no canal (em *tics*);
- "*Data d*" – duração de uma trama de dados (tempo entre o envio da trama *Data* e a geração do evento *Data_end*, em tics);
- "*tic*" – duração de cada tic, em ms;
- "*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 ou se é deixada a decisão ao utilizador, carregando no botão "*Active*". No modo automático a simulação termina quando não se recebem eventos durante um segundo ou o maior tempo de envio e confirmação de trama, se for maior,
- "*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 *Channel* e nos *Protocols*, e com várias medidas de desempenho do sistema para os *Protocols* A e B. Destacam-se entre estas medidas:

- *Last event* – tempo do último evento que o *Protocol* 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;
- *Data frames dropped(C)* – número de tramas de dados perdidas por transmissão concorrente de outras tramas;
- *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=13 tics);
ii) cenário com erros e com *timeout* ajustado (PER=50% e Timeout=13 tics);
iii) cenário com erros e *timeout* longo (PER=50% e Timeout=26 tics);
iv) cenário com fluxo contínuo de pacotes, sem erros (PER=0), Delay=Data d=1 (no canal), 20 pacotes, janela emissão 7);
v) o mesmo cenário de iv) com PER=20%.

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

4. ESPECIFICAÇÕES DETALHADAS

4.1. TIPOS DE TRAMAS E ENVIO DE RECONHECIMENTOS

O *Protocol* 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`)
- Vetor com tramas recebidas fora de ordem (`ackvec`)
- 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** (isto é, é seguida a convenção do livro na Fig 3-17) e o campo `ackvec` contém uma lista com os números de sequência das tramas recebidas fora de ordem. Finalmente, o campo `info` contém os dados da trama.

Como se disse, as tramas de dados têm um tempo de transmissão não nulo. Isto foi implementado nas seguintes duas fases:

1) É iniciado o envio da trama com o método `to_physical_layer`. Este método não bloqueia e deve ser entendido como despoletar o início da transmissão;
2) É recebido um evento `handle_Data_end`, (que de fato é um método que é invocado quando o evento chega) 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. Caso isso aconteça, aborta-se a transmissão da trama de dados que está a ser enviada e começa-se a transmissão da nova. Usando o método `is_sending_data()` é possível verificar se está a ser transmitida uma trama de dados.

Tramas de confirmação de receção (ACK)

A trama ACK contém os seguintes campos:

- Confirmação (`ack`)
- Vetor com tramas recebidas fora de ordem (`ackvec`)

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 que o tempo de transmissão da trama ACK é instantâneo, usando-se apenas a invocação do método `to_physical_layer`.

Como é mais eficiente enviar esta informação (`ack`, `ackvec`) através de tramas de dados (por *piggybacking*), deve-se dar preferência a isso usando o seguinte procedimento: define-se um temporizador auxiliar (`ack_timer`) que é usado para se esperar que uma trama de dados possa ser enviada e transportar esta informação. Se não aparecer nenhuma trama de dados, envia-se então o ACK após este tempo. Explicando de um modo mais sistemático, após receber uma trama de dados deve-se:

1) Armar o temporizador de ACK, usando o método `start_ack_timer()` caso não esteja ativo (pode testar se eatá ativo com o método `isactive_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` (mais uma vez um método é chamado), devendo-se enviar a trama ACK.

4.2. PROTOCOLOS DE NÍVEL LÓGICO

Relativamente aos protocolos do nível Lógico devem-se ter em atenção os seguintes aspetos:

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. Neste exemplo é verificada a ordem de receção, embora tal não fosse necessário. 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.

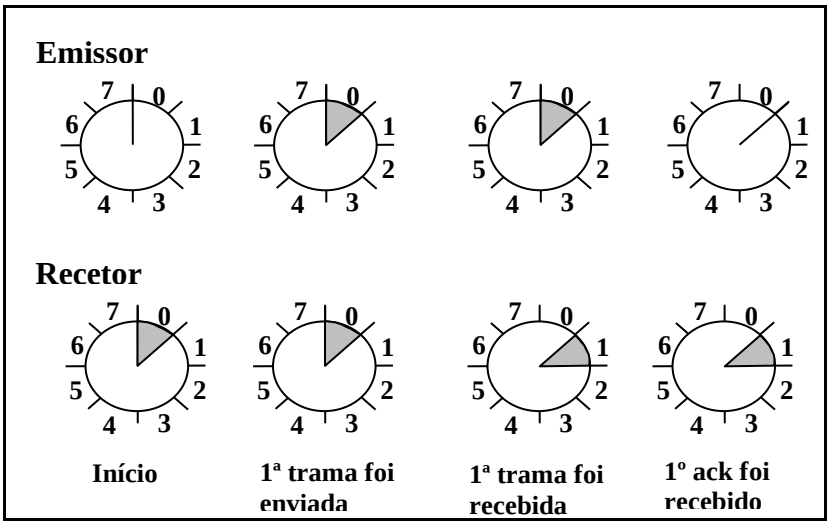
Protocolo *Stop&Wait* Simplex

O protocolo *Stop&Wait* simplex 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.



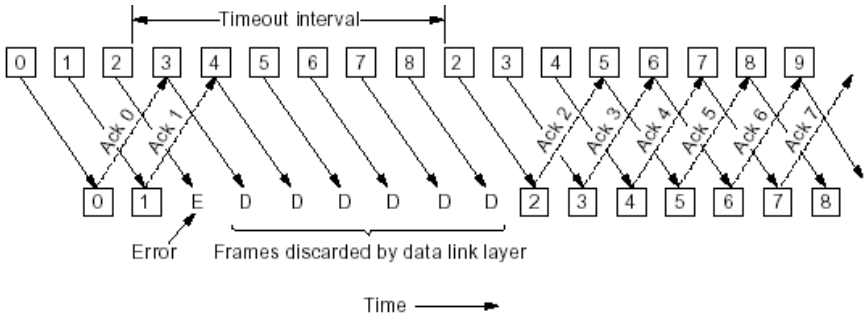
Na prática, este protocolo

junta o emissor e o recetor do protocolo anterior 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 ser feita pelo campo ACK da trama de dados. Recorre-se ao temporizador (*ACK_timer*) para adiar o envio da trama ACK, à espera que surja uma trama de dados para enviar poupando-se a 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 vetor 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.

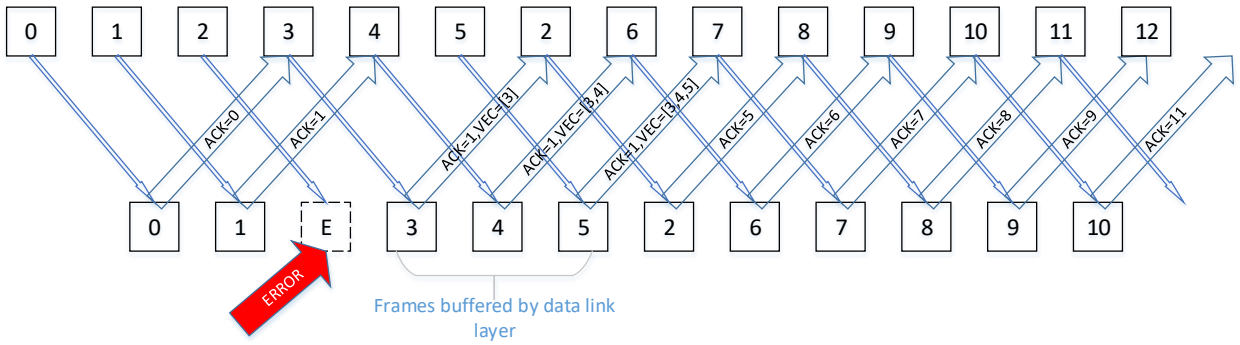


Vai ser usado um único temporizador para todas as tramas de dados. Cada vez que é confirmada uma trama de dados deve ser cancelado o temporizador e ser reiniciado, caso existam tramas enviadas não confirmadas. Quando o temporizador expira, deve ser reenviada a trama mais antiga não confirmada.

Protocolo Retransmissão Seletiva (*Selective Repeat*)

O protocolo com retransmissão seletiva é descrito na secção 3.4.3 do livro. 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.

No trabalho deste ano não serão usadas tramas NAK (*Negative Acknowledgement*). Em substituição, acrescenta às tramas ACK e DATA um vetor `ackvec`, com os números de sequência das tramas que já foram recebidas fora de sequência. Desta forma, o emissor sabe o que tem de retransmitir, e pode fazer a retransmissão assim que recebe a informação que há uma falha na sequência recebida.



No caso dos NAKs, só se pode enviar um NAK. Isto é, depois de se enviar um NAK de uma certa trama não se pode enviar outro dessa trama. No caso deste vetor, quem recebe o vetor e decide que uma trama tem de voltar a ser enviada, só a envia uma vez. Ao se receber, por

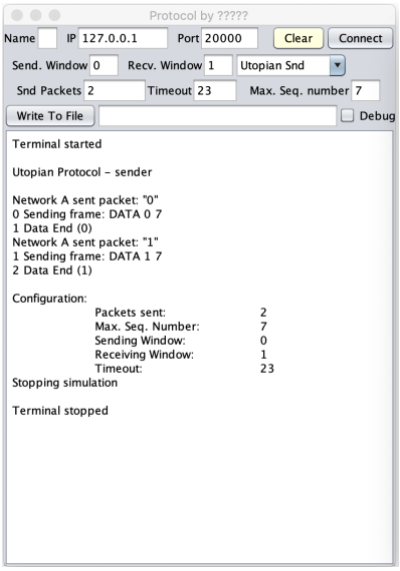
exemplo, `ack=1`, `ackvec=[3,5]`, isto significa que o outro lado recebeu todas as tramas até ao número de sequência 1, e que também já recebeu as tramas com os números de sequência 3 e 5. Pode-se portanto concluir, que se perderam as tramas com os números de sequência 2 e 4. Se nenhuma destas tramas foi reenviada por uma decisão baseada no vetor, voltam a ser enviadas. Se alguma já foi enviada o vetor não traz informação nova e as tramas não são reenviadas.

Este método de retransmissão permite reduzir o tempo para a retransmissão das tramas perdidas face a um mecanismo de recuperação baseado somente num temporizador. Caso este mecanismo falhe (por exemplo o ACK se perde), a recuperação continua a ser feita, tal como no protocolo Go-back-N, através do temporizador. Quando o temporizador expira, o emissor deve retransmitir as tramas pendentes a partir da mais antiga não confirmada, saltando as que já constam de `ackvec`, que sabe que foram bem recebidas.

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.

4.3. APLICAÇÃO *PROTOCOL*

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 *Protocol* 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 *Channel* é feita a ligação (pode ser noutra máquina), escolhendo o IP e o porto. A simulação arranca quando se prime o botão "*Connect*".

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 (em *tics*).

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*. Repetindo, o objetivo do trabalho é desenvolver o código para os quatro últimos protocolos.

4.3.1. Funcionamento Geral

A aplicação *Protocol* tem uma estrutura muito genérica para poder ser a base de muitos trabalhos de Sistemas de Telecomunicações. Assim, existe mesmo algum código que nem vai ser usado neste trabalho.

O elemento mais básico do funcionamento da aplicação *Protocol* é o **evento**. Tudo se faz trocando eventos entre o *Protocol* e o *Channel* e reagindo a eventos recebidos. Um evento pode ter vários tipos (`kind`):

UNDEFINED_EVENT,	STAT_EVENT,	TIME_EVENT,	FRAME_EVENT,
START_TIMER,	TIMER_EVENT,	END_EVENT,	DATA_END,
START_EVENT,	STOP_EVENT,	REQ_CONFIG,	CONFIGURATION.

Dois destes eventos são compostos:

- O evento do tipo FRAME_EVENT que contém uma trama (DATA, ACK ou NAK);
- O evento de estatísticas (STAT_EVENT) que se subdivide em chaves (`key`):
  STAT_RETRANSMITED, STAT_PAYLOADS_TX, STAT_PAYLOADS_RX, STAT_PAYLOADS_RX_INVALID, STAT_PAYLOADS_RX_BUFFERFULL

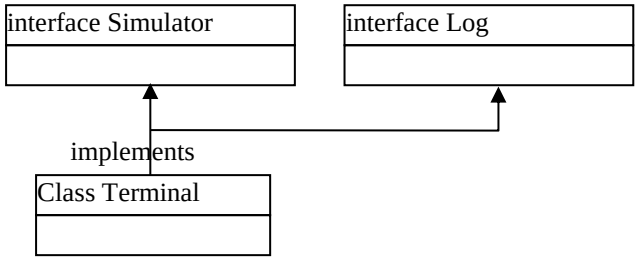
Os temporizadores foram feitos no *Channel*. A solução teve de ser esta pois é o *Channel* que controla o "tempo de simulação", que pode ser diferente do tempo real. A simulação tem *tics* que se relacionam com os segundos reais. Assim, o arranque e a paragem de relógios é feitas por eventos (com chaves para os identificar). Quando um temporizador expirar, um evento é recebido.

O código fornecido divide-se em três pacotes:

No pacote **`terminal`** existe uma interface e três classes muito importantes, que estão completas (Os alunos não têm de codificar nada nelas):

- *Simulator.java* (completa) – Interface que define todos os comandos que podem ser usados na realização de um protocolo de nível Lógico. A classe *Terminal* implementa esta interface.
- *Connection.java* (completa) – *Thread* que implementa o nível Físico. Recebe os eventos já em forma de String para enviar para o *Channel*, e recebe do *Channel* eventos e passa-os ainda na forma de String;
- *NetworkLayer.java* (completa) – Classe que realiza o nível Rede. A informação trocada é também em forma de String.
- *Terminal,java* (completa) – Classe principal com interface gráfica e que tem as seguintes funcionalidades:
  - Reage ao botão "*Connect*" criando os objetos Connection, NetworkLayer e o objeto específico do protocolo de nível Lógico (um dos cinco listados atrás).
  - Contém métodos que simplificam o envio de eventos para o *Channel*. Por exemplo, `start_ack_timer ()`, ou `cancel_ack_timer ()`. Ver a lista em baixo, sob o título "Comandos Terminal".
  - Contém o método `receive_message(String message)` que recebe o evento do *Channel* ainda em String e trata-o chamando as funções específicas do protocolo (um dos cinco). Num dos casos, no evento de configuração, responde logo de imediato enviando um evento sem chamar nada. Um dos eventos é especial: o evento TIME_EVENT quando o `time` é igual zero provoca a chamada ao método `start_simulation(time)` que começa com a simulação.
  - Funções genéricas de serviço às janelas gráficas.
  - Funções para escrever os logs (incluindo a escrita em ficheiro).

A figura em baixo mostra a relação entre a classe *Terminal* e duas interfaces (a *Simulator* descrita em cima e a *Log* descrita em baixo):



Nível Rede

No caso da classe do nível Rede existem os seguintes métodos mais importantes:

- Verificar se há pacotes por enviar no nível Rede:
```
boolean has_more_packets_to_send();
```
- Obter um novo pacote do nível Rede (caso não exista mais nenhum, retorna `null`):
```
String from_network_layer();
```
- Enviar um novo pacote para o nível Rede (caso exista um erro no conteúdo ou falta de espaço para o receber, devolve `false`, indicando que falhou o envio):
```
boolean to_network_layer(String packet);
```

Comandos Terminal

No código que implementa os protocolos, podem-se invocar os seguintes métodos sobre o **objeto `sim`** (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 (para retransmissão seletiva):
```
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);
```
- Verificar se se está a transmitir uma trama de dados para o nível físico (i.e. para o canal):
```
boolean is_sending_data();
```
- Armar o temporizador para tramas de dados. Caso seja armado duas vezes, cancela-se o temporizador mais antigo:
```
void start_data_timer();
```
- Cancelar o temporizador de dados:
```
void cancel_data_timer();
```

- Testar se o temporizador de dados está ativo:

```
boolean isactive_data_timer ();
```

- 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();
```

Os temporizadores de dados e de ACK são realizados usando um objeto temporizador genérico com as chaves `key=1` e `key=-1`, respetivamente para os dados e de ACK. Portanto, existe apenas um temporizador de cada tipo. Não se podem ter vários temporizadores de dados (cada um com o identificador do número de sequência da trama, por exemplo).

O pacote **`simulator`** tem três classes (e também está completo):

- *Event.java* (completa) – Classe que guarda um evento. Tem métodos para: passar de eventos para String e de String para eventos; definir os vários campos dos eventos (todos de uma vez, como por exemplo `new_Frame_Event ()`; ou definir ou ler campo a campo, como por exemplo `set_kind()` ou `kind()`);
- *Frame.java* (completa) – Classe que guarda uma trama. Tem métodos para passar de trama para String e de String para trama. Tal como para a classe *Event*, tem métodos para definir tramas e para ler os campos das tramas;
- *Log.java* (completa) – Interface que define a função *Log* e que depois tem de ser implementada por um objeto (neste caso o objecto *Terminal*);

O pacote **`protocol`** é onde os alunos vão trabalhar. Tem uma interface:

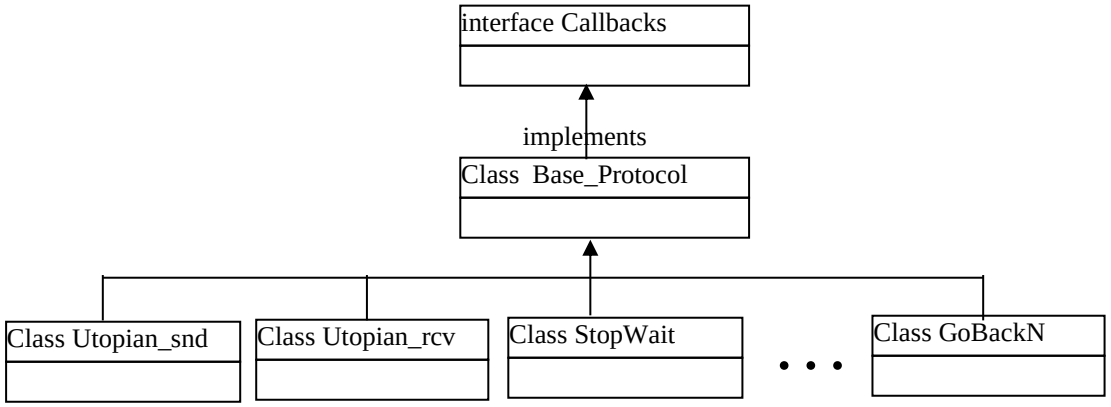
- *Callbacks.java* (completa) – Interface que define todos os métodos que têm de ser implementados na realização de um protocolo de nível lógico;

Este pacote **`protocol`** tem ainda as seguintes classes:

- *Base_Protocol.java* (completa) – Ver descrição em baixo;
- *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 de retransmissão seletiva.

A figura em baixo mostra as relações entre as classes. A classe *Base_Protocol* implementa a interface *Callbacks* e nos métodos apenas se escrevem logs. Os métodos são:

`start_simulation()`, e `end_simulation()`; `from_physical_layer()`; e o tratamento dos três relógios: `handle_Data_Timer()`, `handle_Data_Timer()`, e `handle_ack_Timer()`.



Os alunos devem programar classes que herdem da classe *Base_Protocol* e codificar os métodos que se vão sobrepor aos métodos dela. A classe *Base_Protocol* também contém alguns métodos para manusear números de sequência.

4.3.1.1. 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`, `ackvec` e `info`) e 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, int[] ackvec, String info);
public static Frame new_Ack_Frame(int ack, int[] ackvec);
```

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, ACK_FRAME
public int[] ackvec();  // for DATA_FRAME, ACK_FRAME; is null when it is empty
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);
```

4.3.1.2. 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 next_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 prev_seq(int n);
```

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

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

- 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 (`b-a`), tendo em conta a ordem circular:

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

4.3.2. 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 definir o número de sequência usado nas tramas de dados:

```
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(); // Get packet 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)/* before 0*/,
                                       null/*no ackvec*/, packet);
    sim.to_physical_layer(frame);
    next_frame_to_send= next_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, após um evento *Data_end*:

```
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, testa se o nível rede recebe o pacote com sucesso 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 = next_seq(frame_expected);
    }
  }
}
```

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

4.3.3. Protocolo Simplex *Stop&Wait* (Aula 1)

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:

1) 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;
2) O recetor deve enviar um ACK por cada trama de dados recebida.
3) No emissor é necessário implementar o método *handle_Data_Timer*, para lidar com o expirar do timer.

4.3.4. Protocolo Duplex *Stop&Wait* (Aula 2)

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 caraterísticas 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 as tramas 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. Lembre-se que o recetor tem SEMPRE de enviar uma confirmação para cada trama de dados recebida, preferencialmente através de uma trama de dados.

4.3.5. Protocolo *Go-Back-N* (Aula 3)

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 temporizador de dados únicos. As modificações ocorrem no emissor, que passa a poder ter uma janela de transmissão maior do que um. Assim, a realização passa por dois passos:

1) Na parte do emissor, adicionar um *buffer* de transmissão e as variáveis de controlo necessárias, e implementar o algoritmo de transmissão da janela sem considerar erros no canal;
2) Modificar o emissor para armar o temporizador de dados após uma trama de dados quando não está ligado, retransmitindo tudo a partir da mais antiga, quando o temporizador expira;

Sugestões: Recomenda-se que declare um vetor 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.

4.3.6. Protocolo *Selective Repeat* (Aulas 4 e 5)

A classe *SelectiveRepeat* deve ser programada a partir da classe *GoBackN*, de forma a realizar o protocolo de janela deslizante *Selective Repeat*. As modificações ocorrem tanto no emissor como no recetor.

O recetor passa a poder ter uma janela de receção superior a um. Portanto, passa a poder receber tramas fora de ordem, podendo confirmar blocos de tramas recebidas com um único ACK, e enviá-las para o nível Rede. Passa também a enviar um vetor de ACKs (`ackvec`) com todas as tramas recebidas fora de ordem; desta forma, o emissor pode só retransmitir as tramas ainda não recebidas.

Por outro lado, o emissor pode usar o `ackvec` recém-recebido para desencadear a retransmissão de tramas perdidas no canal. A realização passa por quatro passos:

1) Modificar o recetor para funcionar com uma janela de receção maior do que um, aceitando tramas fora de ordem;
2) Modificar o recetor para enviar o campo `ackvec` com as tramas DATA e ACK;
3) Modificar o emissor para deixar de retransmitir tramas já confirmadas pelo recetor no campo `ackvec`;
4) Modificar o emissor de forma a detetar a falha de receção em tramas, ao analisar o `ackvec` recebido, e para desencadear uma retransmissão rápida inicial, uma única vez.

Sugestões: Declare um vetor 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. Declare também uma estrutura de dados auxiliar para guardar o conteúdo dos vetores `ackvec` recebidos. Novamente, poderá ser necessário declarar outras variáveis adicionais para controlar a receção de dados.

5. 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* na classe `GoBackN`. Comece por copiar o conteúdo da classe `StopWait` para a classe `GoBackN`;
4. Programar o protocolo *Selective Repeat* na classe `SelectiveRepeat`. Comece por copiar o conteúdo da classe `GoBackN` para a classe `SelectiveRepeat`. Conclua o recetor antes de passar ao emissor.

TODOS os alunos devem tentar concluir **as três primeiras fases**. 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 terminado a fase 2. Têm uma semana para completar e testar a fase 3. Nas quarta e quinta semanas devem ter a fase 4 completa e preparar um relatório final. No entanto, devem ter sempre em conta que é preferível fazer menos e bem (a funcionar e sem erros), do que tudo e nada funcionar.

No final do trabalho os alunos devem entregar um ficheiro ZIP ao docente por email com

1) **o código desenvolvido**, e
2) **um breve relatório sobre o protocolo mais completo realizado** (que vai ser avaliado com mais detalhe na discussão final).

No relatório devem enumerar, para a parte de **emissão** (envio de tramas) e para a parte de

receção (receção de tramas): todas as variáveis de estado, eventos recebidos (e.g. temporizadores, tramas, Data_end) e as ações desencadeadas (e.g. envio de tramas, temporizadores, etc.). Juntamente com o enunciado é fornecido um ficheiro *Relatorio_P0_00000.doc* com um exemplo de relatório para o protocolo Utópico fornecido com o enunciado.

POSTURA DOS ALUNOS

Cada aluno 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...)

DATA DE ENTREGA DO TRABALHO

O trabalho deve ser entregue antes das 20:00 do dia 30 de Maio de 2021.