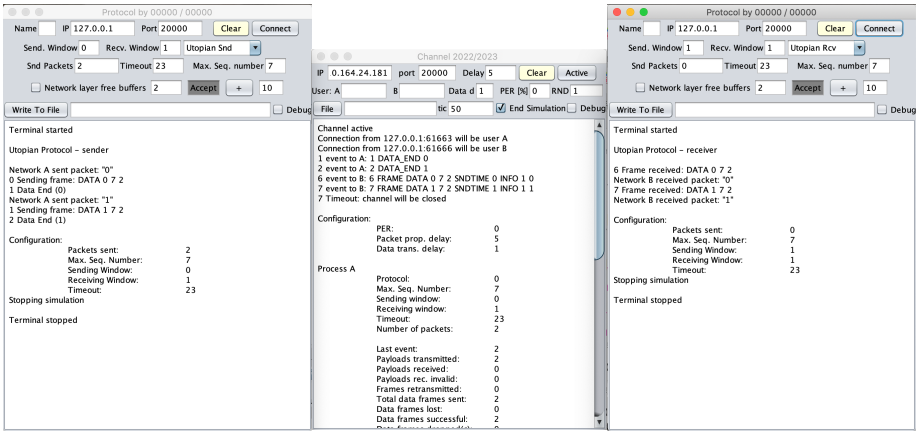
NOVA SCHOOL OF SCIENCE & TECHNOLOGY

DEPARTMENT OF ELECTRICAL AND COMPUTER ENGINEERING



Sistemas de Telecomunicações

2022/2023

Trabalho 5:
Protocolos de nível Lógico com janela deslizante

Aulas 8 a 12

Mestrado integrado em Engenharia Eletrotécnica e de Computadores

https://tele1.deec.fct.unl.pt **V1.0**

Luis Bernardo
Paulo da Fonseca Pinto

Índice

1. Objetivo 1
2. Especificações Gerais 1
2.1. Funcionamento do nível Rede 3
2.2. Protocolos de nível Lógico 4
2.3. Funcionamento do nível Físico 4
3. Aplicação Channel 4
4. Especificações Detalhadas 6
4.1. Tipos de Tramas e Envio de Reconhecimentos 6
4.2. Protocolos de nível Lógico 7
4.3. Aplicação *Protocol* 9
4.3.1. Funcionamento Geral 10
4.3.2. Protocolo Utópico 15
4.3.3. Protocolo *Stop & Wait Simplex* (Aula 1) 16
4.3.4. Protocolo *Stop&Wait Duplex* (Aula 2) 16
4.3.5. Protocolo *Go-Back-N* (Aulas 3 e 4) 17
4.3.6. Protocolo *Go-Back-N* com controlo de fluxo melhorado (Aula 5) 17
5. Metas 17
Postura dos Alunos 18

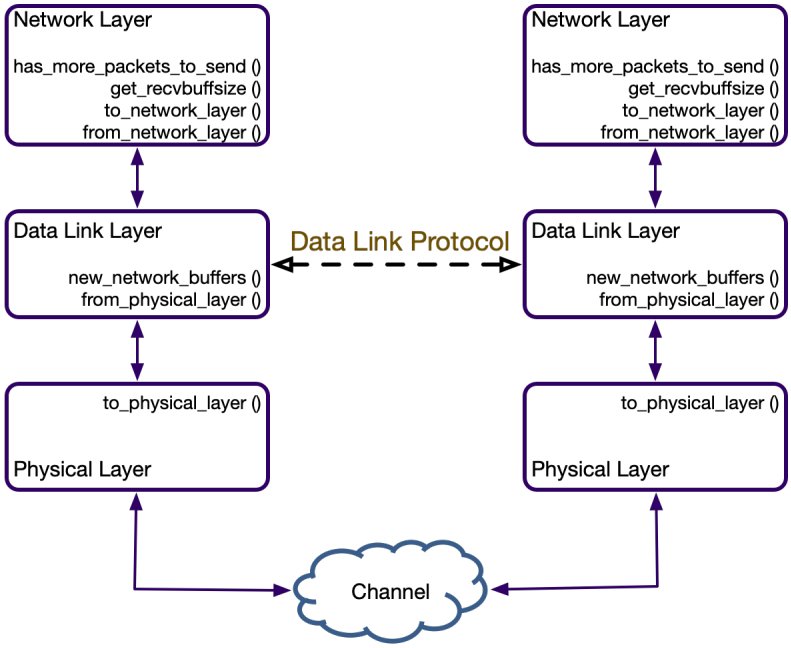
1. OBJETIVO

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

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).



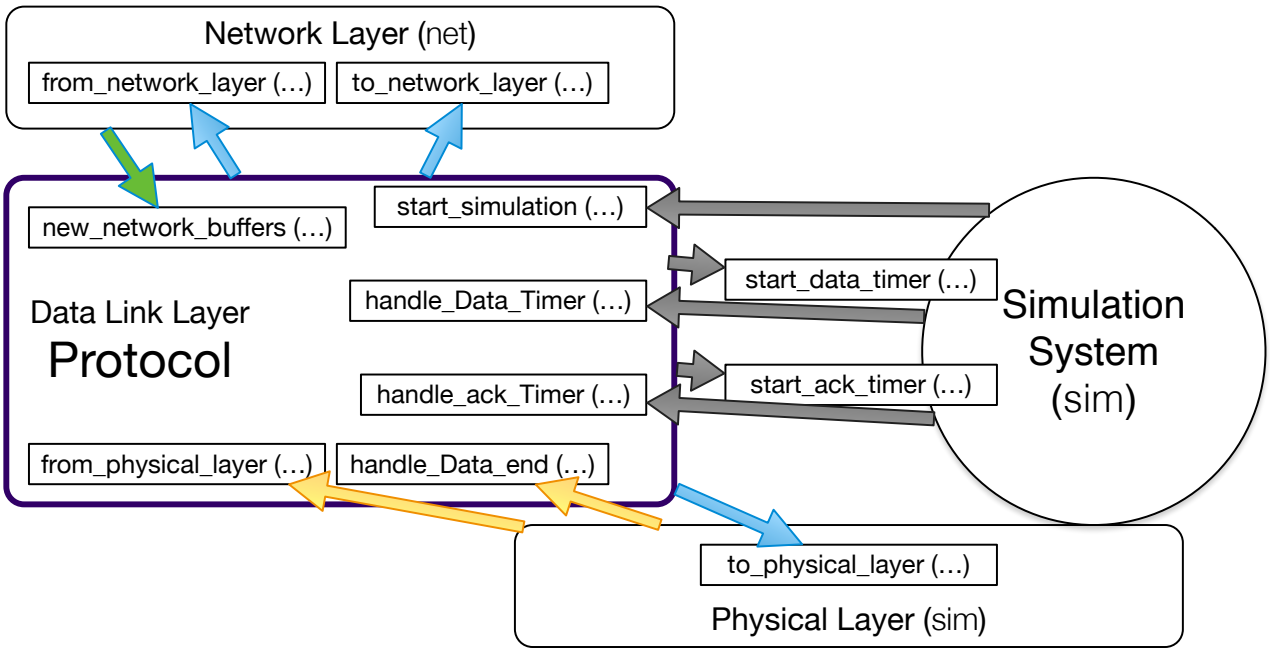
A abordagem proposta foi inspirada no simulador descrito nas secções 3.3 e 3.4 do livro recomendado (Computer Networks 5ª edição), mas esta simulação vai ser um pouco mais próxima da realidade (e.g. considera o tempo de transmissão das tramas de dados). O protocolo de nível lógico vai reagir a eventos 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 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, o nível Lógico invoca 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. O nível Lógico pode saber o espaço disponível no nível Rede para receber dados usando o método do nível Rede `get_recvbuffsize()`. Cada vez que são adicionados buffers livres ao nível Rede, é chamado o método do nível Lógico `new_network_buffers()`. O nível Lógico também pode saber se o nível Rede tem mais dados para enviar usando o método `has_more_packets_to_send()`. Os dados têm de ser entregues

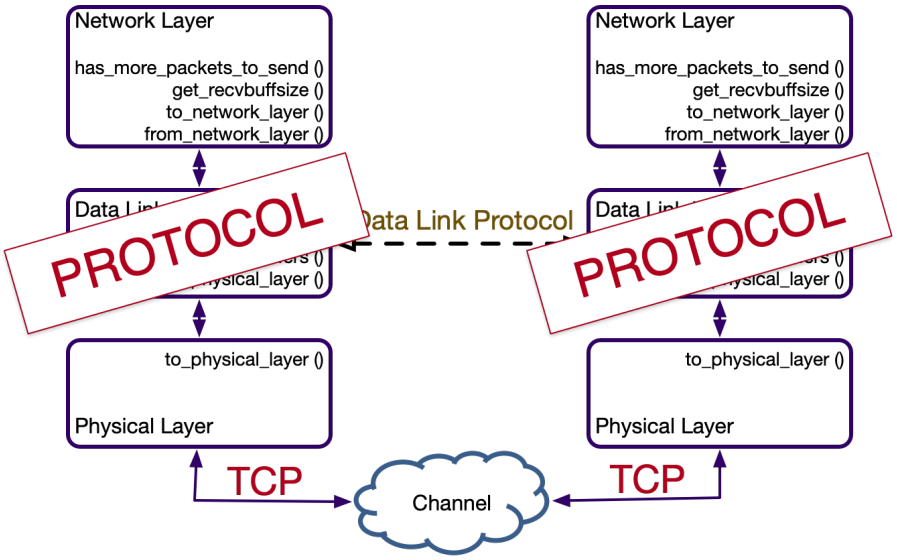
pela mesma ordem em que foram recebidos do nível Rede do outro lado, recuperando de erros que possam existir no canal ou de falhas na entrega ao nível Rede (ver adiante).

Na interface com o nível Físico existem as primitivas de serviço `to_physical_layer()` para enviar tramas e `from_physical_layer()`, para receber tramas. No envio de tramas apenas pode existir uma transmissão de cada vez, obviamente. Se estiver a ser transmitida uma trama (de dados ou controlo), não podem ser transmitidas outras tramas, nem mesmo as pequenas de controlo. O método `handle_Data_end()` é usado para informar que terminou o envio de uma trama de dados. Não existe notificação de fim de envio de tramas de controlo por serem pequenas.

Internamente ao nível Lógico existe um conjunto de funcionalidades de suporte para gerir temporizadores suportado pelo sistema de simulação. 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()`. De forma semelhante, existem os métodos `start_ack_timer`, `cancel_ack_timer` e `handle_ack_Timer()` para poder implementar o envio atrasado de tramas ACK. **Na descrição neste enunciado por vezes usa-se indistintamente "método" e "evento" 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 a caixa designada 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. Fornecido feito.

Ao arrancar, o *Channel* aceita uma ligação TCP de cada um dos 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. Os eventos são ordenados de acordo com o tempo de simulação em que acontecem. O tempo de simulação é medido em unidades de tempo virtuais, designadas de *tics*, que têm uma correspondência com o tempo real – a aplicação permite dizer quantos segundos reais tem um *tic*.

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 a primeira transmissão, que não será recebida no protocolo de destino. As tramas ACK e NAK 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 limitado de buffers para receber pacotes do nível Lógico (no exemplo do livro o número de buffers é ilimitado). Quando a checkbox "*Network layer free buffers*" está selecionada, o número de buffers livres no nível Rede é igual ao número à direita da checkbox, e a transferência fica bloqueada quando o número fica igual a zero. Em qualquer altura de uma simulação, podem ser acrescentados buffers premindo o botão "+". Após acrescentar buffers, é gerado o evento `new_networkbuffers()` no *Protocol*, a informar desta modificação. Em qualquer altura, o *Protocol* pode saber o número de buffers livres usando o método `get_recvbuffsize()`.

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 obtidas pelo nível Lógico a cada invocação do método `from_network_layer()`, que passa a devolver `null` quando se esgotam as mensagens. Pode saber se há mais dados para enviar usando o método `has_more_packets_to_send()`.

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 ao livro para uma correta realização do trabalho. Na realidade, vão existir cinco, e não quatro, tipos de protocolos:

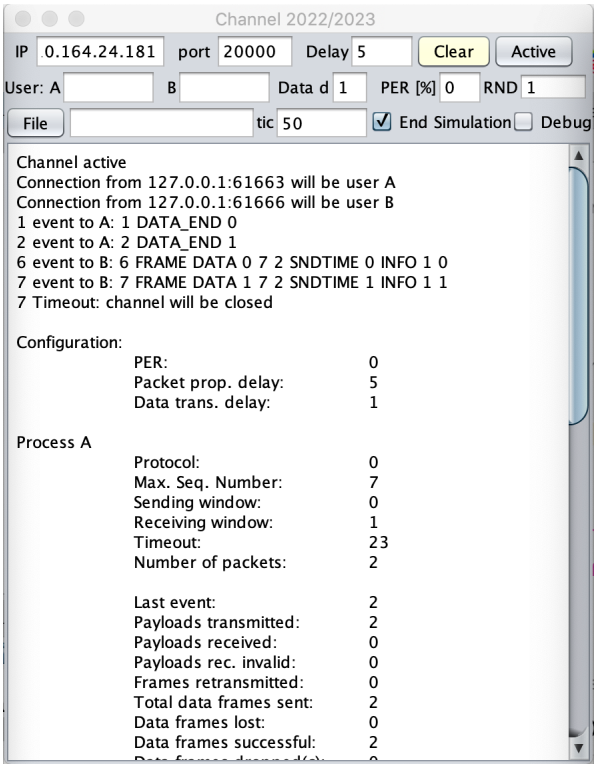
1. Protocolo Utópico – É fornecido o código completo de um emissor e de um recetor deste protocolo juntamente com o enunciado.
2. Protocolo Simplex Stop & Wait – Para ser executado pelos alunos.
3. Protocolo Stop & Wait (duplex) – Para ser executado pelos alunos.
4. Protocolo Go-Back-N – Para ser executado pelos alunos.
5. Protocolo Go-Back-N com controlo de fluxo melhorado (*Go-Back-N_FlowC*) – 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* cria 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 do primeiro bit 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;
- "*RND*" – Semente do gerador de números pseudo-aleatórios usado para simular perda de tramas no canal. Usando o mesmo valor de semente, é possível repetir várias vezes o mesmo padrão de erros no canal;
- "*End Simulation*" – controla se o canal termina automaticamente uma simulação quando não recebe eventos durante um segundo ou o maior tempo de envio e confirmação de trama, se for maior, 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 *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;
- *Payloads rec. full buf* – número de pacotes recebidos com o buffer do nível rede cheio;
- *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;
- *Nak frames lost* – número de tramas NAK perdidas por erros no canal;
- *Nak frames successful* – número de tramas NAK 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 transmitidos. 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 sem erros e com buffers insuficientes para o número de pacotes transmitidos (PER=0, Timeout=13 tics, 10 pacotes, janela emissão 5, buffers= 5, aumentados durante o teste);
v) o mesmo cenário de iv) com PER=50%.

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 três tipos de tramas: DATA, ACK ou NAK.

Tramas de dados (DATA)

As tramas de dados têm os seguintes campos:

- Número de sequência (`seq`)
- Confirmação (`ack`)
- Espaço disponível no buffer de receção (`rcvbufsize`)
- 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 e o campo `rcvbufsize` contém o espaço disponível no buffer de receção. Finalmente, o campo `info` contém os dados da trama.

Como se disse, as tramas de dados têm um tempo de transmissão não nulo. O modo como foi implementado tem estas duas fases:

1) É iniciado o envio da trama com o método `to_physical_layer`;
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 DATA, ACK ou NAK. Caso seja enviada uma nova trama (com o método `to_physical_layer`) o comportamento do `to_physical_layer` depende do argumento `interrupt_if_occupied`:

i) Se for `false`, o envio desta segunda trama falha e o método retorna `false`.
ii) Se for `true`, a transmissão da trama de dados anterior é abortada e começa-se a transmissão da segunda trama, retornando `true`.

```
boolean to_physical_layer(simulator.Frame frame, boolean interrupt_if_occupied);
```

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 dois campos:

- Confirmação (`ack`)
- Espaço disponível no buffer de receção (`rcvbufsize`)

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 foi dito anteriormente, a trama ACK não pode ser enviada enquanto uma transmissão de uma trama de dados está a decorrer.

Como é mais eficiente enviar esta informação (`ack`) através de tramas de dados (por *piggybacking*), deve-se usar o seguinte procedimento:

> Define-se um temporizador auxiliar (`ack_timer`) que é usado para esperar por uma trama de dados que possa transportar esta informação. Se não aparecer nenhuma trama de dados, envia-se o ACK após este tempo.

Concluindo, 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 está 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 temporizador expirar, é gerado o evento `handle_ack_Timer` (ou dito de outro modo, este método é chamado), devendo-se enviar a trama ACK.

Tramas de confirmação negativa (NAK)

A trama NAK contém dois campos:

- Confirmação (`ack`).
- Espaço disponível no buffer de receção (`rcvbufsize`)

O campo Confirmação (`ack`), contém o número de sequência da trama de dados que deve ser retransmitida. Confirma todas as tramas anteriores à `ack` (i.e. `prev_seq(ack)`) e requere a retransmissão da trama de dados cujo número de sequência é `ack`.

As tramas de confirmação negativa (NAK) podem ser geradas em protocolos de janela deslizante quando são recebidas tramas de dados fora de ordem (e.g. está-se à espera da trama 0 e recebe-se a 1) em resultado de se ter saltado algum número de sequência. Em termos de tempo de transmissão, também é considerada uma trama instantânea que só pode ser transmitida quando o meio está livre.

O recetor desta trama deverá retransmitir a trama de dados pedida o mais depressa possível. Para evitar que o emissor esteja continuamente a retransmitir a mesma trama, o NAK só pode ser gerado uma vez para cada situação de tramas fora de ordem, tal como explicado no livro recomendado. A situação considera-se terminada quando se recebe a trama pedida.

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. No código fornecido como 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. Um objetivo é também o de ajudar os alunos a perceberem como os outros protocolos "encaixam" no package Java.

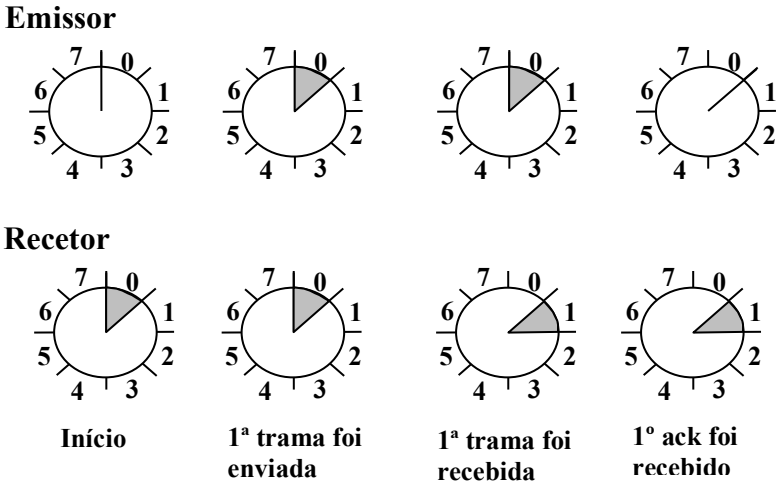
Protocolo *Stop & Wait Simplex*

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 a receber, de acordo com o esquema à direita.

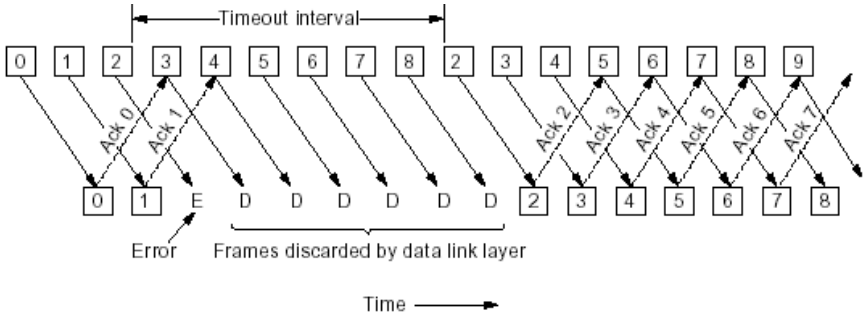


Na prática, este protocolo junta o emissor e o recetor do protocolo *Stop & Wait Simplex* num único objeto que pode enviar e receber dados em paralelo. A grande diferença é que a confirmação da receção de tramas passa a poder ser feita pelo campo ACK da trama de dados. Recorre-se ao temporizador (`ACK_timer`)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

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.



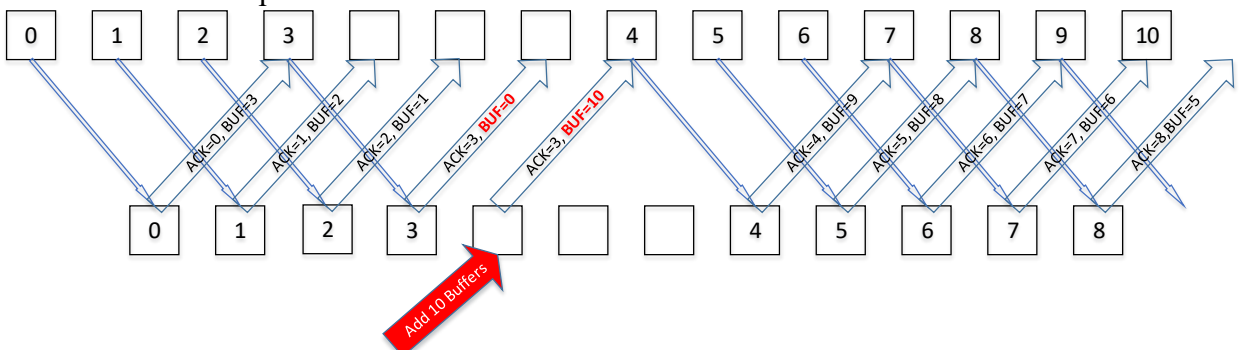
Deve usar um temporizador para cada trama enviada. Cada vez que é confirmada uma trama de dados deve ser cancelado o temporizador associado. Quando este expira, devem ser cancelados todos os temporizadores pendentes, e deve ser reenviada a trama mais antiga não confirmada.

Protocolo *Go-Back-N com controlo de fluxo melhorado* (*Go-Back-N_FlowC*)

O protocolo *Go-Back-N com controlo de fluxo melhorado* melhora o desempenho do protocolo Go-Back-N utilizando o campo `rcvbufsize` presente nas tramas DATA, ACK e NAK para limitar o tamanho da janela de transmissão. O valor da janela é o menor dos seguintes dois: a janela de transmissão definida no início da simulação e o espaço disponível no buffer de receção. Desta forma, previne-se a perda de pacotes durante a entrega ao nível rede no recetor por falta de espaço.

A figura seguinte ilustra um exemplo do funcionamento do protocolo com uma janela de transmissão com 6 tramas, em que o recetor apenas tem 4 buffers livres no início. Após receber as

tramas 0 a 3 o recetor esgotou esse espaço, e envia a trama ACK=3,BUF=0 para sinalizar que o emissor não deve enviar tramas novas – porque se vão perder por não haver espaço no buffer do nível Rede do recetor. O emissor, ao receber a trama ACK=0,BUF=3, fica a saber que só pode enviar até *ACK+BUF* que é igual a *3* (considerando a sequência cíclica). Desta forma, embora receba mais tarde a confirmação de todas as tramas de dados enviadas com a trama ACK=3,BUF=0, vai suspender o envio de NOVAS TRAMAS de dados. No caso de serem acrescentados 10 buffers novos (i.e. ser libertado o espaço no buffer do nível Rede para receber 10 pacotes), é gerada a trama ACK=3,BUF=10 a informar o emissor pode voltar a enviar.



Em caso de perda da trama ACK=3,BUF=10, entra-se num cenário de bloqueio, em que o emissor não pode transmitir porque o último tamanho de buffer que recebeu foi BUF=0. Desta forma, quando o emissor receber BUF=0 com todas as tramas enviadas confirmadas, caso ainda existam pacotes no nível Rede para enviar, deve ir reenviando continuamente a última trama confirmada. Desta forma, o recetor vai repetir o envio da trama com o ACK e BUF em resposta à trama de dados. Uma maneira simples de originar esta repetição de retransmissão é ignorar a confirmação da última trama de dados enviada (quando recebeu BUF=0 e há pacotes no nível rede por transmitir).

A abordagem descrita neste protocolo foi inspirada em protocolos de janela deslizante mais usados no nível transporte do modelo OSI, como o protocolo TCP. Pode encontrar uma descrição do funcionamento deste protocolo na secção 6.2.4 do livro em Inglês, nas páginas 525 e 526. Veja com atenção a figura 6.16.

A única diferença face ao protocolo anterior é que o emissor vai regular a janela de transmissão em função do valor do campo `rcvbufsize` recebido. O recetor deve ser igual ao emissor usado no Go-back-N simples.

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.

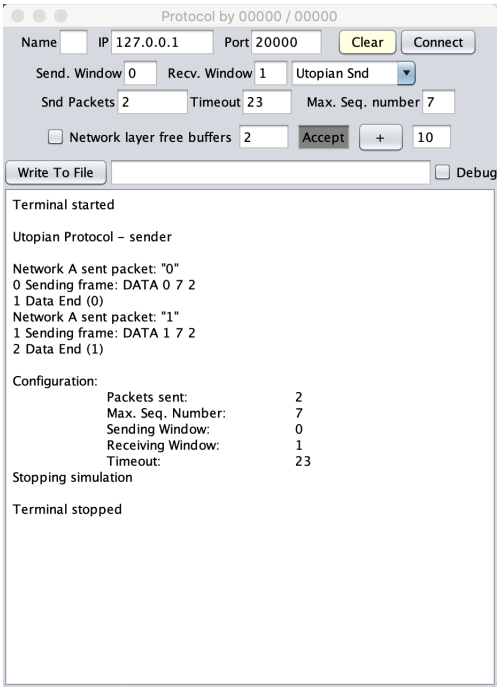
A interface gráfica permite identificar a aplicação *Channel* a usar, através do IP e do porto (pode ser noutra máquina). A simulação arranca quando se prime o botão "*Connect*".

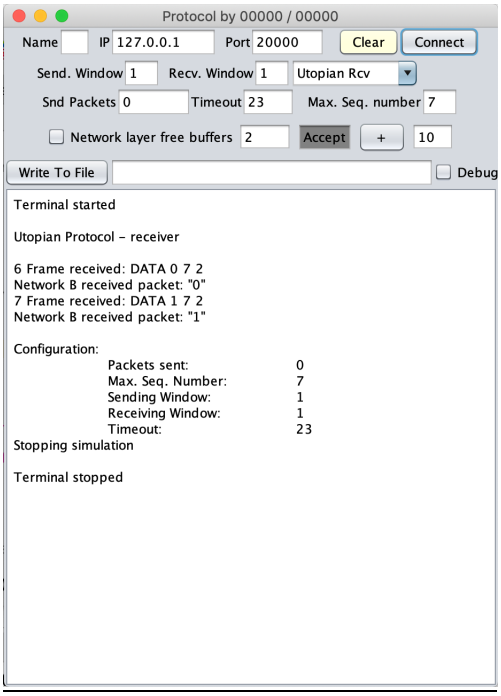
Através da interface gráfica, é possível definir:

- *Snd 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 (é sempre 1 neste trabalho);
- *Timeout* – o tempo de espera por uma confirmação antes de reenviar uma trama de dados (em tics);
- *Network layer free buffers* – o número de buffers livres no nível rede para receberem pacotes (controlado pela checkbox à esquerda);

- "+" – o número de buffers livres incrementados quando se prime o botão '+'.

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 *Go-Back-N FlowC*. 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,
CANCEL_FRAME_EVENT.

Notar que existe um evento do tipo Trama (FRAME_EVENT). Este evento contém uma trama (DATA, ACK ou NAK). No caso do evento de estatísticas (STAT_EVENT), ele ainda se subdivide em chaves (`key`):

STAT_RETRANSMITED, STAT_PAYLOADS_TX, STAT_PAYLOADS_RX,
STAT_PAYLOADS_RX_INVALID, STAT_PAYLOADS_RX_BUFFERFULL

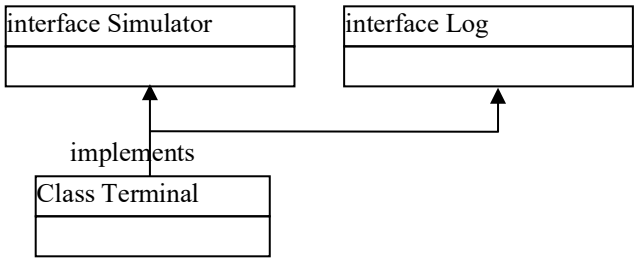
Os temporizadores foram feitos no *Channel*. A solução teve de ser esta pois é o *Channel* que controla o "tempo de simulação", que pode ser diferente do tempo real. Isto é, um segundo de simulação pode ter uma duração diferente de um segundo real. É por isto que se enviam eventos para começar e parar temporizadores (com uma chave como identificação). 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 está também como *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 "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 com `time` 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 interfaces (a interface Log está descrita mais abaixo):



Nível Rede

No caso da classe do nível Rede, podem-se invocar sobre o objeto **`net`** 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 já não exista mais nenhum para enviar, retorna `null`):

```
String from_network_layer();
```

- Verificar o espaço disponível no buffer de receção do nível Rede:

```
int get_recvbuffsize ();
```

- Enviar um novo pacote para o nível Rede (caso exista um erro no conteúdo ou falta de espaço devolve `false`, indicando que falhou o envio):

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

Comandos Terminal

No código que implementa os protocolos, podem-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), podendo-se optar por interromper uma transmissão pré-existente, ou falhar a nova transmissão, dependendo do valor de `interrupt_if_occupied`. Retorna `true` ou `false` como explicado atrás:

```
void to_physical_layer(simulator.Frame frame,
                       boolean interrupt_if_occupied);
```

- Verificar se se está a transmitir uma trama de dados para o nível físico (i.e. para o canal):

```
boolean is_sending_data();
```

- Armar um temporizador para tramas de dados associado à chave `key`. A chave deve ser maior ou igual a 0 e geralmente é igual ao número de sequência da trama de dados associada. Caso seja armado duas vezes com a mesma chave, cancela-se o temporizador mais antigo:

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

- Cancelar o temporizador de dados associado à chave `key`:

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

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

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

- 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();
```

O temporizador de ACK é realizado usando uma chave `key=-1`. Portanto, todos os registos de temporizador que aparecem no canal com a chave `-1` referem-se a eventos de temporizador ACK.

O pacote **`simulator`** tem três classes e três interfaces (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; para 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*);
- *DataFrameIF.java* (completa) – Interface que define os métodos para aceder aos campos existentes numa trama do tipo DATA;
- *AckFrameIF.java* (completa) – Interface que define os métodos para aceder aos campos existentes numa trama do tipo ACK;
- *NakFrameIF.java* (completa) – Interface que define os métodos para aceder aos campos existentes numa trama do tipo NAK.

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

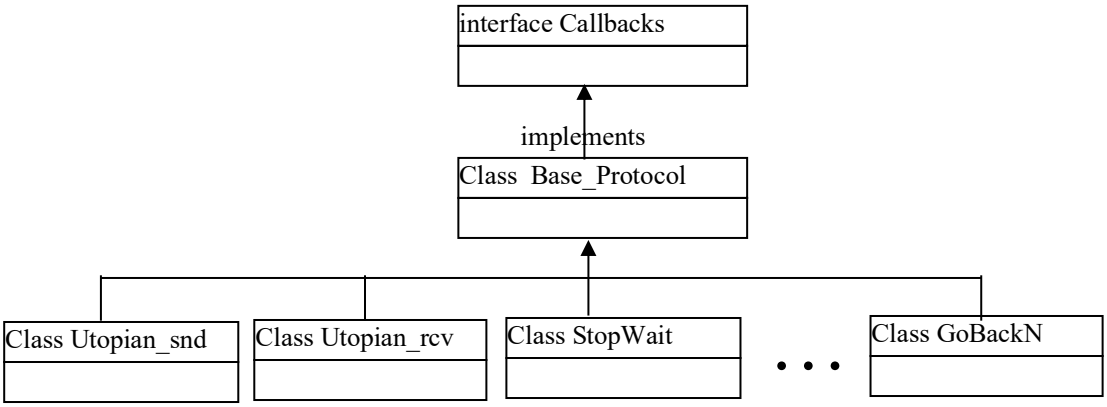
- *Callbacks.java* (completa) – Interface que define todos os métodos que têm de ser implementados na realização de um protocolo de nível Lógico;

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;
- *GoBackN_FlowC.java* (**a completar**) – Realização do protocolo de nível lógico Go-Back-N com controlo de fluxo melhorado.

A figura em baixo mostra as relações entre as classes. A classe *Base_Protocol* implementa a interface *Callbacks* mas os métodos apenas escrevem logs. Exemplos desses métodos são `start_simulation()`, `from_physical_layer()`, `handle_Data_Timer()`, e `handle_ack_Timer()`

Os alunos devem programar classes que herdem da classe *Base_Protocol* e que tenham métodos que se sobreponham 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`, `rcvbufsize` 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 e existem três válidas: `Frame.DATA_FRAME`, `Frame.ACK_FRAME` e `Frame.NAK_FRAME`; e uma inválida `Frame.UNDEFINED_FRAME` (quando a trama 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 rcvbufsize, String info);
public static Frame new_Ack_Frame(int ack, int rcvbufsize);
public static Frame new_Nak_Frame(int nak, int rcvbufsize);
```

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 nak();       // for NAK_FRAME
public int rcvbufsize();// for DATA_FRAME, ACK_FRAME, NAK_FRAME
public long snd_time(); // time when it was sent
```

Para atualizar os campos usam-se os métodos:

```
public int set_seq(int seq);       // for DATA_FRAME
public String set_info(String info);   // for DATA_FRAME
public int set_ack(int ack);       // for DATA_FRAME, ACK_FRAME
public int set_nak(int nak);       // for NACK_FRAME
public int set_rcvbufsize(int size); // for DATA_FRAME, ACK_FRAME, NAK_FRAME
```

Para aceder aos campos de uma trama, deve usar a interface associada a cada tipo de trama. Desta forma, apenas estão disponíveis os campos válidos da trama:

```
DataFrameIF dframe= frame;      // for DATA_FRAME
AckFrameIF aframe= frame;       // for ACK_FRAME
NakFrameIF nframe= frame;       // for NAK_FRAME
```

Por exemplo, para aceder aos campos de uma trama ACK é usada uma variável do tipo `AckFrameIF`. Esta não permite usar campos inexistentes (e.g. `seq()`, que só existe para tramas do tipo DATA).

```
if (frame.kind() == Frame.ACK_FRAME) {      // Check the frame kind
            AckFrameIF aframe= frame;  // Auxiliary variable to access the frame fields.
            if (dframe.ack() == ack_expected) {    // Check the sequence number
                …
            }
        }
```

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 S*tring* 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, ACK e NAK, 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:

```
void 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 /*seq*/,
       prev_seq(0) /* ack= the one before 0 */,
       net.get_recvbuffsize() /* receiver buffer space available in the network layer */,
       packet);
    sim.to physical layer(frame, false /* do not interrupt an ongoing transmission*/);
    next_frame_to_send= next_seq(next_frame_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` que despoleta a chamada ao método `handle_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
    DataFrameIF dframe= frame;  // Auxiliary variable to access the Data frame fields.
    if (dframe.seq() == frame_expected) {     // Check the sequence number
      if (net.to_network_layer(dframe.info())) { // Send the frame to the network layer
        // If the network layer accepted the data
        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 *Stop & Wait Simplex* (Aula 1)

As classes *Simplex_snd* e *Simplex_rcv* devem ser programadas de maneira a implementar o protocolo *Stop & Wait Simplex*, 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 receptor 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 *Stop&Wait Duplex* (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 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. 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* (Aulas 3 e 4)

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 por trama de dados. As modificações ocorrem no emissor, que passa a poder ter uma janela de transmissão maior do que um. Assim, a realização passa por três 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 o envio de uma trama de dados quando não está ligado, retransmitindo tudo a partir da mais antiga, quando o temporizador expira;
3) Modificar o recetor para gerar tramas NAK quando se recebe a primeira fora de ordem, e o emissor para iniciar a transmissão da trama pedida. Não se esqueça que a trama NAK também confirma tramas recebidas.

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.

4.3.6. Protocolo *Go-Back-N* com controlo de fluxo melhorado (Aula 5)

A classe *GoBackN_FlowC* deve ser programada a partir da classe *GoBackN*, de forma a realizar o controlo da janela do emissor utilizando a informação recebida nas tramas sobre o número de *buffers* de receção disponíveis. As modificações ocorrem apenas no emissor, que passa a usar uma janela de emissão inferior à configurada na aplicação, se o número de *buffers* for inferior. Para lidar com erros no canal, quando o último número de *buffers* recebido é zero, tem também de garantir que continua a retransmitir uma trama, de maneira a forçar a retransmissão da trama ACK pelo recetor.

Sugestões: Declare uma variável do tipo inteiro para guardar o valor da janela de transmissão efetiva, e atualize-a cada vez que recebe uma nova trama. Use a variável para controlar a transmissão de tramas. Teste o funcionamento deste protocolo desbloqueando o recetor após estar algum tempo sem buffers livres, com e sem erros.

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 relevante 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*. Cada um vai ter os papeis de emissor e recetor.
3. Programar o protocolo *Go-Back-N* na classe *GoBackN*. Comece por copiar o conteúdo da classe *StopWait* para a classe *GoBackN*. Na parte do emissor acrescente a parte da janela deslizante.

4. Programar o protocolo *Go-Back-N com fluxo melhorado* na classe GoBackN_FlowC. Comece por copiar o conteúdo da classe *GoBackN* para a classe *GoBackN_FlowC*. Acrescente a gestão da janela baseada nos *buffers*.

TODOS os alunos devem tentar concluir **as três primeiras fases (excluindo as tramas NAK)**. 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 duas semanas para completar e testar a fase 3. Na quinta semana 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 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, separadamente 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.). A partir do protocolo *Stop&Wait* inclusive, dentro da mesma classe há partes relacionadas com o envio de dados (herdadas do *Simplex_snd*) e receção de dados (herdadas do *Simplex_rcv*). 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...)