CISTER
Research Center in
Real-Time & Embedded
Computing Systems

Technical Report

Implementação da rede Flexible Time Trigger para Switched Ethernet no Simulador NS-3

Fábio André Gomes Oliveira

CISTER-TR-150108

Implementação da rede Flexible Time Trigger para Switched Ethernet no Simulador NS-3

Fábio André Gomes Oliveira

CISTER Research Center

Polytechnic Institute of Porto (ISEP-IPP)

Rua Dr. António Bernardino de Almeida, 431

4200-072 Porto

Portugal

Tel.: +351.22.8340509, Fax: +351.22.8321159

E-mail:

http://www.cister.isep.ipp.pt

Abstract

This work describes the implementation of the Flexible Time Trigger paradigm for Switched Ethernet on the NS-3 network simulator

Implementação da rede Flexible Time Triggered para Switched Ethernet no Simulador NS-3

CISTER

2014 / 2015

1090566 Fábio André Gomes Oliveira

Simulação da tecnologia FTT-SE no NS-3

CISTER

2014 / 2015

1090566 Fábio André Gomes Oliveira

Licenciatura em Engenharia Informática

Janeiro de 2015

Orientador ISEP: **Prof. Dr. Luis Lino Ferreira**

Supervisor Externo: **Prof. Dr. Michele Albano e Eng. Ricardo Garibay-Martinez**

Aos meus pais,

À minha família,

À minha namorada

E aos meus amigos.

Agradecimentos

Em primeiro lugar, quero endereçar um forte agradecimento ao Professor Doutor Luís Lino Ferreira, ao Professor Doutor Michele Albano e ao Engenheiro Ricardo Garibay-Martínez por me darem todo o apoio ao longo deste projeto e pela dedicação e disponibilidade em ajudar-me nas fases mais complicadas do mesmo, orientando-me sempre no sentido de melhorar o meu trabalho.

Quero também agradecer aos meus pais todos os sacrifícios que enfrentaram para me dar a melhor educação possível e para que pudesse ter uma boa formação e concretizar os meus sonhos. Seguidamente, quero agradecer à minha irmã, cunhado e sobrinhos por estarem sempre ao meu lado e me ajudarem sempre que preciso. Não posso deixar de recordar o meu irmão, que tenho a certeza que estaria orgulhoso de mim. Um agradecimento especial à minha namorada Bárbara Cabeça por toda a dedicação e apoio ao longo desta etapa importante da minha vida e pela paciência nas fases mais difíceis. Aos meus amigos e colegas de curso, em especial ao Bernardo Godinho, padrinho e amigo, presente em todos os momentos, ao Tiago Cerqueira fiel companheiro de Licenciatura e estágio e ao meu afilhado Simão Ribeiro pela amizade. Um grande obrigado também à Maria de Castro e ao João Ribeiro, grandes apoios nos momentos de *stress*.

Resumo

Nos últimos anos tem-se assistido a um crescimento na utilização de sistemas embebido, sendo que muitos destes sistemas se encontram espacialmente separados, realizando comunicações distribuídas de modo a cumprirem os requisitos de tempo-real das suas aplicações. Muitas destas aplicações apresentam exigências temporais muito restritas e requerem um alto nível de determinismo no que diz respeito aos tempos em que as suas tarefas são executadas. O protocolo *Flexible Time Triggered – Switched Ethernet* (FTT-SE), baseado no paradigma *Flexible Time Triggered* (FTT), oferece garantias do determinismo temporal exigido por estas aplicações, apresentando igualmente flexibilidade e uma gestão dinâmica do serviço (QdS).

Neste trabalho é apresentada a implementação do protocolo FTT-SE no simulador de redes NS-3, a primeira implementação deste protocolo para um simulador. O trabalho compreende o desenvolvimento de aplicações, no NS-3, que simulem o comportamento das aplicações *Master*, que realiza o escalonamento de tráfego, e *Slave*, que comunica com outras aplicações *Slave*, numa rede FTT-SE. A implementação define também dois tipos de comunicação entre aplicações *Slave*: sequencial (*end-to-end*) e de acordo com o paradigma *Fork-Join Parallel/Distributed*. Para estes tipos de comunicação, é descrito neste trabalho a integração da simulação de um mecanismo de escalonamento de tarefas por prioridades, único no NS-3, seguindo a política de escalonamento preemptiva *Rate Monotonic*. Através da implementação são simulados vários cenários de rede variando as características das aplicações utilizadas, procurando analisar os resultados obtidos e justificar os mesmos.

A implementação descrita no relatório torna-se, deste modo, um contributo para a comunidade de investigação, uma vez que oferece a possibilidade de estudar e analisar a rede FTT-SE de uma forma prática, fiável e a um baixo custo.

Palavras Chave (Tema): FTT-SE, NS-3, sistemas embebidos, sistemas distribuídos, tempo-real, Switched Ethernet

Palavras Chave (Tecnologias): C++, Linux

Índice

Índice de Figuras

Índice de Tabelas

Notação e Glossário

AW	Asynchronous Window
BSD	Burkeley Sockets
C++	Linguagem de programação orientada a objetos
CPU	Processador
CSMA/CD	Carrier Sense Multiple Access With Collision Detection
DEP	Distributed Execution Path
EC	Elementary Cycle
EDF	Earliest Deadline First
FTT	Flexible Time Triggered
FTT-SE	Flexible Time Triggered - Switched Ethernet
NES	Network Embedded System
NRBD	Node Requirements Database
RBF	Request Bound Function
RM	Rate Monotonic
RT	Response Time
SBF	Supply Bound Function
SRDB	System Requirements Database
SW	Synchronous Window
TCP	Transmission Control Protocol
TM	Trigger Message
UDP	User Datagram Protocol
WCET	Worst-Case Execution Time
WCML	Worst-Case Message Length

WCRT Worst-Case Response Time

1 Introdução

Neste capítulo, é introduzido o projeto realizado, fazendo o seu enquadramento e apresentação dos seus principais objetivos. São também descritas brevemente as tecnologias utilizadas no trabalho e, por fim, é apresentada a organização do relatório.

1.1 Enquadramento

O presente relatório é desenvolvido no âmbito da unidade curricular de Projeto/Estágio (PESTI), da Licenciatura em Engenharia Informática (LEI), relativa ao ano letivo 2014/2015, do Instituto Superior de Engenharia do Porto (ISEP). Em PESTI, os alunos procuram aplicar os conhecimentos adquiridos ao longo do curso e as suas competências num ambiente real de trabalho, tendo a referida unidade curricular um papel bastante importante de integração no contexto profissional.

O projeto de estágio foi realizado no Centro de Investigação em Sistemas Embebidos e de Tempo-Real (CISTER) [1], entre os meses de março de 2014 e janeiro de 2015, e teve como objetivos a implementação no simulador NS-3 [2] do protocolo de comunicação para sistemas embebidos de tempo-real, designado por *Flexible Time Triggered – Switched Ethernet* (FTT-SE), e a realização de simulações afim de obter resultados, considerando vários cenários, e fazer a respetiva análise.

1.2 Apresentação do projeto/estágio

A utilização de sistemas embebidos tem tido um grande crescimento ao longo dos últimos anos, estando presente em múltiplos objetos com os quais interagimos no nosso dia-a-dia, por exemplo: carros, telemóveis, máquinas de café, sistemas industriais, etc. Alguns destes sistemas - geralmente denominados de *networked embedded systems* (NES) - são inerentemente distribuídos, cooperando entre si de modo a cumprirem os requisitos das suas aplicações. Estas aplicações requerem um grau elevado de determinismo temporal, uma vez que apresentam, muitas vezes, exigências temporais que têm de ser cumpridas. Deste modo, é fundamental que a infraestrutura de comunicação garanta o cumprimento dos requisitos de tempo-real. É neste contexto que é introduzido o FTT-SE, baseado no paradigma Flexible Time Triggered (FTT), protocolo que dá garantias de cumprimento dos requisitos de tempo-real, bem como apresenta propriedades de flexibilidade e gestão dinâmica de qualidade do serviço. [3]

Pretende-se neste projeto que seja implementado num simulador, o NS-3, o protocolo supra referido de forma que, através de simulações, se obtenham resultados bastante aproximados daqueles que, na teoria, são esperados de acordo com o protocolo, e consequentemente seja possível, mais facilmente, estudar e analisar o comportamento destes sistemas. Esta implementação é particularmente importante para estudar o comportamento de sistemas com um elevado número de nós e testar múltiplas configurações do sistema.

É também objetivo do projeto realizar um conjunto de experiências em diferentes cenários, usando aplicações sequenciais, e também aplicações paralelas distribuídas, onde é pretendido processamento de dados em nós remotos, com base no paradigma *Parallel/Distributed* [4], fazendo variar as topologias de rede, a quantidade de tráfego transmitido, a natureza do tráfego, a largura de banda reservada para a transmissão do mesmo, bem como o tempo de processamento dos dados remotamente. Através dos resultados obtidos, pretende-se analisar, para cada cenário, as condições que trazem maiores benefícios para a realização das tarefas requeridas, assim como identificar os cenários onde esta abordagem deixa de ser vantajosa. Para finalizar os objetivos, acrescentar a possibilidade de exportar resultados em ficheiros de texto.

1.2.1 Planeamento de projeto

O planeamento para este projeto compreendeu essencialmente três fases: análise, implementação e obtenção e análise de resultados. As perspetivas iniciais apontavam para a conclusão do trabalho a 22-08-2014. No entanto, pela sua natureza de investigação, teve uma

componente teórica bastante acentuada, pelo que a fase de análise revelou-se bastante mais extensa do que foi inicialmente previsto. O facto de nunca ter trabalhado anteriormente com o simulador NS-3 e também, devido à complexidade inerente ao protocolo *Flexible Time Triggered – Switched Ethernet,* associada ao meu completo desconhecimento relativamente ao mesmo, em muito contribuíram para o desfasamento do que foi previamente planeado para a conclusão desta fase.

Por consequência, devido às dificuldades acima enumeradas, a fase de implementação também sofreu alguns atrasos, pelo que o planeamento foi sendo reajustado. Com o avançar do projeto, novos objetivos também foram sendo definidos. No sentido de responder adequadamente a todos os objetivos, a fase de implementação também foi prolongada.

O diagrama de Gantt demonstrativo do planeamento final é apresentado na Figura 1. Este diagrama pode também ser encontrado, em tamanho ampliado, no Anexo 1.

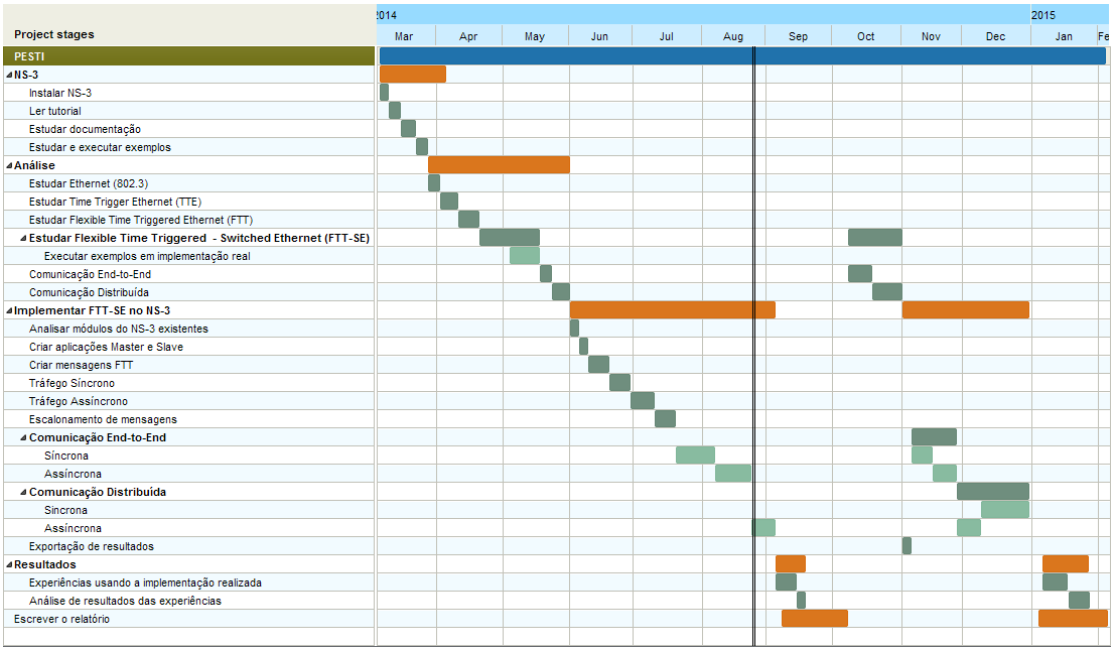


Figura 1: Diagrama de Gantt do planeamento do projeto

1.2.2 Reuniões de acompanhamento

Nesta secção, são apresentadas as reuniões de acompanhamento realizadas ao longo do projeto, onde frequentemente foi debatido o estado do trabalho e foram esclarecidas dúvidas relativamente aos assuntos do projeto. A Tabela 1 expõe o essencial relativamente às reuniões realizadas, sendo que as mesmas tiveram sempre lugar no CISTER.

Acrescentar também que outras reuniões, de carácter mais informal, foram realizadas semanalmente durante um evento da própria organização, denominado *"Coffee Morning"*, onde todos os membros da organização se juntam, num ambiente mais descontraído.

Tabela 1: Reuniões de acompanhamento do projeto

Data	Participantes	Assunto
07-03-2014	Prof. Luís Lino Ferreira e Prof. Michele Albano	Discussão sobre objetivos do projeto
17-03-2014	Prof. Michele Albano	Esclarecimento de dúvidas
18-03-2014	Prof. Luís Lino Ferreira e Prof. Michele Albano	Discussão sobre NS-3
25-03-2014	Prof. Luís Lino Ferreira e Prof. Michele Albano	Apresentação sobre FTT-SE
01-04-2014	Prof. Luís Lino Ferreira, Prof. Michele Albano, Eng. Ricardo Garibay-Martinez e Roberto Duarte	Progresso do trabalho
07-04-2014	Prof. Luís Lino Ferreira	Progresso do trabalho
24-04-2014	Prof. Luís Lino Ferreira, Eng. Ricardo Garibay-Martinez e Roberto Duarte	Progresso código testado
02-05-2014	Prof. Luís Lino Ferreira, Eng. Ricardo Garibay-Martinez, Roberto Duarte e Eng. Ricardo Marau	Teleconferência de esclarecimento de dúvidas sobre FTT-SE com Eng. R. Marau
15-05-2014	Prof. Luís Lino Ferreira, Eng. Ricardo Garibay-Martinez e Roberto Duarte	Progresso do trabalho
19-05-2014	Prof. Luís Lino Ferreira	Progresso do trabalho
20-05-2014	Prof. Luís Lino Ferreira	Discussão sobre implementação
27-05-2014	Eng. Ricardo Garibay-Martinez	Esclarecimento de dúvidas
06-06-2014	Prof. Luís Lino Ferreira	Progresso do trabalho
16-06-2014	Prof. Luís Lino Ferreira	Progresso do trabalho
17-06-2014	Prof. Michele Albano	Esclarecimento de dúvidas
27-06-2014	Prof. Michele Albano	Esclarecimento de dúvidas
04-08-2014	Prof. Luís Lino Ferreira e Eng. Ricardo Garibay-Martinez	Esclarecimento de dúvidas
13-08-2014	Prof. Luís Lino Ferreira	Discussão sobre o relatório
27-08-2014	Eng. Ricardo Garibay-Martinez e Roberto Duarte	Discussão sobre experiências e resultados
03-09-2014	Prof. Luís Lino Ferreira	Discussão sobre o relatório
04-09-2014	Prof. Luís Lino Ferreira	Discussão sobre o relatório

15-09-2014	Eng. Ricardo Garibay-Martinez	Progresso do trabalho
16-09-2014	Prof. Luís Lino Ferreira e Eng. Ricardo Garibay-Martinez	Discussão sobre o relatório
22-09-2014	Eng. Ricardo Garibay-Martinez	Discussão sobre assuntos relacionados com FTT-SE
25-09-2014	Prof. Luís Lino Ferreira	Discussão sobre o relatório
07-10-2014	Prof. Luís Lino Ferreira e Roberto Duarte	Discussão sobre testes e o relatório
22-10-2014	Eng. Ricardo Garibay-Martinez	Progresso do trabalho e esclarecimento de dúvidas
10-11-2014	Eng. Ricardo Garibay-Martinez	Progresso do trabalho e esclarecimento de dúvidas
11-11-2014	Prof. Luís Lino Ferreira e Prof. Michele Albano	Esclarecimento de dúvidas
12-11-2014	Eng. Ricardo Garibay-Martinez	Progresso do trabalho e esclarecimento de dúvidas
14-11-2014	Eng. Ricardo Garibay-Martinez	Progresso do trabalho e esclarecimento de dúvidas
25-11-2014	Eng. Ricardo Garibay-Martinez	Progresso do trabalho
28-11-2014	Eng. Ricardo Garibay-Martinez	Progresso do trabalho
11-12-2014	Prof. Luís Lino Ferreira e Eng. Ricardo Garibay-Martinez	Progresso do trabalho e relatório
17-12-2014	Eng. Ricardo Garibay-Martinez	Progresso do trabalho
19-12-2014	Prof. Luís Lino Ferreira, Prof. Michele Albano e Eng. Ricardo Garibay-Martinez	Discussão de assuntos relacionados com o projeto
22-12-2014	Eng. Ricardo Garibay-Martinez	Progresso do trabalho
23-12-2014	Eng. Ricardo Garibay-Martinez	Progresso do trabalho

1.3 Tecnologias utilizadas

Uma das tecnologias utilizadas neste projeto foi o simulador **NS-3**, iniciado em 2006, tendo como vantagens ser *open source* e ter sido desenvolvido principalmente para uso académico e de investigação. A escolha deste simulador deveu-se, principalmente, pela sua vasta utilização em projetos de investigação, por ser um simulador robusto e com suporte para variadas tecnologias. É uma tecnologia escrita em **C++**, aberta a investigadores que pretendam partilhar os seus projetos com o simulador, sendo a contribuição o espírito da tecnologia.

O foco central deste projeto assenta no protocolo ***Flexible Time Triggered – Switched Ethernet* (FTT-SE)**, uma implementação baseada no paradigma FTT sobre uma rede Ethernet com topologia segmentada por um ou mais *switches.* Este protocolo foi proposto na Universidade de Aveiro por Ricardo Marau. O paradigma FTT, da autoria de Paulo Pedreira e Luís Almeida, compreende uma rede coordenada por um nó *Master* e vários nós *Slave.*

Para o desenvolvimento do código da implementação do FTT-SE no NS-3, foi usado o **Netbeans IDE**.

Em fase de análise de resultados foi utilizado o **Wireshark**, um programa que tem com finalidade a análise do tráfego de uma rede. Através deste programa é possível verificar e analisar todos os pacotes que são trocados em determinadas interfaces.

Para finalizar, acrescentar que todo o projeto foi realizado em ambiente Linux, usando para isso a distribuição **OpenSUSE**.

1.4 Apresentação da organização

O Centro de Investigação em Sistemas Embebidos e de Tempo-Real (CISTER) é uma unidade de investigação Portuguesa de referência, baseada no Instituto Superior de Engenharia do Porto (ISEP) do Politécnico do Porto (IPP). O Centro foca a sua atividade de investigação na análise, projeto e implementação de sistemas de computadores embebidos e de tempo-real, sendo um dos líderes mundiais na investigação em diversos tópicos dentro das áreas das redes de sensores sem fio, das plataformas *multi-core* embebidas ou de *software* de tempo-real.

Desde que foi criado (em 1997), o CISTER cresceu até se tornar a unidade mais proeminente do ISEP, sendo o único Centro de I&D Português nas áreas da Engenharia Eletrotécnica e Informática a obter consecutivamente a avaliação de Excelente nos últimos dois processos de avaliação de I&D em Portugal, realizada por painéis internacionais. O Centro participa consistentemente em projetos de Investigação, Desenvolvimento e Inovação (I&D&I) nacionais e internacionais, com parceiros como a Portugal Telecom, a Critical Software, a ISA, a Thales, a EADS, a Infineon, a SAP, a Schneider Electric ou a Embraer. [1]

1.5 Contributos deste trabalho

Através do NS-3 é possível simular e analisar várias tecnologias de rede existentes atualmente, sendo por isso, uma ferramenta bastante popular na comunidade de investigação.

Até ao momento, nenhum simulador existente tem implementado o protocolo FTT-SE, pelo que apenas através do *software* da implementação real seria possível ter contacto com o mesmo. Desta forma, com este projeto torna-se possível aos estudantes e investigadores realizar experiências de simulações desta tecnologia com resultados bastante aproximados da realidade duma forma mais fácil e prática.

Também é contributo deste trabalho a análise realizada aos resultados obtidos, através de experiências realizadas usando a implementação desenvolvida para o NS-3. Os resultados

permitirão identificar as melhores condições de comunicação em sistemas distribuídos de tempo-real, em diferentes requisitos de tráfego.

1.6 Organização do relatório

Este relatório encontra-se dividido em 7 capítulos principais, estando organizado da seguinte forma:

Capítulo 1 – Introdução: é realizada uma apresentação e enquadramento do projeto, apresentação da organização e os contributos do trabalho.

Capítulo 2 – Contexto: é realizada uma contextualização do trabalho, apresentando noções teóricas essenciais para a compreensão do trabalho realizado.

Capítulo 3 – *Flexible Time Triggered over Switched Ethernet* (FTT-SE): é descrito em detalhe o protocolo FTT-SE.

Capítulo 4 – *Network Simulator* (NS-3): são introduzidos os aspetos gerais do simulador NS-3, e descrito com maior detalhe os componentes relevantes para o projeto.

Capítulo 5 – Descrição técnica: são apresentados os requisitos do trabalho e uma descrição da implementação realizada neste projeto.

Capítulo 6 – Resultados e validação: são realizadas algumas experiências usando a implementação realizada no projeto, fazendo uma análise aos resultados obtidos.

Capítulo 7 – Conclusão: são apresentadas as conclusões finais do trabalho realizado, mencionando os objetivos concretizados, o trabalho futuro e apreciação final ao projeto.

2 Contexto

Neste capítulo será apresentado uma contextualização detalhada do tema do projeto abordando os conceitos essenciais relativamente aos sistemas distribuídos de tempo-real, a definição das comunicações *Event* e *Time Triggered*, políticas de escalonamento e a descrição de uma rede *Switched Ethernet*.

2.1 Sistemas de tempo-real

Existe um grande número de aplicações ligadas, por exemplo, à robótica, à indústria e aos sistemas de transporte que são suportadas por sistemas embebidos. Estes sistemas controlam normalmente um sistema físico capaz de recolher dados, que posteriormente são processados e geram uma resposta. Devido à dinâmica destes sistemas, a resposta terá que estar pronta num tempo adequado. Muitas destas aplicações têm exigências temporais muito precisas no que diz respeito às respostas criadas, formando sistemas de tempo-real, sendo o tempo uma característica fundamental no seu algoritmo. [3] [5]

Pode-se definir sistema de tempo-real como um sistema onde a sua exatidão não depende apenas dos seus resultados computacionais, mas também dos instantes em que os mesmos acontecem. Computadores de sistemas de tempo-real devem reagir a estímulos dentro de intervalos de tempo estipulados pelo ambiente em que estão inseridos, e devem produzir os seus resultados antes de determinados instantes, chamados *deadlines*. (Figura 2) [5]

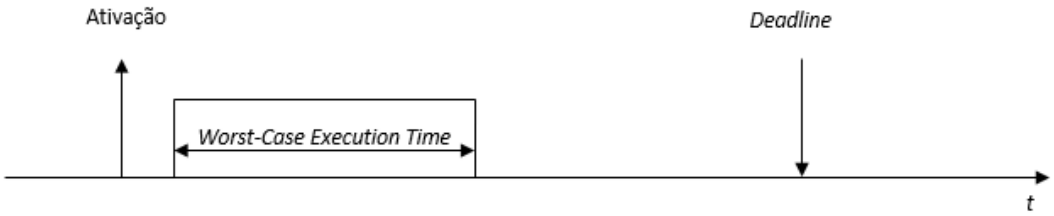


Figura 2: Deadline de uma tarefa

No entanto, nem sempre as respostas são produzidas nos instantes em que são esperadas falhando, por isso, o *deadline*. Avaliando o impacto destas falhas pode-se classificar um *deadline* como *soft* ou *firm*. Atribui-se a primeira classificação quando, apesar da violação do *deadline*, o resultado produzido continuar, ainda assim, a ser proveitoso. Caso contrário, o *deadline* terá a classificação de *firm*. Se um *firm deadline* não é cumprido, e dessa falha resultar uma catástrofe, o *deadline* terá então a designação *hard*. Um sistema de tempo-real que tenha pelo menos um *hard deadline* é denominado sistema de tempo-real *hard*. Caso tal não se verifique, terá a classificação de sistema de tempo-real *soft*.

A sinalização, por meio de um semáforo, do cruzamento com um caminho-de-ferro é um exemplo de um sistema de tempo-real *hard.* Se a luz do semáforo não alterar para vermelho antes da travessia do comboio, pode gerar um acidente catastrófico. Deste modo, estes sistemas devem ser cuidadosamente desenhados, de forma a garantir o cumprimento das restrições temporais exigidas.

A execução de um sistema de tempo-real pode ser realizada em um ou vários CPUs. Esta execução pode ser efetuada de forma centralizada, por meio de memória partilhada, ou então pode ser efetuada de forma distribuída onde existe a intervenção de vários nós ligados em rede. [3]

Este trabalho é focado em sistemas embebidos distribuídos de tempo-real e é abordado com maior detalhe na próxima secção.

2.2 Sistemas distribuídos de tempo-real

Os sistemas embebidos distribuídos formam um subconjunto dos sistemas embebidos, onde o trabalho realizado encontra-se distribuído por vários nós, que comunicam por meio de rede. Neste contexto, o sistema de comunicação tem um papel fulcral, uma vez que, em caso de falha, tem como consequência a falha de todos os serviços do sistema. Portanto, é exigível ao sistema de comunicação um alto grau de confiabilidade para que o sistema funcione como esperado. [5]

Existem diferenças, do ponto de vista funcional, entre uma abordagem centralizada e uma abordagem distribuída, na implementação de um sistema de tempo-real *hard*. Esta temática tem merecido grande interesse por parte das comunidades das áreas da indústria e da investigação, particularmente no que diz respeito às vantagens que a abordagem distribuída oferece comparativamente com os sistemas centralizados.

Um dos principais argumentos apontados é a composição. Muitas vezes, os sistemas são construídos através da integração de vários subsistemas sendo fundamental que a integração dos mesmos seja realizada de forma correta e de forma não prejudicial aos subsistemas já integrados. Tomando como exemplo o sistema carro, muitos são os subsistemas integrados, como, por exemplo, a travagem ou a suspensão. É fulcral que estes subsistemas continuem a funcionar corretamente sem a necessidade de redesenhar e testar novamente outros já integrados.

Também a escalabilidade é uma característica importante. Ao longo da vida de um sistema novos requisitos podem ser adicionados, provocando mudanças ao nível das funções a executar, bem como, na possibilidade de existir novas a serem adicionadas, o sistema deve permitir que as mudanças se possam realizar de forma ilimitada. Novos nós podem ter de ser adicionados no sentido de aumentar a capacidade do sistema.

Nos sistemas distribuídos, a tolerância a falhas é uma especificação onde se deve focar particular cuidado. Aquando da deteção de uma falha, esta deve ser prontamente corrigida de forma a não colocar em causa todo o sistema.

Para terminar a enumeração dos principais argumentos favoráveis à abordagem dos sistemas distribuídos, falta referir a instalação física. O desenho das funções dos subsistemas têm em atenção a redução da mão-de-obra e o custo da instalação. [5]

Os diversos nós deste tipo de sistemas comunicam através de mensagens. Estas mensagens são geradas internamente nos nós através de tarefas, que concorrem entre si pelo seu processamento. Sabendo que os sistemas de tempo-real têm exigências temporais muito rigorosas, é importante garantir que os tempos de resposta a cada tarefa (*Task*) sejam de acordo com os requisitos do próprio sistema.

Através da Figura 3 é possível perceber melhor o percurso de uma mensagem correspondente a uma tarefa, numa comunicação entre um nó local e um nó remoto, e identificar os atrasos inerentes ao seu trajeto que definem o pior caso do tempo de resposta (*Worst-Case Response Time*).

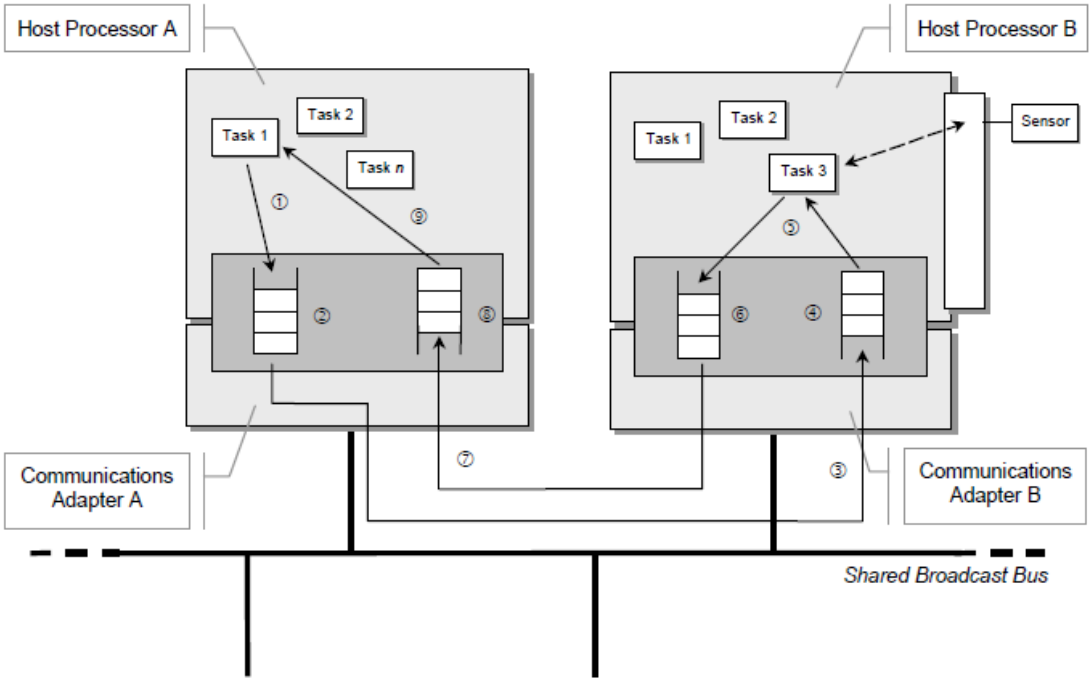


Figura 3: Worst-Case Response Time de uma tarefa num sistema distribuído [6]

Inicialmente, a tarefa tem de concorrer com outras tarefas pelo processador (1). Seguidamente, a mensagem referente à tarefa é inserida numa fila (*queue*), onde tem que aguardar até que seja enviada para a rede (2). A próxima etapa está relacionado com o tempo de envio através da rede, que é condicionado, essencialmente, pela velocidade da mesma e pelo atraso da própria rede (3). Chegada ao dispositivo de rede do nó remoto, a mensagem é inserida numa fila (4) até ser, posteriormente, processada por uma tarefa naquele nó. Desse processamento, resulta uma mensagem de resposta que será inserida numa fila (5) que terá de realizar o percurso inverso, sendo afetado pelo mesmo tipo de atrasos que a primeira mensagem sofreu. A mensagem aguarda na fila até que seja enviada para a rede (6), é transmitida (7), inserida numa fila aquando da chegada ao dispositivo de rede do nó local, onde espera (8) até ser processada, sendo gerada uma mensagem de resposta (9). [6]

2.3 Comunicações Event e Time triggered

Nos sistemas de tempo-real existem duas abordagens distintas no que diz respeito às comunicações: *time-triggered* (TT) e *event-triggered* (ET). Na primeira abordagem, as comunicações são realizadas tendo em conta a linha do tempo, sendo iniciadas em instantes predefinidos, em períodos de igual duração. (Figura 4) Esta característica confere um alto grau

de determinismo neste tipo de comunicação. Uma vez que nos sistemas TT as atividades são realizadas periodicamente, os recursos são pré-alocados e utilizados em tempo de execução.

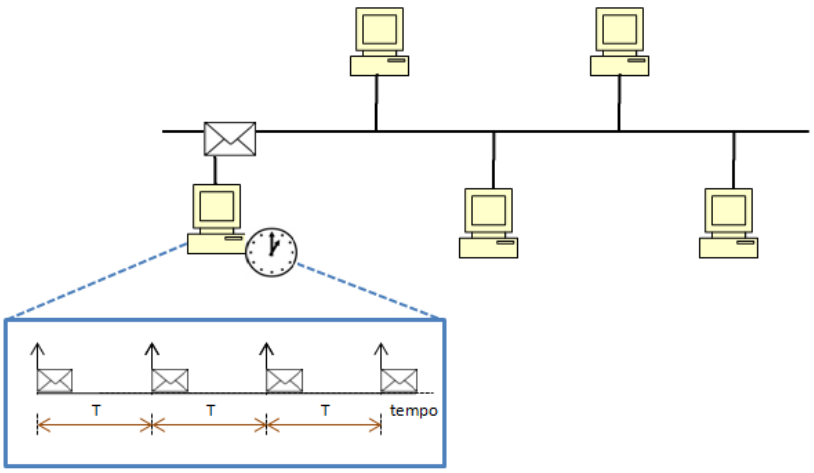


Figura 4: Comunicação Time-triggered

Nos sistemas ET, a ativação de uma comunicação está diretamente relacionado com ocorrência de eventos. Um evento pode ser definido como a alteração do estado de um determinado atributo que seja significativo para o sistema. Desta forma, os instantes em que estes eventos acontecem não são conhecidos. Estas alterações podem resultar na execução de tarefas ou envio de mensagens. (Figura 5) Comparativamente com os sistemas TT, este tipo de comunicação oferece grau de determinismo mais baixo. Relativamente à alocação de recursos, devido à imprevisibilidade resultante da ocorrência de eventos, é considerado o pior caso, ou seja, são alocados recursos sempre que for cumprido o *minimum inter-arrival time* (T_{mit}) e usado apenas quando necessário.

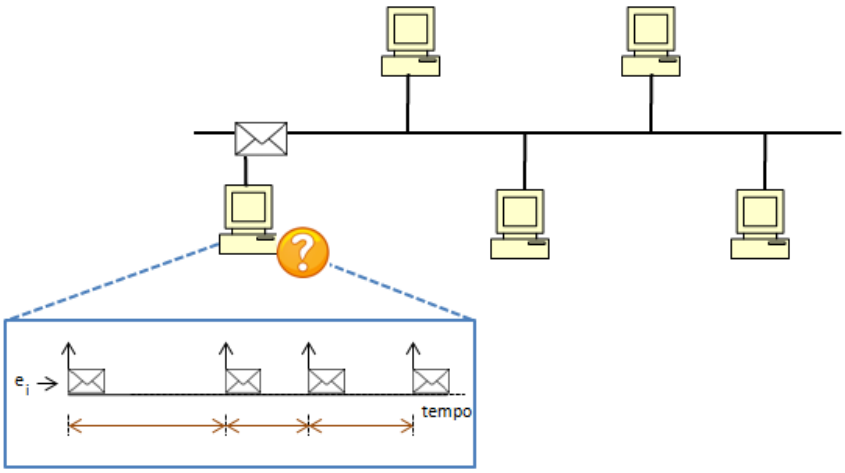


Figura 5: Comunicação Event-triggered

Pelo seu alto nível de determinismo, a abordagem TT é a mais indicada para sistemas que têm de lidar com requisitos temporais. No entanto, existem sistemas que utilizam ambos os paradigmas de comunicação, tirando vantagem de cada uma delas. Porém, para estes sistemas torna-se necessário reservar larguras de banda diferentes para os diferentes tipos de tráfego, de forma que não haja prejuízo para nenhuma das comunicações devido às suas diferentes características. [3] [5]

2.4 Escalonamento

Num sistema de tempo-real, existe um conjunto de tarefas (*Tasks*) que concorrem entre si para a obtenção de recursos para serem executadas. Muitos foram os estudos sobre políticas de escalonamento num sistema com recursos limitados, que possibilitem que todas as tarefas possam ser executadas dentro dos tempos definidos para as suas execuções, isto é, cumprindo os seus respetivos *deadlines*.

Podemos, portanto, definir tarefa como uma atividade realizada pelo sistema computacional. De acordo com as suas características, uma tarefa pode ser considerada *time-triggered* ou *event-triggered* (ver secção 2.3). A primeira classificação é atribuída quando os instantes de ativação são realizados em intervalos de tempo fixos, e está diretamente relacionada com tarefas síncronas (periódicas). Já a segunda está relacionada com tarefas assíncronas (aperiódicas ou esporádicas), onde os instantes de ativação não são conhecidos, uma vez que depende da ocorrência de eventos. As tarefas assíncronas com restrições temporais podem ser incluídas num sistema de tempo-real, uma vez que têm definido um *minimum inter-arrival time* que será, no fundo, o seu período. [3]

$$\Gamma = \{\tau_i\ (C_i, T_i, Ph_i, D_i, Pr_i), i = 1, \dots, n\} \quad (1)$$

A equação 1 pode definir um conjunto de tarefas (τ) de um sistema. Uma dada tarefa τ_i tem como propriedades o *Worst-Case Execution Time* (WCET) (C_i), o período T_i, a fase inicial (Ph_i), o *deadline* (D_i) e a prioridade (Pr_i).

Vários algoritmos de escalonamento foram criados para decidir a ordem pela qual são reservados recursos para as tarefas do sistema que se encontram em espera para serem executadas. (Figura 6)

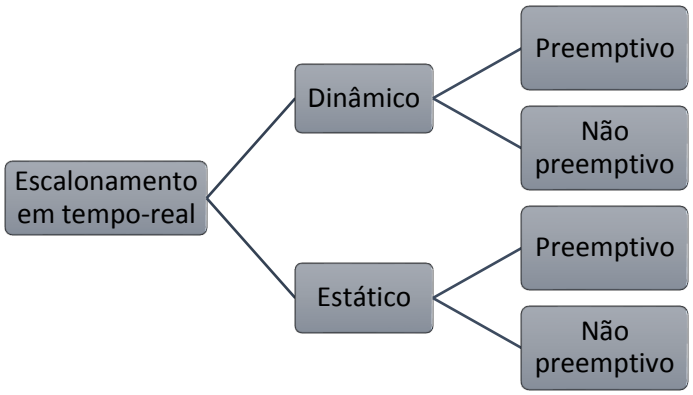


Figura 6: Algoritmos de escalonamento em hard real-time

Cada um dos algoritmos pode ser classificado de acordo com a sua natureza. Um escalonamento é dinâmico caso as suas decisões sejam realizadas em tempo de execução (*online*). Este tipo de algoritmos são considerados flexíveis, uma vez que as decisões são tomadas de forma a melhor se adaptarem às tarefas em espera. Por outro lado, existem os algoritmos de escalonamento estáticos, onde as suas decisões são tomadas em tempo de compilação, sendo, para isso, necessário um conhecimento *a priori* das características de todas as tarefas do sistema.

Os algoritmos de escalonamento podem ser classificados quanto à preempção. Por um lado, nos algoritmos preemptivos, qualquer tarefa em execução pode ser interrompida, no sentido de atribuir recursos a uma outra tarefa considerada mais prioritária. Já nos algoritmos não preemptivos, tal não se verifica. Neste caso, uma tarefa não pode ser interrompida. Se durante a execução de uma dada tarefa, uma outra mais prioritária seja ativada, esta terá de ficar em espera até que termine a execução da primeira. [5]

Tabela 2: Características de um conjunto de tarefas

Tarefa	T = D	C
τ1	4	2
τ2	6	2
τ3	11	1

Rate Monotonic

Rate Monotonic (RM) é um algoritmo de escalonamento dinâmico e preemptivo, publicado em 1973 [7], que usa prioridades fixas, ou seja, as prioridades são determinadas antes de execução e permanecem constantes. É um algoritmo adequado, e muito utilizado, no escalonamento de tarefas de tempo-real, limitado a um único processador. Este algoritmo, atribui as prioridades com base no período das tarefas, sendo mais prioritária aquela com menor período e menos prioritária aquela com maior período. Na política RM, todas as tarefas são consideradas independentes umas das outras e o *deadline* de cada uma tem valor igual ao seu respetivo período. [5]

A título de exemplo, o escalonamento das tarefas reunidas na Tabela 2, utilizando RM, seria efetuado de acordo com a Figura 7.

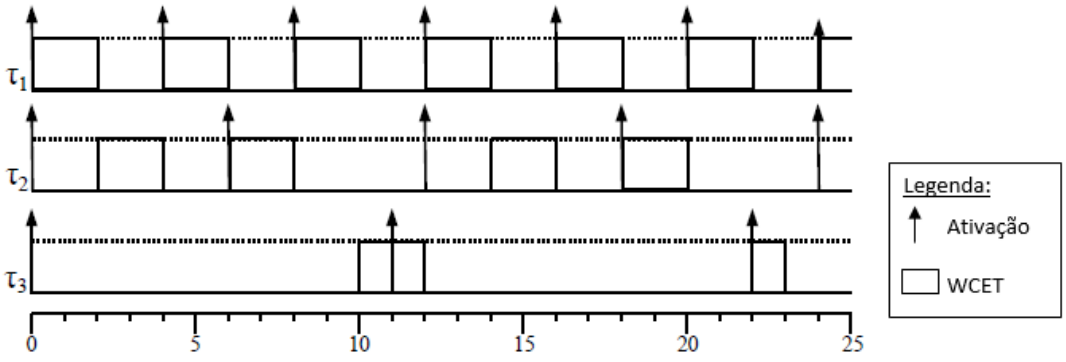


Figura 7: Escalonamento utilizando o algoritmo Rate Monotonic [3]

Earliest Deadline First

Earliest Deadline First (EDF) é um algoritmo de escalonamento dinâmico e preemptivo [3], que usa prioridades dinâmicas. Ao contrário do que se verifica no RM, as prioridades são atualizadas sempre que ocorre um novo evento. Neste algoritmo, é mais prioritária a tarefa cujo *deadline* esteja mais próximo e menos prioritária aquela cujo *deadline* esteja mais distante. Todas as tarefas são independentes entre si e apresentam *deadline* igual ao seu período, à semelhança do que acontece no RM.

A Figura 8 ilustra o escalonamento das tarefas apresentadas na Tabela 2, desta vez, utilizando o algoritmo EDF. [5]

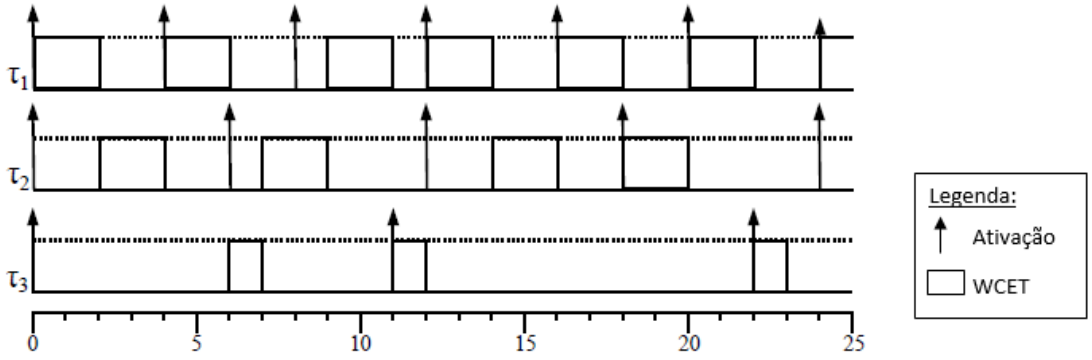


Figura 8: Escalonamento utilizando o algoritmo Earliest Deadline First [3, p. 15]

O fator de utilização de um conjunto de tarefas, utilizando as políticas de escalonamento referidas acima, é dado pela seguinte equação:

$$\mu = \sum_{i=1}^{n} \frac{C_i}{Ti} \leq n(2^{\frac{1}{n}} - 1) \qquad (2)$$

Se todas as restrições forem cumpridas, este algoritmo oferece garantias do escalonamento de todas as tarefas sem falharem o seu respetivo *deadline*, desde que se cumpra a condição. [3]

2.5 Switched Ethernet

O ano de 1973 introduziu a tecnologia *Ethernet* que foi desenvolvida para definir as camadas 1 e 2, camadas Física e *Data Link*, respetivamente, do modelo OSI. Inicialmente desenvolvida para suportar transmissões a 2.94 Mbps, foi evoluindo até aos dias de hoje conseguindo atualmente atingir velocidades até 10 Gbps.

Não só nas velocidades de transmissão a *Ethernet* sofreu modificações. Também na topologia onde era usada foi evoluindo, transitando de uma topologia inicial em barramento para uma topologia em estrela ligada através de *hubs*. No entanto, neste contexto está implícita a existência de um único domínio de colisão, o que obriga a um cuidado especial na transmissão das *frames* (ver Figura 9). É neste ponto que o protocolo *Carrier Sense Multiple Access with Collision Detection* (CSMA/CD) desempenha um papel importante através do seu mecanismo de arbitragem da transmissão de dados. Cada nó deve esperar que o barramento passe para o estado *idle* antes de enviar qualquer *frame*.

No entanto, as colisões não são totalmente evitadas, uma vez que todos os nós cumprem o mesmo algoritmo e, por isso, colisões podem ocorrer após a transmissão de uma *frame*. Aquando da ocorrência de uma colisão, um mecanismo de recuperação é ativado. Todos os nós a transmitir abortam as suas transmissões, enviam uma sequência *jam*, esperando em seguida um período aleatório de tempo antes de iniciar novamente as suas transmissões.

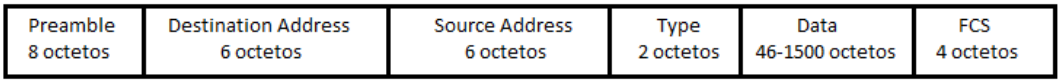


Figura 9: Estrutura de uma frame Ethernet II

A possível ocorrência de colisões num único domínio de colisão causa um problema de transferência de dados e consequentemente diminui a eficiência da própria rede. Para resolver esta problemática, foram introduzidos, nos anos 90, os *switches*. Através de um *switch*, é possível segmentar a rede criando vários domínios de colisão, um por cada segmento definido. Esta solução oferece uma melhoria significativa da performance da rede, uma vez que as colisões são totalmente evitadas. Para além disso, a utilização de *switches* torna possível que apenas o nó a quem é destinada uma *frame* a receba. Isto é conseguido devido à existência de uma tabela de endereços MAC que contém os endereços de cada um dos nós ligados ao *switch*, que pode ser definida estaticamente ou dinamicamente, atualizando a mesma à medida que as *frames* são trocadas.

Relativamente ao envio de tráfego utilizando *switches*, é importante mencionar as duas possíveis formas de transmissão. Na forma *store-and-foward*, o *switch* guarda o que recebe em memória e apenas após receber a totalidade do pacote a enviar é que inicia a sua transmissão. Já na forma *cut-through*, o *switch* não espera que a totalidade do pacote seja recebida, iniciando a sua transmissão logo que o endereço de destino seja processado. [3]

3 Flexible Time Triggered over Switched Ethernet (FTT-SE)

Nesta secção é descrito em detalhe o protocolo *Flexible Time Triggered – Switched Ethernet*. É efetuada uma breve apresentação do protocolo, uma abordagem à arquitetura do sistema, às características inerentes ao protocolo e aos modelos de escalonamento de tráfego mais utilizados.

Também é abordada, neste capítulo, a implementação do protocolo, descrevendo as principais características da mesma, bem como são apresentados alguns exemplos elucidativos.

3.1 Breve apresentação

Flexible Time Triggered – Switched Ethernet (FTT-SE), desenvolvido por Ricardo Marau, é um protocolo de comunicação em tempo-real adaptado do protocolo *FTT – Ethernet* [8]. Ambos os protocolos seguem o paradigma FTT [9], isto é, apresentam um esquema *Master/Slave*, que compreende a existência de um nó dedicado, denominado *Master*, que coordena o trafego trocado na rede pelos nós *Slaves*. A grande vantagem da existência de um nó coordenador é o facto de todo o processo de escalonamento de mensagens estar centralizado num único nó, tornando mais fácil a mudança da política de escalonamento das mensagens e os requisitos de comunicação, assim como facilita a introdução de um mecanismo de controlo de admissão,

que garanta o cumprimento dos requisitos temporais do sistema. Todas estas características oferecem um alto nível de flexibilidade ao sistema.

A comunicação nos protocolos FTT tem como base a existência de ciclos, com duração pré-fixa, que são chamados *Elementary Cycles* (ECs). O *Master* sincroniza o sistema com a transmissão de uma mensagem - *Trigger Message* (TM) - no início de cada EC, que contém no seu *payload* as mensagens a serem trocadas naquele EC, previamente escalonadas por aquele nó.

Cada *Elementary Cycle* compreende duas janelas importantes, a *Synchronous Window* (SW) e a *Asynchronous Window* (AW), que reservam, respetivamente, largura de banda para a transmissão de tráfego síncrono (periódico) e tráfego assíncrono (aperiódico). Acrescentar também a existência de largura de banda reservada antes da SW e da AW, primeiramente para as transmissões da TM e posterior processamento da mesma (*Turn-around*) por parte dos nós *Slaves* e também para a transmissão de mensagens relacionadas com o mecanismo de *signalling* do protocolo. Esta largura de banda é pré-fixa e os seus tamanhos são baseados, para o primeiro caso, no número de nós da rede, e para o segundo caso na capacidade dos nós *Slaves*. [10]

O *Master* realiza escalonamento uma vez por EC, contruindo a respetiva TM e envia a mensagem em *broadcast* para todos os restantes nós. Por sua vez, aquando da receção da TM, os nós *Slaves* processam a mesma, verificando se são os transmissores de alguma das mensagens escalonadas para aquele EC. Caso tal se verifique, o processo de envio das mensagens é iniciado.

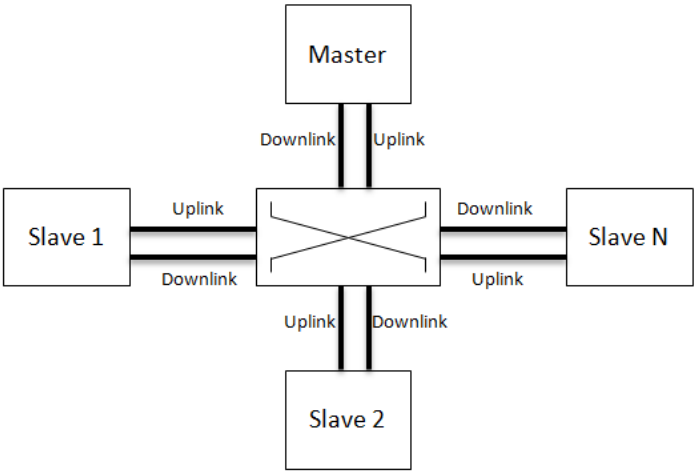


Figura 10: Ligações numa rede segmentada por um switch

O protocolo FTT-SE veio trazer algumas melhorias relativamente ao protocolo *FTT – Ethernet*. *O FTT – Ethernet* foi desenvolvido para trabalhar sobre uma rede *Ethernet* partilhada, onde existe um único domínio de colisão. Apesar da introdução do protocolo CSMA/CD, no sentido de lidar com as colisões detetadas no meio partilhado, este mecanismo revela-se não determinístico, pelo que o *Master* teria de ter em consideração esta limitação no escalonamento das mensagens.

A *Trigger Message*, além de incluir as mensagens escalonadas, também teria de definir os instantes em que as mesmas deveriam ser transmitidas, através de *offsets* específicos, que deveriam ser rigorosamente respeitados pelos *Slaves* para uma eficiente utilização da rede. Com o FTT-SE, é introduzido um *switch* no sistema e passamos a ter uma rede micro-segmentada com apenas um nó na extremidade de cada segmento, apresentando um domínio de colisão privado por cada segmento e também ligações *full-duplex* (ver Figura 10). Esta característica permite a existência de dois *links* dedicados entre cada nó e o *switch*. Um *uplink,* que liga o nó ao *switch,* e um *downlink,* que liga o *switch* ao nó, que torna possível a um nó enviar e receber mensagens ao mesmo tempo sem qualquer interferência entre as mesmas. Este protocolo permite também a existência de topologias com mais do que um *switch*, sendo as ligações entre os mesmos igualmente *full-duplex*.

Desta forma, a ausência de colisões é uma das principais vantagens deste protocolo. Consequentemente, a construção da *Trigger Message* torna-se mais simplificada, uma vez que já não se verifica necessária a definição dos instantes em que as mensagens escalonadas devem ser trocadas. Aquando da receção da TM, por parte dos nós *Slaves*, estes processa-a e caso sejam transmissores podem de imediato iniciar a transmissão, dentro da janela adequada ao tráfego a enviar, ficando a serialização das mesmas a cargo do próprio *switch*. [3]

3.2 Elementary Cycle

Tal como descrito na secção 3.1, a comunicação nos protocolos FTT é organizada em ciclos de tamanho fixo, chamados *Elementary Cycles.* É o *Master* que marca o início de cada EC sincronizando toda a rede, com o envio em *broadcast* da *Trigger Message* que posteriormente será processada por cada um dos restantes nós.

A Figura 11 ilustra a divisão das janelas que compõem cada EC. Na fase inicial do EC, é reservada a largura de banda necessária à transmissão da TM, que se denomina *Guard Window.* Seguidamente, aparece a *Turn-around Window* que permite aos nós *Slaves* terem tempo para interpretar a *Trigger Message*. O restante espaço do EC é guardado para a divisão

de tráfego que o protocolo realiza. Primeiramente, aparece a *Synchronous Window,* onde é trocado o tráfego *time-triggered,* e posteriormente a *Asynchronous Window* utilizado na troca de tráfego do tipo *event-triggered*. Contudo, o tamanho da SW pode sofrer variação, uma vez que existe um limite máximo estipulado para esta janela. Porém, o tamanho efetivo da janela corresponde, apenas, à largura de banda necessária para a transmissão de todas das mensagens síncronas escalonadas para aquele EC.

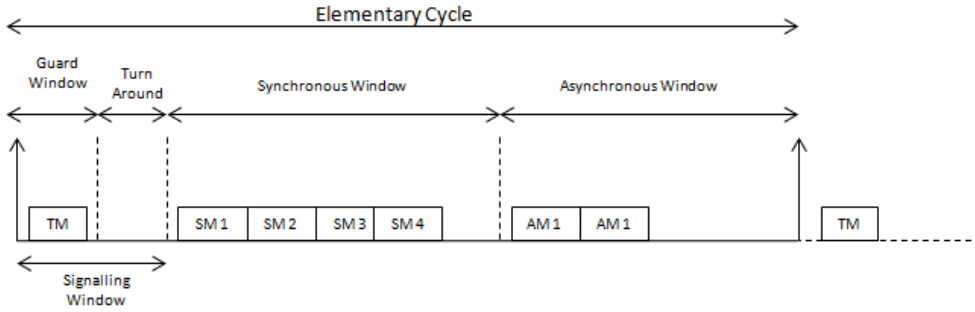


Figura 11: Estrutura do Elementary Cycle

As *streams* síncronas, utilizadas no FTT-SE, correspondem a tráfego *time-triggered,* cujo escalonamento é executado no *Master* em tempos predefinidos, de acordo com o período definido para as mesmas. Já as *streams* assíncronas correspondem a comunicações *event-triggered*, ou seja, criadas a partir da geração de eventos dentro dos nós *Slaves*, pelo que o seu escalonamento não está unicamente dependente do nó *Master*. Neste ponto, o protocolo compreende um mecanismo de sinalização destes eventos, através do envio, por parte dos *Slaves*, de s*ignalling messages*, que contêm informação sobre o estado das filas associadas às suas aplicações que produzem tráfego assíncrono, no sentido de dar a conhecer ao *Master* a existência de mensagens assíncronas pendentes.

Dentro do EC, as *Signalling Messages* são enviadas durante a *Signalling Window*, que resulta da junção do tempo reservado para a *Guard Window* e para a *Turn-around Window*. Ao receber uma *signalling message*, o Master vai processá-la e considerar a informação contida na mesma para o escalonamento do tráfego aperiódico do EC seguinte. Deste modo, o envio de qualquer mensagem assíncrona tem a duração de, pelo menos, dois *Elementary Cycles.* (ver capítulo 3.4) [3]

3.3 Arquitetura interna dos nós FTT-SE

A arquitetura interna dos nós no protocolo FTT-SE é organizada em três camadas: *Interface*, *Management* e *Core*. No entanto, existem algumas diferenças entre *Master* e *Slave* nesta matéria. A Figura 12 ilustra a organização dos vários componentes que fazem parte da arquitetura interna dos nós FTT-SE.

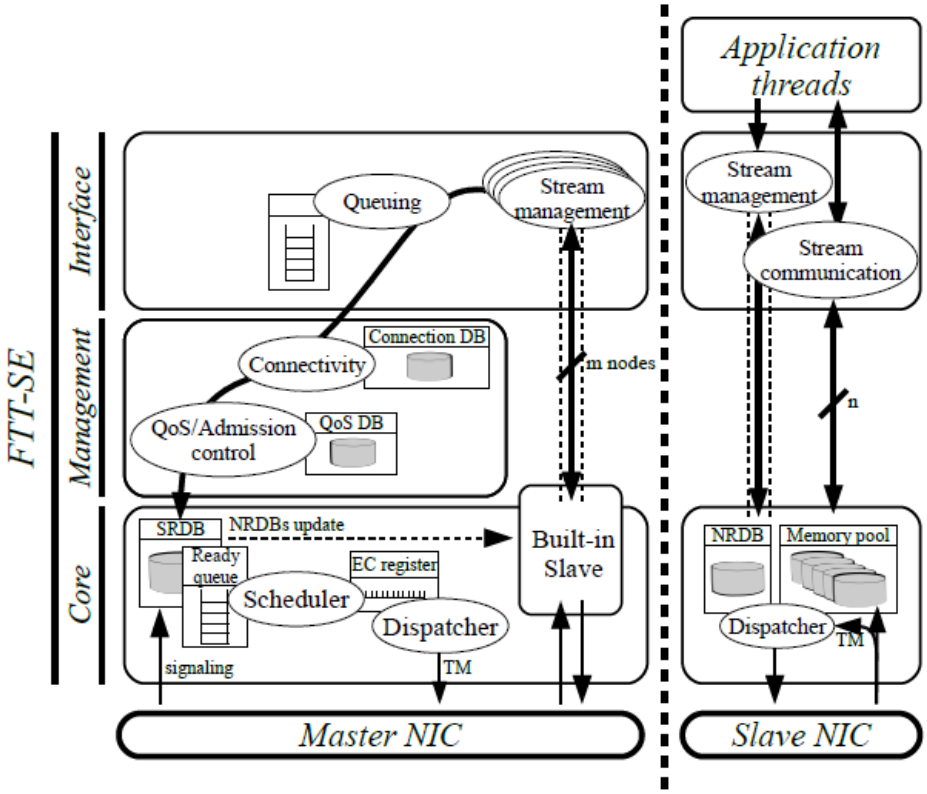


Figura 12: Componentes da arquitetura interna dos nós no FTT-SE [3]

A camada *Interface* fornece serviços de comunicação e manutenção à aplicação. A camada *Management* apenas está presente na arquitetura do *Master*. Esta camada desempenha um papel importante na Qualidade do Serviço do sistema, uma vez que, através do componente *QoS manager*, são realizados dinamicamente ajustamentos aos recursos disponibilizados para cada *stream*, tendo em consideração os requisitos das aplicações do sistema. Também é realizada, nesta camada, um controlo de admissão aquando do registo de uma nova *stream*, que apenas é permitido por este controlo se a mesma for possível de ser escalonada pelo *Master*. Por fim, a camada *Core* realiza as operações principais na comunicação. Na perspetiva do *Master*, existe uma base de dados denominada *System Requirements Database* (SRDB), onde são armazenadas informações sobre as características das *streams* registadas. Durante o processo de escalonamento efetuado no início de cada EC, a componente *Scheduler* percorre

a SRDB na procura de mensagens pendentes para serem trocadas, guardando-as numa *Ready queue*. As mensagens contidas nesta fila são escalonadas de acordo com a política de escalonamento definida no *Master* (ver capítulo 2.4), e as que poderem ser transmitidas naquele EC são registadas no *EC Register*, que será utilizado pelo *Dispatcher* na construção da *Trigger Messager.* Na perspetiva dos *Slaves,* existe igualmente uma base de dados, idêntica à encontrada no *Master*, denominada *Node Requirements Database* (NRDB). O nó *Slave*, após a receção da TM, vai processar a informação contida no mesmo, no sentido de verificar se é o produtor de alguma das mensagens escalonadas. Caso tal se verifique, o componente *Dispatcher* vai captar essa mensagem na *memory pool* e iniciar a sua transmissão. [3] [10]

3.4 Tipos de tráfego

O modelo de escalonamento realizado no *Master* da rede, tem um papel fundamental no funcionamento da mesma. Porém, existem diferenças na forma como o escalonamento é efetuado, tendo em conta a natureza das próprias mensagens. Nesta secção, são apresentadas as principais diferenças entre o tráfego síncrono e assíncrono, mencionando como as suas características influenciam o escalonamento das *streams* da rede.

Tráfego síncrono

As informações relativas às *streams* síncronas são guardadas pelo *Master* na *Synchronous Requirements Table* (SRT), integrada na sua base de dados SRDB (ver secção 3.3). É nesta tabela que o nó coordenador da rede armazena as informações sobre o tráfego síncrono da rede. O conjunto *streams* síncronas é definido por:

$$SRT = \{SM_i : SM_i = \left(C_i, D_i, T_i, O_i, Pr_i, S_i, \left\{R_i^1, \dots, R_i^{k_i}\right\}\right), i = 1, \dots, n\} \quad (3)$$

Cada *stream* é, portanto, definida pelo *Worst-Case Message Length* (WCML) C_i, correspondente ao máximo de tempo de transmissão da mensagem, o *deadline* D_i, o período T_i, o *offset* O_i, sendo estes três últimos apresentados em números inteiros correspondentes ao número de ECs, a prioridade Pr_i, o ID do nó produtor S_i e o conjunto de IDs dos nós consumidores $\left\{R_i^1, \dots, R_i^{k_i}\right\}$. [3]

É com base nas propriedades referidas que o *Master* implementa uma fila com as informações de mensagens periódicas pendentes para serem transmitidas, sendo que a construção desta fila segue a política de escalonamento especificada (RM ou EDF). Pela sua natureza *time-triggered*, o *Master* verifica em todos os ECs quais são as *streams* com mensagens pendentes e guarda informações sobre as mesmas na fila. Esta fila é usada no processo de codificação da

TM, que ao ser recebida e processada pelos nós *Slave*, permite a transmissão de mensagens pendentes, dentro da janela reservada para este tráfego, *Synchronous Window*.

Tráfego assíncrono

À semelhança do que acontece com o tráfego síncrono, as informações relativas às *streams* assíncronas também são guardadas na base de dados do *Master*, a SRDB, contudo, numa outra tabela, chamada *Asynchronous Requirements Table* (ART). O conjunto *streams* assíncronas é definido por:

$$ART = \{AM_i : AM_i = \left(C_i, Tmit_i, Pr_i, S_i, \{R_i^1, \dots, R_i^{k_i}\}\right), i = 1, \dots, n\} \quad (4)$$

A definição destas *streams* não difere muito das síncronas. No lugar do período e *dealine* é guardado o *minimum inter-arrival time* $Tmit_i$, que define o intervalo de tempo mínimo entre duas mensagens assíncronas de uma *stream* que, no fundo, terá a função de período e também é definido em números inteiros correspondentes ao número de ECs. Para além disso, não existem informações sobre *offsets* neste tipo de tráfego. [3]

A maior diferença relativamente ao tráfego síncrono, é a forma como as *streams* assíncronas são ativadas, uma vez que estas são geradas a partir de eventos (*event-triggered*) nos nós produtores. Desta forma, também o modelo de escalonamento para o tráfego assíncrono é diferente e compreende a existência de um mecanismo fundamental, chamado mecanismo de *signalling*. Este procedimento tira vantagem das ligações *full-duplex*, devido à existência de *switches Ethernet*, havendo a possibilidade de enviar e receber mensagens em simultâneo, sem colisões entre as mesmas. Os nós *Slaves* são sincronizados através da *Trigger Message* enviadas pelo nó *Master* no início de cada EC. A esta sincronização segue-se o envio, por parte dos nós *Slaves*, de uma mensagem, chamada *Signalling Message*, contendo informação relativamente ao estado das suas filas de mensagens assíncronas. (ver Figura 13)

É através deste mecanismo que o *Master* obtém conhecimento de mensagens assíncronas pendentes. Após a receção de uma *Signalling Message* de um determinado nó, e esta informar da existência de mensagens pendentes, o *Master* regista numa fila, à semelhança do que realiza no tráfego síncrono, as informações relativas a mensagens pendentes. Estas informações serão usadas para o escalonamento do EC seguinte e incluídas na TM, no caso de as mensagens poderem ser transmitidas naquele EC. Deste modo, o tempo de resposta de uma mensagem síncrona nunca será inferior a dois ECs. [3]

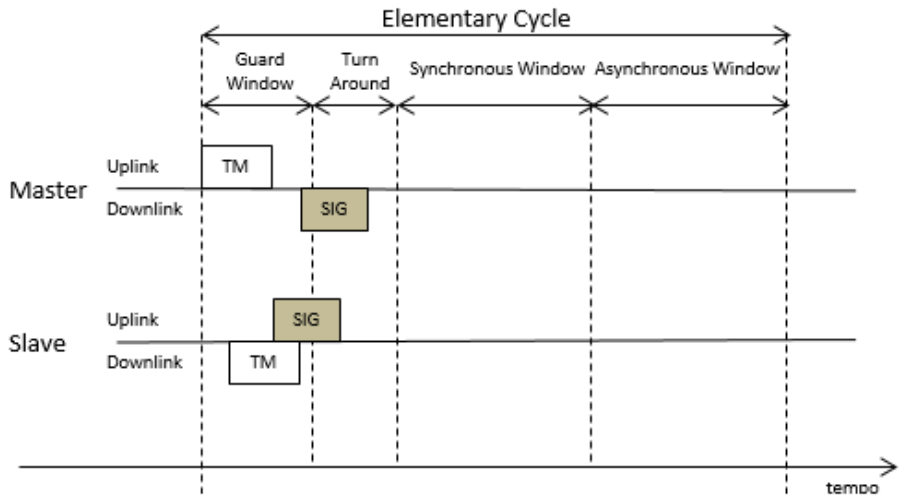


Figura 13: Mecanismo de signalling

3.5 Construção de um Elementary Cycle

O escalonamento realizado internamente no nó *Master* da rede para todos os *Elementary Cycles* é um processo determinante no protocolo FTT-SE, uma vez que é neste ponto que são decididas quais as mensagens que podem ser trocadas dentro de um determinado EC. Portanto, torna-se fundamental entender muito bem como todo este processo é desencadeado, de forma a entender o funcionamento deste protocolo.

Como já abordado na secção 3.5, existe uma clara divisão entre tipos de tráfego (síncrono e assíncrono) dentro de um EC, havendo duas janelas que reservam largura de banda para cada um dos tipos (SW e AW, respetivamente). O modo como o nó *Master* decide quais as mensagens que podem ser transferidas dentro de cada janela é um pouco complexo e deve merecer particular atenção. Internamente, o *Master* tem uma base de dados, SRDB (ver secção 3.3), onde são registadas as *streams* criadas pelos nós *Slaves* e cada uma com as suas propriedades. Nesta componente, também é realizada uma divisão entre tipos de tráfego, havendo uma tabela que regista as *streams* síncronas, *Synchronous Requirements Table* (SRT), e outra que regista as assíncronas, *Asynchronous Requirements Table* (ART).

Um dado importante a relembrar é a existência de ligações *full-duplex* entre os nós da rede e o *switch* a que se encontram ligados, resultando em duas ligações, chamadas *Uplink* e *Downlink,* em cada segmento da rede. Isto permite que simultaneamente um nó possa estar a enviar e receber dados. No caso de existir mais do que um *switch* na rede, as ligações entre os mesmos também são *full-duplex*.

Existem duas abordagens diferentes no escalonamento de cada tipo de tráfego. Começando pelo tráfego síncrono, este processo inicia-se pela identificação, por parte do *Master*, das mensagens que estão pendentes para serem trocadas. Pela sua natureza, cada *stream* síncrona só é ativada periodicamente. Portanto, as *streams* são verificadas, uma a uma, no sentido de verificar se o seu escalonamento não viola as suas próprias propriedades (período). Caso tal o *Master* verifique que uma *stream* se encontra pendente, a mesma é adicionada a uma fila de mensagens pendentes, denominada *Ready queue*.

Quanto ao tráfego assíncrono a abordagem é efetuada de forma diferente devido às suas características aperiódicas, diretamente relacionadas com a geração de eventos. Quando um evento é gerado dentro de um nó *Slave*, o mesmo é informado ao *Master* por meio de uma *Signalling Message*, enviada pelo *Slave* onde esse evento foi gerado, durante a *Signalling Window.* O *Master* quando recebe essa mensagem, adiciona a *stream* associada a esse evento a uma outra fila de mensagens pendentes (*Ready queue*) para que seja escalonada quando possível (ver Figura 14).

Quando o tamanho de uma *stream* ultrapassa o valor do MTU definido para a rede *Ethernet*, a mesma tem de sofrer fragmentação em várias mensagens de modo a ser respeitado o MTU. Cada fragmento criado é adicionado à respetiva *Ready queue* herdando as características da *stream* que o originou.

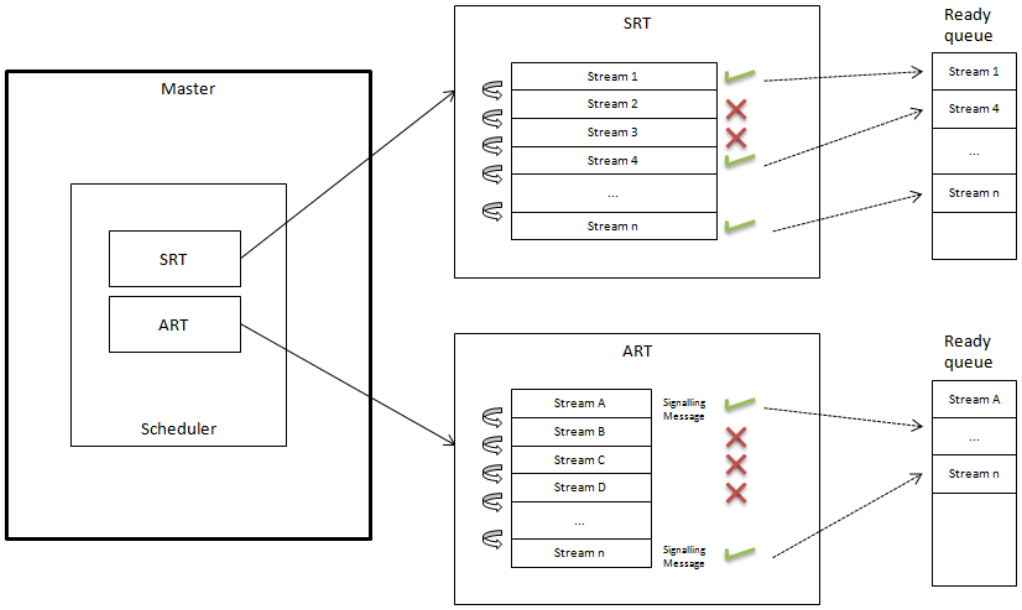


Figura 14: Armazenamento de mensagens pendentes em Ready queues

O início de cada EC é marcado pelo envio da *Trigger Message* por parte do *Master*, sendo a sua construção baseada no escalonamento das mensagens pendentes e guardadas nas *Ready queues*. Deste modo, é neste ponto que o processo de escalonamento é efetivamente iniciado. O primeiro passo é ordenar cada *Ready queue* de acordo com a política de escalonamento utilizada, *Rate Monotonic* (Figura 15) ou *Earliest Deadline First*. Existe uma pequena diferença na ordenação da fila das mensagens assíncronas, uma vez que existem 3 subtipos de tráfego assíncrono e são atribuídas diferentes prioridades a cada um deles. Por ordem decrescente de prioridade: *Hard real-time, Soft real-time* e *Best effort*. Desta forma, a primeira regra de ordenação na *Ready queue* de mensagens assíncronas está relacionada com a prioridade do subtipo de tráfego.

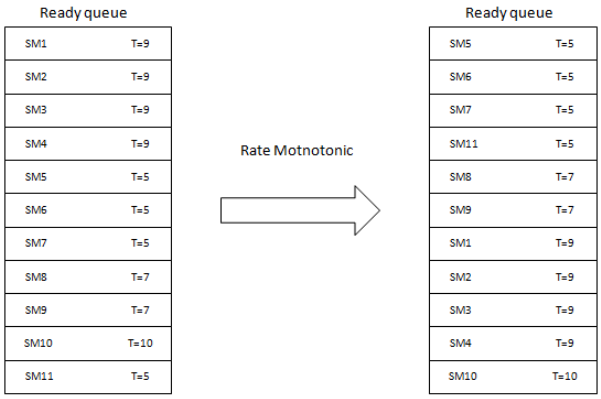


Figura 15: Ordenação de Ready queue com mensagens síncronas de acordo com o modelo Rate Monotonic

No passo seguinte, é analisada cada mensagem pendente, uma a uma, no sentido de verificar se existe largura de banda suficiente para a transmissão da mesma. No caso do tráfego síncrono, as suas transmissões não podem ultrapassar o limite máximo (LSW) definido para a sua janela. Já no caso assíncrono, não pode ser ultrapassada a largura de banda reservada para o seu tipo de tráfego, nem terminar a transmissão fora do *Elementary Cycle*.

O percurso de uma mensagem numa transmissão envolve obrigatoriamente pelo menos duas ligações, iniciando no *Uplink* que liga o nó produtor ao primeiro *switch* do percurso e terminando no *Downlink* que liga o último *switch* do percurso ao nó consumidor. Sendo assim, é por esta ordem que a análise de cada mensagem é realizada. No caso da topologia de rede compreender mais do que um *switch*, as ligações entre os mesmos também têm de ser consideradas.

Primeiramente, é escalonado o tráfego síncrono e só após terminar esse processo é que o *Master* realiza o escalonamento do tráfego assíncrono.

Tomando como exemplo o primeiro processo, o *Master* tem de analisar todas as mensagens guardadas na *Ready* queue, começando da primeira até à última, isto é, da mais prioritária até à menos prioritária, de acordo com a ordenação realizada previamente. Analisando uma mensagem, o referido nó avalia se o tempo de transmissão da totalidade dessa mensagem no seu respetivo *Uplink* não ultrapassa o limite da sua janela respetiva. No caso de o percurso compreender mais do que um *switch*, terá de haver a mesma verificação nas ligações entre os *switches.* Se esse limite não for ultrapassado o mesmo procedimento é realizado para o respetivo *Downlink*. É importante referir que o escalonador deve considerar, quer nas ligações entre os *switches* quer nos *Downlink*, o instante em que a transmissão na ligação imediatamente anterior terminou. Se, após todas as análises, se verificar que cumpre a referida regra, a mensagem é registada num outro componente da arquitetura do *Master*, o *EC Register* (Figura 16) e é eliminada da fila de mensagens pendentes. Para além disso, são também são registados os períodos de tempo considerados para a transmissão da mensagem em cada ligação, de forma que, na análise às restantes mensagens, estes períodos não sejam considerados disponíveis para transmissão.

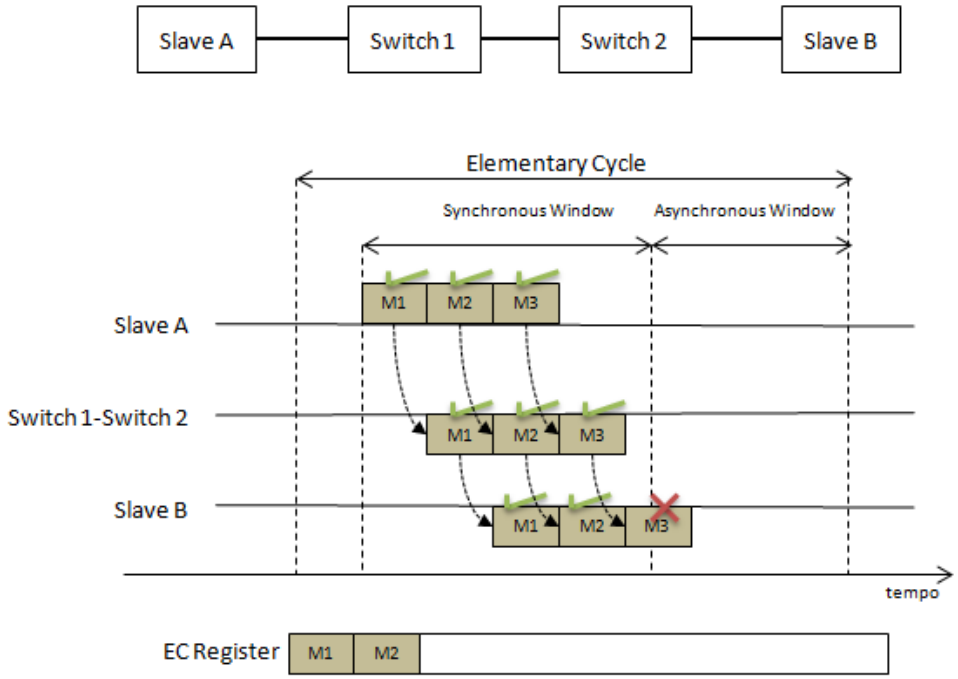


Figura 16: Escalonamento de mensagens síncronas e registo no EC Register

No entanto, na análise de uma determinada mensagem basta que a transmissão numa das duas ligações que irá percorrer ultrapasse o limite estipulado (LSW) para que a transmissão seja abortada. Neste caso, a mensagem é mantida como pendente na *Ready queue* para que no EC seguinte volte a ser analisada. Nenhum período tempo é, então, registado nas ligações envolvidas, uma vez que a transmissão daquela mensagem não será realizada. No entanto, a análise às mensagens da *Ready queue* não está terminada, sendo este procedimento aplicado até à última mensagem contida naquela fila.

No que diz respeito ao escalonamento do tráfego assíncrono, o procedimento é muito semelhante. A única diferença está no limite a considerar nas análises realizadas. Neste caso, deve ser considerada a largura de banda reservada para este tipo de tráfego.

Por fim, resta apenas construir a *Trigger Message*, a mensagem que irá informar os restantes nós da rede do escalonamento para aquele EC, sendo a mesma enviada no início do EC, marcando o seu início. Esta mensagem vai incorporar no seu *payload* todos os registos do *EC Register* que foram sendo adicionados ao longo de todo o processo de escalonamento do *Master.* [3]

3.6 Fork-Join Parallel Distributed Real-time Tasks no protocolo FTT-SE

Atualmente, muitas aplicações embebidas distribuídas de tempo-real têm de lidar com restrições temporais, que por vezes, não são possíveis de cumprir devido a limitações como por exemplo, a existência de um único processador. Uma possível solução é a distribuição do trabalho por nós remotos, ligados por uma rede de tempo-real, realizando processamento em paralelo.

Num sistema distribuído de tempo-real existem aplicações distribuídas que compreendem a execução de um conjunto de tarefas *Fork-Join Parallel/Distributed* (tarefas P/D). Neste paradigma, uma tarefa P/D inicia através da execução sequencial de uma *thread* principal que seguidamente é dividida, através da operação de *fork* (*D-Fork*), e executada em paralelo quer no nó local, quer nos nós remotos. Terminada a execução em paralelo em todos os nós, há ainda que realizar a operação de *join* (*D-Join*) no nó local, que agrega o resultado das execuções na *thread* principal. Estas operações são semelhantes às realizadas em sistemas multiprocessador. O procedimento pode ser repetido várias vezes. (Figura 17) [11]

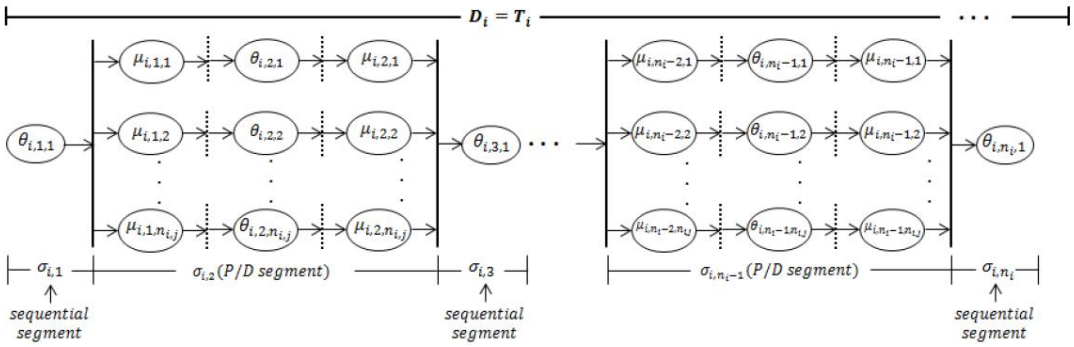


Figura 17: Paradigma fork-join parallel distributed real-time tasks [4].

Para o escalonamento de tarefas P/D, devem ser consideradas todas as interações entre todas as *threads* em execução nos vários nós e as mensagens que são trocadas entre as mesmas. Para se ter garantias de que uma tarefa é sempre executada antes de terminar o seu *deadline* é fundamental calcular o *Worst-Case Response Time* (WCRT) da mesma. Este cálculo é obtido através de duas etapas: o cálculo do WCRT das mensagens que são trocadas na rede e o cálculo do WCRT da execução das *threads* no processador.

3.6.1 *Worst-Case Response Time* na rede FTT-SE

O *Worst-Case Response Time* das mensagens na rede FTT-SE é obtido calculando os atrasos numa transmissão *end-to-end* (apenas num sentido) das mensagens existentes neste protocolo. Esta abordagem compreende a análise de dois pontos fundamentais: *resquest bound function* (*rbf*) e *supply bound function* (*sbf*). [12]

O *resquest bound function* $\boldsymbol{rbf_i(t)}$ representa os requisitos máximos para a transmissão de uma mensagem (m_i), considerando todos os atrasos relacionados com a transmissão de outras mensagens com maior prioridade, num intervalo [0,t]. O cálculo do $rbf_i(t)$ é dado por:

$$rbf_i(t) = C_i + sn_i \times (SFD_i + \Delta) + Wl_i(t) + Wr_i(t) \qquad (5)$$

onde C_i representa o tempo máximo para a transmissão de m_i, sn_i é o número de *switches* que m_i atravessa no seu percurso, desde o nó de origem até o nó de destino, $SFD_i + \Delta$ representa os atrasos relacionados com a computação do *switch*, $Wl_i(t)$ é o "*Shared Link Delay*" e $Wr_i(t)$ é o "*Remote Link Delay*". [12]

O *Shared Link Delay* define o tempo máximo que a mensagem em análise fica bloqueada por outras mensagens com prioridade mais alta, que partilham pelo menos um *link* com m_i. O cálculo do *Shared Link Delay* é dado por:

$$Wl_i(t) = \sum_{\substack{\forall j \in [1,n], j \neq i \\ \wedge R_j \cap R_i \neq 0 \\ \wedge m_j \in hp(m_i) \\ \wedge m_j \in WT(m_i)}} \left\lceil \frac{t}{T_j} \right\rceil (C_j) + Is_i(t) \tag{6}$$

onde R_i é o conjunto de *switches* percorridos na transmissão de m_i, $hp(m_i)$ é o conjunto de mensagens com mais prioridade que m_i, $WT(m_i)$ é o conjunto de mensagens do mesmo tipo de tráfego que m_i e $Is_i(t)$ representa o efeito dos atrasos na passagem das mensagens pelos *switches*. Para o cálculo de $Is_i(t)$, é considerado um vetor $G_i(t)$ que armazena os atrasos num *switch* correspondente a cada mensagem que é considerada para o cálculo do *Shared Link Delay*. No entanto, como estamos a analisar o pior caso, no instante *t* apenas devem ser considerados os primeiros $z(t)$ elementos de G_i^{sort}, o vetor ordenado por ordem decrescente. $z(t)$ representa o número de ECs num intervalo [0, *t*] e é dado por: $z(t) = \left\lceil \frac{t}{EC} \right\rceil$. No caso de $z(t)$ ser superior o número de elementos inseridos em G_i^{sort}, é atribuído o valor zero aos elementos excedentes. O cálculo do efeito dos atrasos resultantes da passagem das mensagens pelos *switches* é dado por: [12]

$$Is_i(t) = \sum_{l=1}^{z(t)} G_i^{sort}(t)[l] \tag{7}$$

O *Remote Link Delay* define o tempo máximo que a mensagem em análise fica bloqueada indiretamente por outras mensagens com prioridade mais alta, mas que não partilham *links* com m_i. Em [13] é dado um exemplo elucidativo desta interferência. São consideradas três mensagens $\{m_1, m_2, m_3\}$, sendo que a primeira é a que tem mais prioridade e a última a que tem menos prioridade. m_1 partilha um *link* com m_2 mas não partilha com m_3, e m_2 partilha um *link* com m_3. Devido à partilha de um *link*, m_1 pode atrasar a transmissão de m_2 e daí resultar, de forma indireta, o adiamento da transmissão de m_3 para o EC seguinte. Para o cálculo do *Remote Link Delay*, são consideradas todas as mensagens consideradas no âmbito do *Shared Link Delay*, sendo excluídas aquelas que efetivamente foram utilizadas para o cálculo do mesmo. Desta forma, o *Remote Link Delay* é dado por:

$$Wr_i(t) = \sum_{\substack{\forall k,j \in [1,n], k \neq j \neq i \\ \wedge R_k \cap R_j \neq 0 \wedge R_k \cap R_i = 0 \wedge R_j \cap R_i \neq 0 \\ \wedge m_k \in hp(m_j) \\ \wedge m_k \in WT(j)}} \left\lceil \frac{t}{T_k} \right\rceil (C_k) \tag{8}$$

O *supply bound function* $\boldsymbol{sbf_i(t)}$ representa o mínimo de capacidade que a rede oferece no intervalo de tempo [0,t]. O seu cálculo é dado por:

$$sfb(t) = \left(\frac{BW - I}{EC}\right) \times t \tag{9}$$

onde *BW* representa a largura de banda reservada para a janela correspondente ao tipo de tráfego da mensagem em análise e *I* representa o tempo inativo da janela.

O *Worst-Case Response Time* da transmissão da mensagem m_i é obtido através da determinação do instante t^* que é obtido pela comparação entre os resultados dos cálculos do *request bound function* e *supply bound function*:

$$t^* = \min(t > 0) : sbf(t) \geq rbf(t) \tag{10}$$

Devido ao desconhecimento relativamente ao instante exato em que as mensagens são transmitidas num EC, o WCRT é dado em número de ECs. Para uma mensagem síncrona, utilizando a abordagem escrita, o WCRT da transmissão da mesma é dado por: [12]

$$WCRT_i = \left\lceil \frac{t^*}{EC} \right\rceil \tag{11}$$

Relativamente a mensagens assíncronas, para além da abordagem referida há que considerar o atraso provocado pelo cumprimento do mecanismo de *signalling*, necessário na transmissão deste tipo de tráfego. Deste modo, o WCRT é obtido cumprindo todos os cálculos apresentados, somando apenas ao resultado 2 ECs.

3.6.2 *Worst-Case Response Time* da execução no processador

Para uma análise completa ao WCRT de uma tarefa P/D τ_i, torna-se também necessário calcular o WCRT da execução das *threads* P/D. Estando a considerar tarefas preemptivas com prioridades fixas, o *Worst-Case Execution Time* (WCET) de uma *thread* P/D θ_i é dado pela seguinte equação recursiva:

$$r_{\theta_i}^{n+1} = C_i + \sum_{\theta_i \in hp(\theta_i)} \left\lceil \frac{r_{\theta_i}^n}{T_j} \right\rceil \times C_j \tag{12}$$

onde r_{θ_i} representa o WCET de θ_i, $hp(\theta_i)$ representa o conjunto de todas as *threads* com mais prioridade do que θ_i que executam no mesmo processador. As iterações desta equação recursiva começam com $r_{\theta_i}^1 = \theta_i$ e terminam quando for cumprida a seguinte igualdade: $r_{\theta_i}^{n+1} = r_{\theta_i}^n$ sendo este o valor do WCET da *thread* θ_i. [4]

3.7 Implementação do protocolo FTT-SE

Neste capítulo é realizada uma abordagem a alto nível à implementação real do protocolo FTT-SE, identificando as suas principais características, o propósito para o projeto e as limitações encontradas durante a sua utilização.

A utilização desta implementação não teve relação direta com este projeto, mas justifica-se pelo trabalho em conjunto realizado no âmbito de outro projeto, com o colega de curso Roberto Duarte. Esta fase do projeto teve particular interesse no sentido de ser possível analisar, por intermédio da ferramenta *Wireshark*, todos os pacotes trocados entre os vários nós da rede, a forma como estavam construídos e os instantes em que os mesmos eram transmitidos.

A implementação real do FTT-SE, da autoria de Ricardo Marau, Paulo Pedreiras e Luís Almeida, consiste numa biblioteca para *Linux* desenvolvida em linguagem C, que pode ser adicionada a aplicações para gerir a transmissão de dados de acordo com o protocolo referido, e pode ser obtida em [14]. Apresenta uma arquitetura modular, onde se podem verificar a implementação de cada um dos módulos da arquitetura interna dos nós FTT: *Interface*, *Management* e Core. (capítulo 3.3) A Figura 18 resume os vários componentes da implementação real.

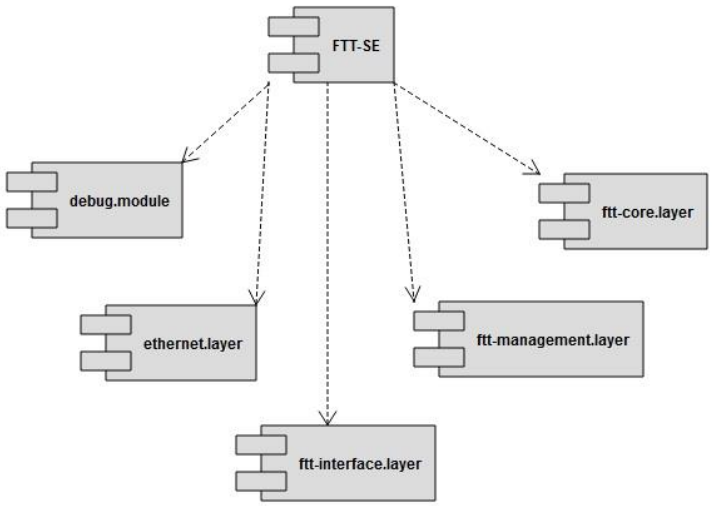


Figura 18: Componentes da implementação real do FTT-SE

Para a utilização desta biblioteca, é necessário executar uma aplicação que vai correr na máquina que vai agir como *Master* e também executar aplicações para agirem como *Slaves* nos respetivos nós. A Figura 19 apresenta um diagrama de instalação das aplicações acima referidas, num cenário que envolve um *Master* e três *Slaves*.

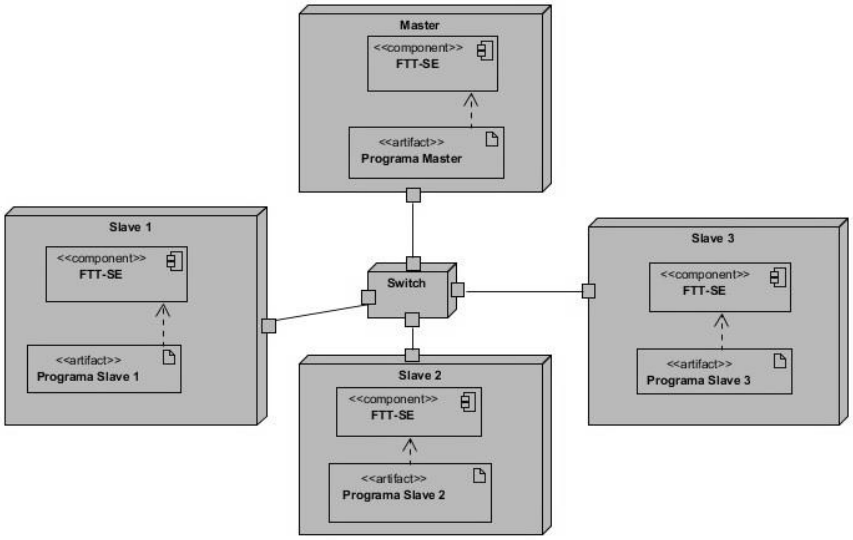


Figura 19: Diagrama de instalação de aplicações para 3 nos e 1 master FTT-SE

Entre todas as funções existentes nesta implementação, pode-se destacar as seguintes:

- `ftt_master_start_up()`: Inicia a execução do *Master*.
- `ftt_master_close_up()`: Termina a execução do *Master*.
- `ftt_slave_start_up()`: Inicia a execução do *Slave*.
- `ftt_slave_close_up()`: Termina a execução do *Slave*.
- `S_FTT_INTERFACE_L_init_block()`: Bloqueia a execução do *Slave* até todas as inicializações terem sido efetuadas, sendo a sua chamada obrigatória antes do uso de outras funções.
- `S_FTT_INTERFACE_L_var_add_load`: Faz o registo de uma nova *stream*.
- `S_FTT_INTERFACE_L_attach_tx()`: Regista o programa como produtor de uma *stream*.
- `S_FTT_INTERFACE_L_attach_rx()`: Regista o programa como consumidor de uma *stream*.
- `S_FTT_INTERFACE_L_dettach()`: Cancela o registo do programa como consumidor ou produtor de uma *stream*.
- `S_FTT_INTERFACE_L_bind()`: Liga o programa a uma *stream* existente.
- `S_FTT_INTERFACE_L_unbind()`: Remove ligação do programa a uma *stream* existente.

- `S_FTT_INTERFACE_L_pre_send()`: Reserva memória para preparar a transmissão de uma *stream*.
- `S_FTT_INTERFACE_L_send()`: Inicia a transmissão de uma *stream*.
- `S_FTT_INTERFACE_L_pre_receive()`: Reserva memoria para receber uma *stream*.
- `S_FTT_INTERFACE_L_recieve()`: Inicia a receção de uma *stream*.

Problemas e limitações detetados

Durante o estudo da implementação real do FTT-SE foram detetados alguns problemas e limitações na sua utilização. Inicialmente, foi identificado um problema relativamente à transmissão de tráfego síncrono, uma vez que o mesmo não era transmitido. Após reportar o problema ao autor da implementação, Ricardo Marau, o problema foi corrigido sendo enviada uma nova versão da implementação.

Para além disso, algumas limitações foram sendo identificadas. A implementação não consegue suportar a utilização de *Elementary Cycles* de duração reduzida, sendo que o mínimo que foi possível utilizar sem existência de falhas, foram ECs de 8 milissegundos.

Por fim, aquando da utilização do protocolo com o paradigma *Parallel/Distributed*, o mesmo só foi possível utilizando *streams* assíncronas, uma vez que a implementação não suporta esta integração com *streams* síncronas, devido ao facto de não existir uma sincronização entre as aplicações do nó local e as dos nós remotos.

4 Network Simulator (NS-3)

Nesta secção é apresentado de forma genérica o NS-3, uma parte fundamental deste trabalho, uma vez que foi neste simulador que se realizou a implementação do protocolo FTT-SE. Desta forma, esta secção vai ser iniciada com uma breve descrição do simulador, seguido da apresentação da arquitetura do mesmo. Por fim, serão enumerados e explicados os passos fundamentais para a configuração de uma simulação no NS-3 e uma abordagem à integração das classes desenvolvidas neste projeto com as classes existentes no simulador.

4.1 Descrição genérica

O *Network Simulator* (NS), atualmente na sua terceira versão, o NS-3, é um simulador de redes bastante utilizado no âmbito e de investigação. É um projeto *open source* e gratuito que pode ser executado nos sistemas operativos *Linux*, *Mac OS X* e *Windows* (através de *Cygwin*), desenvolvido essencialmente em C++ mas também com possibilidade de executar *scripts* em *Python*. O seu *design* foi desenvolvido em módulos, tornando mais fácil e rápida a implementação de novos protocolos ou alteração de classes já existentes. Os investigadores partilham muitas vezes o seu *software* ajudando na adição de novos módulos, correção de erros e mostrando os resultados das suas experiências. Sendo a contribuição o espírito inerente à utilização do simulador, revela-se, portanto, fulcral na evolução do mesmo.

O NS-3 tem como grandes vantagens permitir o estudo de muitas tecnologias existentes de uma forma rápida, eficaz, com um alto nível de realismo e a um baixo custo. Desta forma, a comunidade associada não necessita de investir em tecnologias reais para realizar os seus estudos e experiências. As suas simulações são baseadas em eventos discretos, isto é, cada evento é executado até ao fim (por exemplo, o envio de um pacote), sendo ordenada de imediato (pelo escalonador de eventos do simulador) a execução do próximo evento definido. Para além disso, possui mecanismos de *tracing* para obter facilmente mensagens de registo sobre o estado das simulações efetuadas. Também permite visualizar animações das simulações através de *NetAnim* e interação com sistemas reais.

Por fim, torna-se relevante mencionar a excelente documentação que é disponibilizada, usando *Doxygen*, as APIs do simulador, sendo apresentadas de forma detalhada descrições sobre os métodos e atributos de cada classe do NS-3.

4.2 Arquitetura

O NS-3 tem uma arquitetura baseada em módulos. Devido ao *design* implementado orientado a objetos e ao polimorfismo das suas classes, torna-se possível de uma forma mais prática a extensão de novos módulos relacionados com novas tecnologias e até mesmo a correção do código já existente.

A Figura 20 reúne os elementos dos principais módulos do NS-3, ilustrando a forma como estes se encontram organizados. Os módulos *core* e *network* implementam os componentes mais genéricos que podem ser utilizados em qualquer configuração de rede. Já os módulos que se encontram acima destes no esquema apresentado implementam os componentes específicos da rede da simulação. A título de exemplo, o módulo *Internet* implementa os protocolos Arp, IPv4, IPv6, TCP e UDP, no módulo *Applications* encontram-se aplicações que geram tráfego como por exemplo `UdpClient`, `UdpServer`, `UdpEchoClient`, `UdpEchoServer`, `OnOffApplication` ou `PacketSink` e nos *Devices* são implementados `Csma`, `Wifi` *e* `Point-to-Point`. O NS-*3* providencia também *helpers* específicos de cada módulo que facilita a configuração e a utilização no próprio módulo. Por fim, e também transversal a todos os módulos, o simulador fornece ferramentas para testes e validação de *software* desenvolvido.

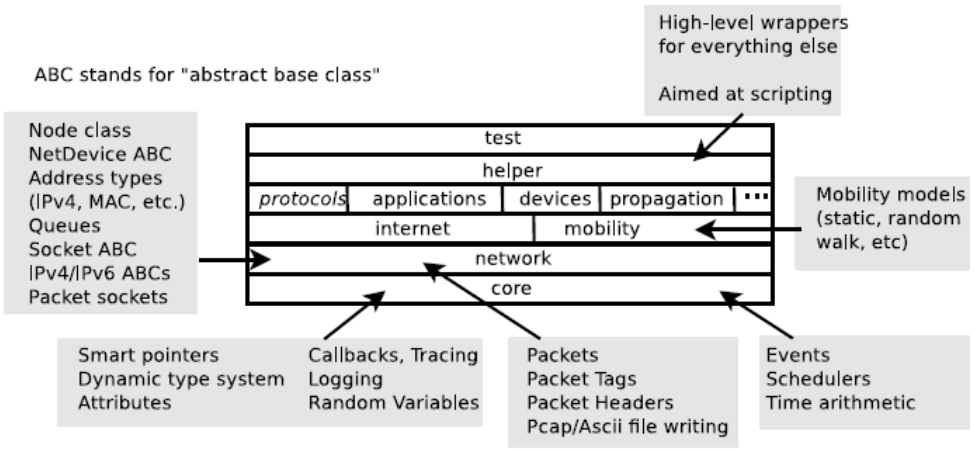


Figura 20: Organização dos módulos do NS-3 [15]

Numa simulação existem elementos que são considerados fundamentais. São eles os *Packets*, os *Nodes*, as *Applications*, os *Sockets*, os *NetDevices* e os *Channels*. (ver Figura 21)

Os *Packets* representam unidades de informação que são enviadas e recebidas pelas aplicações. Compreendem um *buffer* de *bytes* que espelha um pacote real de um determinado protocolo. Os *Nodes* representam os dispositivos computacionais que pertencem à rede. Cada nó contém uma lista de aplicações, uma pilha protocolar e uma lista de *NetDevices* que pertencem ao mesmo. Opcionalmente também é possível adicionar mobilidade ao nó através do módulo de *Mobility*.

As *Applications* são geradores de tráfego que são executados dentro dos nós. O NS-3 tem a classe abstrata `Application` que tem várias implementações tendo em conta o tipo de tráfego pretendido. O trafego gerado é enviado para a pilha protocolar através de um `Socket` adequado ao mesmo. Por fim, como já referido acima, um nó compreende a existência de *NetDevices*, que são a representação das interfaces físicas dos nós. Tal como nas aplicações, o NS-3 tem várias implementações da classe abstrata `NetDevice`, de acordo com a tecnologia pretendida (por exemplo, *Ethernet*). Cada interface contém um apontador para o nó respetivo, um endereço de MAC, um MTU, um nome e um estado, e encontra-se associada a um `Channel` do mesmo tipo, como por exemplo, um `CsmaNetDevice` tem de ser associado a um `CsmaChannel`.

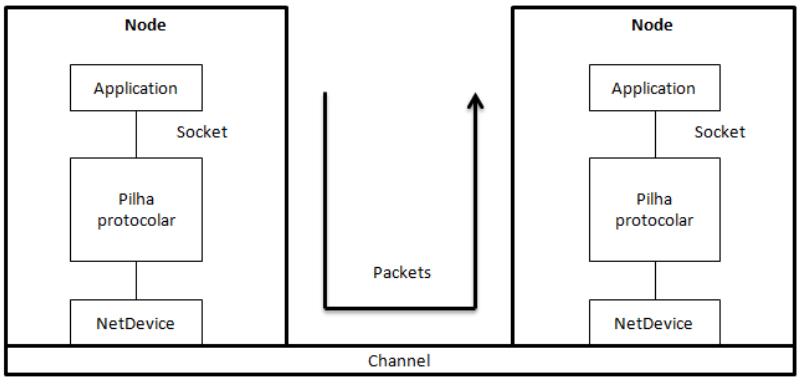


Figura 21: Troca de tráfego entre nós

4.3 Criação de simulações em NS-3

Antes de iniciar uma simulação no NS-3, é necessário escrever um *script* com todos os elementos necessários à realização da mesma. Desta forma, a escrita de um *scprit* compreende vários passos importantes gerais, a seguir enumerados:

- Incluir os módulos necessários para a simulação
- Declarar o *namespace ns3*
- Ativar mensagens de *logging*, caso pretendido
- Iniciar a função principal *main*
- Criar os nós da rede
- Criar os *NetDevices* em cada nó
- Associar um *Channel* a cada *NetDevice*
- Instalar uma pilha protocolar em cada nó
- Atribuir endereços às interfaces de rede
- Instalar aplicações em cada nó e definir os tempos de início e fim das mesmas
- Ativar mecanismo de *tracing*
- Executar a simulação

O NS-3 providencia um tutorial que auxilia os utilizadores que estão a dar os primeiros passos, explicando os conceitos básicos na realização de simulações através de alguns exemplos elucidativos. O primeiro exemplo (*first.cc)*, cujo código é apresentado na Figura 22, define uma simulação onde é trocado tráfego entre dois nós com ligação ponto-a-ponto.

```
#include "ns3/core-module.h"
#include "ns3/network-module.h"
#include "ns3/internet-module.h"
#include "ns3/point-to-point-module.h"
#include "ns3/applications-module.h"

using namespace ns3;

NS_LOG_COMPONENT_DEFINE ("FirstScriptExample");

int
main (int argc, char *argv[])
{
  Time::SetResolution (Time::NS);
  LogComponentEnable ("UdpEchoClientApplication", LOG_LEVEL_INFO);
  LogComponentEnable ("UdpEchoServerApplication", LOG_LEVEL_INFO);

  NodeContainer nodes;
  nodes.Create (2);

  PointToPointHelper pointToPoint;
  pointToPoint.SetDeviceAttribute ("DataRate", StringValue ("5Mbps"));
  pointToPoint.SetChannelAttribute ("Delay", StringValue ("2ms"));

  NetDeviceContainer devices;
  devices = pointToPoint.Install (nodes);

  InternetStackHelper stack;
  stack.Install (nodes);

  Ipv4AddressHelper address;
  address.SetBase ("10.1.1.0", "255.255.255.0");

  Ipv4InterfaceContainer interfaces = address.Assign (devices);

  UdpEchoServerHelper echoServer (9);

  ApplicationContainer serverApps = echoServer.Install (nodes.Get (1));
  serverApps.Start (Seconds (1.0));
  serverApps.Stop (Seconds (10.0));

  UdpEchoClientHelper echoClient (interfaces.GetAddress (1), 9);
  echoClient.SetAttribute ("MaxPackets", UintegerValue (1));
  echoClient.SetAttribute ("Interval", TimeValue (Seconds (1.0)));
  echoClient.SetAttribute ("PacketSize", UintegerValue (1024));

  ApplicationContainer clientApps = echoClient.Install (nodes.Get (0));
  clientApps.Start (Seconds (2.0));
  clientApps.Stop (Seconds (10.0));

  Simulator::Run ();
  Simulator::Destroy ();
  return 0;
}
```

Figura 22: Código do exemplo first.cc do tutorial do NS-3

Este exemplo disponibilizado pelo NS-3 inclui os passos principais de um *script* de uma simulação. Tudo começa com a inclusão dos módulos necessários para definir a simulação. Seguidamente, é declarado o *namespace ns3*. Todo o código do simulador foi implementado usando o este *namespace*.

O próximo passo é ativar os componentes de *logging*, bastante importante para recolher informações sobre o estado da simulação. A instrução `NS_LOG_COMPONENT_DEFINE` permite definir um nome para as mensagens de *logging*. Mais instruções deste tipo encontram-se já dentro da função *main* para registar informações sobre as aplicações da simulação:

```
LogComponentEnable ("UdpEchoClientApplication", LOG_LEVEL_INFO);
LogComponentEnable ("UdpEchoServerApplication", LOG_LEVEL_INFO);
```

Neste ponto do *script* inicia-se a construção da rede criando os nós. O NS-3 providencia assistentes que auxiliam na criação e manipulação de objetos, como o `NodeContainer`. Através do método `Create`, são criados dois nós e armazenados numa lista dentro do assistente. Após criados os nós da rede, o passo seguinte foca-se na definição da topologia da rede. Os *helpers* são assistentes que facilitam este processo e para definir a ligação ponto-a-ponto é usado o `PointToPointHelper`. Através deste assistente, é possível configurar atributos dos dispositivos e canais ponto-a-ponto pelos métodos `SetDeviceAttribute` e `SetChannelAttribute`. Já o método `Install` recebe como parâmetro os nós guardados no `NodeContainer` e instala em cada um deles uma interface de rede ponto-a-ponto sendo estas, por sua vez, guardadas num assistente próprio, `NetDeviceContainer`. As seguintes instruções estão relacionadas com a instalação da pilha protocolar e a atribuição de endereços às interfaces dos nós. Para isso, recorre-se novamente ao auxílio dos *helpers* `InternetStackHelper` e `Ipv4AddressHelper`, respetivamente através dos métodos `Install` e `Assign`. Por fim, resta apenas criar e instalar as aplicações em ambos os nós. Num dos nós é instalado uma aplicação `UdpEchoServer` e noutro uma aplicação `UdpEchoClient` através do método `Install` de cada um dos *helpers* respetivos, `UdpEchoServerHelper` e `UdpEchoClientHelper`. Pelo método `SetAttribute` é possível configurar alguns atributos das aplicações, como é visível no código do exemplo. Também é necessário indicar o tempo do início e do fim das aplicações, através dos métodos `Start` e `Stop`. Para executar a simulação falta apenas acrescentar a instrução `Simulator::Run ()` que executará todos os eventos escalonados pelas aplicações nos tempos relativos da simulação, definidos previamente. Após isso, torna-se necessário destruir os objetos que foram criados para a simulação através da instrução `Simulator::Destroy ()`.

4.4 Análise das classes do NS-3

Nesta secção são mostradas as classes do NS-3 relevantes para o trabalho proposto com uma detalhada análise às suas funções e atributos.

A análise do NS-3 revelou-se ser uma parte fundamental para o sucesso deste projeto. O simulador envolve um grande número de classes e, por isso, esta fase do trabalho foi particularmente exigente devido ao meu completo desconhecimento relativamente à estrutura do NS-3. Para isso, inicialmente segui o tutorial disponibilizado [16], que apresenta alguns exemplos básicos, para ter algumas noções de como configurar e executar uma simulação. Posteriormente, havia que identificar as potenciais classes existentes que poderiam ser utilizadas no trabalho a realizar e aprofundar os conhecimentos relativamente às funcionalidades, métodos e atributos inerentes às mesmas.

A Figura 23 apresenta um diagrama de classes simplificado, não detalhando os atributos e métodos de cada classe, para apenas demonstrar a relação entre as várias classes. É possível visualizar o diagrama completo no Anexo 2 deste trabalho.

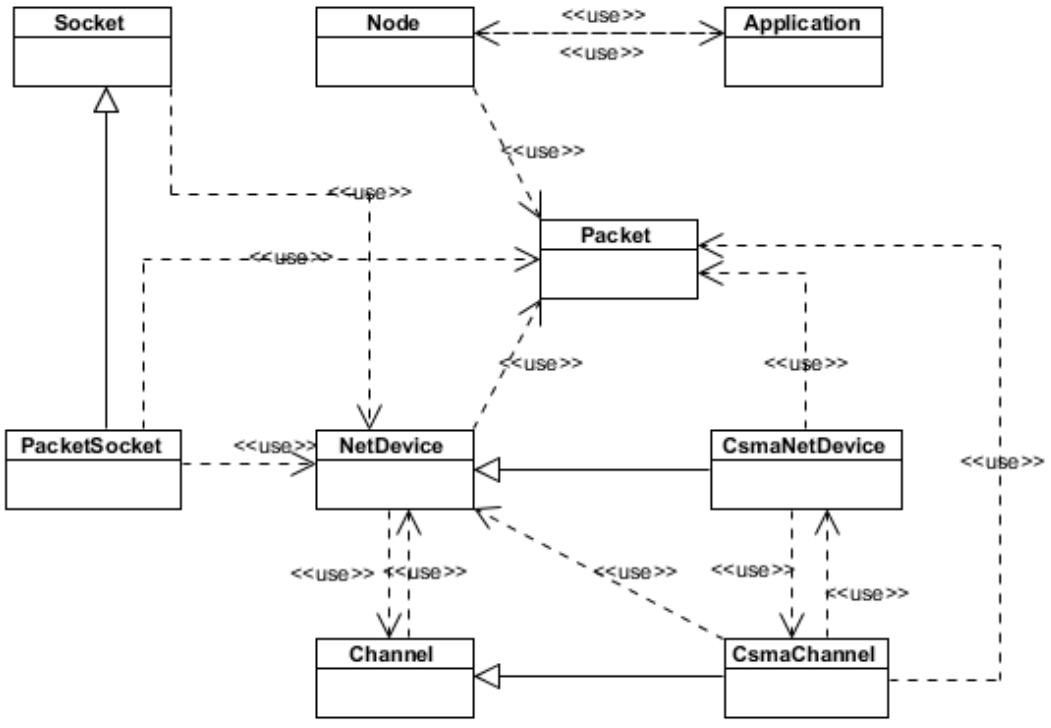


Figura 23: Diagrama de classes simplificado das classes do NS-3 relevantes para o projeto

Seguidamente, é apresentada individualmente cada classe detalhando os seus métodos e atributos e feita uma breve descrição das mesmas.

Node:

Um *Node* representa um nó na rede. Neste objeto existe uma lista de aplicações, onde são guardadas as aplicações instaladas no nó, que interagem com o mesmo por meio da implementação de um `Socket`. Também existe uma lista onde são guardados as interfaces de rede (`NetDevice`) instaladas no nó. Além disso, cada nó tem associado um único ID e também um ID de sistema utilizado para simulações de paralelismo.

Dois dos métodos mais utilizados e importantes nesta classe são os seguintes:

- `AddApplication` – Através desta função, é possível instalar uma aplicação no nó.
- `AddDevice` – Esta função permite associar um dispositivo de rede ao nó.

Application:

É a classe base para todas as aplicações do NS-3. Qualquer aplicação desenvolvida no simulador deve ser implementada a partir da classe `Application`. O utilizador deve definir nesta classe os atributos `StartTime` e `StopTime` que especificam o instante em que a aplicação inicia e o instante em que termina, respetivamente. Estes atributos são utilizados pelos seguintes métodos:

- `StartApplication` – Este método é chamado no instante inicial, definido no atributo `StartTime`, da execução de uma aplicação e tem de ser reescrito nas classes que implementam esta classe.
- `StopApplication` – Este método é chamado no instante definido pelo atributo `StoptTime`, assinalando o final da execução de uma aplicação. Também tem de ser reescrito nas classes que implementam esta classe.

Socket:

Esta classe é uma interface, baseada nos *Burkeley sockets* (BSD), para protocolos de transporte, camada 4 do modelo OSI, como por exemplo TCP ou UDP, e também como interface para *packet sockets*. As funções mais utilizadas nesta classe são as seguintes:

- `Bind` – Esta função associa um endereço ao *socket*.
- `Connect` – Através desta função é iniciada a ligação do *socket* ao endereço de destino.
- `Recv` – Esta função permite a leitura de dados recebidos pelo *socket*.

- `Send` – Esta função é chamada quando se pretende enviar dados (`Packet`) para uma máquina remota.

No NS-3 existem implementações de vários tipos de s*ockets*, que herdam os atributos e métodos desta classe. Em seguida são enumeradas algumas dessas implementações:

- `TcpSocket` – Implementa um *socket* para comunicações que utilizem o protocolo TCP.
- `UdpSocket` – Implementa um *socket* em comunicações que utilizem o protocolo UDP.
- `PacketSocket` – Este tipo de *socket* permite uma ligação de dados entre uma aplicação e um dispositivo de rede instalado no nó.

`NetDevice`:

A classe `NetDevice` é uma interface que descreve a forma como a camada três do modelo OSI interage com a camada dois do mesmo modelo. Desta forma, todas as implementações da classe `NetDevice` emulam um dispositivo de rede real.

A classe `NetDevice` define, então, algumas funções importantes, tais como:

- `Send` – Esta função é responsável pelo envio de pacotes (*frames*) pelo `NetDevice`. Recebe como argumentos um `Packet` a enviar, um endereço MAC de destino e um inteiro que define o protocolo. Este último parâmetro facilita, mais tarde, a chamada do protocolo de camada 3 correto, na altura da receção do pacote.
- `SetReceiveCallback` – Esta função é responsável por associar uma `ReceiveCallback` a um dado `NetDevice`. Na prática, isto significa que, de cada vez que um pacote atinge este `NetDevice`, a `ReceiveCallback` passada por parâmetro é executada.

Alguns exemplos de `NetDevices` relevantes para este projecto são:

- `CsmaNetDevice` – Implementa uma interface de rede para ligações que utilizem o protocolo CSMA, usado em redes como *Ethernet* (IEEE 802.3).
- `WifiNetDevice` – Este dispositivo emula uma interface IEEE 802.11. Atualmente, o ns-3 suporta a camada física de 802.11a e 802.11b, variados modelos de propagação de sinal e modelos de atraso de propagação de sinal, o que confere algum realismo à modelação de redes que fazem uso deste `NetDevice`.

- `WaveNetDevice` – O modelo de WAVE foca-se na camada MAC, pelo que faz uso da camada física do 802.11a implementada. Neste contexto, é capaz de realizar comunicações baseadas no *standard* IEEE 802.11p, sendo capaz de transmitir em modo "*Outside the context of a Basic Service Set*"(OCB), uma funcionalidade central do *standard* IEEE 802.11p, que permite aos nós comunicarem sem estarem associados a um *Basic Service Set*.

`Channel`*:*

É superclasse de um canal de rede, que faz a ligação entre duas interfaces de nós distintos. A sua implementação está diretamente relacionada com o tipo de rede. Deste modo deve ser implementada com o mesmo tipo do `NetDevice`.

Alguns exemplos de `Channel` são:

- `CsmaChannel` – Implementa a ligação por cabo entre dois nós numa rede CSMA (*Ethernet*).
- `WifiChannel` – Esta classe implementa as propriedades físicas de um canal *Wifi* real. É usada em conjunto com a classe `WifiPhy`.

`Packet`:

Os pacotes modelados contem um *buffer* de *bytes* e um conjunto de *byte tags* e *packet tags*. O *buffer* de bytes contem a informação serializada do pacote (por exemplo, *headers*, *trailers* e dados), o que permite facilmente modelar um dado protocolo de interesse para o utilizador. É esperado que a estrutura do pacote seja implementada de forma a corresponder, *bit* a *bit*, ao pacote real, contudo, os utilizadores podem escolher modelar pacotes como desejarem, de forma a irem ao encontro do objetivo das suas investigações.

`Callback`:

As *callbacks*, no ns-3, fazem uso de uma funcionalidade da linguagem C chamada *pointer to function returning integer*. Este método permite ao NS-3 obter um acoplamento mais baixo, uma vez que deixa de ser necessário instanciar objetos sem relação.

Esta prática faz com que seja possível associar um pedaço de código a executar quando um dado evento é ativado. Esta funcionalidade é usada, por exemplo, no tratamento dos pacotes que chegam a um dado `NetDevice`. O utilizador pode, então, especificar a forma como quer tratar os pacotes, conferindo uma grande flexibilidade ao simulador.

`Simulator`:

Este é das classes mais importantes do NS-3. Como já referido do capítulo 4.1, o NS-3 é baseado em eventos discretos, ou seja, o simulador mantem uma fila de eventos que vão sendo executados numa ordem sequencial de tempo. A classe Simulator é responsável por manipular todos os eventos na simulação. Em seguida são apresentadas as principais funções desta classe:

- `Run` – Executa a simulação definida. O simulador vai executar todos os eventos escalonados na simulação até que se esgotem ou que seja chamada a função `Stop` pelo utilizador.
- `Stop` – Quando esta função é chamada, mais nenhum evento é executado na simulação.
- `Schedule` – Esta função permite ao utilizador escalonar um evento para um determinado tempo relativo na simulação. Quando a simulação chega ao tempo relativo especificado, é chamada a função associada ao evento escalonado.
- `Cancel` – Permite cancelar um determinado evento escalonado.
- `Now` – Ao chamar esta função é possível obter o instante relativo da simulação.

***Tracing*:**

O NS-3 oferece aos seus utilizadores variadas formas de obter informação a partir das suas simulações. A maioria dos seus módulos possui métodos de *tracing* implementados, tais como ficheiros PCAP, que podem ser usados, em conjunto com programas como o *Wireshark* ou o *Tcpdump*, para analisar os pacotes recebidos e transmitidos por uma dada interface.

De forma a poder acomodar as necessidades dos utilizadores, é possível definir funções a serem executadas quando um determinado evento ocorre (por exemplo, quando um pacote está pronto para ser transmitido pela camada física). O utilizador pode, assim, registar uma variedade de eventos que ocorrem durante uma simulação, bem como as propriedades desse evento (por exemplo, a informação de um pacote que é aceite pela camada física mas rejeitado pela camada MAC)

O NS-3 possui, também, um modelo de *logging*, que é um requerimento de qualquer módulo implementado. Como tal, todos os módulos possuem algum tipo de *logging*, o que facilita a depuração de uma simulação, bem como a análise dos eventos à medida que estes são processados pelo ambiente de simulação. Este método imprime informação para o descritor de ficheiros *stderr* e possui sete níveis de severidade, de forma a não sobrecarregar o utilizador com informação.

Modelo de Erros:

De forma a obter um maior realismo, a maioria dos módulos possuem algum tipo de modelo de erros, de forma a modelar os problemas que poderão surgir durante a comunicação. Estes modelos podem ir desde modelos simples, que calculam a possibilidade de erros aleatoriamente, a modelos de erros complexos e específicos para uma dada tecnologia que, pela sua elevada complexidade computacional, são, por vezes, computados previamente e distribuídos com o simulador.

Para a realização deste projeto, foi necessário aplicar um *patch* sobre as classes CsmaNetDevice e CsmaChannel, uma vez que a implementação existente não suportava a ligações *full-duplex*, ou seja, transmissões e receções em simultânea por parte de um nó. Com este *patch* [17] torna-se possível, no NS-3, a utilização de redes *Ethernet* com suporte para ligações *full-duplex*.

5 Descrição técnica

Neste capítulo é feita a descrição técnica do trabalho desenvolvido, nomeadamente o levantamento dos requisitos, funcionais e não funcionais, o estudo da documentação do NS-3 relativamente às classes relevantes para o projeto, a modelação das novas classes criadas para a implementação realizada, bem como a respetiva descrição.

5.1 Levantamento de requisitos

O levantamento de requisitos é um processo importante na fase de análise de um projeto, uma vez que têm de ser mantidos em consideração na abordagem à solução a realizar para o problema. O trabalho desenvolvido também compreendeu esta fase de levantamento de requisitos que são seguidamente enumerados e analisados, sendo divididos em requisitos funcionais e não funcionais.

5.1.1 Requisitos funcionais

Os requisitos funcionais determinam as funcionalidades que devem ser integradas no *software* a desenvolver. O trabalho realizado procurou cumprir ao máximo os requisitos definidos, que são em seguida enumerados.

***Master*:**

Deve ser criada uma aplicação que simule o comportamento de um nó *Master*, que será responsável pela coordenação da rede e do escalonamento do tráfego que será trocado entre os restantes nós da rede. A construção da *Trigger Message* desempenha um papel fundamental neste processo.

Slave:

Deve ser criada uma aplicação que simule o comportamento de um nó *Slave*, que tem como função o envio de tráfego nos instantes definidos pelo nó *Master* da rede. Também terá de integrar um mecanismo de envio de *Signalling Messages* no âmbito da troca de tráfego aperiódico (assíncrono).

Cenário *Plug-and-play* (PnP):

Qualquer nó ou aplicação (*Master* e *Slave*) que seja adicionado à rede que vai integrar a simulação a realizar deve cumprir um mecanismo de *Plug-and-play* sendo automaticamente registados na rede.

Escalonamento

A aplicação *Master* deve realizar internamente o escalonamento de mensagens, para todos os *Elementary Cycles*, tendo como base o modelo *Rate Monotonic* (ver secção 2.4) e a natureza das mesmas, definindo aquelas que são possíveis de trocar entre as aplicações *Slave* durante aquele EC.

Trigger Message

O início de cada EC é marcado pelo envio em *broadcast* da *Trigger Message*, por parte do *Master,* que incorpora no seu *payload* as mensagens que, após o escalonamento efetuado para aquele EC, foram definidas para serem trocadas pelos restantes nós.

Envio de tráfego síncrono e assíncrono

Cada aplicação deve gerar e enviar tráfego (*streams*) para os respetivos consumidores, de tipo síncrono ou assíncrono, sendo que o último pode ser dividido em 3 subtipos: *Hard real time*, *Soft real time* e *Best effort*.

Fork-Join Parallel-Distributed

Deve ser possível realizar simulações onde haja o processamento duma determinada quantidade de dados por vários nós distribuídos, de acordo com o paradigma *Fork-Join Parallel/Distributed.* Neste cenário deve existir uma aplicação no nó local que envia dados para

aplicações em nós remotos que devem ser processar os mesmos (de forma simulada) e reenviar para a aplicação do nó principal. Deve suportar a utilização deste paradigma em *streams* assíncronas e também síncronas.

Cálculo de *Offset* de sincronização.

Nas simulações que compreendem o paradigma *Fork-Join Parallel-Distributed*, utilizando tráfego síncrono, o *Master deve* calcular um *offset* de sincronização entre as aplicações do nó local e dos nós remotos, considerando sempre o pior caso (*Worst-Case Response Time*). (ver capítulo 3.6)

Processamento de tarefas

Deve ser implementada a simulação de processamento de tarefas em sistemas monoprocessadores, de acordo com a política de escalonamento *Rate Monotonic* (ver capítulo 2.4)

Exportação de resultados

Deve ser possível a exportação dos resultados das simulações para ficheiros de texto, sendo criado um ficheiro por cada uma das *streams* existentes na simulação.

Helper

Deve ser criado um assistente (*helper*) para facilitar ao utilizador a configuração de uma rede FTT-SE. O assistente deve auxiliar na construção de cenários quer em comunicações sequenciais quer em cenários *Fork-Join Parallel/Distributed*.

5.1.2 Requisitos não funcionais

Os requisitos não funcionais estão relacionados com as caracterização do *software* de acordo com os seus atributos e/ou restrições inerentes ao mesmo. Apesar de não estarem diretamente relacionados com as funcionalidades têm também grande importância e dever ser considerados no desenvolvimento do trabalho. Seguidamente são apresentados os requisitos não funcionais deste trabalho.

Usabilidade

Deve ser fácil de configurar os atributos de uma simulação, iniciá-la e recolher os resultados da mesma.

Implementação

O trabalho deve ser desenvolvido na linguagem C++, uma vez que é a linguagem utilizada no desenvolvimento dos módulos do NS-3. Também devem ser respeitados os estilos de código utilizados no NS-3.

5.2 Modelação e implementação

Nesta secção é apresentada a abordagem realizada para a solução do problema deste projeto. É descrita a forma como foi implementado o protocolo FTT-SE no NS-3, detalhando cada uma das funcionalidades introduzidas.

5.2.1 Classes implementadas

Inicialmente foram definidas quais seriam as classes a desenvolver e perceber como se poderiam integrar com as classes existentes do NS-3, nomeadamente as referidas na secção 4.4.

Após a análise realizada, foi decidido desenvolver a solução a partir do módulo *Applications* do simulador, uma vez que o foco central do trabalho incide em criar aplicações que simulassem o comportamento de aplicações *Master* e *Slaves* numa rede FTT-SE. Para além disso, tornou-se imprescindível o desenvolvimento de classes correspondentes a componentes específicos da arquitetura da tecnologia, nomeadamente no que diz respeito às bases de dados internas e às *streams* relacionadas com este tipo de comunicações. Como também um dos objetivos do trabalho é simular a execução de tarefas, foi necessário criar uma classe que modelasse uma tarefa. Por fim, havia que criar um assistente para facilitar a configuração de uma rede FTT-SE.

Desta forma, chegou-se à conclusão que a arquitetura ilustrada na Figura 24 seria a melhor solução para o problema do presente trabalho. A figura mostra um diagrama de classes simplificado, no sentido apenas de mostrar a arquitetura do sistema. Detalhes relativamente às classes desenvolvidas serão abordados, individualmente, mais à frente no relatório.

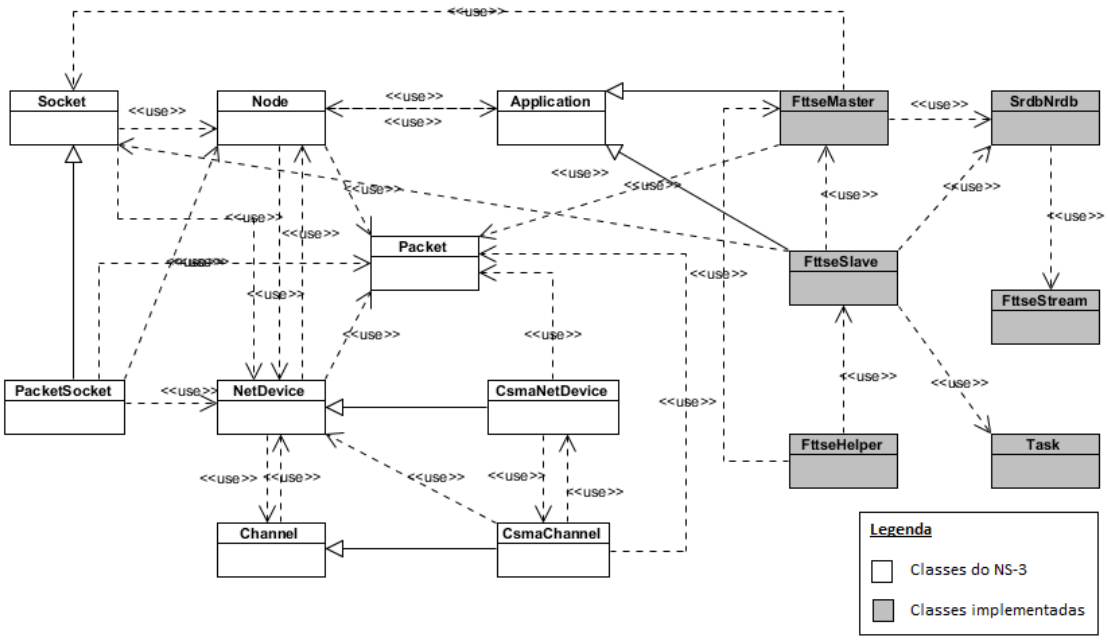


Figura 24: Diagrama de classes simplificado da solução encontrada para o projeto

A classe `FttseMaster` (Figura 25) representa uma aplicação *Master* da rede FTT-SE. A classe `FttseSlave` (Figura 26) define uma aplicação *Slave* da rede FTT-SE. A classe `FttseStream` (Figura 27) modela as caraterísticas de uma *stream* a ser trocada na rede pelas aplicações `FttseSlave` que pode. A classe `SrdbNrdb` (Figura 28) representa uma base de dados de `FttseStream`, fazendo parte dos atributos, quer da aplicação `FttseMaster`, quer da aplicação `FttseSlave`. Cada aplicação `FttseSlave` tem nos seus atributos uma `Task` (Figura 29), que representa uma tarefa associada àquela aplicação. Por fim, a classe `FttseHelper` (Figura 30) define um assistente de configuração da rede FTT-SE.

Seguidamente, é elaborada uma descrição de cada uma das classes desenvolvidas, apresentando os seus atributos e funções e enumerando as principais funcionalidades das mesmas.

`FttseMaster`

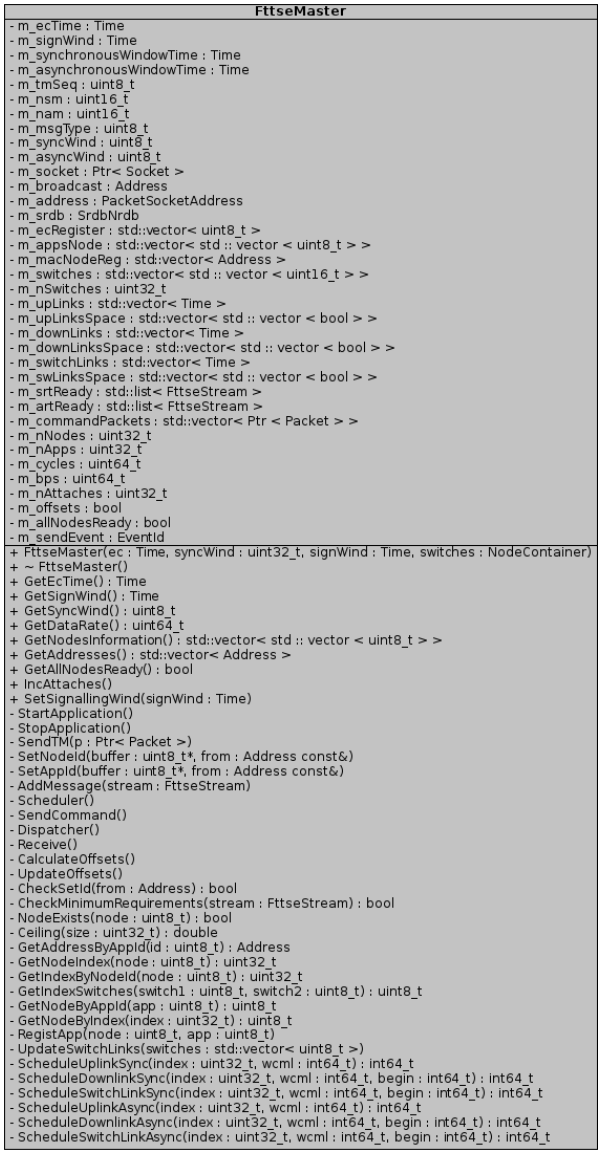


Figura 25: Classe FttseMaster

Esta classe é uma implementação da classe *Application* do NS-3. É a aplicação que é executada do nó *Master*, que fará a coordenação da rede.

Principais funcionalidades:

- Envio de mensagens de controlo
- Registo de *streams*
- Escalonamento de mensagens seguindo a política *Rate Monotonic*
- Envio de *Trigger Messages*

- Cálculo de *Offsets* de sincronização entre aplicações produtoras de tráfego síncrono em comunicações *Fork-Join Parallel/Distributed*.

É na construção desta classe que o utilizador define as características da rede.

```
FttseMaster(uint64_t ec, uint32_t syncWindow, Time signWindow, Node-
Container switches)
```

O seu construtor recebe como argumentos a duração de cada *Elementary Cycle*, em microssegundos, através do parâmetro `ec`, a percentagem da *Synchronous Window*, isto é a largura de banda reservada em cada EC para a transmissão de tráfego síncrono (`syncWindow`), o tempo reservado para a *Signalling Window* (`signWindow`), em microssegundos e os *switches* que integram a topologia utilizada para a simulação (`switches`).

`FttseSlave`

É também uma implementação da classe *Application* e representa as aplicações executadas nos nós *Slave*, que trocam tráfego síncrono ou assíncrono. Cada uma destas aplicações está diretamente relacionada com uma *stream* de dados de uma determinada natureza, isto é, síncrona ou assíncrona.

Neste ponto, é importante lembrar que existem nós produtores, que geram uma *stream* de dados, e nós consumidores, que consomem uma *stream* de dados. Para permitir a criação de aplicações de ambos os tipos, foram definidos construtores, nesta classe, para criar aplicações produtoras ou consumidoras.

Principais funcionalidades:

- Envio de mensagens de pedidos de ID, no cenário *Plug-and-Play*
- Envio de *Signalling Messages*
- Envio de mensagens síncronas e assíncronas
- Possibilidade de realizar comunicação sequencial e distribuída (paradigma *Fork-Join Parallel/Distributed*)
- Simulação do processamento de tarefas considerando a existência de apenas um único processador

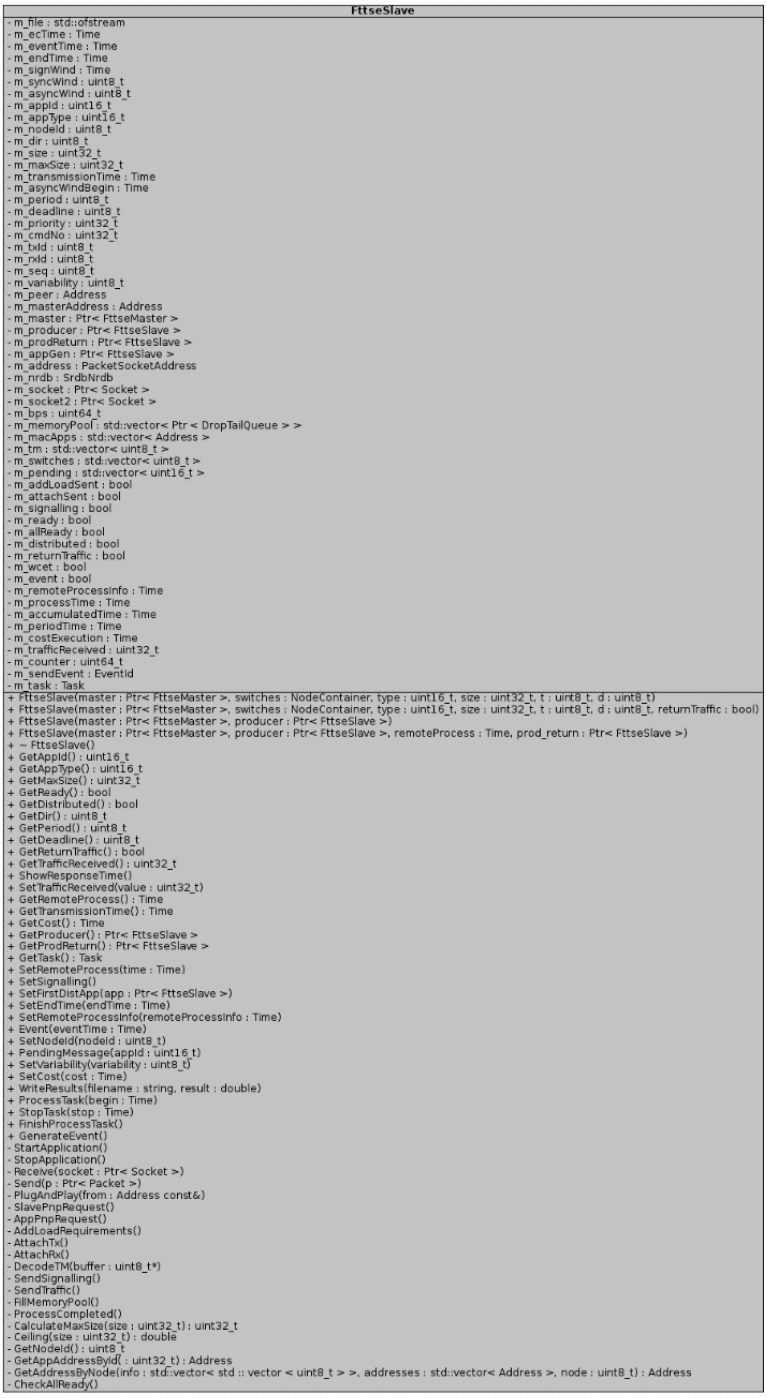


Figura 26: Classe FttseSlave

Um cuidado especial foi tido em conta no desenvolvimento desta classe, no sentido de cumprir os objetivos propostos para este trabalho. Teria de ser possível simular a comunicação direta (sequencial) entre nós *Slave* através deste protocolo, contudo também é objetivo do trabalho a possibilidade de realizar comunicação distribuídas (*Fork-Join Parallel/Distributed*) por vários nós na rede. Desta forma, a solução encontrada resultou nas seguintes definições de construtores.

```
FttseSlave(Ptr<FttseMaster> master, NodeContainer switches, uint16_t
type, uint32_t size, uint8_t t, uint8_t d)
```

Este construtor cria uma aplicação *Slave* produtora para uma comunicação sequencial. Recebe como argumentos um apontador para a aplicação `FttseMaster` da rede (`master`), um contentor com os *switches* que fazem parte do percurso da *stream* (`switches`), ordenado de acordo com a sequência do percurso, o tipo de tráfego (`type`), o tamanho das mensagens a produzir (`size`), o período da *stream* (`t`) e o *deadline* associado à mesma (`d`).

```
FttseSlave(Ptr<FttseMaster> master, NodeContainer switches, uint16_t
type, uint32_t size, uint8_t t, uint8_t d, bool returnTraffic)
```

Este construtor define uma aplicação produtora numa comunicação distribuída. Os argumentos que recebe são os mesmos que o construtor da aplicação produtora sequencial, contudo é adicionado um último argumento (`returnTraffic`) que define se a aplicação está instalada no nó local ou no nó remoto da comunicação.

```
FttseSlave(Ptr<FttseMaster> master, Ptr<FttseSlave> producer)
```

Este construtor define uma aplicação *Slave* consumidora de uma comunicação sequencial. Recebe como argumentos um apontador para a aplicação `FttseMaster` (`master`), bem como um apontador para a aplicação produtora da *stream* que vai consumir (`producer`).

```
FttseSlave(Ptr<FttseMaster> master, Ptr<FttseSlave> producer, Time
process, Ptr<FttseSlave> producer_return)
```

Este construtor é utilizado para criar uma aplicação *Slave* consumidora de uma comunicação distribuída. Também recebe como argumentos um apontador para a aplicação `FttseMaster` (`master`) e um apontador para a aplicação produtora da *stream* que vai consumir (`producer`), contudo inclui também argumentos para definir o tempo de processamento de tráfego recebido (`remoteProcess`) e, caso pertença a um nó remoto, um apontador para a aplicação produtora responsável pelo retorno do tráfego para o nó local (`producer_return`). No caso de ser um consumidor no nó local, este último argumento recebe um apontador para uma aplicação criada a partir do construtor vazio desta classe.

FttseStream

Figura 27: Classe FttseStream

Esta classe representa uma *stream* de dados numa rede FTT-SE. É no seu construtor que são definidas as características das *stream* de dados.

```
FttseStream(uint16_t id, uint16_t type, uint32_t size, uint32_t
max_size, Time c, uint8_t t, uint8_t d, uint32_t p)
```

O seu construtor recebe, então, o ID da *stream*, igual ao ID da aplicação (`id`), tipo de tráfego (`type`), tamanho das mensagens (`size`), tamanho máximo que a mensagem pode ter (`max_size`), o *Worst-Case Message Length* (`c`), o período (`t`), o *deadline* (`d`) e a prioridade definida para a *stream* (`p`). É baseado nestas informações que as aplicações geram as suas mensagens. Nota para o facto da utilização da política *Rate Monotonic* na implementação deste trabalho. Desta forma, o *deadline* é igual ao período sendo a *stream* mais prioritária quanto menor for o seu período.

SrdbNrdb

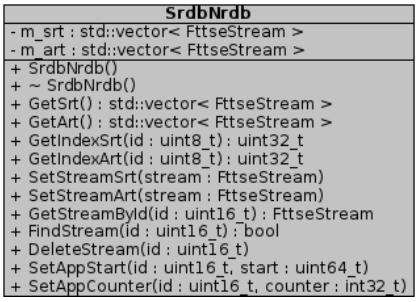


Figura 28: Classe SrdbNrdb

Esta classe define as bases de dados que compõem a arquitetura dos nós no protocolo FTT-SE, SRDB no *Master* e NRDB nos *Slaves*. O seu construtor não recebe argumentos.

```
SrdbNrdb()
```

Devido à semelhança entre as bases de dados de *Master* e *Slave*, esta classe tanto define a base de dados de `FttseMaster` como de `FttseSlave`. Cada uma compreende dois vetores de `FttseStream`, um que armazena as *streams* relativas ao tráfego síncrono e outro que guarda as *streams* assíncronas.

Task

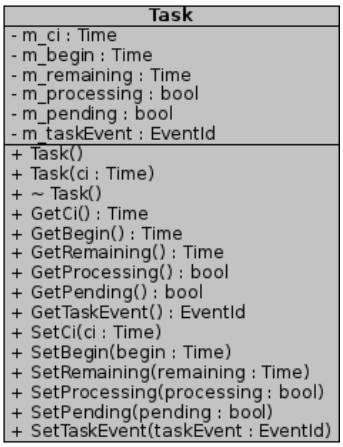


Figura 29: Classe Task

Esta classe define uma tarefa associada a uma aplicação `FttseSlave`.

```
Task(Time ci)
```

No seu construtor é definido o tempo de processamento associado, o *Wors-Case Execution Time* (WCET), que é recebido como argumento (`ci`). Este objeto é usado na simulação de processamento de tarefas nas aplicações `FttseSlave`.

FttseHelper

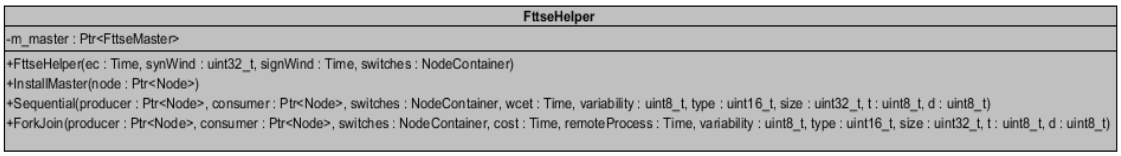


Figura 30: Classe FttseHelper

Esta classe tem a função de assistente na configuração de uma rede FTT-SE. É no seu construtor que recebe as características da rede.

```
FttseHelper(uint64_t ec, uint32_t syncWindow, Time signWindow, Node-
Container switches)
```

O seu construtor recebe, então, como argumentos o tamanho do EC (`ec`), a percentagem reservada para a transmissão de tráfego síncrono (`syncWindow`), o tempo reservado para a *Signalling Window* (`signWindow`) e um contentor com todos os *switches* utilizados na simulação.

Principais funcionalidades:

- Instalação da aplicação *FttseMaster* no nó *Master*
- Configuração de comunicações sequenciais nos nós *Slave*
- Configuração de comunicações distribuídas, segundo paradigma *Fork-Join Parallel/Distributed* nos nós *Slave*.

Após apresentada a descrição genérica de cada uma das classes criadas, é importante entender as atividades que ocorrem, quer na aplicação `FttseMaster` quer nas aplicações e `FttseSlave`. Para isso, são apresentados na Figura 31 Figura 32 diagramas de atividades que oferecem uma perspetiva a alto nível do que acontece na aplicação `FttseMaster` e `FttseSlave` durante cada *Elemtentary Cycle*.

No diagrama apresentado na Figura 31 é possível ver o fluxograma da classe `FttseMaster` desde o início do EC até ao seu fim. Inicialmente, é verificada a existência de mensagens pendentes. As informações relativamente a mensagens pendentes são guardadas em duas filas (m_srtReady e m_artReady), uma para mensagens síncronas e outra para mensagens assíncronas.

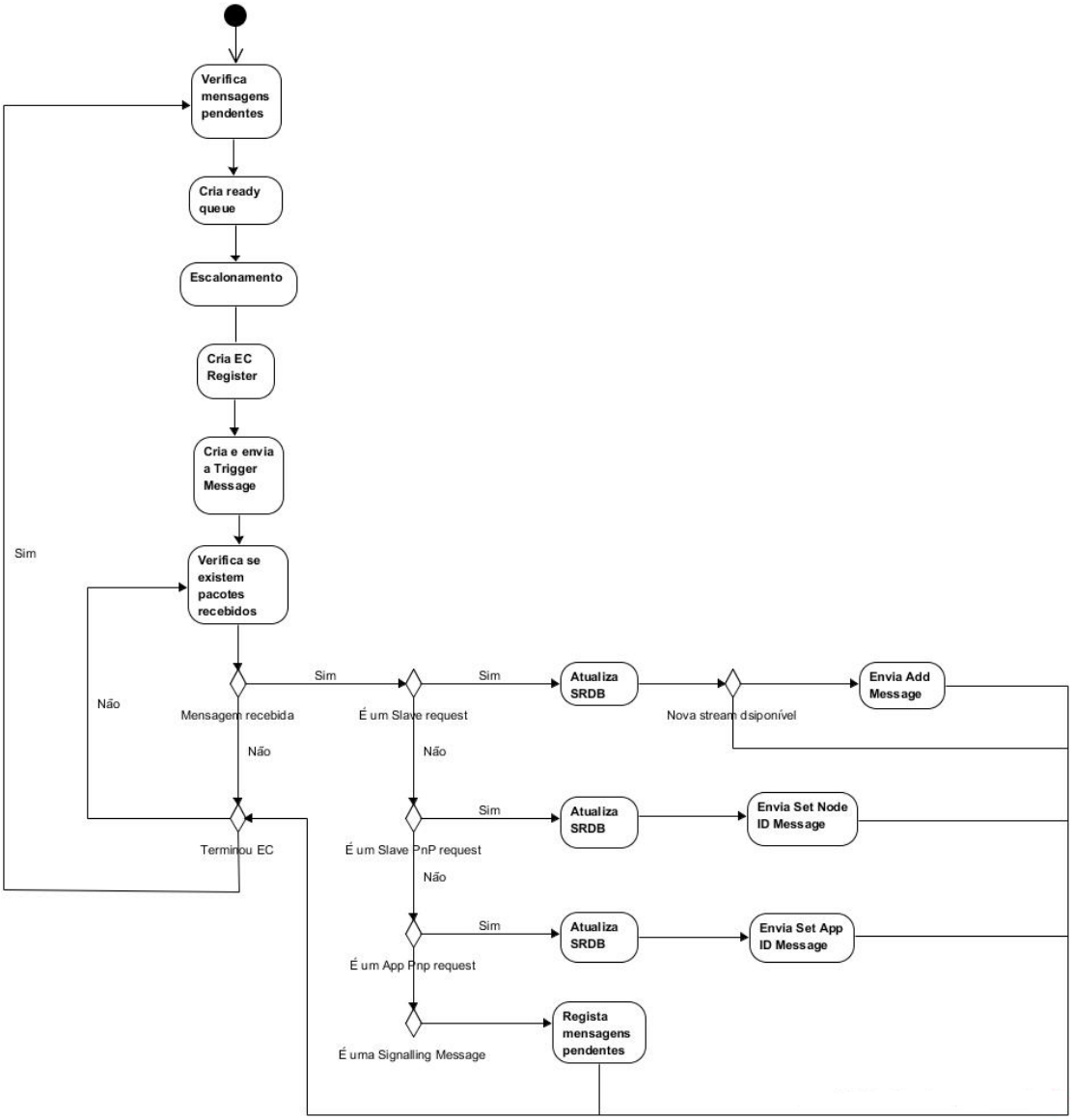


Figura 31: Diagrama de atividades da classe `FttseMaster`

Seguidamente, as *Ready queues* são utilizadas no processo de escalonamento, de forma a verificar quais as mensagens que podem ser transmitidas naquele EC. À medida que o escalonamento se vai processando, informações sobre as mensagens escalonadas são registadas num vetor, chamado `m_ecRegister`. É com base no `m_ecRegister` que é contruída a *Trigger Message*, que em seguida é enviada em *broadcast* para todas as aplicações `FttseSlave`. Contudo, não termina aqui o trabalho da aplicação durante o *Elementary Cycle*. A aplicação `FttseMaster` deve reagir a eventuais pedidos efetuados pelos restantes nós da rede e responder convenientemente. Desta forma, seguidamente ao envio da TM, o *Master* fica imediatamente à espera da chegada de mensagens para poder reagir a cada um dos pedidos recebidos, quer relacionados com o mecanismo de *Plug-and-play,* quer com o mecanismo de *signalling*. Relativamente ao mecanismo de *Plug-and-play*, o `FttseMaster`,

ao receber pedidos de ID (para nó ou aplicação), constrói uma mensagem de resposta, onde incluí no conteúdo da mesma o ID atribuído.

Mais detalhes sobre o cenário de *Plug-and-play* são apresentados no capítulo 5.2.2. No que diz respeito ao mecanismo de s*ignalling*, o `FttseMaster` ao receber uma *Signalling Message*, guarda a informação recebida, registando mensagens assíncronas pendentes na *Ready queue* (`m_srtReady ou m_artReady`) respetiva.

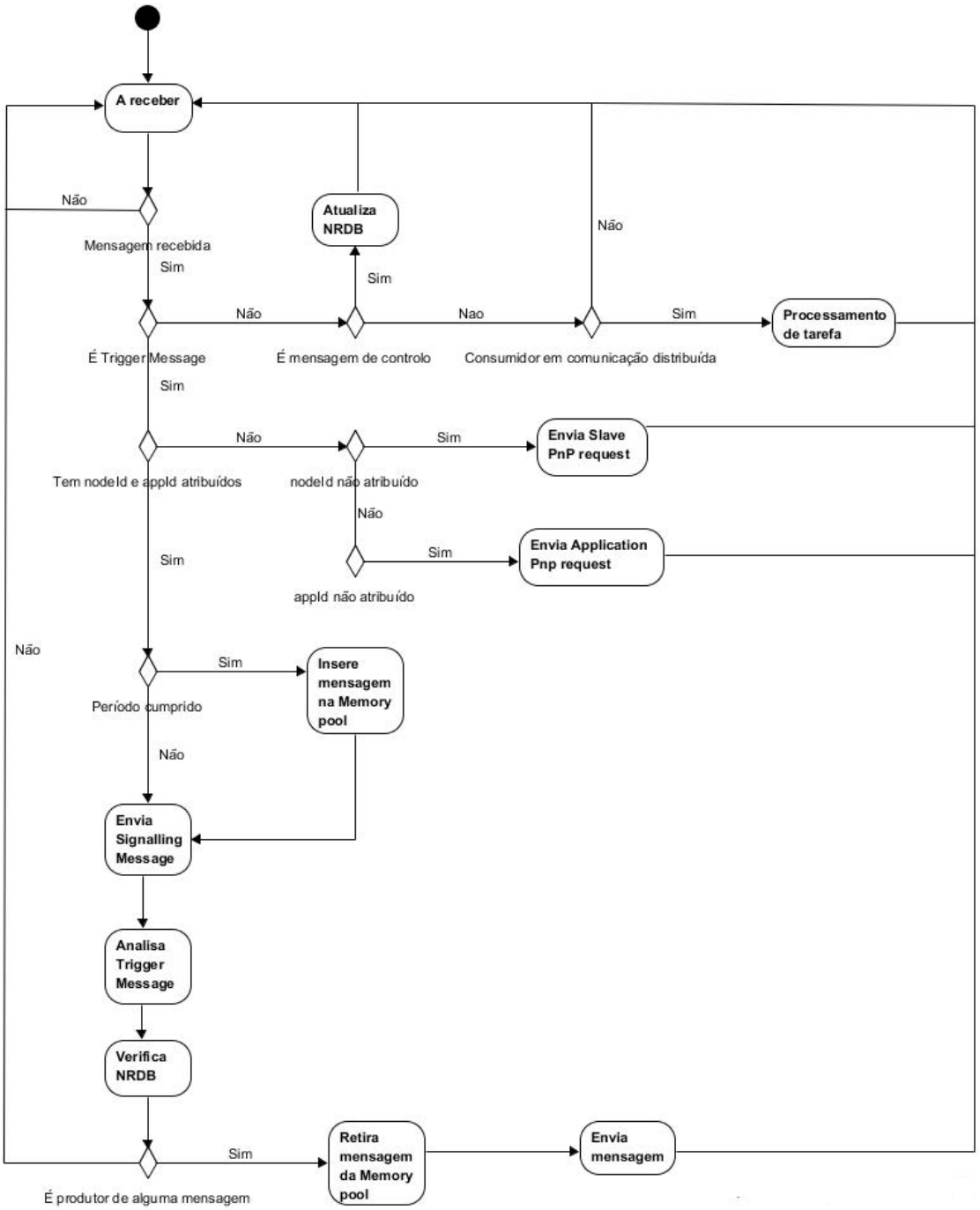


Figura 32: Diagrama de atividades da classe `FttseSlave`

Já a Figura 32 ilustra a perspetiva do que acontece dentro da classe `FttseSlave`. Mal o EC inicia, a aplicação fica imediatamente à espera de receber mensagens, a *Trigger Message* ou mensagens de controlo, por parte do `FttseMaster`. Se a aplicação ainda não tiver cumprido o mecanismo de *Plug-and-Play* ao receber a TM, inicia o referido processo. (ver capítulo 5.2.2)

Também é função das aplicações `FttseSlave`, caso sejam produtores, a verificação no início de cada EC (marcado pela receção da *Trigger Message*) se o período associado à *stream* da aplicação está cumprido. Se o período já tiver sido cumprido, a aplicação gera uma mensagem correspondente à sua *stream* e insere-a numa fila de mensagens pendentes (m_memoryPool). Seguidamente, é enviado para o *Master* a informação relativamente ao estado das filas de mensagens assíncronas, ou seja, se existem mensagens assíncronas pendentes, através de uma *Signalling Message*.

A *Trigger Message* contém no seu *payload* a informação sobre as mensagens que vão ser transmitidas naquele EC. Desta forma, cada aplicação produtora procede à análise do conteúdo da TM, afim de verificar se é produtora de alguma das mensagens escalonadas para aquele EC. No caso de ser produtora de algumas dessas mensagens, a aplicação inicia imediatamente o envio da mensagem, retirando-a da sua fila de mensagens pendentes e procedendo, em seguida, ao início da sua transmissão.

Relativamente às aplicações consumidoras integradas numa comunicação distribuída (paradigma *Fork-Join Parallel/Distributed*), após a receção da totalidade da mensagem associada à *stream* que está a consumir, no caso de este se encontrar em nó remoto, inicia a simulação do processamento dos dados recebidos.

É importante referir que, após a execução de qualquer procedimento acima referido, a aplicação fica novamente à espera da chegada de novas mensagens.

5.2.2 Cenário de Plug-and-Play

Após mostrada uma visão geral das classes implementadas e mensagens trocadas entre as mesmas é importante perceber em maior detalhe cada fase da comunicação na tecnologia FTT-SE. As mensagens iniciais são relativas ao mecanismo de *Plug-and-Play.* Nesta fase, as aplicações `FttseSlave` fazem pedidos de ID, quer para o nó quer, para as suas aplicações, sendo estas informações registadas pela aplicação `FttseMaster` e pela aplicação `FttseSlave` que efetuou o pedido. Neste mecanismo, é verificado em primeiro lugar se já foi atribuído um ID ao nó (`m_nodeId`) e se tal não se verificar, o `FttseSlave` envia um pedido ao `FttseMaster` (*Slave PnP request*). No caso de já estar atribuído um `m_nodeId`, é verificado se a aplicação tem um ID (`m_appId`) atribuído e se não se verificar é enviado para

o `FttseMaster` um outro pedido (*Application PnP request*). A atribuição destes identificadores é concretizada por mensagens de controlo recebidas do `FttseMaster`.

Nota para a duração que compreende esta etapa. Para cada um dos pedidos (nó e aplicação) são necessários pelo menos dois ECs, resultando num mínimo de quatro ECs para a conclusão deste processo.

Os diagramas de sequência simplificados apresentados a seguir demonstram os principais aspetos implementados para a execução deste cenário. O primeiro diagrama (Figura 33) está relacionado com a atribuição de ID ao nó da aplicação `FttseSlave` e o segundo (Figura 34) relativo à atribuição de ID à própria aplicação `FttseSlave`. Ambas as atribuições são realizadas por parte do `FttseMaster`.

O cenário de *Plug-and-Play* inicia com a aplicação `FttseSlave` a enviar um pedido, chamado *Slave PnP request*, após a receção da *Trigger Message*. Esta mensagem é dirigida ao `FttseMaster` e é efetuada devido ao facto de a aplicação `FttseSlave` verificar que o seu ID do seu nó ainda não se encontra atribuído. Por sua vez, o nó coordenador da rede, após receber o pedido, vai construir uma mensagem de resposta ao mesmo, onde vai introduzir o ID atribuído no *payload* da mensagem. No entanto, o envio da resposta apenas é realizado no EC seguinte, posteriormente à *Signalling Window*.

O próximo EC começa novamente com envio da respetiva TM. No momento da sua receção, o `FttseSlave` continua sem ID de nó atribuído, uma vez que a resposta ao seu pedido ainda não foi recebida e, por isso, é realizado novo pedido. Após a *Signalling Window*, a resposta é enviada por parte do `FttseMaster` e analisada pelo nó recetor, que ao verificar o conteúdo da mesma, atualiza o seu ID (`m_nodeId`) com o identificador atribuído. (Figura 33)

O cenário PnP ainda não se encontra finalizado, uma vez que a aplicação `FttseSlave` ainda não tem ID definido. Este processo é muito semelhante ao descrito anteriormente. No terceiro EC deste cenário, após a receção da TM, o *Slave* realiza um pedido de ID para a sua aplicação (*Application PnP request*). O *Master* ao recebê-lo constrói uma mensagem de resposta que apenas será enviada no EC seguinte, posteriormente à *Signalling Window*.

Chegado o quarto, e último EC, deste processo, novo *Application PnP request* é enviado para o `FttseMaster`, uma vez que a aplicação `FttseSlave` ainda não obteve o ID pretendido, que por sua vez envia efetivamente a resposta. Por fim, o `FttseSlave` analisa o *payload* da mensagem de resposta e regista na aplicação o ID atribuído pelo nó central da rede (`m_appId`). (Figura 34)

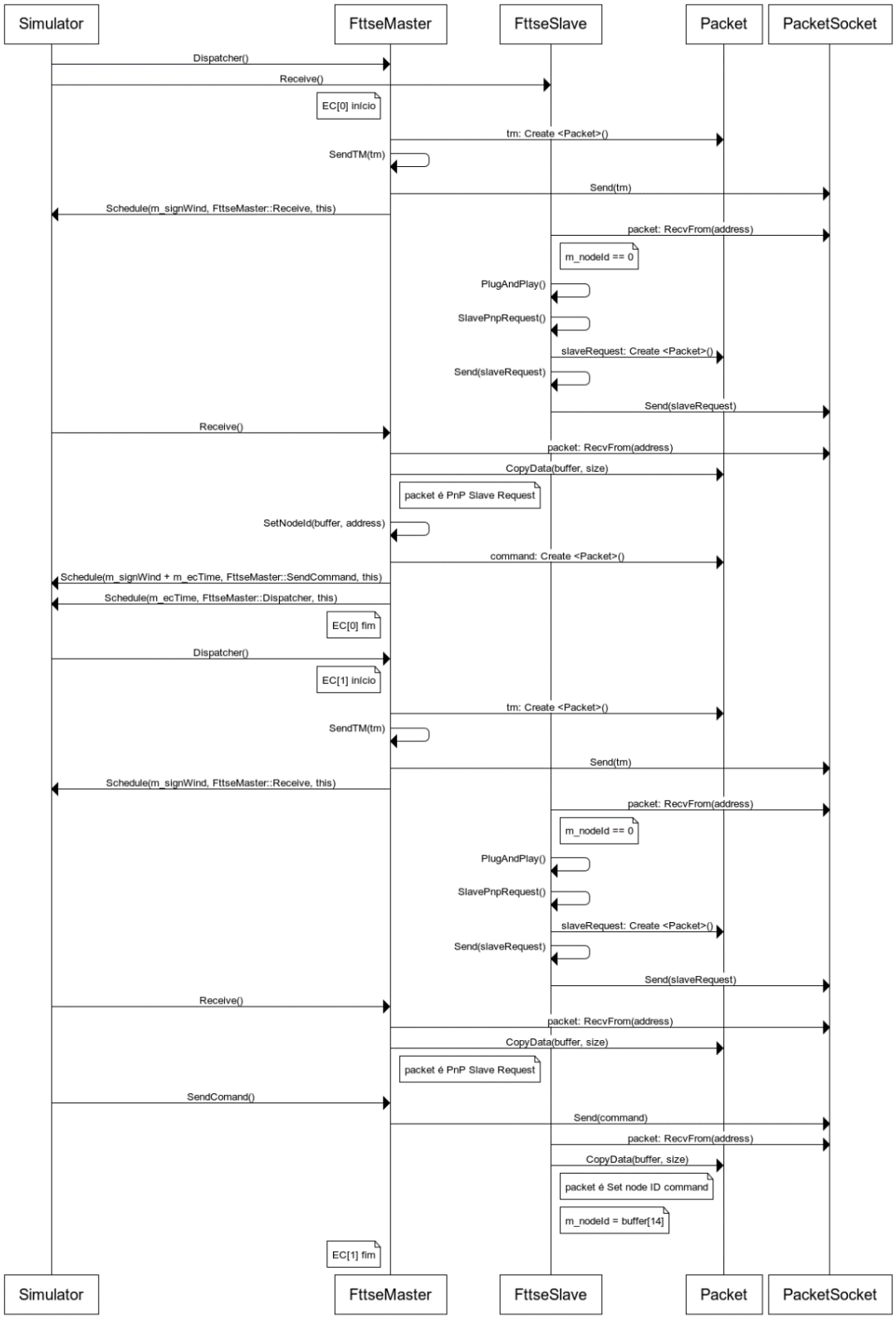


Figura 33: Diagrama de sequências da atribuição de ID ao nó de uma aplicação `FttseSlave`

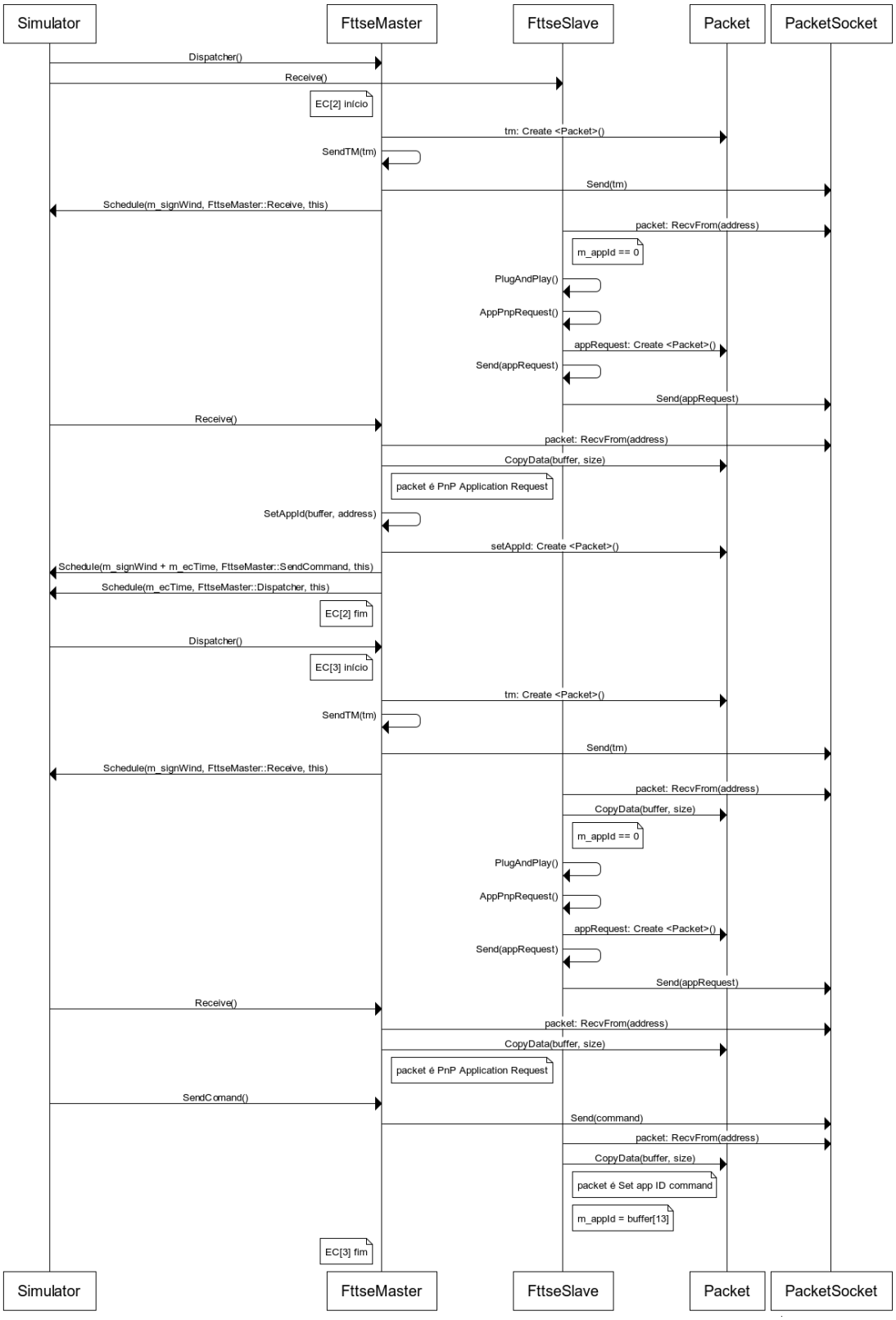


Figura 34: Diagrama de sequências da atribuição de ID a uma aplicação `FttseSlave`

5.2.3 Registo de *streams*

O registo das *streams*, representadas pela classe `FttseStream`, permitem, principalmente ao nó `FttseMaster`, ter um total conhecimento das mensagens que vão ser trocadas na rede. Contudo, também permite às aplicações `FttseSlave` produtoras conhecerem o consumidor das suas `FttseStream`s. Neste processo, os componentes SRDB e NRDB (representadas por `SrdbNrdb`), bases de dados das aplicações `FttseMaster` e `FttseSlave`, respetivamente, desempenham uma função importante, uma vez que permitem armazenar as características de cada `FttseStream`.

Em seguida são apresentados diagramas de sequência simplificados, evidenciando as principais execuções realizadas no processo de registo das `FttseStream`s nas referidas bases de dados.

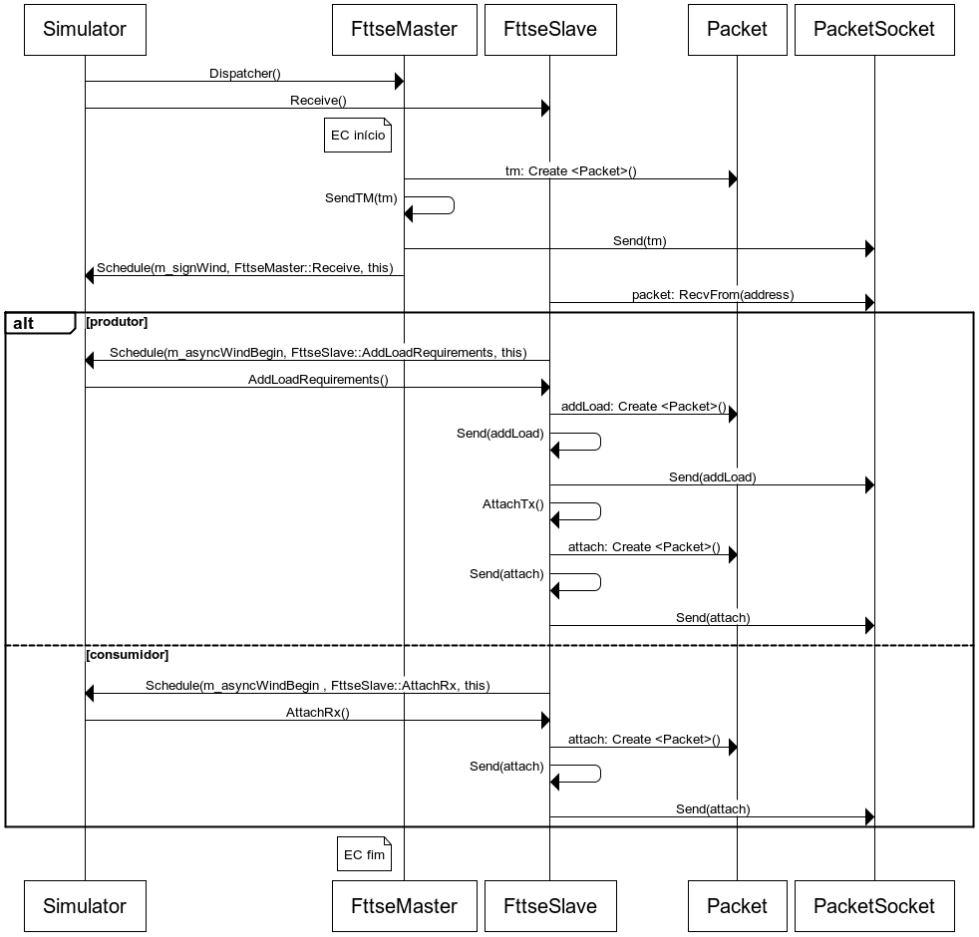


Figura 35: Diagrama de sequências dos pedidos de Attach e informação das características de `FttseStream`

Terminado o cenário de *Plug-and-Play*, essencial para atribuir IDs a cada um dos nós e aplicações de forma a torná-los únicos na rede, o passo seguinte está relacionado com o registo das `FttseStream`s associadas a cada uma das aplicações nas bases de dados, quer da aplicação `FttseMaster`, quer nas aplicações `FttseSlave` produtoras e consumidoras. Desta forma, após a receção da *Trigger Message*, a aplicação `FttseSlave` produtora inicia a construção de uma mensagem (*Add load requirements*) onde reúne no seu conteúdo as informações relativamente a todas as características da `FttseStream` que irá produzir: ID, tipo, tamanho, tamanho máximo, tempo de transmissão, período, *deadline* e prioridade. Seguidamente, cria também uma mensagem (*AttachTx*) para informar ao nó controlador da rede que é o produtor daquela `FttseStream`. Por sua vez, a aplicação `FttseSlave` consumidora apenas tem de informar que é a consumidora daquela `FttseStream` (*AttachRx*). Estas mensagens são enviadas em *unicast* para o `FttseMaster`. (ver Figura 35)

Ao receber as mensagens informativas (*Add load requirements*, *AttachTx* e *AttachRx*), o `FttseMaster` regista na sua base de dados (`SrbdNrdb`) os dados recebidos e verifica se estão reunidas as condições mínimas para o início da transmissão daquela `FttseStream`, isto é, se existe um produtor e consumidor definido para aquela `FttseStream`. Caso tal se verifique, aquele nó informa a rede através de uma mensagem em *broadcast* (*Add Message*), que inclui no seu conteúdo as informações relativamente à `FttseStream`, à semelhança do conteúdo da mensagem *Add load requirements*, adicionando a informação sobre o ID do produtor e do consumidor da `FttseStream`. Desta forma, o `FttseMaster` informa toda a rede que uma nova `FttseStream` se encontra disponível.

Através da receção da *Add Message*, as aplicações `FttseSlave` verificam se são produtoras ou consumidores da `FttseStream` indicada por aquela mensagem, e caso se verifique registam igualmente aquelas informações nas suas próprias bases de dados. (ver Figura 36)

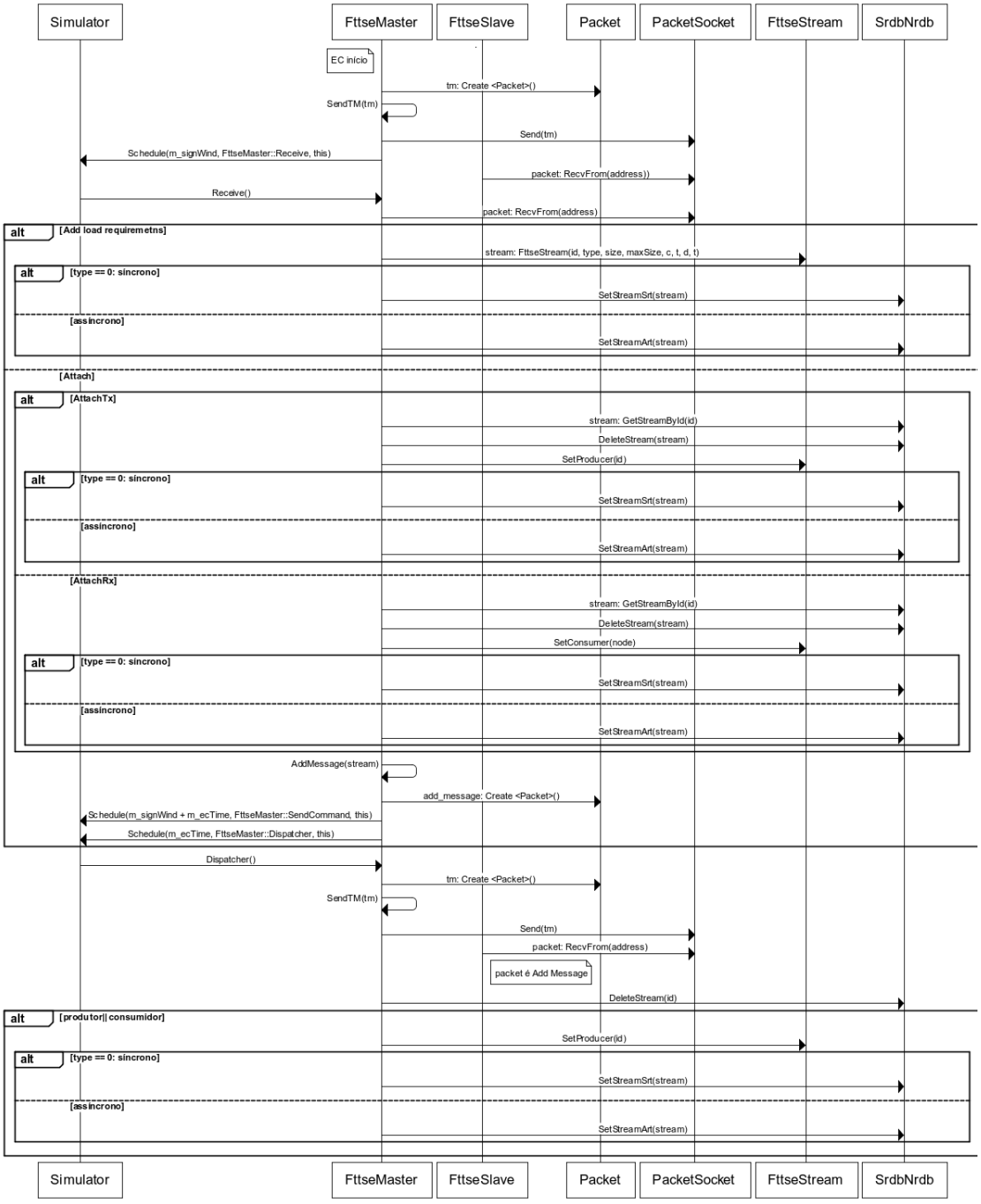


Figura 36: Diagrama de sequências do registo de `FttseStream`'s nas aplicações `FttseMaster` e `FttseSlave`

5.2.4 Geração de tráfego

Cada aplicação `FttseSlave` produtora está associada a um tipo de tráfego: síncrono ou assíncrono. Como descrito na secção 3.4, as aplicações síncronas são de natureza *time-triggered*, onde a ativação das suas mensagens são realizadas em instantes pré-definidos, e as aplicações assíncronas são de natureza *event-triggered*, onde as suas mensagens são ativadas por meio da geração de eventos, cujos instantes não são conhecidos.

A implementação da classe `FttseSlave` teve em consideração a natureza do tráfego gerado pela mesma, nomeadamente na forma como o tráfego é gerado. As aplicações `FttseSlave` produtoras de tráfego síncrono verificam, no início de cada EC, aquando da receção da *Trigger Message*, se já cumpriu o período definido para a sua `FttseStream`. Caso o período tenha sido cumprido, é gerada de imediato uma mensagem, de acordo com as características definidas para a sua `FttseStream`, inserindo a mesma, na sua fila de mensagens pendentes.

As aplicações `FttseSlave` produtoras de tráfego assíncrono verificam no início de cada EC, aquando da receção da TM, se já foi cumprido o *minimum inter-arrival time* (que tem a função de período), agendando em seguida um evento para um instante aleatório até 1 EC de duração. Este evento vai gerar uma mensagem assíncrona de acordo com as características definidas para a sua `FttseStream` inserindo a mesma, na sua fila de mensagens pendentes.

Desta forma, o tráfego síncrono é gerado no `FttseSlave` sempre no início do EC, quando cumprido o período definido para a `FttseStream`, aquando da chegada da TM. No caso do tráfego assíncrono, é introduzida alguma aleatoriedade, no sentido de simular o baixo determinismo inerente às comunicações *event-triggerent*.

A Figura 37 apresenta um diagrama de sequências simplificado do processo de geração de tráfego em aplicações `FttseSlave` de acordo com a natureza da mesma.

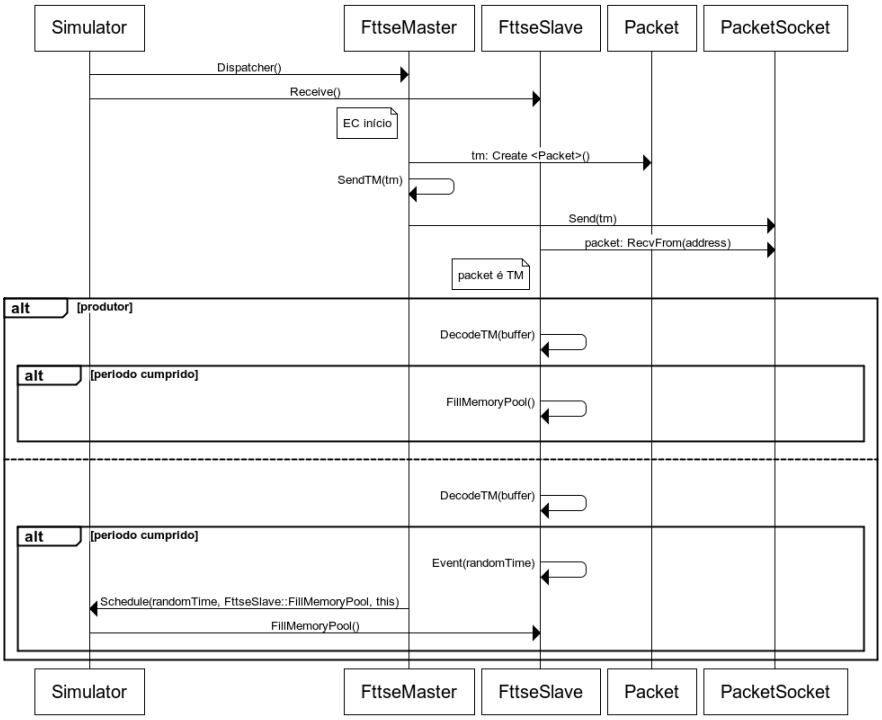


Figura 37: Diagrama de sequências simplificado do processo de geração de tráfego nas aplicações produtoras `FttseSlave`

5.2.5 Escalonamento de tráfego síncrono

A fase de escalonamento é das mais importantes deste protocolo, uma vez que é através deste procedimento que é realizada a coordenação do tráfego transmitido na rede. Neste capítulo é abordado o processo de escalonamento do tráfego síncrono realizado por parte do `FttseMaster` da rede.

Como já referido anteriormente neste relatório, o tráfego síncrono está diretamente relacionado com comunicações *time triggered*, ou seja, eventos ocorrem em instantes separados por intervalos de tempo predefinidos (ver secção 2.3).

Os diagramas (simplificados) apresentados na Figura 38 e Figura 39 demonstram a sequência de execuções que foram implementadas no NS-3 para simular este processo.

A construção da *Trigger Message* está diretamente relacionada com o escalonamento realizado previamente. As `FttseStream`s síncronas, mal são registadas por parte da aplicação `FttseMaster` são imediatamente integradas na tabela de `FttseStream`s desse tipo (`m_srt` da base de dados `SrdbNrdb`). O escalonamento síncrono inicia a análise a todas as `FttseStream`s armazenadas naquela tabela, no sentido de verificar, uma a uma, quais as que cumpriram o período associado à mesma, ou seja, aquelas que estão pendentes. Este ponto reflete a abordagem realizada no capítulo 3.5 e ilustrada na Figura 14. Uma vez verificado que

uma mensagem está pendente, é necessário avaliar se há necessidade de fragmentação da mesma. É importante referir que numa rede *Ethernet*, nenhuma *frame* pode ultrapassar o valor definido para o MTU (1450 bytes neste trabalho). No caso de o tamanho da mensagem ser superior ao MTU é realizada uma fragmentação em várias mensagens. Nesta situação, cada fragmento compreende um tamanho igual ao MTU, com exceção do último que poderá ter tamanho inferior. Por outro lado, se o tamanho da mensagem for inferior ao MTU resulta apenas em uma mensagem, mantendo o seu tamanho. Cada fragmento/mensagem é adicionado a uma lista de `FttseStream`s (`m_srtReady`).

Seguidamente, é gerada uma cópia da `m_srtReady` para uma lista temporária (`srt_ready_sorted`), que contém exatamente as mesmas `FttseStream`s. Esta lista é posteriormente ordenada de acordo com a política de escalonamento utilizada na rede, *Rate Monotonic* neste trabalho. (ver Figura 38)

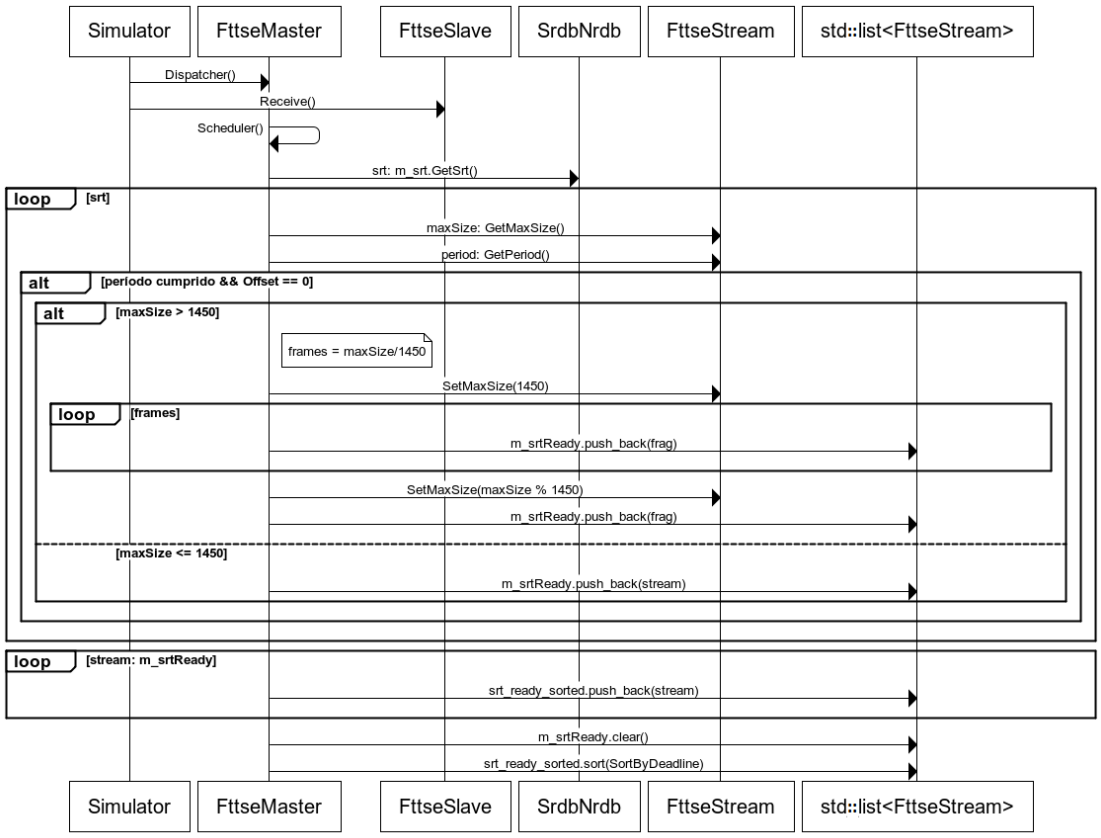


Figura 38: Diagrama de sequências simplificado da identificação de mensagens síncronas pendentes

Uma vez obtida uma lista com as mensagens síncronas pendentes ordenadas por prioridades, é necessário avaliar, mensagem a mensagem, desde a mais prioritária até à menos prioritária,

se é possível de ser escalonada tendo em consideração todo o seu percurso. A implementação realizada vai de encontro à abordagem efetuada no capítulo 3.5 e ilustrada na Figura 16. Desta forma, para a mensagem em análise é avaliado o escalonamento em cada uma das suas ligações. A transmissão da mensagem, desde o nó de origem até ao de destino, nunca pode exceder a largura de banda reservada para a *Synchronous Window*. A primeira ligação a ser avaliada é o *Uplink*, que liga o nó ao primeiro *switch* do percurso. No caso de ser possível de transmitir naquela ligação, a avaliação prossegue. Se o percurso da mensagem compreender mais do que um *switch*, o mesmo procedimento é realizado para cada uma das ligações que ligam os *switchs*, respeitando a ordem no percurso da mensagem. A última ligação a ser analisada é o *Downlink* que liga o *switch* mais distrante do percurso ao nó de destino. Contudo, é necessário ter em atenção que o escalonamento em cada ligação tem de considerar o instante em que é estimado que a transmissão termine na ligação imediatamente anterior, caso exista.

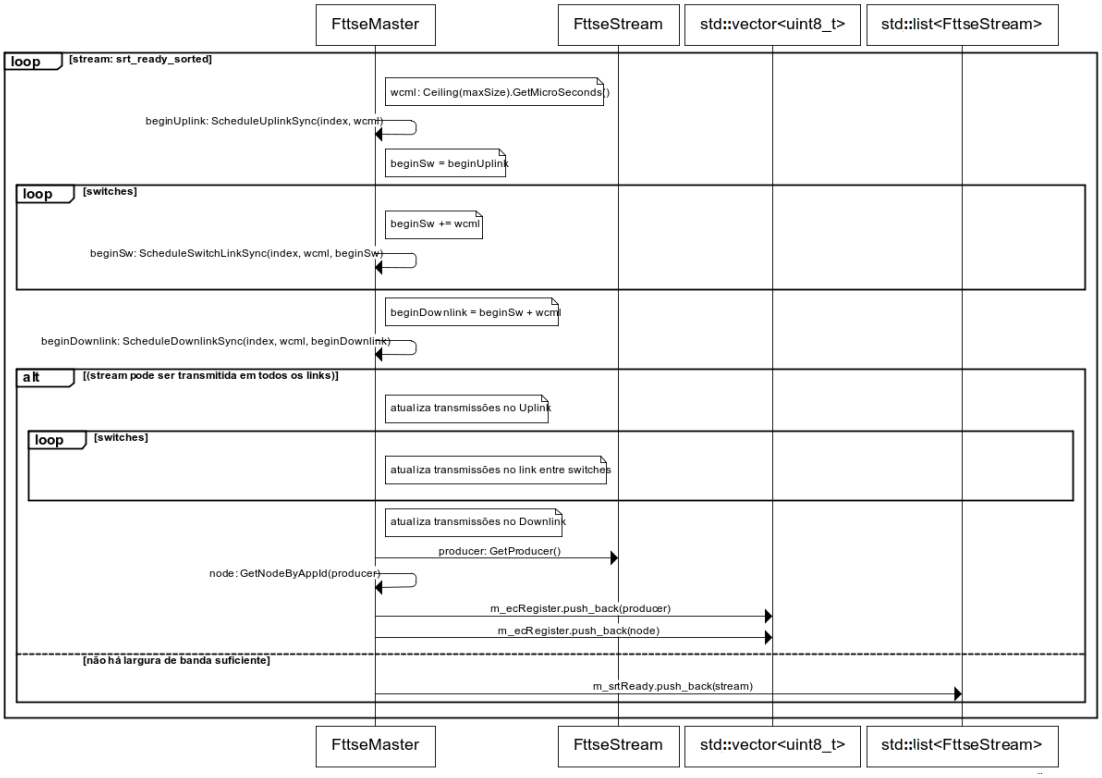


Figura 39: Diagrama de sequências simplificado do escalonamento de mensagens síncronas

Se, após a avaliação de todas as ligações do percurso da mensagem, se concluir que a transmissão não excederá a largura de banda definida para a *Synchronous Window*, a aplicação `FttseMaster` regista os intervalos de tempo estimados para a transmissão daquela mensagem em cada uma das ligações e insere informação acerca daquela mensagem no `m_ecRegister`. O registo dos intervalos de transmissão torna-se fundamental, uma vez

que, numa determinada ligação, não pode existir mais do que uma mensagem a ser transmitida ao mesmo tempo (no mesmo sentido). Caso em alguma ligação se verifica não ser possível a transmissão dentro da *Synchronous Window*, a mensagem é novamente inserida na lista de mensagens pendentes (`m_srtReady`), para poder ser considerada no processo de escalonamento do EC seguinte. (Figura 39)

5.2.6 Escalonamento de tráfego assíncrono

Neste capítulo é realizada a abordagem ao escalonamento de tráfego assíncrono. Ao contrário do que acontece com o tráfego síncrono, a ativação de mensagens assíncronas não está relacionada com a linha do tempo, mas sim com eventos. Desta forma, a única maneira de o `FttseMaster` obter informação sobre mensagens assíncronas pendentes, é através do mecanismo de *signalling.* (ver capítulo 3.4)

A Figura 40 apresenta um diagrama de sequências simplificado, ilustrando a implementação realizada para este mecanismo.

Após a receção da *Trigger Message*, a aplicação `FttseSlave` produtora de `FttseStream` assíncrona verifica se já cumpriu o *minimum inter-arrival time* definido para a sua `FttseStream`. Caso tal se verifique, é agendado um evento no simulador, através do método `Schedule` da classe `Simulator` para um tempo aleatório, até o máximo de 1 EC. Durante a execução desse evento, a aplicação informa a primeira aplicação existente no seu nó, através no método `PendingMessage`, que existem mensagens assíncronas pendentes originadas da sua aplicação.

No EC seguinte, após a receção da TM e dentro da *Signalling Window*, a primeira aplicação de cada nó envia para o *Master* uma *Signalling Message*, informando se existem mensagens assíncronas pendentes nas aplicações daquele nó. Ao receber uma *Signalling Message*, o `FttseMaster` interpreta-a e, caso o conteúdo da mensagem informe acerca de mensagens pendentes, informações sobre a mesma é inserida na lista `FttseStream`s assíncronas pendentes (`m_artReady`), fragmentada se for o seu tamanho for superior ao MTU.

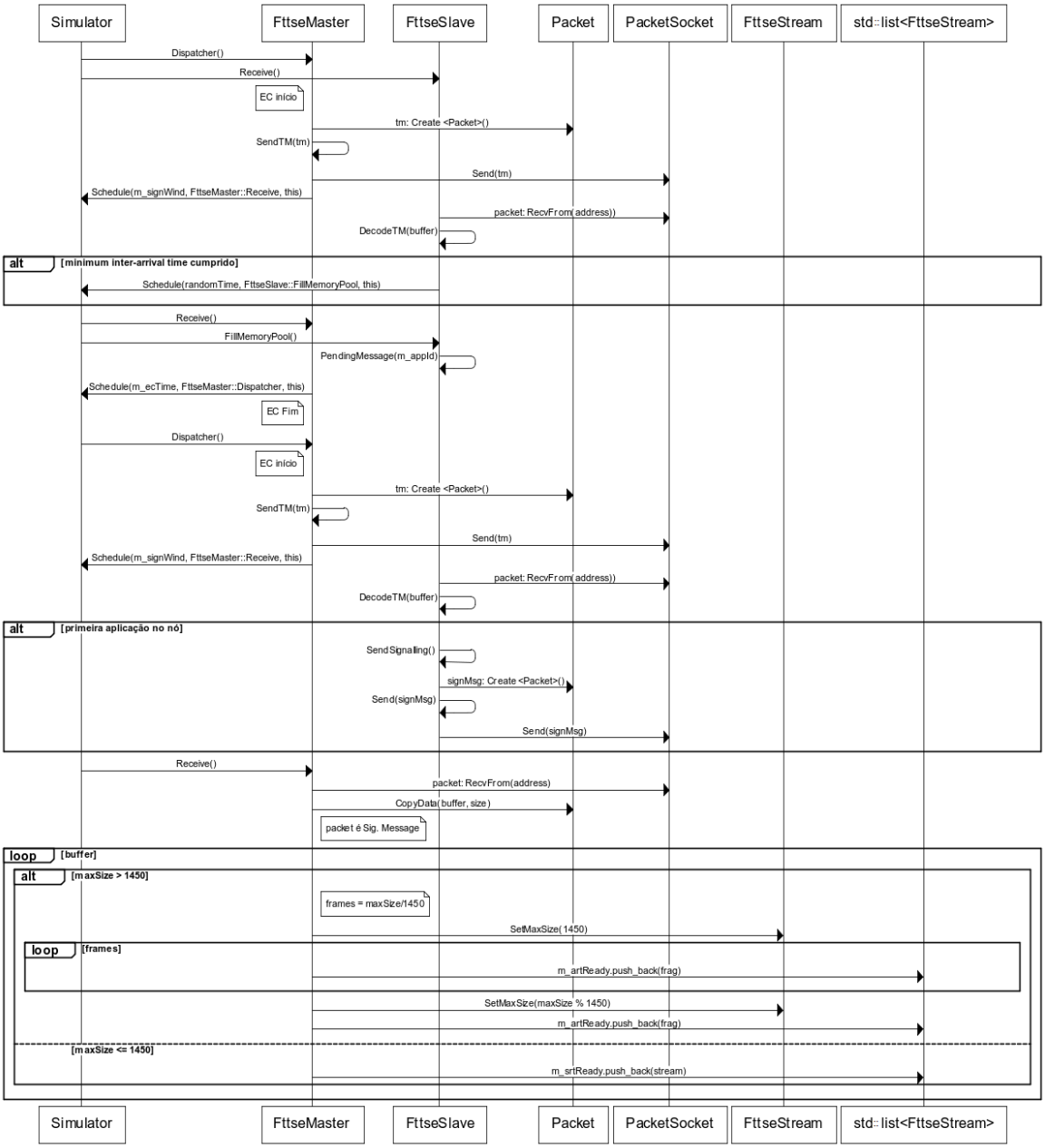


Figura 40: Diagrama de sequências simplificado da implementação do mecanismo de signalling

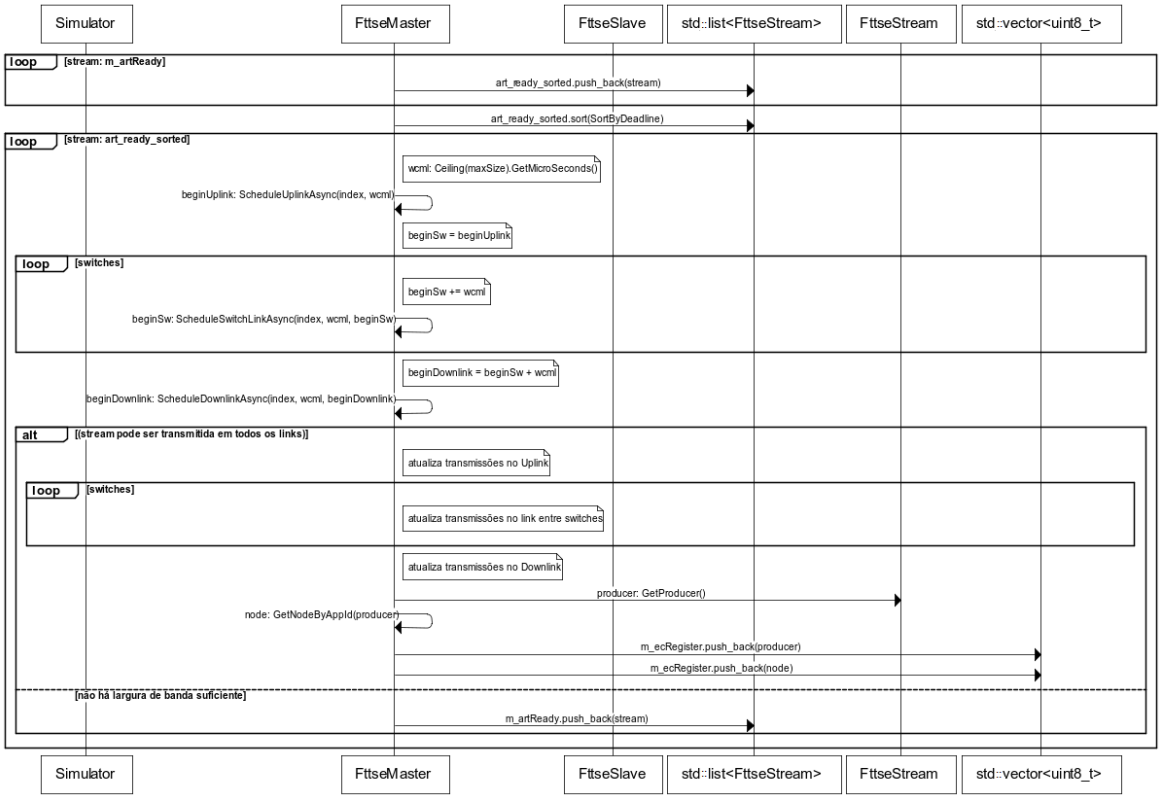


Figura 41: Diagrama de sequências simplificado do escalonamento de mensagens assíncronas

O processo ilustrado na Figura 41 é muito semelhante ao apresentado na Figura 39, relacionado com o escalonamento de mensagens síncronas. Uma vez que já estão definidas as `FttseStream`s assíncronas pendentes numa lista (`m_artReady`), o primeiro passo é criar uma cópia temporária da lista (`art_ready_sorted`) e proceder à sua ordenação por prioridade.

Seguidamente, cada mensagem é analisada no sentido de verificar se é possível ser escalonada naquele EC. Para isso, cada ligação do percurso da mensagem tem de ser avaliado, sendo um processo idêntico ao descrito no capítulo 5.2.5. A diferença está no limite associado ao escalonamento de mensagens assíncronas, onde as transmissões não podem exceder a largura de banda reservada para tráfego assíncrono, nem acontecer fora no próprio EC. As mensagens que forem possível de ser transmitidas são registadas no `m_exRegister`. No caso de existirem mensagens que não possam ser escalonadas, são inseridas novamente na lista de `FttseStream`s pendentes (`m_artReady`) para que no próximo EC sejam alvo do mesmo processo.

5.2.7 Construção da Trigger Message e envio de tráfego

Neste capítulo é abordada a forma como é construída a *Trigger Message* e como as aplicações `FttseSlave` procedem ao envio de tráfego ao analisar a mesma.

Nos capítulos 5.2.5 e 5.2.6, foi descrita a implementação realizada para o escalonamento quer de mensagens síncronas quer assíncronas. Comum a ambos os processos, a lista `m_ecRegister` é utilizada para registar as mensagens escalonadas pelo `FttseMaster` para aquele EC. É com base no `m_ecRegister` que a *Trigger Message* é construída. Toda a informação contida naquela lista é adicionada ao conteúdo da TM. Seguidamente, a aplicação `FttseMaster` procede ao seu envio.

Do lado dos `FttseSlave`'s, ao receberem a TM, é agendada a sua análise para o instante inicial da *Synchronous Window*. Durante a análise ao conteúdo da TM, cada aplicação produtora verifica se é produtora de alguma das mensagens contidas no *payload* da TM. Caso isso se verifique, é iniciado o processo de transmissão da mensagem. A mensagem é, então, retirada da fila de mensagens pendentes e enviada para o nó de destino.

O processo acima descrito encontra-se ilustrado no diagrama de sequências simplifica, na Figura 42

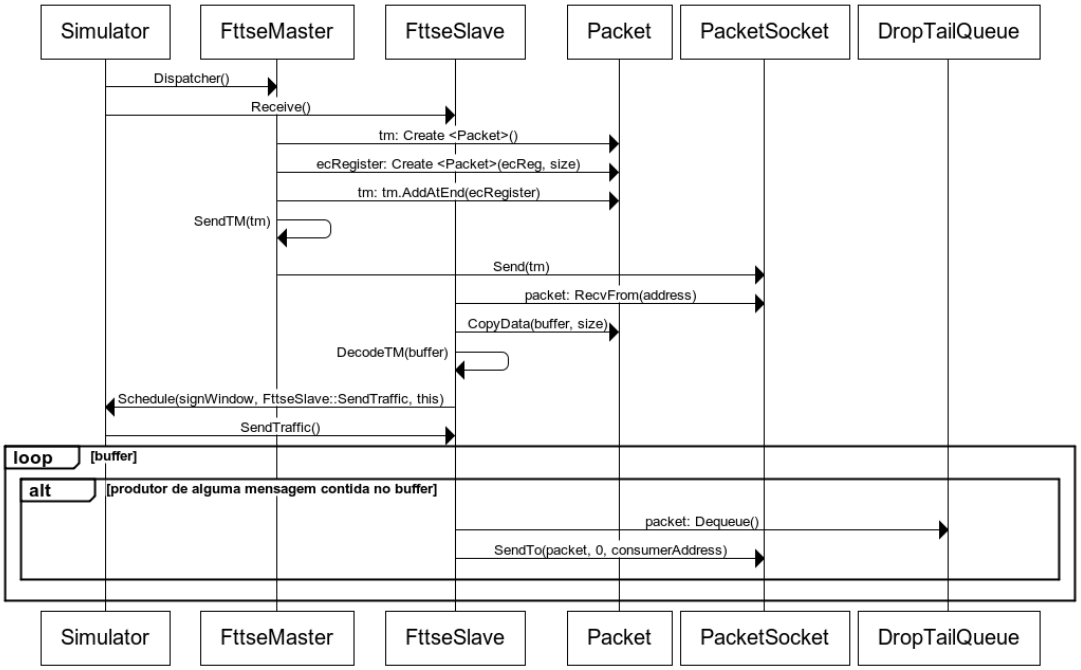


Figura 42: Diagrama de sequências simplificado da construção da Trigger Message e envio de tráfego

5.2.8 Cálculo de Offset de sincronização de aplicações P/D síncronas

Neste capítulo é abordada a implementação realizada para a sincronização de aplicações P/D do tipo síncrono. É importante relembrar a natureza *time-triggered* das aplicações síncronas, onde mensagens são geradas em instantes predefinidos de tempo. Desta forma, numa comunicação paralela distribuídas, a aplicação do nó remoto tem de estar sincronizada com a do nó local, considerando sempre o *Worst-Case Response Time* da aplicação do nó local.

A sincronização entre a aplicação `FttseSlave` do nó local e a aplicação `FttseSlave` do nó remoto é efetuada pela atribuição de um *Offset* de sincronização que é calculado pela aplicação `FttseMaster`, conforme a abordagem descrita no capítulo 3.6, sendo que a implementação realizada foi baseada nessa abordagem. Para o cálculo do *Offset* é necessário considerar o *Worst-Case Response Time* da transmissão da mensagem pela rede e *Worst-Case Response Time* da execução no processador remoto. O Algoritmo 1 descreve as instruções principais para a computação deste *Offset*.

Entre as linhas 7 e 37 do Algoritmo 1 são calculados e comparados o *resquest bound function* (`rbf`), que indica os requisitos máximos necessários para a transmissão da mensagem, e o *suply bound function* (`sbf`), que indica a capacidade mínima que a rede oferece num determinado período de tempo. As operações realizadas para o cálculo destes valores estão de acordo com a descrição do capítulo 3.6.1.

Entre as linhas 40 e 47 do Algoritmo um é calculado o pior caso para a execução de *threads* num processador, de acordo com a descrição do capítulo 3.6.2.

O *Offset* é obtido somando os *Elementary Cycles* correspondentes resultado do cálculo do *Worst-Case Response Time* da transmissão da mensagem pela rede e os *Elementary Cycles* correspondentes resultado do cálculo do *Worst-Case Response Time* do processamento da execução da *thread*.

Algoritmo 1

```
 1: streams = listaStreams
 2: fragmentos = listaFragmentos
 3: streams.sort(SortByDeadline)
 4: fragmentos.sort(SortByDeadline)
 5: loop fragmentos: i
 6:    if pertenceNoRemoto
 7:        while time < deadline && sbf < rbf
 8:        Wli = 0
 9:        Wri = 0
10:        Isi = 0
11:            loop fragmentos: j < i
12:                if partilhaSwitch
13:                    listaGi_j.insere(gi_j)
14:                    Wli += time/period_j * C_j
15:                    //identifica os k
16:                   loop fragmentos: k < j
17:                    if (!k_partilhaSwitch_i && k_partilhaSwitch_j)
18:                        lista_k.insere(k) //se k não existir em lista_k
19:                    end loop
20:                end if
21:            end loop
22:            listaGi_j.sort()
23:            z = time/ec
24:            loop listaGi_j: h
25:                Isi += h
26:            end loop
27:            Wli += Isi
28:            loop lista_k: h
29:                Wri += time/ec * C_k
30:            end loop
31:            rbf = C_i + (nSwitches*(gi_i)+1) + Wli + Wri
32:            sbf = (syncWindow/ec)*time
33:            if sbf >= rbf
34:                timeAst = time
35:            end if
36:            time++
37:        end while
38:        r_new = wcet_i
39:        r_old = 0
40:        while r_new != r_old && r_new < deadline
41:            r_old = r_new
42:            r_new = 0
43:            loop streams: j < i
44:                r_new += (r_old/period_j) * wcet_j
45:            end loop
46:            r_new += wcet_i
47:        end while
48:        RT = timeAst/ec
49:        RP = r_new/ec
50:        //stream: stream de i
51:        stream.SetOffset(RT + RP)
52:    end if
53: end loop
```

5.2.9 Processamento de tarefas

Neste capítulo é descrita a implementação realizada para o processamento de tarefas dentro de um nó, um dos objetivos deste trabalho.

Esta implementação, que permite a definição de um cenário holístico, é única no NS-3 e permite simular o mecanismo de escalonamento por prioridades, seguindo a política de escalonamento preemptiva *Rate Monotonic* (ver capítulo 2.4). A Figura 43 ilustra a implementação realizada no NS-3, onde uma tarefa (`Task`) com mais prioridade (`Task` 1) interrompe a execução da `Task` 2, que tem menos prioridade, para executar a sua tarefa. Aquando da finalização da execução da `Task` 1, a `Task` 2 (que se encontrava pendente) retoma a sua execução.

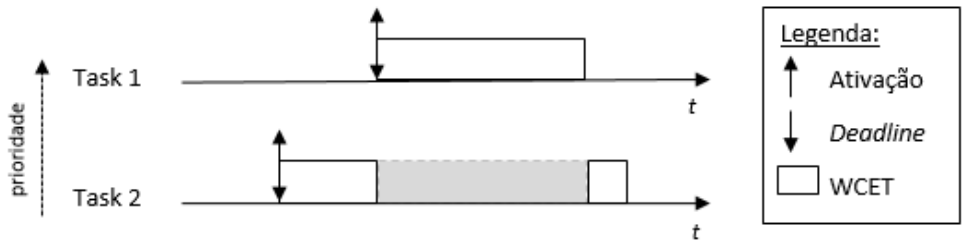


Figura 43: Implementação do mecanismo de escalonamento de tarefas por prioridades

A Figura 44 representa um diagrama de atividades que ilustra a implementação realizada.

Cada aplicação `FttseSlave` tem associada uma tarefa (`Task`), conforme representado na Figura 26. O processamento de `Task`s é realizado por aplicações `FttseSlave` consumidoras. Uma determinada aplicação consumidora aguarda até receber a totalidade da mensagem que está associada à `FttseStream` que consome. Quando tal se verifica, um evento é gerado iniciando de imediato o processo de concorrência pelo processador, por parte da `Task` associada (θ_i). O primeiro passo é verificar se existe alguma outra `Task`, proveniente de outra aplicação `FttseSlave`, em processamento naquele momento. Esta verificação é realizada analisando o estado da `Task` correspondente a cada uma das aplicações `FttseSlave` do nó a que pertence.

Se não existir nenhuma `Task` em execução, de imediato é iniciada a execução da `Task` θ_i. Quando terminado o processamento, é necessário averiguar se existem `Task`s pendentes. Se existirem, verifica qual a que tem mais prioridade e a aplicação a que pertence essa `Task` inicia o seu processamento.

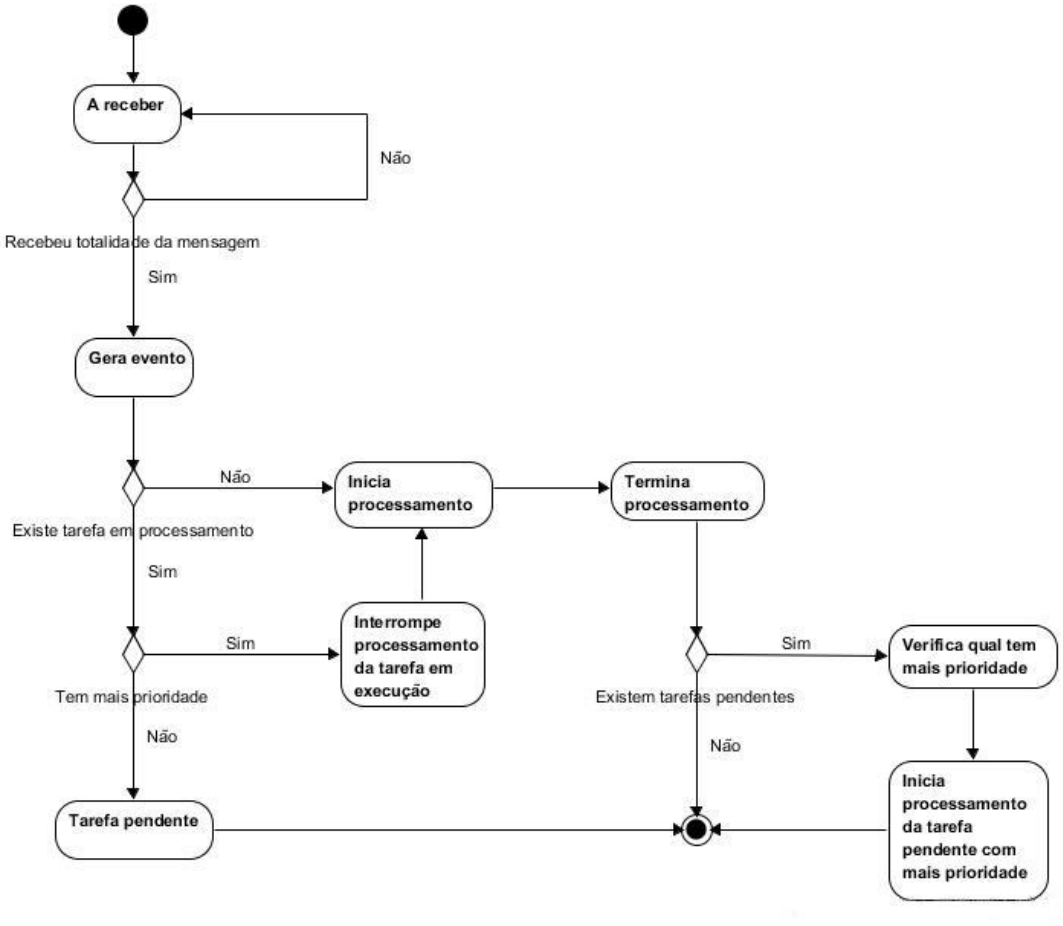


Figura 44: Diagrama de atividades do processamento de uma tarefa num nó de uma aplicação `FttseSlave`

No caso de, após a geração do evento, existir uma `Task` em processamento, é necessário verificar qual, entre ambos, tem mais prioridade. No caso de a `Task` em execução ter mais prioridade, θ_i atualiza o seu estado para pendente. Se se verificar o contrário, a `Task` em execução é interrompida e passa para o estado de pendente, e θ_i inicia o seu processamento. Quando terminar o processamento, identifica qual das `Task`s pendentes tem mais prioridade, e a respetiva aplicação inicia, então, a sua execução.

5.2.10 Exportação de resultados

Neste capítulo, é explicado como é realizada a exportação de resultados obtidos nas simulações utilizando a implementação realizada.

Para a apresentação do resultado das simulações, nomeadamente o *response time* de uma mensagem foram definidas duas soluções, que podem ser utilizadas em conjunto. A primeira está relacionada com a utilização de mensagens de `LOG`, uma das funcionalidades de registo que o NS-3 integra (ver capítulo 4.1). No entanto, esta solução apresenta os resultados de

todas as `FttseSlaves` no mesmo *output*, pelo que poderá dificultar a análise dos mesmos. Uma segunda solução foi desenvolvida para contornar as limitações apresentadas pelas mensagens de `LOG` do NS-3, uma vez que os resultados são separados e guardados em ficheiros de texto, havendo um ficheiro por cada `FttseStream`. Para tornar mais intuitiva a análise dos resultados, cada ficheiro é definido com um *filename* que incluiu o ID da `FttseStream`, o ID do nó produtor e o tamanho dos dados da `FttseSteam`. Além disso, a primeira linha de cada ficheiro tem na sua primeira linha mais informações, quer sobre a `FttseStream` quer sobre a própria rede, como por exemplo, o *Worst-Case Message Length*, o período, o *deadline*, a duração do *Elementary Cycle* e a percentagem reservada para a *Synchronous Window*.

Obtenção do *Response Time*

Na implementação realizada, existem duas formas para a obtenção do *Response Time* (RT) da tarefa de uma aplicação, que estão relacionadas com o tipo de comunicação que as aplicações realizam: sequencial ou de acordo com o paradigma *Fork-Join Parallel/Distibuted.*

Nas aplicações sequenciais, um RT é definido pelo tempo desde o momento em que a aplicação `FttseSlave` produtora gera uma mensagem, sendo registado esse instante como o instante inicial, e a insere na fila de mensagens pendentes até que a respetiva aplicação `FttseSlave` consumidora receba a mensagem e processe os dados recebidos. O instante final do RT é o momento em que é finalizado o processamento dos dados. O valor do RT é obtido pela diferença de tempo entre o instante final e o inicial da comunicação. A Figura 45 ilustra a forma como é registado o RT na comunicação de uma aplicação sequencial, tomando como exemplo, a comunicação de uma aplicação sequencial `FttseSlave` 1 com uma aplicação `FttseSlave` 2.

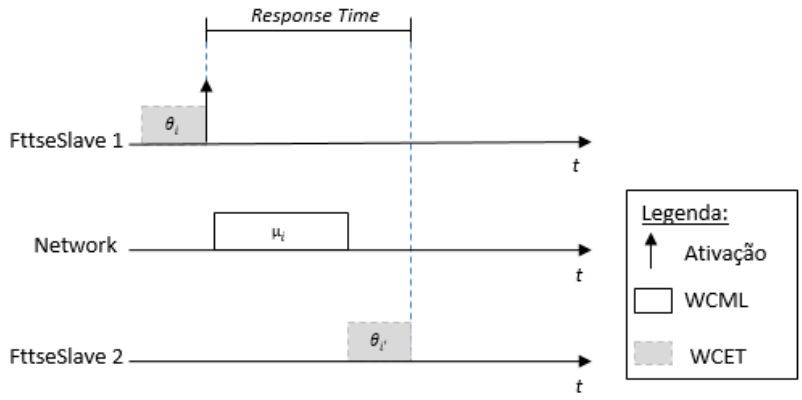


Figura 45: Obtenção de Response Time em aplicações sequenciais

Nas aplicações que realizam comunicações paralelas distribuídas, a obtenção do RT é um pouco mais complexa. Da mesma forma que nas aplicações sequenciais, a contabilização do RT começa no instante em que a aplicação `FttseSlave` produtora gera uma mensagem, sendo este o instante inicial. Essa mensagem é enviada para um nó remoto, recebida por uma aplicação `FttseSlave` consumidora, que processa os dados recebidos remotamente. Após o processamento dos dados a aplicação `FttseSlave` do nó remoto envia uma mensagem de volta à aplicação `FttseSlave` do nó local, sendo registado o instante em que a totalidade da mensagem é recebida. Este é o instante final da comunicação. O valor do RT é obtido pela diferença de tempo entre o instante final e o instante inicial, sendo adicionado ao resultado um custo (atribuído na aplicação `FttseSlave` produtora do nó local). A Figura 46 ilustra a forma como é registado o RT na comunicação de uma aplicação paralela distribuída, tomando como exemplo, a comunicação de uma aplicação paralela distribuída `FttseSlave` 1 com uma aplicação `FttseSlave` 2.

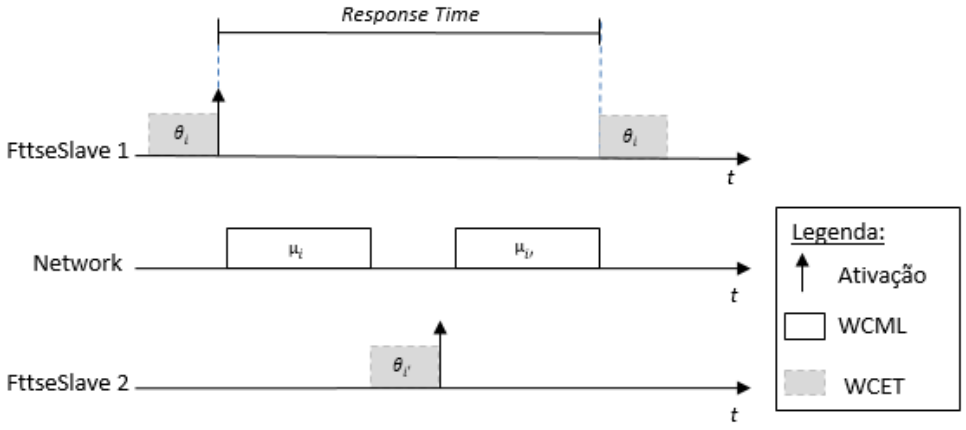


Figura 46: Obtenção de Response Time em aplicações paralelas distribuídas

6 Resultados e validação

Neste capítulo são apresentados os resultados de simulações realizadas no NS-3, no âmbito deste projeto, realizando uma adequada análise, comparando diferentes cenários e identificando as principais variações de resultados entre os mesmos.

6.1 Validação (FTT-SE)

Nesta secção são apresentadas simulações no sentido de validar a implementação da rede FTT-SE e estudar o seu comportamento em várias configurações.

6.1.1 Tráfego síncrono

As simulações apresentadas neste capítulo foram baseadas em simulações realizadas em [18], sofrendo apenas ligeiras adaptações. No artigo referido são realizadas experiências no sentido de analisar a influência de várias topologias de *switched Ethernet* em veículos. Será usada a rede FTT-SE, que oferece garantias no cumprimento dos requisitos das comunicações de tempo-real. As aplicações executadas em veículos exigem um bom desempenho por parte da rede, de forma a cumprir todos os requisitos, como por exemplo, o cumprimento dos *deadlines* das mesmas.

Nas simulações apresentadas neste capítulo apenas serão usadas mensagens síncronas.

6.1.1.1 Configuração da rede

Nestas experiências são usadas três topologias de rede: estrela, *daisy chain* e árvore. Todas estas topologias pode ser usadas em redes *switched Ethernet*. Todas as ligações entre os vários nós da rede suportam a velocidade de 100 Mbits/s.

A topologia em estrela (Figura 47) baseia-se na existência de um único *switch* que interliga todos os nós finais. Esta topologia tem como vantagens o baixo custo de instalação e manutenção.

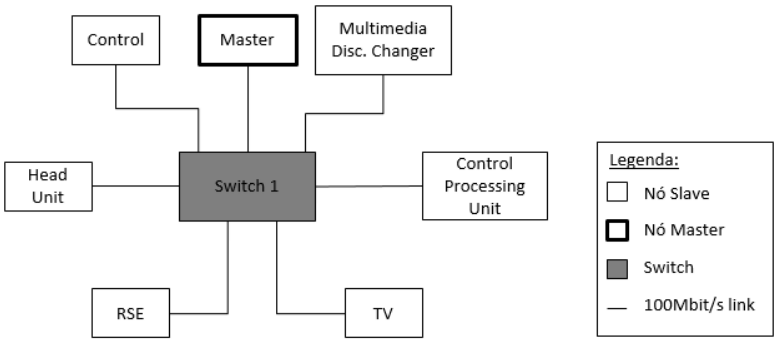


Figura 47: Topologia em estrela

A topologia *daisy chain* (Figura 48) caracteriza-se por ter os *switches* em cadeia. Para estas experiências foram usados três *switches*. A principal vantagem está relacionada com a facilidade de configuração da rede usando *switches* com um número reduzido de portas.

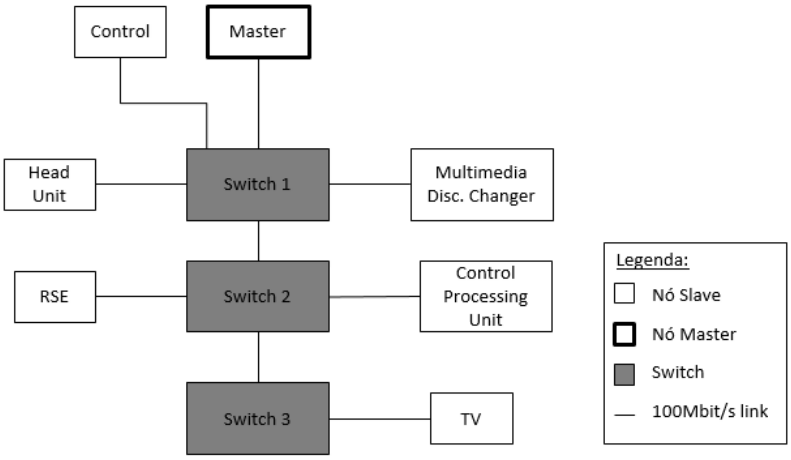


Figura 48: Topologia daisy chain

A topologia em árvore (Figura 49) resulta de uma mistura das duas topologias anteriores e, deste modo, existe um balanceamento entre as vantagens e desvantagens de ambas. É também uma topologia mais equilibrada, uma vez que também se adapta melhor à estrutura física das aplicações, diminuindo os custos e o peso da cablagem.

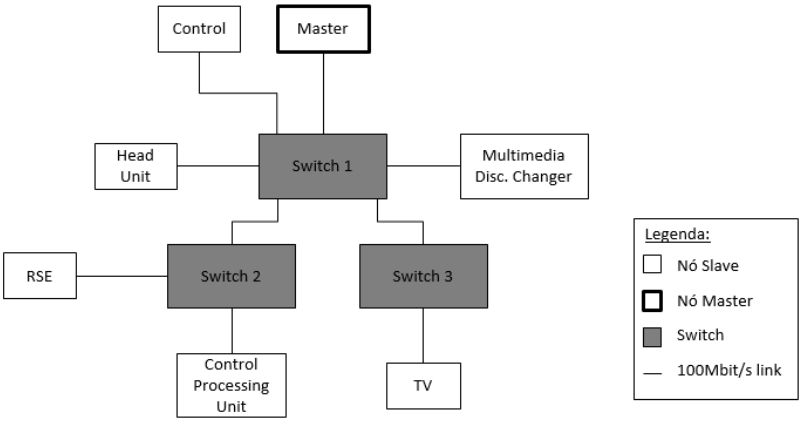


Figura 49: Topologia em árvore

As experiências compreendem a utilização de aplicações sequenciais que produzem *streams* típicas em veículos, com elevadas exigências relativamente à largura de banda. Quatro categorias de *streams* são trocadas entre os nós da rede: controlo, navegação, multimédia vídeo e áudio, e TV vídeo e áudio. É importante relembrar que aplicações sequenciais realizam comunicações que compreendem o envio de mensagens uma aplicação de um nó remoto, seguido da execução de uma tarefa (remotamente). Porém, como neste capítulo apenas é pretendido validar a rede, não é considerada a execução de tarefas remotamente.

As *streams* de controlo são produzidas pelas aplicações sequenciais μ_1 e μ_2 e ambas têm origem no nó "Control" e produzem mensagens de 64 *bytes*, que correspondem a um *Worst-Case Message Length* (WCML) de 7 µs (i.e. numa rede com velocidade de 100 Mbits/s, uma mensagem de 64 *bytes* demora 7 µs a ser transferida numa ligação), com um período de 11 ECs. No caso da aplicação μ_1, a *stream* tem como destino uma aplicação consumidora no nó "Control Processing Unit". Já a μ_2 tem como destino uma aplicação consumidora no nó "RSE" (*Rear Seat Entertainment*).

A aplicação sequencial μ_3, executada a partir do nó "Head Unit", envia para uma aplicação no nó "RSE" 5000 *bytes* (equivale a um WCML de 406 µs) a cada 100 ECs. As *streams* de multimédia, com mensagens de 1400 *bytes* (equivale a um WCML de 114 µs) são enviadas a partir duma aplicação do nó "Multimedia Disc. Changer" para uma aplicação no "RSE", sendo que a aplicação μ_4 (Vídeo) tem 1 EC como período e a aplicação μ_5 (Áudio) tem período de 2 ECs. Por fim, as aplicações μ_6 (Vídeo) e μ_7 (Áudio) produzem dados de TV, com mensagens de 1400 *bytes* (equivale a um WCML de 114 µs) e enviam desde o nó "TV" para uma aplicação instalada no nó "Head Unit", sendo que as mensagens de vídeo tem um período de 2 ECs e as de Áudio tem um período de 3 ECs. As características das aplicações encontram-se resumidas na Tabela 3.

Tabela 3: Características das aplicações sequenciais síncronas

Aplicação	Categoria	Período (ECs)	Tamanho (bytes)	WCML	Origem	Destino
μ_1	Controlo	11	64	7 μs	Control	CPU
μ_2	Controlo	11	64	7 μs	Control	RSE
μ_3	Navegação	100	5000	406 μs	Head Unit	RSE
μ_4	MM Vídeo	1	1400	114 μs	Mult. Disc	RSE
μ_5	MM Áudio	2	1400	114 μs	Mult. Disc	RSE
μ_6	TV Vídeo	2	1400	114 μs	TV	Head-Unit
μ_7	TV Áudio	3	1400	114 μs	TV	Head-Unit

6.1.1.2 Comparação entre topologias estrela, *daisy chain* e árvore utilizando tráfego síncrono

Relativamente às características do protocolo FTT-SE utilizadas para esta experiência, foram definidos 1000 μs como duração do *Elementary Cycle*, 60% do EC (540 μs) foi reservado para a *Synchronous Window* e 100 μs são reservados para a *Signalling Window*. Todas as *streams* utilizadas nestas simulações são do tipo síncrono. A Tabela 4 resume as características da rede utilizadas nestas experiências

Tabela 4: Características da rede FTT-SE em simulações com aplicações sequenciais síncronas em várias topologias de rede

Elementary Cycle	**1000 μs**
Synchronous Window	60% (540 μs)
Asynchronous Window	40% (360 μs)
Singnalling Window	100 μs

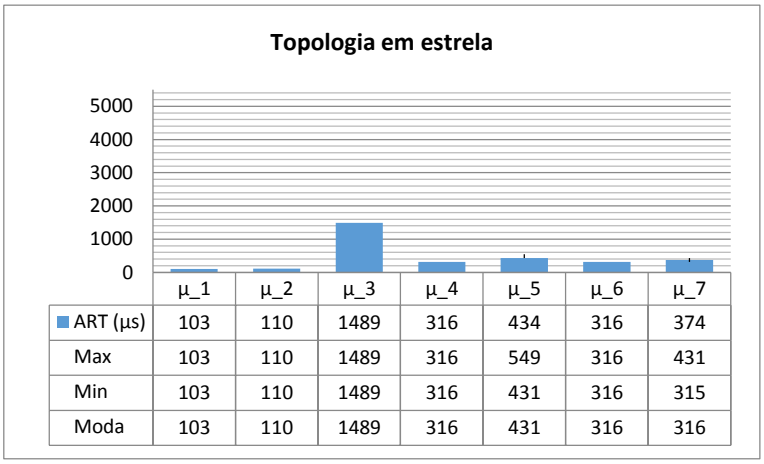
No processo de recolha e análise de resultados é importante ter em consideração dois aspetos. O primeiro aspeto é determinar a média dos *response times* (ART) obtidos pelas aplicações sequenciais. Um *response time* (RT) é o tempo desde o instante em que uma mensagem foi gerada pela aplicação produtora no nó local até ao instante que a aplicação consumidora da mensagem no nó remoto recebe a totalidade da mesma. O segundo aspeto é enquadrar o valor da ART obtida com número de ECs correspondente, isto é, identificar o limite superior em número inteiro de ECs. Para exemplificar, numa experiência com um EC de 1000 μs, um ART de 600 μs revela-se inferior ao tempo de 1 EC, sendo esse o seu limite superior.

A Figura 50 apresenta os resultados da simulação utilizando a topologia em estrela. A aplicação μ_1 obteve uma ART de 103 μs, a aplicação μ_2 de 110 μs, a aplicação μ_3 de 1489 μs, a aplicação

μ_4 de 316 µs, a aplicação μ_5 de 434 µs, a aplicação μ_6 de 316 µs e, por fim, a aplicação μ_7 obteve uma ART de 374 µs. Todas as aplicações obtiveram ARTs inferiores ao tempo de 1 EC (1000 µs), com exceção da aplicação μ_3, onde a sua ART revelou-se inferior ao tempo de 2 ECs (2000 µs).

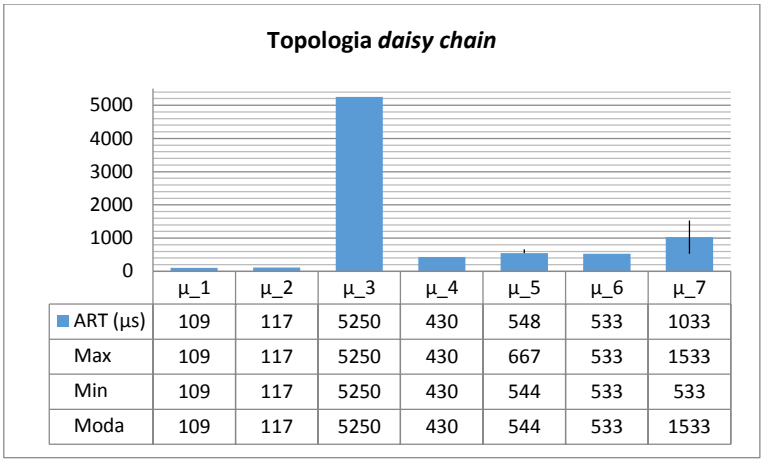
A Figura 51 apresenta os resultados da simulação numa topologia *daisy chain*. Neste caso, a aplicação μ_1 obteve ART de 109 µs, a aplicação μ_2 de 117 µs, a aplicação μ_3 de 5250 µs, a aplicação μ_4 de 430 µs, a aplicação μ_5 de 548 µs, a aplicação μ_6 de 533 µs e a aplicação μ_7 de 1033 µs . As ARTs das aplicações μ_1, μ_2, μ_4, μ_5 e μ_6 verificaram-se inferiores ao tempo de 1 EC (1000 µs), já a ART da aplicação μ_7 é inferior a 2 ECs (2000 µs) e a ART da aplicação μ_3 é inferior a 6 ECs (6000 µs).

A Figura 52 apresenta os resultados da simulação numa topologia em árvore. A aplicação μ_1 obteve uma ART de 109 µs, a aplicação μ_2 de 117 µs, a aplicação μ_3 de 5250 µs, a aplicação μ_4 de 430 µs, a aplicação μ_5 de 548 µs, a aplicação μ_6 de 425 µs e a aplicação μ_7 de 482 µs. Todas as aplicações obtiveram ARTs inferiores ao tempo de 1 EC (1000 µs), com exceção da aplicação μ_3, que obteve uma ART inferior ao tempo de 6 ECs (6000 µs).



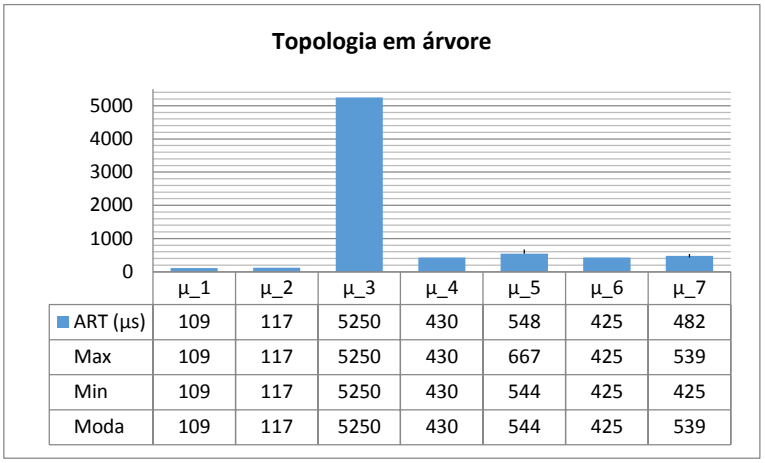
	µ_1	µ_2	µ_3	µ_4	µ_5	µ_6	µ_7
ART (µs)	103	110	1489	316	434	316	374
Max	103	110	1489	316	549	316	431
Min	103	110	1489	316	431	316	315
Moda	103	110	1489	316	431	316	316

Figura 50: Resultados da simulação com tráfego síncrono usando topologia em estrela com EC de 1000 µs, 60% de Synchronous Window e 100 µs de Signalling Window



	μ_1	μ_2	μ_3	μ_4	μ_5	μ_6	μ_7
ART (μs)	109	117	5250	430	548	533	1033
Max	109	117	5250	430	667	533	1533
Min	109	117	5250	430	544	533	533
Moda	109	117	5250	430	544	533	1533

Figura 51: Resultados da simulação com tráfego síncrono usando topologia daisy chain com EC de 1000 μs, 60% de Synchronous Window e 100 μs de Signalling Window



	μ_1	μ_2	μ_3	μ_4	μ_5	μ_6	μ_7
ART (μs)	109	117	5250	430	548	425	482
Max	109	117	5250	430	667	425	539
Min	109	117	5250	430	544	425	425
Moda	109	117	5250	430	544	425	539

Figura 52: Resultados da simulação com tráfego síncrono usando topologia em árvore com EC de 1000 μs, 60% de Synchronous Window e 100 μs de Signalling Window

Comparando os resultados das três simulações, podem-se identificar algumas diferenças entre cada uma delas, principalmente no que diz respeito às *streams* de navegação e TV áudio. Começando por analisar a *stream* de navegação (aplicação μ_3), pode-se verificar que nas topologias *daisy chain* (Figura 51) e em árvore (Figura 52) os resultados são significativamente mais elevados do que na topologia em estrela (Figura 50). Isto acontece devido ao facto de, na topologia em estrela (Figura 47), as mensagens apenas terem de percorrer duas ligações (o *Uplink* que liga o "Head Unit" ao *switch* e o *Downlink* que liga o *switch* ao "RSE"), e nas restantes topologias (ver Figura 48 e Figura 49), para além daquelas ligações, existe ainda a ligação entre o *switch 1* e o *switch 2* para percorrer, que fica sobrecarregado. Consequentemente, menos mensagens provenientes da aplicação de navegação podem ser escalonadas num *Elementary Cycle*, pelo que o seu *response time* será maior.

O número de ligações também está na origem das variações dos resultados referentes à *stream* de TV áudio, uma vez que na topologia em estrela apenas existem duas ligações a percorrer, na topologia em árvore existem três ligações e na topologia *daisy chain* existem quatro ligações. No entanto, analisando os resultados da simulação na topologia *daisy chain* é possível verificar uma diferença significativa entre os *response times* das *streams* TV vídeo (aplicação μ_6) e TV áudio (aplicação μ_7), embora ambas tenham o mesmo nó de origem e de destino. Esta diferença deve-se ao facto de, quando ambas são geradas no mesmo *Elementary Cycle*, a *stream* TV vídeo é escalonada em primeiro, por ter mais prioridade, e já não existir largura de banda suficiente para a transmissão da mensagem da TV áudio no mesmo *Elementary Cycle* sendo, por isso, adiado o seu escalonamento para o EC seguinte.

Seguidamente, são apresentados e analisados histogramas referentes às aplicações onde se verificaram variações de *response times* (RTs) registados nas simulações. Pela natureza síncrona, não existe grande diversidade de RTs obtidos, uma vez que as mensagens das aplicações são sempre geradas em instantes pré-definidos. Desta forma, apenas foi registada variabilidade nos RTs das aplicações μ_5 e μ_7.

A Figura 53 mostra o histograma dos RTs da aplicação μ_5 obtidos durante a simulação na topologia em estrela (ver Figura 47). É possível verificar que 98% dos RT foram de 431 μs e apenas 2% de 549 μs.

A Figura 55 mostra o histograma dos RTs da aplicação μ_7 obtidos durante a simulação na topologia em estrela. Neste caso, 50% dos RT foram de 316 μs e outros 50% foram de 431 μs.

Na Figura 54 é apresentado o histograma dos RTs da aplicação μ_5 obtidos durante a simulação na topologia *daisy chain* (ver Figura 48). 98% dos RT foram de 544 μs e 2% de 677 μs.

A Figura 56 mostra o histograma dos RTs da aplicação μ_7 obtidos durante a simulação na topologia *daisy chain*. Também, nesta topologia, se registaram dois resultados com uma frequência de 50%: 533 μs e 1533 μs.

Na Figura 57 é apresentado o histograma dos RTs da aplicação μ_5 obtidos durante a simulação na topologia em árvore (ver Figura 49). 98% das vezes foram obtidos RT de 544 μs e 2% de 677 μs.

A Figura 58 mostra o histograma dos RT da aplicação μ_7 obtidos durante a simulação na topologia em árvore. Também foram obtidos 50% de RT de 533 μs e 50% de 1533 μs.

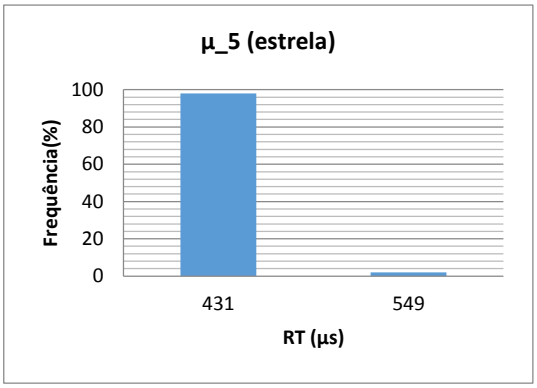


Figura 53: Histograma de μ_5 (síncrono) *na topologia em estrela com EC=1000 µs e SW=60%*

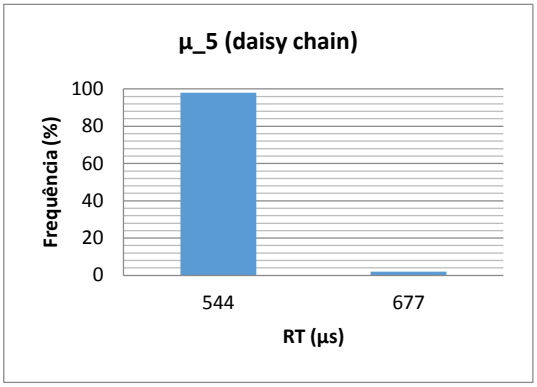


Figura 54: Histograma de μ_5 (síncrono) *na topologia daisy chain com EC=1000 µs e SW=60%*

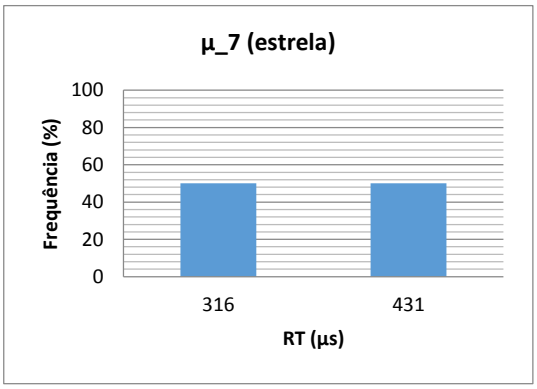


Figura 55: Histograma de μ_7 (síncrono) *na topologia em estrela com EC=1000 µs e SW=60%*

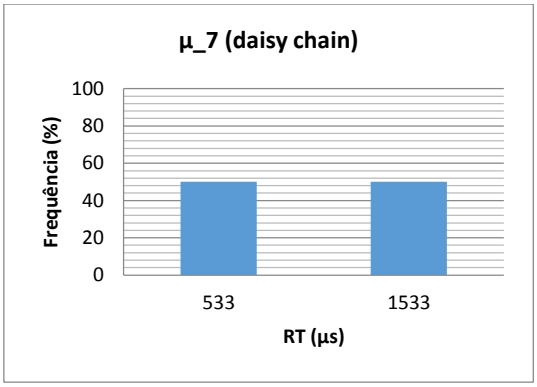


Figura 56: Histograma de μ_7 (síncrono) *na topologia daisy chain com EC=1000 µs e SW=60%*

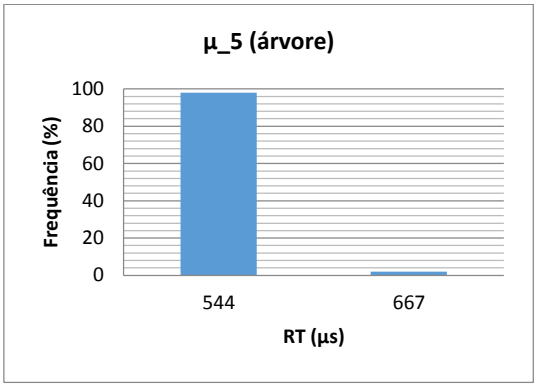


Figura 57: Histograma de μ_5 (síncrono) *na topologia em árvore com EC=1000 µs e SW de 60%*

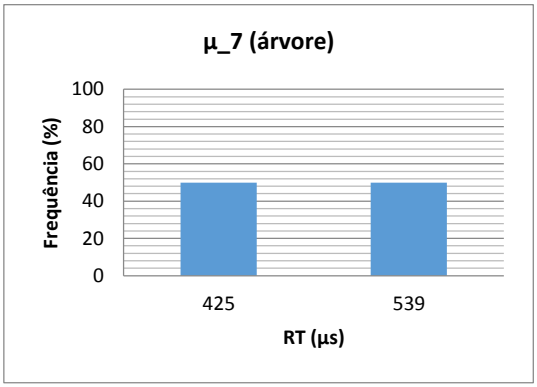


Figura 58: Histograma de μ_7 (síncrono) *na topologia em árvore com EC=1000 µs e SW=60%*

Os resultados demonstram que os *response times* obtidos com tráfego síncrono não revelam muita diversidade. Isto acontece precisamente porque, no tráfego síncrono, a geração do mesmo é realizada sempre que é recebida a *Trigger Message* (ver capítulo 5.2.4). Desta forma, qualquer variabilidade apenas se pode justificar por influência da rede, ou da execução de tarefas remotamente. Como nesta experiência apenas se está a considerar a rede, não havendo execuções remotas, qualquer variabilidade causada é resultado da transmissão de mensagens na própria rede.

Analisando os resultados, verifica-se que, quer para a aplicação μ_5, quer para a aplicação μ_7, as variações registadas são muito semelhantes em cada uma das topologias. No caso de μ_5, a variação deve-se à interferência com o tráfego das aplicações de controlo (μ_1 e μ_2) onde ligações são partilhadas (ver Figura 47, Figura 48 e Figura 49). Da mesma forma, a aplicação μ_7 regista variações devido à interferência com o tráfego produzido pela aplicação μ_6, cujas mensagens têm o mesmo percurso e coincidem em 50% das vezes, devido aos seus períodos. Contudo, é importante referir que, no caso da aplicação μ_7, na topologia *daisy chain*, a interferência registada tem como consequência o atraso de 1 EC em metade das transmissões, uma vez que o percurso das mensagens desta aplicação naquela topologia compreende mais ligações, e quando coincide a sua transmissão num EC com o tráfego proveniente da aplicação μ_6, devido aos seus períodos, não existe largura de banda suficiente para se efetuar ambas as transmissões num único. Como as mensagens da aplicação μ_7 têm menor prioridade, o seu escalonamento é adiado para o EC seguinte.

6.1.1.3 Variação da duração do Elementary Cycle com tráfego síncrono

Uma vez que já foram analisadas as diferenças entre as topologias estrela, *daisy chain* e árvore no capítulo anterior, as próximas simulações são fixadas na topologia em árvore pelo facto de a mesma apresentar um equilíbrio entre as vantagens quer da topologia em estrela e *daisy chain*, e também por ser uma topologia que oferece adaptabilidade à estrutura físicas das aplicações e baixos custos.

Nesta experiência, foram utilizadas exatamente as mesmas aplicações do que na experiência anterior (ver Tabela 3), fazendo variar apenas as características da rede, nomeadamente no que diz respeito à duração do *Elementary Cycle*, mantendo a *Synchronous Window* definida a 60% e 100 µs para a *Signalling Window*. Nas simulações da secção 6.1.1.2 foi utilizado um EC de 1000 µs. Nesta experiência, são utilizados ECs de 1500 µs e 2000 µs. É importante referir que, também houve a tentativa de realizar simulações com um EC de 500 µs, no entanto, o mesmo revelou-se muito curto, tendo algumas aplicações falhado o seu *deadline*, uma vez que

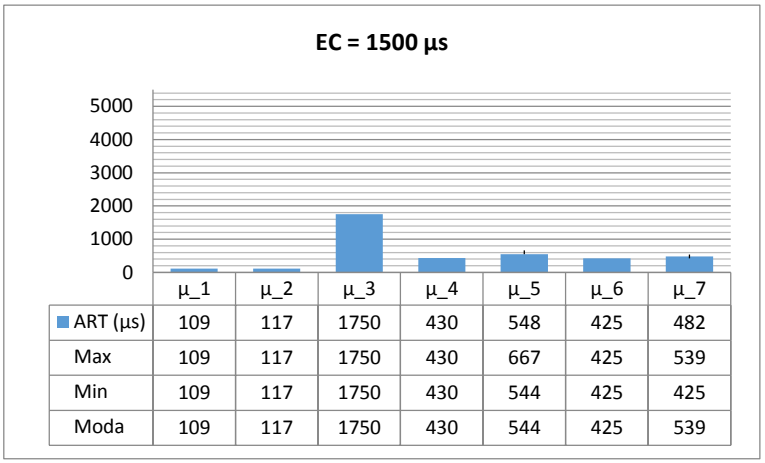
as aplicações μ_4, μ_5, μ_6 e μ_7 têm períodos bastante baixos, e com um EC tão curto não existe largura de banda suficiente para o escalonamento das mensagens de todas as aplicações.

Tabela 5: Características da rede FTT-SE em simulações com aplicações sequenciais síncronas com vários Elementary Cycles

Elementary Cycle	**1500 µs**	**2000 µs**
Synchronous Window	60% (840 µs)	60% (1140 µs)
Asynchronous Window	40% (560 µs)	40% (760 µs)
Singnalling Window	100 µs	100 µs

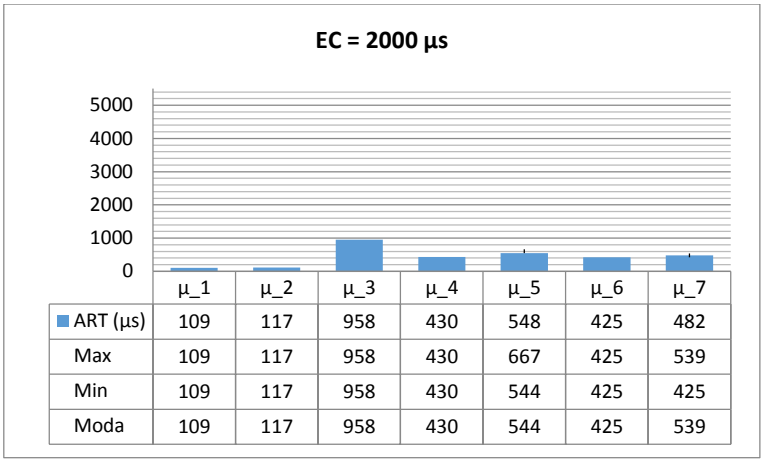
Na Figura 59 são apresentados os resultados da simulação utilizando um EC de 1500 µs. A aplicação μ_1 obteve uma ART de 109 µs, a aplicação μ_2 de 117 µs, a aplicação μ_3 de 1750 µs, a aplicação μ_4 de 430 µs, a aplicação μ_5 de 548 µs, a aplicação μ_6 de 425 µs e a aplicação μ_7 de 482 µs. Todas as aplicações obtiveram ARTs inferiores ao tempo de 1 EC (1500 µs), com exceção da aplicação μ_3, que obteve uma ART inferior ao tempo de 2 ECs (3000 µs).

Já a Figura 60 apresenta os resultados da simulação com EC definido a 2000 µs. A aplicação μ_1 obteve uma ART de 109 µs, a aplicação μ_2 de 117 µs, a aplicação μ_3 de 958 µs, a aplicação μ_4 de 430 µs, a aplicação μ_5 de 548 µs, a aplicação μ_6 de 425 µs e a aplicação μ_7 de 482 µs. Nesta simulação, todas as aplicações registaram ARTs inferiores ao tempo de 1 EC (2000 µs).



	µ_1	µ_2	µ_3	µ_4	µ_5	µ_6	µ_7
ART (µs)	109	117	1750	430	548	425	482
Max	109	117	1750	430	667	425	539
Min	109	117	1750	430	544	425	425
Moda	109	117	1750	430	544	425	539

Figura 59: Resultados da simulação com tráfego síncrono usando topologia em árvore com EC de 1500 µs, 60% de Synchronous Window e 100 µs de Signalling Window



	µ_1	µ_2	µ_3	µ_4	µ_5	µ_6	µ_7
ART (µs)	109	117	958	430	548	425	482
Max	109	117	958	430	667	425	539
Min	109	117	958	430	544	425	425
Moda	109	117	958	430	544	425	539

Figura 60. Resultados da simulação com tráfego síncrono usando topologia em árvore com EC de 2000 µs, 60% de Synchronous Window e 100 µs de Signalling Window

Comparando os resultados de ambas as simulações, a única diferença registada foi relativamente à aplicação μ_3, que com EC de 1500 µs (Figura 59) apresentou uma média de *response time* de 1750 µs e com EC de 2000 (Figura 60) de 958 µs. Esta variação justifica-se pelo facto de, com EC a 2000 µs existe mais largura de banda disponível, sendo suficiente para todas as mensagens de todas as aplicações poderem ser transmitidas no mesmo EC. Já no caso da simulação com EC de 1500 µs, não existe largura de banda suficiente para transmitir todas as mensagens, e como a aplicação μ_3 é aquela com menor prioridade, algumas das mensagens provenientes da sua aplicação têm de ser escalonadas no EC seguinte.

Em seguida, são apresentados histogramas referentes às aplicações onde se registaram variações RTs nas simulações realizadas. Tal como nas simulações apresentadas no capítulo 6.1.1.2, apenas foi registada variabilidade nos RTs das aplicações μ_5 e μ_7, devido à natureza síncrona de todas as aplicações.

Na Figura 61 é apresentado o histograma dos RTs da aplicação μ_5 obtidos durante a simulação com EC de 1500 µs (ver Figura 60). Nesta simulação, 98% dos RTs foram de 544 µs e 2% de 677 µs.

A Figura 63 mostra o histograma dos RTs de μ_7 na simulação com EC de 1500 µs. Nesta simulação, registaram-se dois resultados com uma frequência de 50%: 425 µs e 1533 µs.

Na Figura 62 é apresentado o histograma dos RTs de μ_5 obtidos durante a simulação com EC de 2000 µs (ver Figura 61). Nesta simulação, 98% dos RT foram de 544 µs e 2% de 677 µs.

A Figura 64 mostra o histograma dos RTs de μ_7 na simulação com EC de 2000 µs. Nesta simulação, registaram-se dois resultados com uma frequência de 50%: 425 µs e 1533 µs.

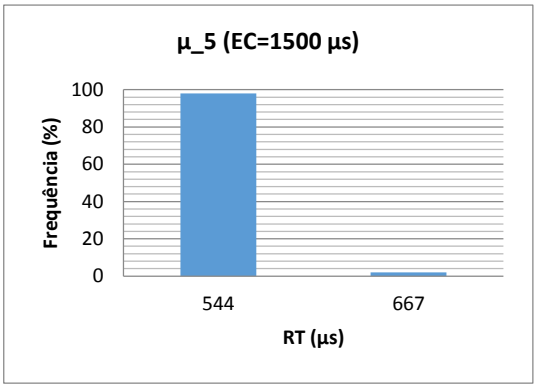


Figura 61: Histograma de μ_5 (síncrono) *na topologia em árvore com EC=1500 µs e SW de 60%*

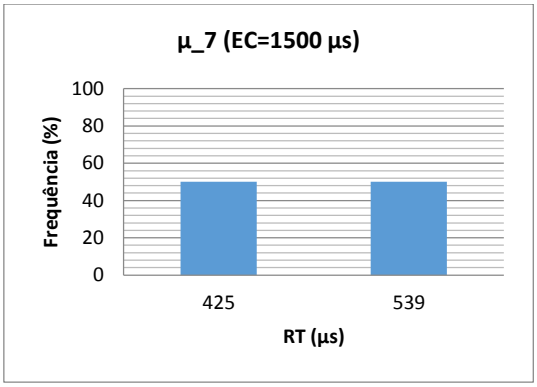


Figura 63: Histograma de μ_7 (síncrono) *na topologia em árvore com EC=1500 µs e SW de 60%*

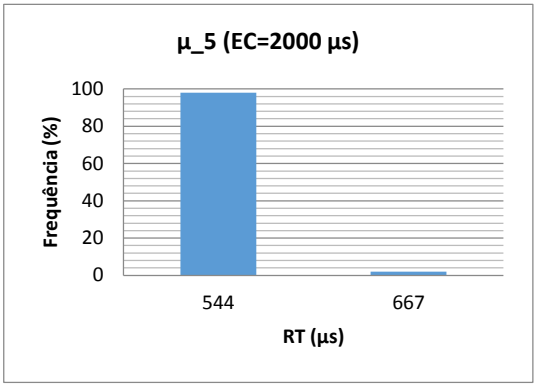


Figura 62: Histograma de μ_5 (síncrono) *na topologia em árvore com EC=2000 µs e SW de 60%*

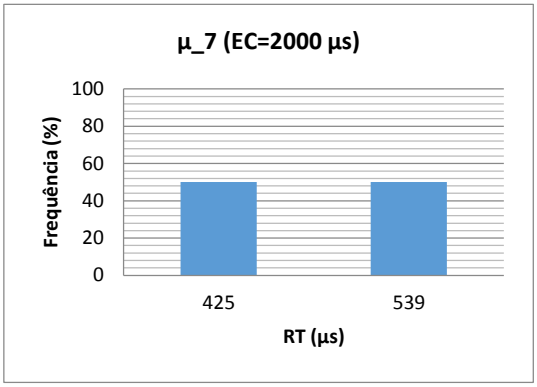


Figura 64: Histograma de μ_7 (síncrono) *na topologia em árvore com EC=2000 µs e SW de 60%*

Analisando os resultados apresentados nos histogramas relativos a esta experiência, as conclusões que se podem retirar são muito semelhantes às referidas no capítulo 6.1.1.2. No caso da aplicação μ_5, a variação justifica-se pela interferência com o tráfego de controlo das aplicações μ_1 e μ_2. Já na aplicação μ_7, existe variação devido à interferência com o tráfego da aplicação μ_6 que coincidem na transmissão em 50% das vezes, devido aos seus períodos. Porém, nas simulações apresentadas, as variações identificadas não são significativas, uma vez que todas as mensagens são enviadas no mesmo EC.

6.1.1.4 Variação da Synchronous Window com tráfego síncrono

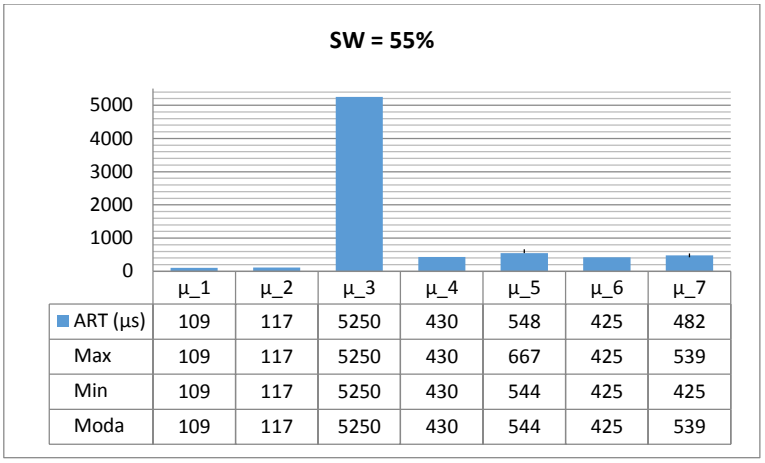
Nas simulações desta experiência, também foram utilizadas as aplicações das experiências anteriores (ver Tabela 3). Contudo, desta vez, é alterada a quantidade definida para a *Synchronous Window*. Realizaram-se simulações com SW de 55% e 65%. Para todas as simulações foi definido um EC de tamanho 1000 µs e 100 µs foram reservados para a *Signalling Window*.

Tabela 6: Características da rede FTT-SE em simulações com aplicações sequenciais síncronas com diferentes Synchronous Window

Elementary Cycle	**1000 µs**	**1000 µs**
Synchronous Window	55% (495 µs)	65% (585 µs)
Asynchronous Window	45% (405 µs)	35% (315 µs)
Singnalling Window	100 µs	100 µs

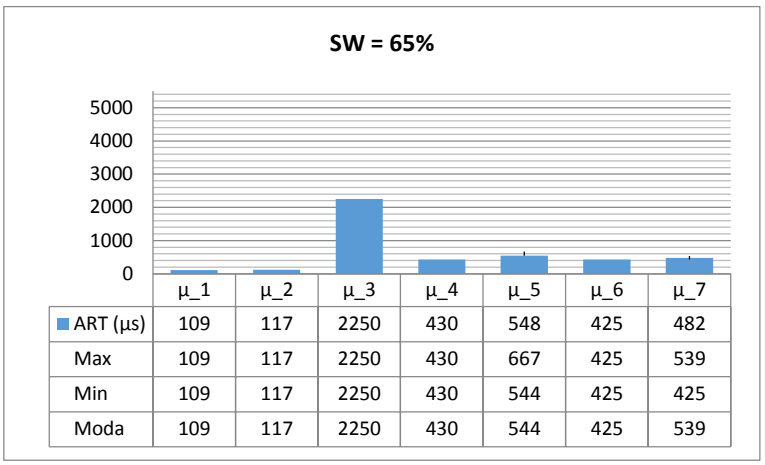
A Figura 65 apresenta os resultados da simulação utilizando um EC de 1000 µs com uma SW de 55%. A aplicação μ_1 obteve uma ART de 109 µs, a aplicação μ_2 de 117 µs, a aplicação μ_3 de 5250 µs, a aplicação μ_4 de 430 µs, a aplicação μ_5 de 548 µs, a aplicação μ_6 de 425 µs e a aplicação μ_7 de 482 µs. Todas as aplicações obtiveram ARTs inferiores ao tempo de 1 EC (1000 µs), com exceção da aplicação μ_3, que obteve uma ART inferior ao tempo de 6 ECs (6000 µs).

Na Figura 66 são revelados os resultados da simulação com EC definido a 1000 µs mas com SW de 65%. A aplicação μ_1 obteve uma ART de 109 µs, a aplicação μ_2 de 117 µs, a aplicação μ_3 de 2250 µs, a aplicação μ_4 de 430 µs, a aplicação μ_5 de 548 µs, a aplicação μ_6 de 425 µs e a aplicação μ_7 de 482 µs. Todas as aplicações obtiveram ARTs inferiores ao tempo de 1 EC (1500 µs), com exceção da aplicação μ_3 que, nesta simulação, obteve uma ART inferior ao tempo de 3 ECs (3000 µs).



	μ_1	μ_2	μ_3	μ_4	μ_5	μ_6	μ_7
ART (μs)	109	117	5250	430	548	425	482
Max	109	117	5250	430	667	425	539
Min	109	117	5250	430	544	425	425
Moda	109	117	5250	430	544	425	539

Figura 65: Resultados da simulação com EC de 1000 μs e 55% de SW com tráfego síncrono



	μ_1	μ_2	μ_3	μ_4	μ_5	μ_6	μ_7
ART (μs)	109	117	2250	430	548	425	482
Max	109	117	2250	430	667	425	539
Min	109	117	2250	430	544	425	425
Moda	109	117	2250	430	544	425	539

Figura 66: Resultados da simulação com EC de 1000 μs, 65% de SW com tráfego síncrono

Analisando e comparando os resultados de ambas as simulações, verifica-se que com uma *Synchronous Window* de 65% (ver Figura 65), a média dos *response times* é significativamente inferior do que com uma SW de 55% (ver Figura 66). No caso da simulação com 55% registou-se 5250 μs de ART para a aplicação μ_3, mas o aumento da SW para 65% fez com que o resultado diminuísse para uma ART de 2250 μs, representando uma diferença de 3000 μs, equivalente ao valor de 3 ECs. Pode-se, então, concluir que com uma SW superior existe mais largura de banda para a transmissão de tráfego síncrono, pelo que resulta em *response times* mais baixos.

A Figura 67 apresenta o histograma dos RTs da aplicação μ_5 obtidos durante a simulação com SW a 55% (ver Figura 65). Nesta simulação, 98% dos RTs foram de 544 μs e 2% de 677 μs.

A Figura 69 mostra o histograma dos RTs de μ_7 na simulação com SW de 55%. Nesta simulação, registaram-se dois resultados com uma frequência de 50%: 425 μs e 1533 μs.

Na Figura 68 é apresentado o histograma dos RTs de μ_5 obtidos durante a simulação com SW de 65% (ver Figura 66). Nesta simulação, 98% dos RTs foram de 544 µs e 2% de 677 µs.

A Figura 70 mostra o histograma dos RT de μ_7 na simulação com SW de 65%. Nesta simulação, registaram-se dois resultados com uma frequência de 50%: 425 µs e 1533 µs.

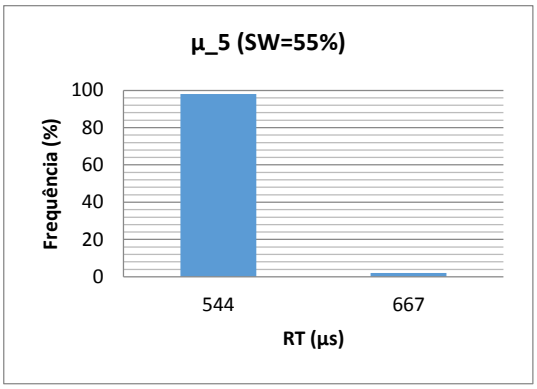


Figura 67: Histograma de μ_5 (síncrono) *com EC=1000 µs e SW de 55%*

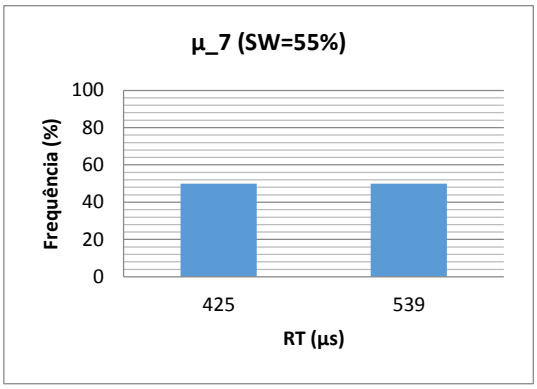


Figura 69: Histograma de μ_7 *(síncrono)* com EC=1000 µs e SW de 55%

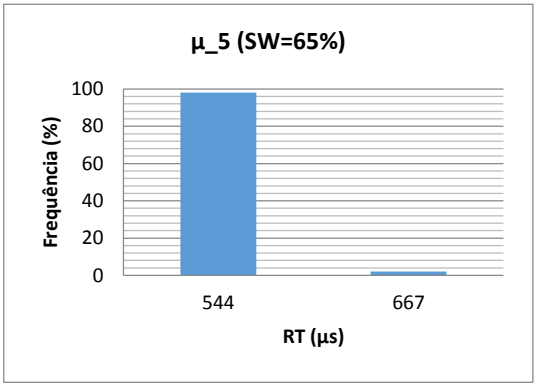


Figura 68: Histograma de μ_5 (síncrono) *com EC=1000 µs e SW de 65%*

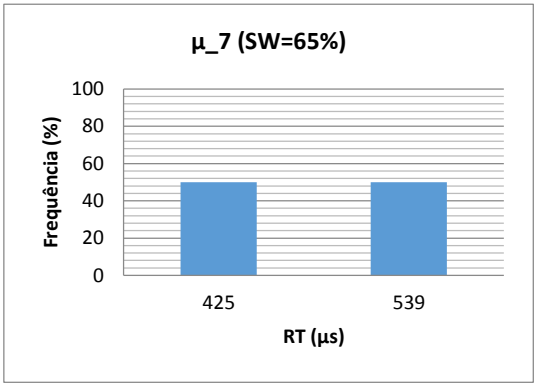


Figura 70: Histograma de μ_7 (síncrono) *com EC=1000 µs e SW de 65%*

Em relação aos resultados apresentados dos histogramas relativos a esta experiência, as conclusões que se podem retirar são similares aos das experiências anteriores (ver capítulos 6.1.1.2 e 6.1.1.3). No caso da aplicação μ_5, existe variação devido à interferência com o tráfego de controlo das aplicações μ_1 e μ_2. Já na aplicação μ_7, a variação está relacionada com a interferência com o tráfego da aplicação μ_6, que coincidem as suas transmissões em 50% das vezes, devido aos seus períodos.

6.1.2 Tráfego assíncrono

Neste capítulo será validada a transmissão de tráfego assíncrono utilizando a implementação do protocolo FTT-SE deste trabalho.

6.1.2.1 Configuração da rede

Tal como no capítulo 6.1.1, as próximas experiências compreendem as topologias de rede em estrela (Figura 47), *daisy chain* (Figura 48) e em árvore (Figura 49). As aplicações utilizadas são as mesmas que as representadas na Tabela 3, contudo, as suas características têm algumas adaptações, uma vez que as aplicações μ_4, μ_5, μ_6 e μ_7 tinham períodos muito baixos para acomodar os atrasos introduzidos pelos mecanismos de *singalling* (ver capítulo 3.4) inerente a este tipo de tráfego. Relembrando a execução do mecanismo de *signalling*, uma aplicação assíncrona produtora tem de sinalizar a existência de mensagens pendentes enviando uma *Signalling Message* para a aplicação *Master* da rede, durante a *Signalling Window*, para que no EC seguinte o *Master* considere essas mensagens no processo de escalonamento. Desta forma, este mecanismo implica um atraso de pelo menos 2 ECs.

A aplicação μ_1 do nó "Control", produz uma *stream* de controlo com tamanho de 64 *bytes* (equivalente a WCML de 7 µs), com período de 11 ECs e envia para uma aplicação instalada no nó "CPU". A aplicação μ_2, também instalada no nó "Control", envia para uma aplicação no nó "RSE" igualmente 64 *bytes* (equivalente a WCML de 7 µs) a cada 11 ECs. Já μ_3 produz, a partir do "Head Unit", 5000 *bytes* (406 µs de WCML) de tráfego de navegação a cada 24 ECs enviando para uma aplicação no nó "RSE". As aplicações μ_4 e μ_5 de "Multimedia Disc. Changer" enviam para uma aplicação no nó "RSE" 2800 *bytes* (equivalente a 227 µs de WCML) cada uma, sendo que μ_4 tem um período de 3 ECs e μ_5 tem período de 6 ECs. Por fim, as aplicações do nó "TV", enviam para aplicações intaladas no nó "Head Unit" 2800 *bytes* (227 µs de WCML), sendo que μ_6 produz a cada 6 ECs e μ_7 produz a cada 9 ECs. As características de todas as aplicações produtoras encontram-se reunidas na Tabela 7

Tabela 7: Características das aplicações sequenciais assíncronas

Aplicação	Categoria	Período (ECs)	Tamanho (bytes)	WCML	Origem	Destino
µ_1	Controlo	11	64	7 µs	Control	CPU
µ_2	Controlo	11	64	7 µs	Control	RSE
µ_3	Navegação	24	5000	406 µs	Head Unit	RSE
µ_4	MM Vídeo	3	2800	227 µs	Mult. Disc	RSE
µ_5	MM Áudio	6	2800	227 µs	Mult. Disc	RSE
µ_6	TV Vídeo	6	2800	227 µs	TV	Head-Unit
µ_7	TV Áudio	9	2800	227 µs	TV	Head-Unit

6.1.2.2 Comparação entre topologias estrela, *daisy chain* e árvore utilizando tráfego assíncrono

Para esta experiência, o *Elementary Cycle* foi definido com uma duração de 1000 μs, sendo reservados 60% do mesmo (540 μs) para a *Synchronous Window*, ou seja 40% (360 μs) para a *Asynchronous* Window, e 100 μs são reservados para a *Signalling Window*. As características da rede para as simulações deste capítulo encontram-se reunidas na Tabela 8. Todas as *streams* utilizadas nestas simulações são do tipo assíncrono. É importante recordar que o tráfego assíncrono é gerado a partir da ocorrência de eventos, sendo imprevisíveis os instantes em que os mesmos acontecem (ver capítulos 2.3 e 3.4).

Tabela 8: Características da rede FTT-SE em simulações com aplicações sequenciais assíncronas em várias topologias de rede

Elementary Cycle	**1000 μs**
Synchronous Window	60% (540 μs)
Asynchronous Window	40% (360 μs)
Singnalling Window	100 μs

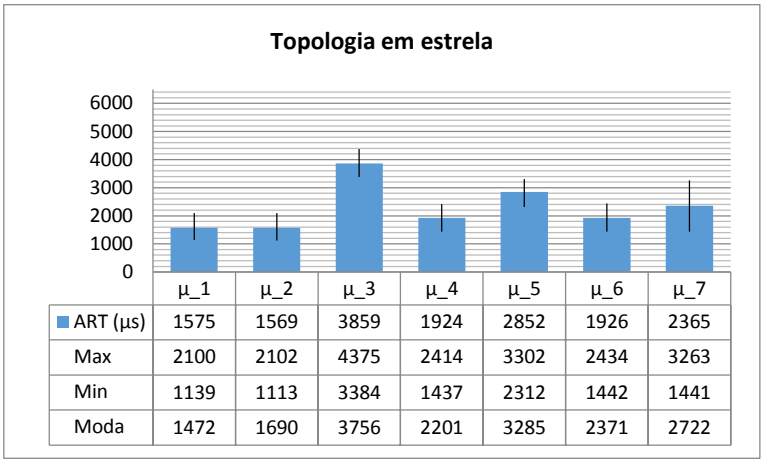
A Figura 71 apresenta os resultados da simulação utilizando a topologia em estrela. A aplicação μ_1 obteve uma ART de 1575 μs, a aplicação μ_2 de 1569 μs, a aplicação μ_3 de 3859 μs, a aplicação μ_4 de 1924 μs, a aplicação μ_5 de 2852 μs, a aplicação μ_6 de 1926 μs e a aplicação μ_7 de 2365 μs. As ARTs das aplicações μ_1, μ_2, μ_4 e μ_6 são inferiores ao tempo de 2 ECs (2000 μs), já as ARTs das aplicações μ_5 e μ_7 são inferiores a 3 ECs (3000 μs) e a ART da aplicação μ_3 é inferior a 4 ECs (4000 μs).

A Figura 72 apresenta os resultados da simulação numa topologia *daisy chain*. Neste caso, a aplicação μ_1 obteve uma ART de 1656 μs, a aplicação μ_2 de 1591 μs, a aplicação μ_3 de 4014 μs, a aplicação μ_4 de 2035 μs, a aplicação μ_5 de 2921 μs, a aplicação μ_6 de 2148 μs e a aplicação μ_7 de 2560 μs. As ARTs das aplicações μ_1 e μ_2 são inferiores ao tempo de 2 ECs (2000 μs), as aplicações μ_4, μ_5, μ_6 e μ_7 são inferiores ao tempo de 3 ECs (3000 μs), já a ART da aplicação μ_3 é inferior a 5 ECs (5000 μs).

Por fim, os resultados da simulação numa topologia em árvore são apresentados na Figura 73. A aplicação μ_1 obteve uma ART de 1642 μs, a aplicação μ_2 de 1640 μs, a aplicação μ_3 de 4006 μs, a aplicação μ_4 de 2078 μs, a aplicação μ_5 de 2957 μs, a aplicação μ_6 de 2103 μs e a aplicação μ_7 de 2478 μs. As ARTs das aplicações μ_1 e μ_2 são inferiores ao tempo de 2 ECs (2000

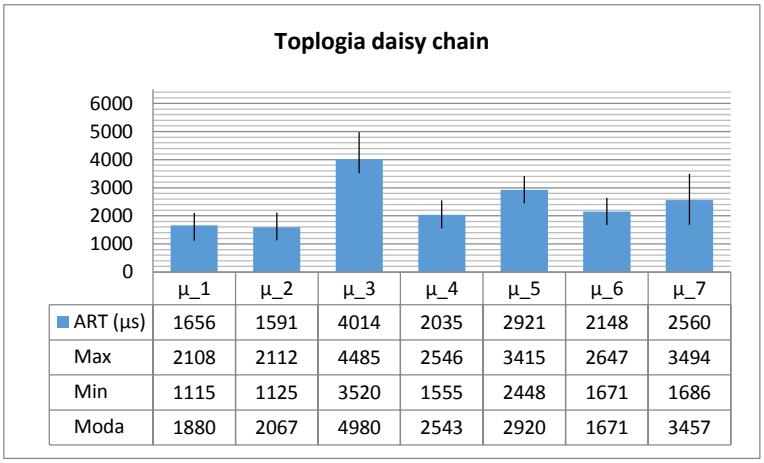
µs), as aplicações μ_4, μ_5, μ_6 e μ_7 são inferiores ao tempo de 3 ECs (3000 µs) e a aplicação μ_3 obteve ART inferior a 5 ECs (5000 µs).

Os gráficos apresentados também fazem referência aos resultados máximos e mínimos de cada uma das aplicações consideradas. Contudo, para efeitos de análise, deve-se considerar a média dos resultados obtidos (ART).



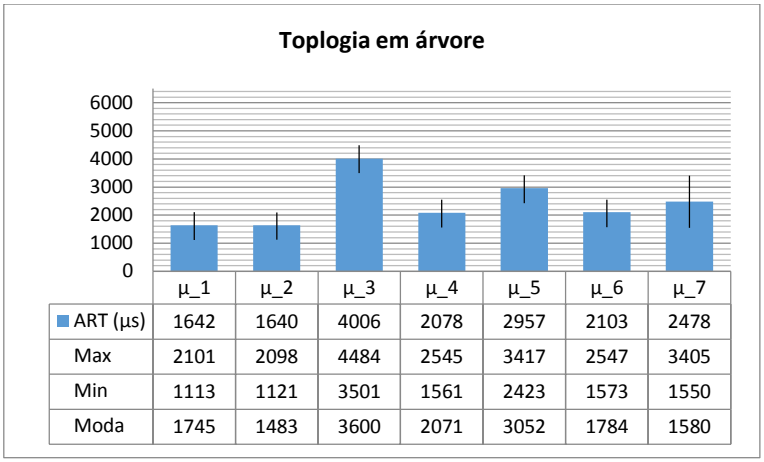
	µ_1	µ_2	µ_3	µ_4	µ_5	µ_6	µ_7
ART (µs)	1575	1569	3859	1924	2852	1926	2365
Max	2100	2102	4375	2414	3302	2434	3263
Min	1139	1113	3384	1437	2312	1442	1441
Moda	1472	1690	3756	2201	3285	2371	2722

Figura 71: Resultados da simulação com tráfego assíncrono usando topologia em estrela com EC de 1000 µs, 60% de Synchronous Window e 100 µs de Signalling Window



	µ_1	µ_2	µ_3	µ_4	µ_5	µ_6	µ_7
ART (µs)	1656	1591	4014	2035	2921	2148	2560
Max	2108	2112	4485	2546	3415	2647	3494
Min	1115	1125	3520	1555	2448	1671	1686
Moda	1880	2067	4980	2543	2920	1671	3457

Figura 72: Resultados da simulação com tráfego assíncrono usando topologia daisy chain com EC de 1000 µs, 60% de Synchronous Window e 100 µs de Signalling Window



	μ_1	μ_2	μ_3	μ_4	μ_5	μ_6	μ_7
ART (μs)	1642	1640	4006	2078	2957	2103	2478
Max	2101	2098	4484	2545	3417	2547	3405
Min	1113	1121	3501	1561	2423	1573	1550
Moda	1745	1483	3600	2071	3052	1784	1580

Figura 73: Resultados da simulação com tráfego assíncrono usando topologia em árvore com EC de 1000 μs, 60% de Synchronous Window e 100 μs de Signalling Window

Analisando os resultados, a primeira conclusão a apontar é que, de facto, nenhuma aplicação registou um *response time* inferior ao tempo de 1 *Elementary Cycle* (1000 μs), mesmo aquelas com prioridade mais alta. Esta evidência justifica-se pelo cumprimento do mecanismo de *signalling* que as aplicações produtoras de tráfego assíncrono têm de cumprir. Relembrando, este mecanismo compreende a sinalização de mensagens pendentes, através do envio de uma *Siginalling Message* para o *Master*, sendo que este mecanismo tem um custo de 2 ECs. (ver capítulo 3.4) Por isso, os resultados apresentados validam o correto funcionamento do mecanismo de *signalling* implementado neste trabalho.

Relativamente aos resultados das várias topologias, podem-se identificar algumas variações entre as registadas em cada uma delas, sendo as mais significativas as variações da *streams* de navegação. No caso desta *stream* (μ_3), os tempos de resposta registados usando uma topologia em estrela são um pouco mais baixos do que nas restantes topologias consideradas. A diferença de resultados tem origem na existência de mais uma ligação no percurso da *stream* usando as topologias *daisy chain* (Figura 48) e em árvore (Figura 49) comparativamente com a topologia em estrela (Figura 47). Esta diferença tem como consequência que menos mensagens daquela aplicação possam ser escalonadas num EC.

Os resultados, em todas as topologias consideradas, revelam uma ligeira diferença nas ARTs obtidas comparando as *streams* TV vídeo (μ_6) e TV áudio (μ_7), mesmo tendo o mesmo percurso. A diferença justifica-se devido à prioridade da *stream* TV vídeo, que é considerada mais prioritária, quando o seu escalonamento coincide com a *stream* de TV Áudio no mesmo EC, devido aos seus períodos. Como apenas é considerada depois, já não existe largura de banda suficiente para o envio da *stream* de TV Áudio, pelo que o seu escalonamento é adiado para o EC seguinte.

Ainda relativamente à *stream* de TV Áudio, analisando apenas as topologias de rede, seria de esperar, tal como verificado no capítulo 6.1.1.2, que se verificassem variações significativas entre os resultados das várias topologias devido ao diferente número de *switches* do seu percurso em cada uma das delas. Na topologia em estrela apenas existem duas ligações a percorrer, enquanto na topologia em árvore existem três ligações e na topologia *daisy chain* existem quatro ligações. Contudo, mesmo numa topologia com poucas ligações (estrela), a largura de banda reservada para esta experiência é insuficiente para que as mensagens de TV Vídeo e TV Áudio possam ser transmitidas no mesmo EC.

A mesma interpretação é realizada para entender a variação entre a ART da *stream* da aplicação μ_4 e da aplicação μ_5. Pelo facto de as mensagens de ambas as aplicações terem o mesmo percurso, e as suas transmissões coincidirem em alguns ECs, devido ao seu período, a aplicação μ_5 tem uma ART mais elevada porque tem menos prioridade que μ_4 e, não existindo largura de banda suficiente para a transmissão das mensagens de ambas as aplicações num mesmo EC, o escalonamento das mensagens de μ_5 é adiado para o ciclo seguinte, sofrendo o atraso de 1 EC.

Seguidamente são apresentados e analisados os histogramas das aplicações assíncronas utilizadas nestas simulações. Contrariamente às simulações síncronas, os resultados nos histogramas das simulações assíncronas são agrupados em classes correspondentes ao número de *Elementary Cycles* equivalente, devido à variabilidade causada pela imprevisibilidade do tráfego *event-triggered*. Além disso, para efeitos de análise dos *response times* é também relevante interpretá-los em número de ECs. Neste contexto, um determinado número de ECs compreende o intervalo de tempo desde que terminou o EC imediatamente anterior até ao tempo do seu EC. Por exemplo, um RT de 2500 μs insere-se na classe correspondente a 3 ECs, que define o intervalo de tempo de 2000 μs até 3000 μs.

A Tabela 9 reúne os intervalos correspondentes a cada classe, nos histogramas desta simulação.

Tabela 9: Intervalos das classes em ECs em simulação com EC de 1000 μs

EC (Classe)	Intervalo
1	0 μs a 1000 μs
2	1000 μs a 2000 μs
3	2000 μs a 3000 μs
4	3000 μs a 4000 μs

5	4000 μs a 5000 μs
6	5000 μs a 6000 μs
7	6000 μs a 7000 μs
8+	Mais de 7000 μs

A Figura 74, Figura 81 e Figura 88 mostram os histogramas referentes à aplicação μ_1 nas topologias estrela, *daisy chain* e árvore, respetivamente. Já a Figura 75, Figura 82 e Figura 89 apresentam os resultados da aplicação μ_2, na mesma ordem de topologias. Em ambas as aplicações, foram obtidos, com uma frequência bastante elevada, RTs na classe de 2 ECs, ou seja, entre os 1000 μs e os 2000 μs.

A aplicação μ_3 é das que regista RTs mais elevados. A Figura 76 apresenta os resultados na topologia em estrela, onde os RTs obtidos se inserem, essencialmente, nas classes correspondentes a 4 (de 3000 μs a 4000 μs) e 5 (de 4000 μs a 5000 μs) ECs, sendo a classe de 4 ECs é a que se regista com mais frequência, com larga margem de diferença. Já os resultados da topologia *daisy chain* (Figura 83) revelam que os RTs obtidos dividiram-se, em número semelhante, pelas classes de 4 e 5 ECs. Também as classes de 4 e 5 ECs foram os resultados obtidos para a topologia em árvore (Figura 90), havendo uma ligeira supremacia na frequência registada na classe de 4 ECs.

Os resultados de μ_4 encontram-se na Figura 77, Figura 84 e Figura 91, respetivamente para as topologias em estrela, *daisy chain* e árvore. Em todas elas, os resultados dos RTs dividem-se entre as classes de 2 (de 1000 μs a 2000 μs) e 3 (de 2000 μs a 3000 μs) ECs.

Já na aplicação μ_5, os resultados dividem-se pelas classes de 3 (de 2000 μs a 3000 μs) e 4 (de 3000 μs a 4000 μs) ECs. Os mesmos podem ser encontrados na Figura 78 (estrela), Figura 85 (*daisy chain*) e Figura 92 (árvore).

A aplicação μ_6, em todas as topologias, registou resultados correspondentes às classes de 2 (de 1000 μs a 2000 μs) e 3 (de 2000 μs a 3000 μs) ECs. Contudo, no caso da topologia em estrela (Figura 79), houve uma incidência maior na classe de 2 ECs, enquanto na topologia *daisy chain* (Figura 86) e árvore (Figura 93), verificou-se o contrário. Os resultados centram-se em RTs correspondentes às classes a 3 e 4 (de 3000 μs a 4000 μs) ECs.

Por fim, na aplicação μ_7 os resultados dividiram-se um pouco pelas classes correspondentes a 2 de (1000 μs a 2000 μs), 3 (de 2000 μs a 3000 μs) e 4 (de 3000 μs a 4000 μs) ECs em todas as topologias - estrela (Figura 80), *daisy chain* (Figura 87) e árvore (Figura 94) – sendo registada mais frequência na classe de 3 ECs.

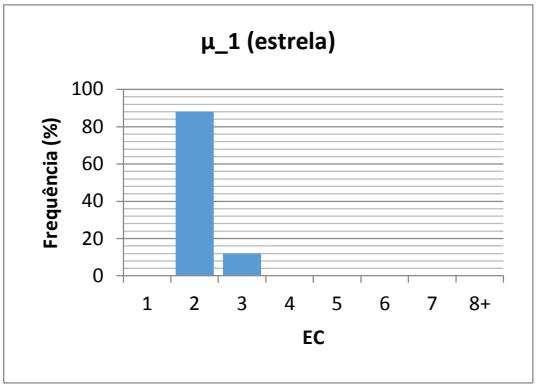


Figura 74: Histograma de μ_1 (assíncrono) *na topologia em estrela com EC=1000 µs e SW=60%*

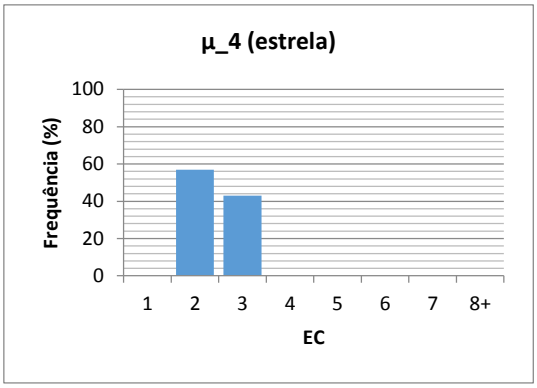


Figura 77: Histograma de μ_4 (assíncrono) *na topologia em estrela com EC=1000 µs e SW=60%*

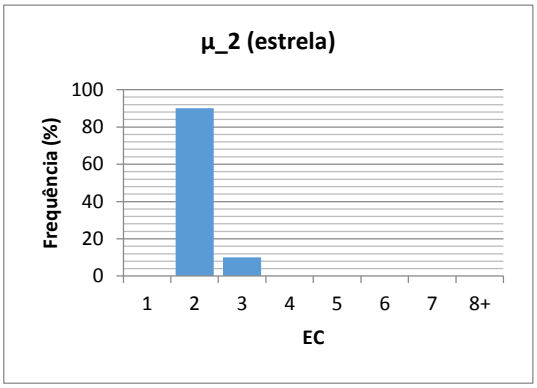


Figura 75: Histograma de μ_2 (assíncrono) *na topologia em estrela com EC= 1000 µs e SW=60%*

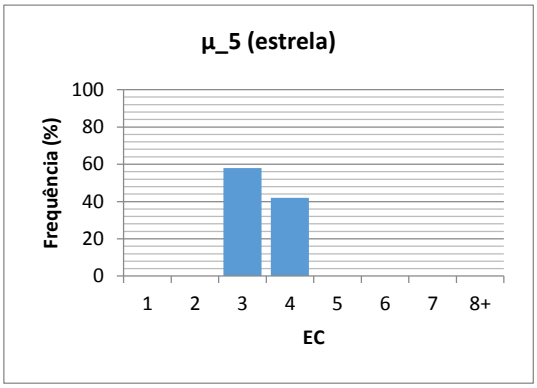


Figura 78: Histograma de μ_5 (assíncrono) *na topologia em estrela com EC=1000 µs e SW=60%*

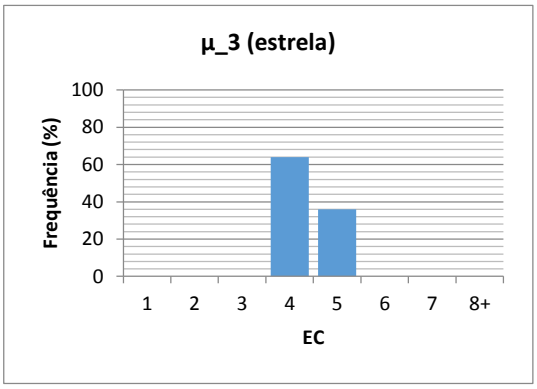


Figura 76: Histograma de μ_3 (assíncrono) *na topologia em estrela com EC=1000 µs e SW=60%*

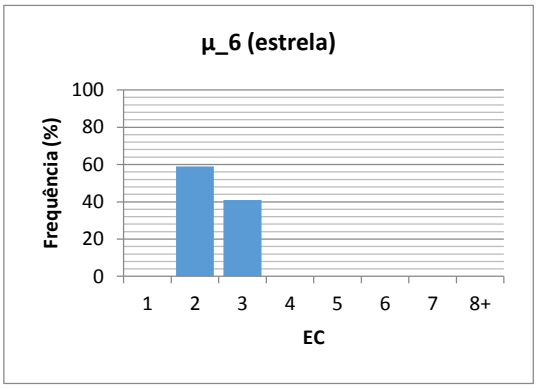


Figura 79: Histograma de μ_6 (assíncrono) *na topologia em estrela com EC=1000 µs e SW=60%*

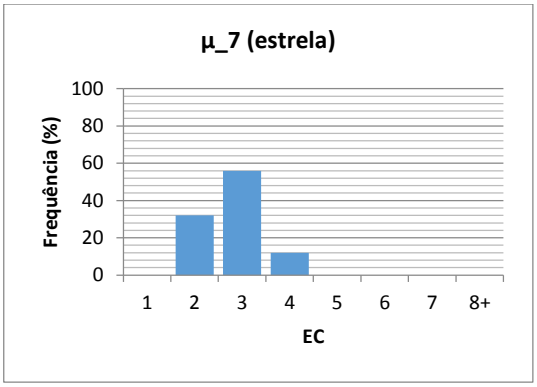


Figura 80: Histograma de μ_7 (assíncrono) *na topologia em estrela com EC=1000 μs e SW=60%*

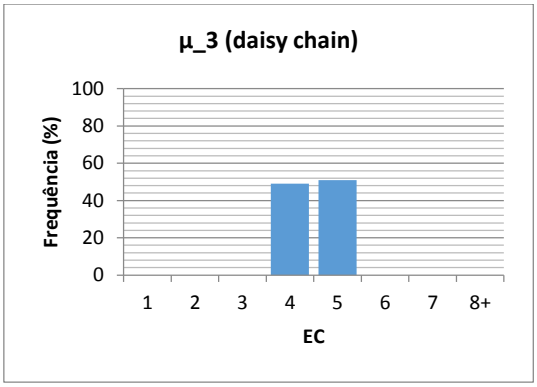


Figura 83: Histograma de μ_3 (assíncrono) *na topologia daisy chain com EC=1000 μs e SW=60%*

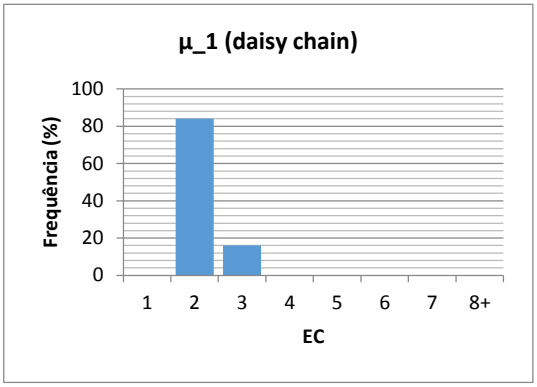


Figura 81: Histograma de μ_1 (assíncrono) *na topologia daisy chain com EC=1000 μs e SW=60%*

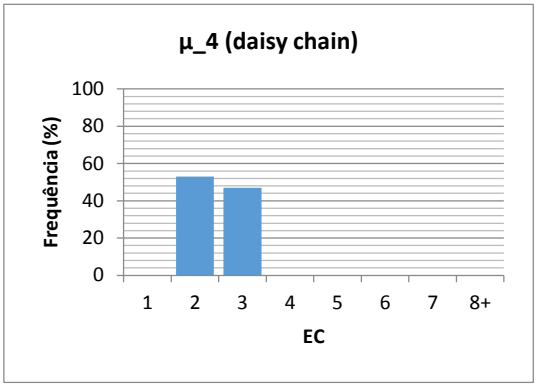


Figura 84: Histograma de μ_4 (assíncrono) *na topologia daisy chain com EC=1000 μs e SW=60%*

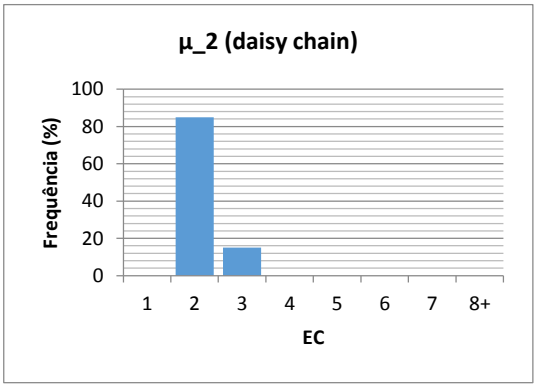


Figura 82: Histograma de μ_2 (assíncrono) *na topologia daisy chain com EC=1000 μs e SW=60%*

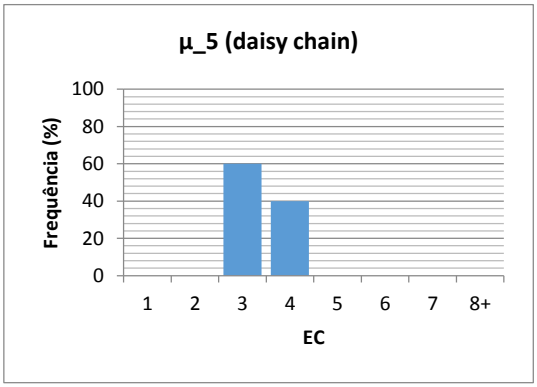


Figura 85: Histograma de μ_5 (assíncrono) *na topologia daisy chain com EC=1000 μs e SW=60%*

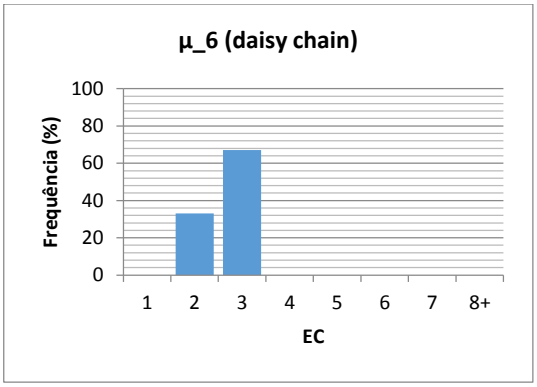


Figura 86: Histograma de μ_6 (assíncrono) *na topologia daisy chain com EC=1000 µs e SW=60%*

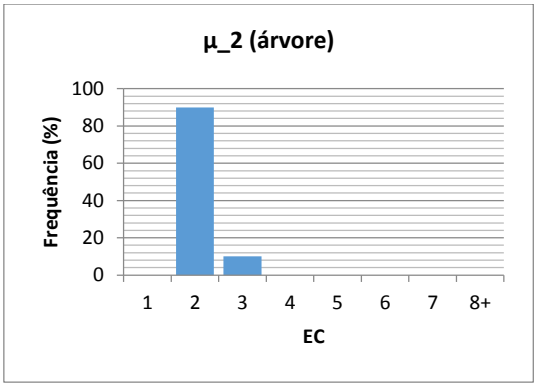


Figura 89: Histograma de μ_2 (assíncrono) *na topologia em árvore com EC=1000 µs e SW=60%*

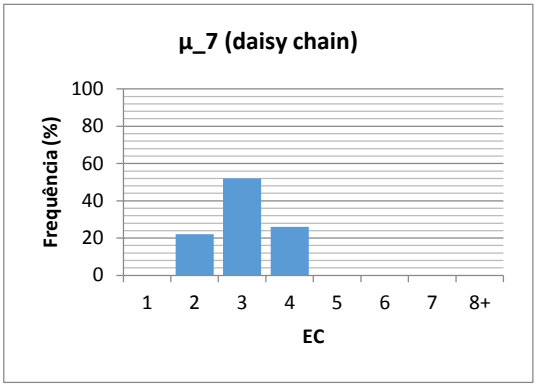


Figura 87: Histograma de μ_7 (assíncrono) *na topologia daisy chain com EC=1000 µs e SW=60%*

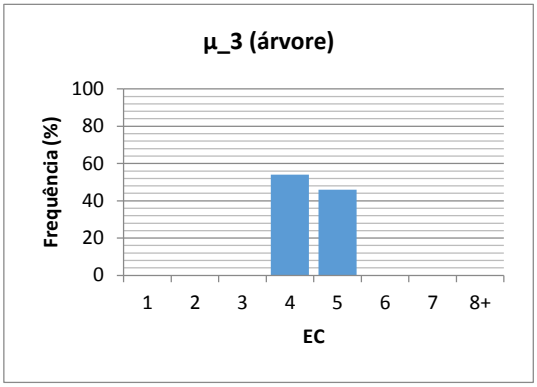


Figura 90: Histograma de μ_3 (assíncrono) *na topologia em árvore com EC=1000 µs e SW=60%*

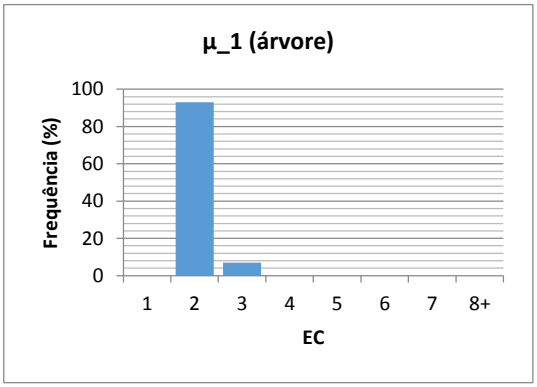


Figura 88: Histograma de μ_1 (assíncrono) *na topologia em árvore com EC=1000 µs e SW=60%*

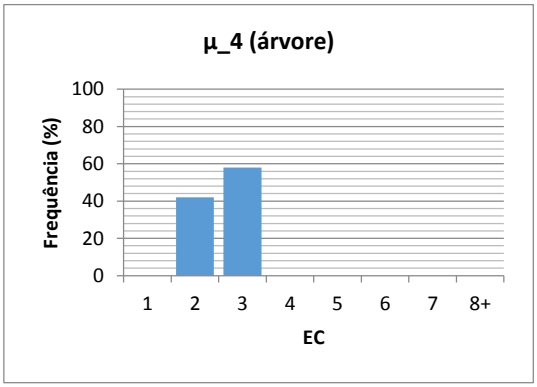


Figura 91: Histograma de μ_4 (assíncrono) *na topologia em árvore com EC=1000 µs e SW=60%*

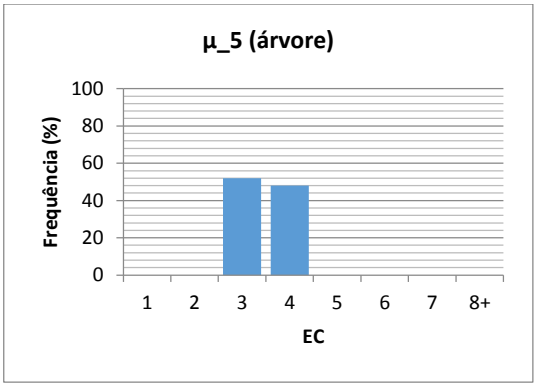


Figura 92: Histograma de μ_5 (assíncrono) *na topologia em árvore com EC=1000 μs e SW=60%*

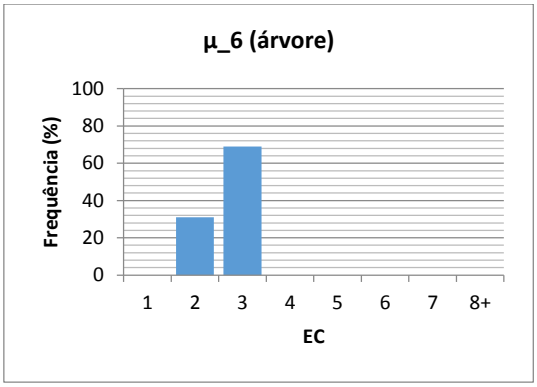


Figura 93: Histograma de μ_6 (assíncrono) *na topologia em árvore com EC=1000 μs e SW=60%*

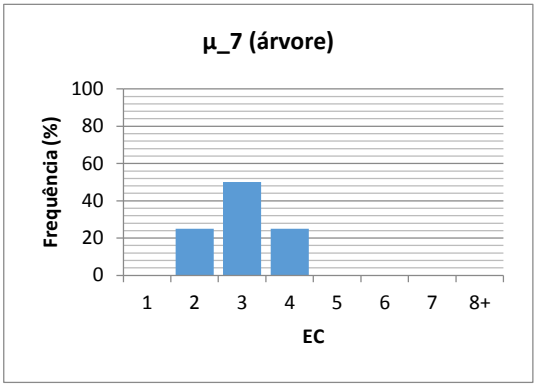


Figura 94: Histograma de μ_7 (assíncrono) *na topologia em árvore com EC=1000 μs e SW=60%*

Analisando os resultados, os resultados obtidos pelas aplicações μ_1, μ_2, μ_4 e μ_5, estas não sofreram variações relevantes, comparando os resultados das simulações nas várias topologias consideradas. No caso das aplicações μ_4 e μ_5, esta evidência justifica-se pelo facto de serem terem uma prioridade elevada. Já no caso de μ_1 e μ_2, as variações não são muito visíveis, uma vez que as suas mensagens têm um tamanho curto e, por isso, não requerem muita largura de banda para a sua transmissão.

Contrariamente às aplicações referidas anteriormente, uma relativa variação foi registada nos resultados da aplicação μ_3. Embora em todas as topologias consideradas, os resultados se centrarem nas classes correspondentes a 4 (de 3000 μs a 4000 μs) e 5 (de 4000 μs a 5000 μs) ECs, na topologia em estrela os resultados incidem com bastante mais frequência na classe de 4 ECs. Já nas topologias *daisy chain* e árvore os resultados obtidos dividem-se um pouco entre as duas classes referidos. A diferença está relacionada com o percurso das mensagens desta *stream*, que percorrem apenas duas ligações na topologia em estrela (ver Figura 47), enquanto

nas topologias *daisy chain* e árvore existe um percurso de três ligações (ver Figura 48 e Figura 49) e consequentemente, alguns dos fragmentos das mensagens geradas por esta aplicação têm menos largura de banda disponível para serem transmitidos.

A aplicação μ_6 regista resultados nas classes de 2 (de 1000 μs a 2000 μs) e 3 (de 2000 μs a 3000 μs) ECs em todas as topologias. Contudo, na topologia *daisy chain* e em árvore, existe bastante mais frequência na classe de 3 ECs. Já na topologia em estrela, a maior incidência recai sobre a classe de 2 ECs. A diferença revelada justifica-se pela existência de quatro ligações, no percurso das mensagens desta aplicação, na topologia *daisy chain* e três ligações na topologia em árvore, havendo largura de banda insuficiente para uma transmissão em apenas 1 EC, forçando a haver um RT superior. Já o percurso na topologia em estrela compreende apenas duas ligações e, por isso, a largura de banda disponível é suficiente para a transmissão num único EC, uma vez que é a mensagem com mais prioridade nas ligações do seu percurso.

A justificação para a variação dos resultados da aplicação e μ_7 divide-se em duas partes. A primeira está relacionada com o a mesma razão apontada para a aplicação μ_6, uma vez que percorrem as mesmas ligações. A segunda parte justifica a razão pela qual os RTs obtidos variam desde a classe correspondente a 2 (de 1000 μs a 2000 μs) a 4 (de 3000 μs a 4000 μs) ECs nas várias topologias. Tal se verifica, devido ao facto de, por vezes, coincidir a transmissão com a da aplicação μ_6, devido aos seus períodos, e por ser menos prioritária tem como consequência o adiamento da mesma.

6.1.2.3 Variação da duração do Elementary Cycle com tráfego assíncrono

Tal como explicado no capítulo 6.1.1.3, uma vez que foi realizada uma análise ao comportamento das aplicações em diferentes topologias, as restantes simulações inserida no capítulo 6.1.2 são fixadas na topologia em árvore pelo equilíbrio que oferece.

Para esta experiência, foram mantidas as mesmas aplicações, com as mesmas características, do que as utilizadas no capítulo anterior (ver Tabela 7). A análise é centrada na variação das características da rede e, neste caso, relativamente à duração do *Elementary Cycle.*

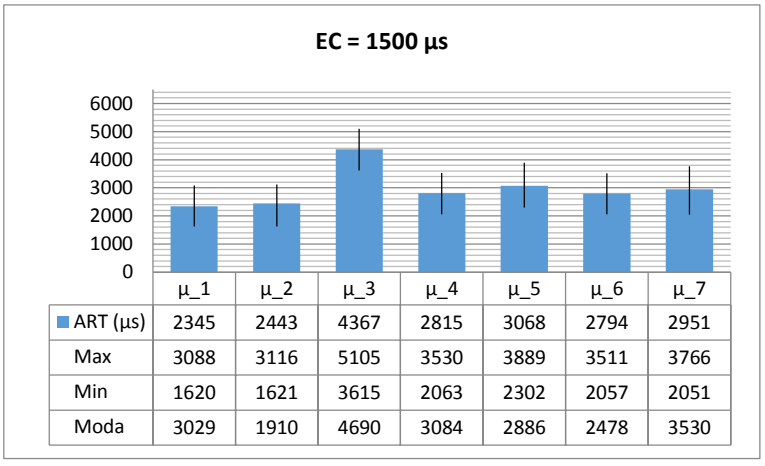
Tanto a *Synchronous Window* (60%) como a *Signalling Window* (100 μs), mantêm os seus valores. Na experiência da comparação entre as topologias de rede foi utilizado um EC de 1000 μs. Nas simulações deste capítulo são usados ECs com 1500 μs e 2000 μs de duração. As características da rede FTT-SE utilizadas nestas simulações encontram-se reunidas na Tabela 10.

Tabela 10: Características da rede FTT-SE em simulações com aplicações sequenciais asíncronas com vários Elementary Cycles

Elementary Cycle	**1500 µs**	**2000 µs**
Synchronous Window	60% (840 µs)	60% (1140 µs)
Asynchronous Window	40% (560 µs)	40% (760 µs)
Singnalling Window	100 µs	100 µs

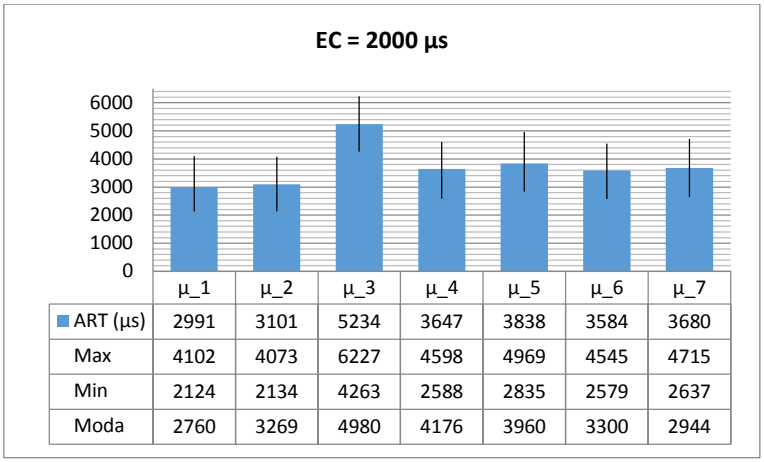
Na Figura 95 são apresentados os resultados da simulação com um EC de 1500 µs de duração. A aplicação μ_1 obteve uma ART de 2345 µs, a aplicação μ_2 de 2443 µs, a aplicação μ_3 de 4367 µs, a aplicação μ_4 de 2815 µs, a aplicação μ_5 de 3068 µs, a aplicação μ_6 de 2794 µs e a aplicação μ_7 de 2951 µs. Todas as aplicações obtiveram ARTs inferiores ao tempo de 2 EC (3000 µs), com exceção das aplicações μ_3 e μ_5, que obtiveram ARTs inferiores ao tempo de 3 ECs (4500 µs).

Já na Figura 96 são apresentados os resultados da simulação com EC definido a 2000 µs. A aplicação μ_1 obteve uma ART de 2991 µs, a aplicação μ_2 de 3101 µs, a aplicação μ_3 de 5234 µs, a aplicação μ_4 de 3647 µs, a aplicação μ_5 de 3838 µs, a aplicação μ_6 de 3584 µs e a aplicação μ_7 de 3680 µs. Todas as aplicações obtiveram ARTs inferiores ao tempo de 2 EC (4000 µs), com exceção da aplicação μ_3, que registou uma ART inferior ao tempo de 3 ECs (6000 µs).



	µ_1	µ_2	µ_3	µ_4	µ_5	µ_6	µ_7
ART (µs)	2345	2443	4367	2815	3068	2794	2951
Max	3088	3116	5105	3530	3889	3511	3766
Min	1620	1621	3615	2063	2302	2057	2051
Moda	3029	1910	4690	3084	2886	2478	3530

Figura 95: Resultados da simulação com tráfego assíncrono usando topologia em árvore com EC de 1500 µs, 60% de Synchronous Window e 100 µs de Signalling Window



	µ_1	µ_2	µ_3	µ_4	µ_5	µ_6	µ_7
ART (µs)	2991	3101	5234	3647	3838	3584	3680
Max	4102	4073	6227	4598	4969	4545	4715
Min	2124	2134	4263	2588	2835	2579	2637
Moda	2760	3269	4980	4176	3960	3300	2944

Figura 96: Resultados da simulação com tráfego assíncrono usando topologia em árvore com EC de 2000 µs, 60% de Synchronous Window e 100 µs de Signalling Window

Comparando os resultados de ambas as simulações, ao converter a média dos *response times* obtidos para ECs, verifica-se que os resultados são idênticos. Contudo, analisando os valores em µs, pode-se verificar que, com um EC definido a 2000 µs, as médias dos RT sobem significativamente comparativamente com um EC de 1500 µs, embora haja mais largura de banda disponível. Pode-se concluir, para esta experiência, que aumentando o tamanho do EC para comunicações assíncronas, embora exista maior largura de banda disponível, o prejuízo de acomodar os atrasos relacionados com o mecanismo de *signalling* é superior aos benefícios de reservar mais largura de banda para os ECs. A Tabela 11 e Tabela 12 reúnem os intervalos a que pertencem cada classe, para facilitar a leitura dos histogramas.

Tabela 11: Intervalos das classes em ECs em simulação com EC de 1500 µs

EC (Classe)	Intervalo
1	0 µs a 1500 µs
2	1500 µs a 3000 µs
3	3000 µs a 4500 µs
4	4500 µs a 6000 µs
5	6000 µs a 7500 µs
6	7500 µs a 9000 µs
7	9000 µs a 10500 µs
8+	Mais de 10500 µs

Tabela 12: Intervalos das classes em ECs em simulação com EC de 2000 µs

EC (Classe)	Intervalo
1	0 µs a 2000 µs
2	2000 µs a 4000 µs
3	4000 µs a 6000 µs
4	6000 µs a 8000 µs
5	8000 µs a 10000 µs
6	10000 µs a 12000 µs
7	12000 µs a 14000 µs
8+	Mais de 14000 µs

Seguidamente, são apresentados os histogramas que ilustram os comportamentos das aplicações nesta experiência.

A Figura 97 e Figura 104 mostram os histogramas referentes à aplicação μ_1, respetivamente nas simulações com EC de 1500 μs e 2000 μs. A Figura 98 e Figura 105 apresentam os resultados da aplicação μ_2 para as mesmas durações de ECs. Em ambas as aplicações foram obtidos RTs com maior frequência na classe correspondente a 2 ECs, ou seja, de 1500 μs a 3000 μs na simulação com EC de 1500 μs, e de 2000 μs a 4000 μs na simulação com EC de 2000 μs.

A aplicação μ_3 na simulação com EC de 1500 μs (Figura 99) obteve os seus RTs nas classes correspondentes a 3 (de 3000 μs a 4500 μs) e 4 (de 4500 μs a 6000 μs) ECs, assim como na simulação com EC de 2000 μs (Figura 106) também registou os seus RTs na classe de 3 (de 2000 μs a 6000 μs) e 4 (de 6000 μs a 8000 μs) ECs. Contudo, com um EC de 2000 μs há uma larga maioria de resultados obtidos na classe correspondente a 3 ECs. Na simulação com EC de 1500 μs, há apenas uma ligeira diferença, favorável à classe de 3 ECs.

Relativamente a μ_4, os seus resultados são apresentados na Figura 100 e Figura 107, respetivamente para 1500 μs e 2000 μs de EC. No caso da simulação com EC de 1500 μs, os RTs foram obtidos nas classes correspondentes a 2 (de 1500 μs a 3000 μs) e 3 ECs (de 3000 μs a 4500 μs), com mais incidência na classe de 2 ECs. Na simulação com EC de 2000 μs os RT foram obtidos na classe de 2 (de 2000 μs a 4000 μs) e 3 (de 4000 μs a 6000 μs) ECs, com maior frequência na classe correspondente a 2 ECs.

Na aplicação μ_5, relativamente à simulação com EC de 1500 μs (Figura 101), os RTs foram obtidos nas classes correspondentes a 2 (de 1500 μs a 3000 μs) e 3 ECs (de 3000 μs a 4500 μs), com mais frequência na classe de 3 ECs. Na simulação com EC de 2000 μs (Figura 109) os RTs foram obtidos na classe de 2 (de 2000 μs a 4000 μs) e 3 (de 4000 μs a 6000 μs) ECs, com maior frequência na classe correspondente a 2 ECs.

A aplicação μ_6 tem os seus resultados apresentados na Figura 102 e Figura 109, respetivamente para 1500 μs e 2000 μs de EC. No caso da simulação com EC de 1500 μs, os RTs foram obtidos nas classes correspondentes a 2 (de 1500 μs a 3000 μs) e 3 ECs (de 3000 μs a 4500 μs), com mais incidência na classe de 2 ECs. Na simulação com EC de 2000 μs, os RTs foram obtidos na classe de 2 (de 2000 μs a 4000 μs) e 3 (de 4000 μs a 6000 μs) ECs, com maior frequência na classe correspondente a 2 ECs.

Para finalizar, a aplicação μ_7 registou resultados divididos de forma semelhante entre as classes de 2 (de 1500 µs a 3000 µs) e 3 (de 3000 µs a 4500 µs) ECs de RT para um EC 1500 µs (Figura 103). Com EC de 2000 µs (Figura 110) foram registados RTs nas classes correspondentes aos 2 (de 2000 µs a 4000 µs) e 3 (de 4000 µs a 6000 µs) ECs, com uma clara frequência na classe de 2 ECs.

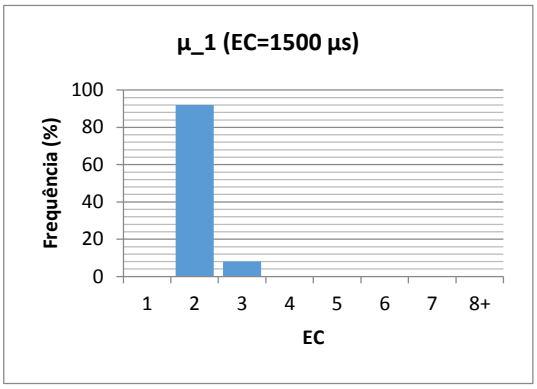


Figura 97: Histograma de μ_1 (assíncrono) *na topologia em árvore com EC=1500 µs e SW de 60%*

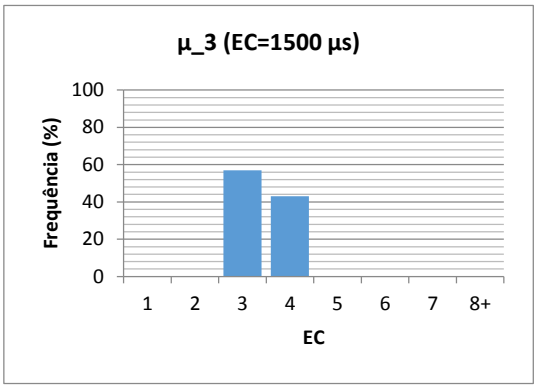


Figura 99: Histograma de μ_3 (assíncrono) *na topologia em árvore com EC=1500 µs e SW de 60%*

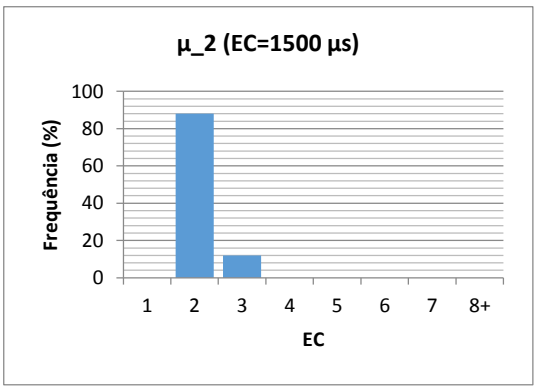


Figura 98: Histograma de μ_2 (assíncrono) *na topologia em árvore com EC=1500 µs e SW de 60%*

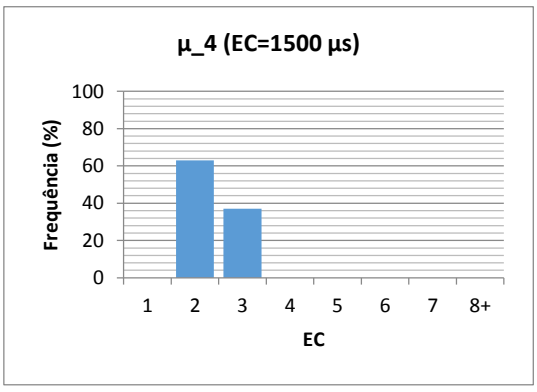


Figura 100: Histograma de μ_4 (assíncrono) *na topologia em árvore com EC=1500 µs e SW de 60%*

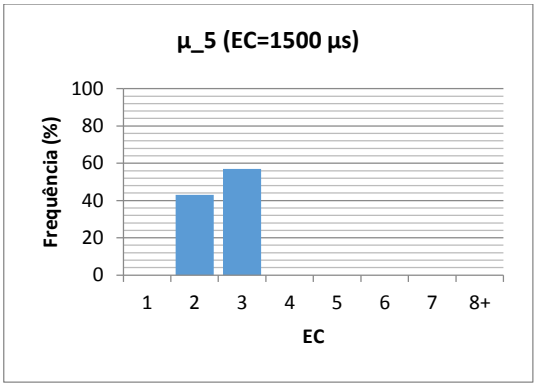


Figura 101: Histograma de μ_5 (assíncrono) *na topologia em árvore com EC=1500 µs e SW de 60%*

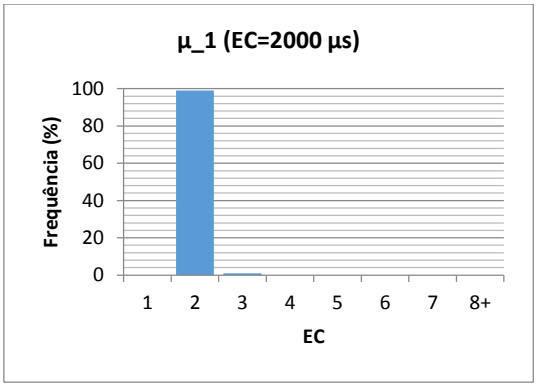


Figura 104: Histograma de μ_1 (assíncrono) *na topologia em árvore com EC=2000 µs e SW de 60%*

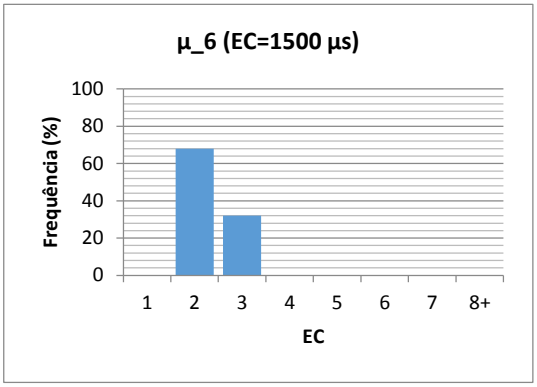


Figura 102: Histograma de μ_6 (assíncrono) *na topologia em árvore com EC=1500 µs e SW de 60%*

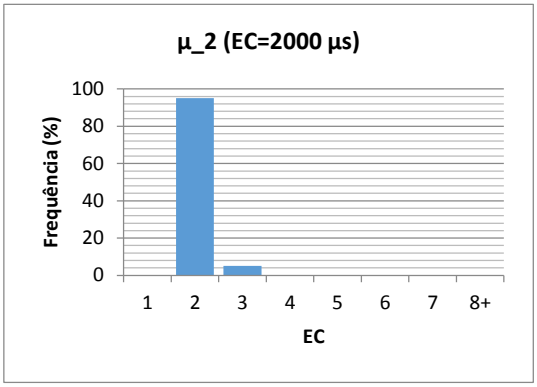


Figura 105: Histograma de μ_2 (assíncrono) *na topologia em árvore com EC=2000 µs e SW de 60%*

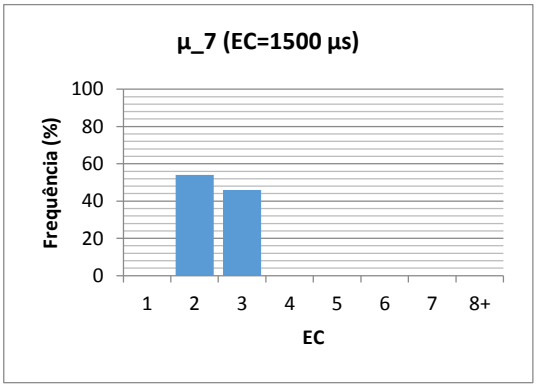


Figura 103: Histograma de μ_6 (assíncrono) *na topologia em árvore com EC=1500 µs e SW de 60%*

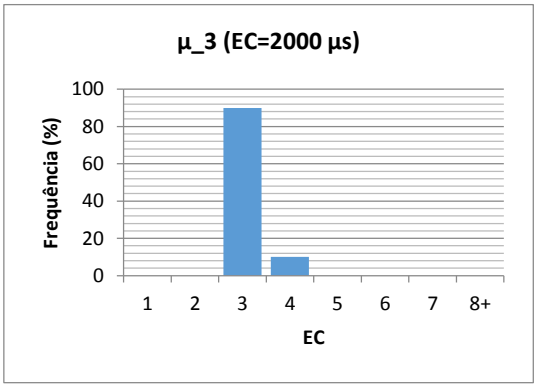


Figura 106: Histograma de μ_3 (assíncrono) *na topologia em árvore com EC=2000 µs e SW de 60%*

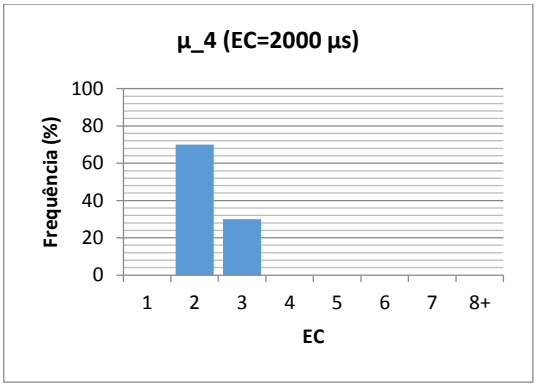


Figura 107: Histograma de μ_4 (assíncrono) *na topologia em árvore com EC=2000 μs e SW de 60%*

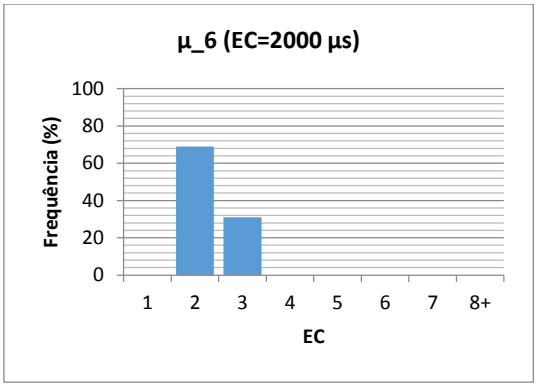


Figura 109: Histograma de μ_6 (assíncrono) *na topologia em árvore com EC=2000 μs e SW de 60%*

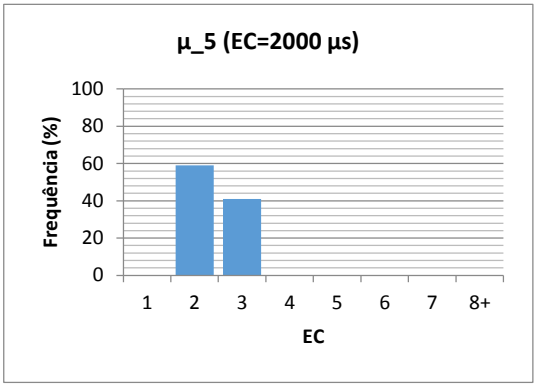


Figura 108: Histograma de μ_5 (assíncrono) *na topologia em árvore com EC=2000 μs e SW de 60%*

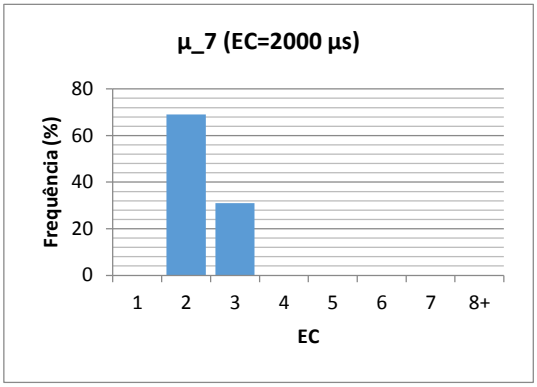


Figura 110: Histograma de μ_7 (assíncrono) *na topologia em árvore com EC=2000 μs e SW de 60%*

Analisando os resultados dos histogramas apresentados acima, considerando apenas os ECs em que as mensagens foram trocadas, verifica-se que as aplicações μ_1, μ_2, μ_4 e μ_6 revelam resultados semelhantes, quer com EC definido a 1500 μs quer a 2000 μs. Nos casos de μ_1 e μ_2, está relacionado com o facto de as suas mensagens terem um tamanho pequeno, pelo quer requerem pouca largura de banda para serem transmitidas. Já nos casos de μ_4 e μ_6, tal se verifica porque as suas mensagens têm prioridades mais altas nas ligações que atravessam. As aplicações μ_3, μ_5 e μ_7 sofrem pequenas variações devido a interferência com os tráfego de multimédia, multimédia (vídeo) e TV Vídeo, respetivamente.

Contudo, embora os resultados (em termos de ECs) sejam semelhantes, quer com EC de 1500 μs, quer com EC de 2000 μs, os resultados com um EC 2000 μs são claramente menos satisfatórios, uma vez que os benefícios de oferecer uma maior largura de banda por EC são

visivelmente inferiores ao prejuízo de acomodar os atrasos relacionados com o mecanismo de *signalling*, como pode ser comprovado pelos resultados apresentados na Figura 95 eFigura 96.

6.1.2.4 Variação da Synchronous Window com tráfego assíncrono

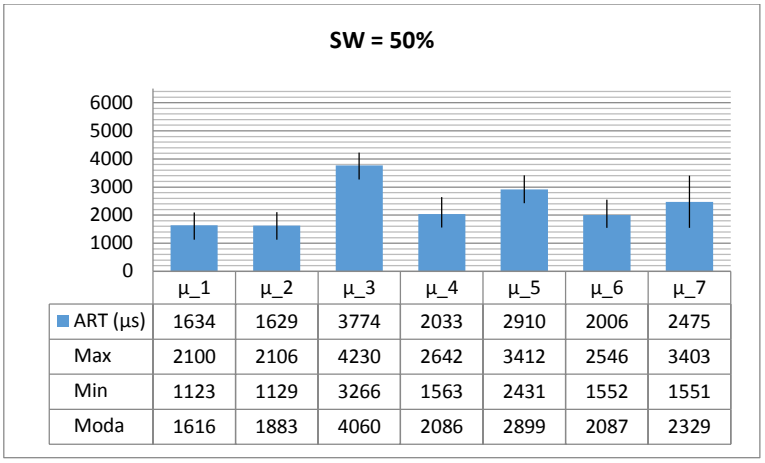
Neste capítulo, são também consideradas as aplicações apresentadas na Tabela 7 para as simulações. Nesta secção, é analisado o impacto que o tamanho definido para a *Synchronous Window* tem transmissão de tráfego assíncrono. Realizaram-se simulações com SW de 50% e 70%. Para todas as simulações foi definido um EC de tamanho 1000 μs e 100 μs foram reservados para a *Signalling Window*. A Tabela 13 reúne as características da rede FTT-SE utilizadas nestas simulações.

Tabela 13: Características da rede FTT-SE em simulações com aplicações sequenciais síncronas com diferentes Synchronous Window

Elementary Cycle	**1000 μs**	**1000 μs**
Synchronous Window	50% (450 μs)	70% (630 μs)
Asynchronous Window	50% (455 μs)	30% (270 μs)
Singnalling Window	100 μs	100 μs

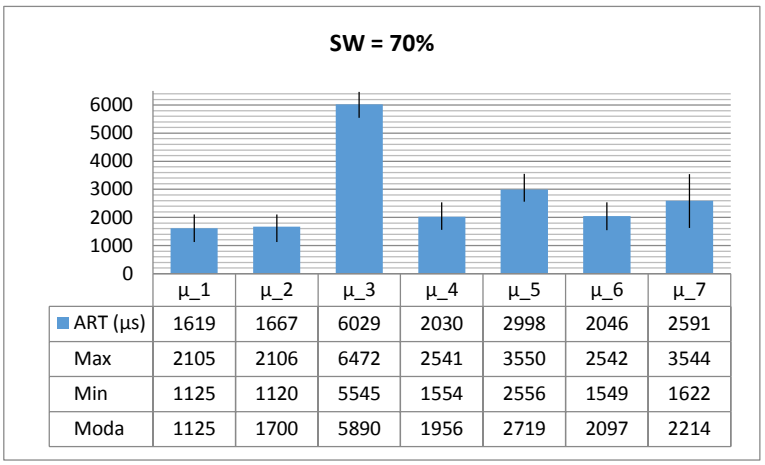
A Figura 111 mostra os resultados da simulação definindo um EC de 1000 μs e uma SW de 50%. A aplicação μ_1 obteve uma ART de 1634 μs, a aplicação μ_2 de 1629 μs, a aplicação μ_3 de 3774 μs, a aplicação μ_4 de 2033 μs, a aplicação μ_5 de 2910 μs, a aplicação μ_6 de 2006 μs e a aplicação μ_7 de 2475 μs. Todas as aplicações obtiveram ARTs inferiores ao tempo de 2 EC (2000 μs), com exceção da aplicação μ_3, que obteve uma ART inferior ao tempo de 4 ECs (4000 μs).

Na Figura 112, são revelados os resultados da simulação com EC definido a 1000 μs mas com SW de 70%. A aplicação μ_1 obteve uma ART de 1619 μs, a aplicação μ_2 de 1667 μs, a aplicação μ_3 de 6029 μs, a aplicação μ_4 de 2030 μs, a aplicação μ_5 de 2998 μs, a aplicação μ_6 de 2046 μs e a aplicação μ_7 de 2591 μs. As aplicações μ_1 e μ_2 obtiveram ARTs inferiores ao tempo de 2 EC (2000 μs), as aplicações μ_4, μ_5, μ_6 e μ_7 registaram ARTs inferiores ao tempo de 3 ECs (3000 μs) e aplicação μ_3 obteve uma ART inferior ao tempo de 4 ECs (4000 μs).



	μ_1	μ_2	μ_3	μ_4	μ_5	μ_6	μ_7
ART (μs)	1634	1629	3774	2033	2910	2006	2475
Max	2100	2106	4230	2642	3412	2546	3403
Min	1123	1129	3266	1563	2431	1552	1551
Moda	1616	1883	4060	2086	2899	2087	2329

Figura 111: Resultados da simulação com EC de 1000 μs e 50% de SW com tráfego assíncrono



	μ_1	μ_2	μ_3	μ_4	μ_5	μ_6	μ_7
ART (μs)	1619	1667	6029	2030	2998	2046	2591
Max	2105	2106	6472	2541	3550	2542	3544
Min	1125	1120	5545	1554	2556	1549	1622
Moda	1125	1700	5890	1956	2719	2097	2214

Figura 112: Resultados da simulação com EC de 1000 μs e 70% de SW com tráfego assíncrono

Comparando os resultados de ambas as simulações, verifica-se que a única diferença encontra-se na ART da aplicação μ_3 que, com uma *Synchronous Window* de 70% (ver Figura 112Figura 122), tem uma ART de 6029 μs e na simulação com SW de 50% (ver Figura 111) tem uma ART de 3774 μs, resultando numa diferença de 2255 μs. Ao aumentar a SW, está a reduzir-se a largura de banda reservada para as comunicações assíncronas, pelo que, como consequência, os RTs das aplicações que produzem mensagens de tamanho elevado, como o caso da aplicação μ_3, aumentam significativamente.

Seguidamente, são apresentados os histogramas referentes às simulações realizadas neste capítulo. As aplicações μ_1, μ_2, μ_4 e μ_6 obtiveram RTs equivalentes a 2 (de 1000 μs a 2000 μs) e a 3 (de 2000 μs a 3000 μs) ECs, sendo que μ_1 e μ_2 obtiveram uma frequência muito grande de RTs na classe de 2 ECs, quer com SW a 50% (Figura 113, Figura 114, Figura 116 e Figura 118, respetivamente), quer com SW a 70% (Figura 120, Figura 121, Figura 123 e Figura 125, respetivamente).

A aplicação μ_3, foi a que apresentou a maior diferenças entre a simulação com 50% de SW (Figura 111) e com 70% (Figura 112), sendo que para SW a 50% foram obtidos RT, maioritariamente na classe correspondente a 4 (de 3000 µs a 4000 µs) ECs, e com SW a 70% os resultados encontram-se nas classes de 6 (de 5000 µs a 6000 µs) e 7 (de 6000 µs a 7000 µs) ECs.

A aplicação μ_5, tanto na simulação com 50% de SW (Figura 111) como na simulação com 70% (Figura 112), os resultados centraram-se nas classes correspondentes de 3 (de 2000 µs a 3000 µs) e 4 (de 1000 µs a 2000 µs) ECs, enquanto a aplicação μ_7 obteve RT divididos desde a classes de 2 (de 1000 µs a 2000 µs) até à classe de 4 ECs com 50% (Figura 102) e também com 70% (Figura 109).

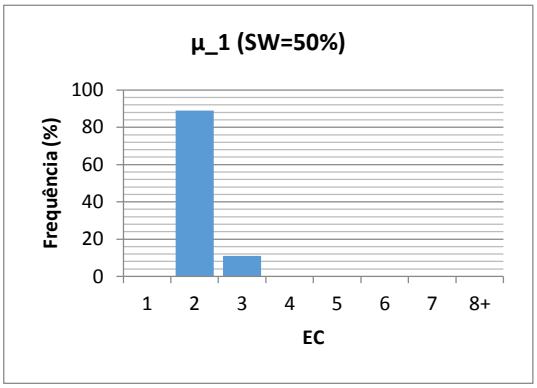


Figura 113: Histograma de μ_1 (assíncrono) *com EC=1000 µs e SW de 50%*

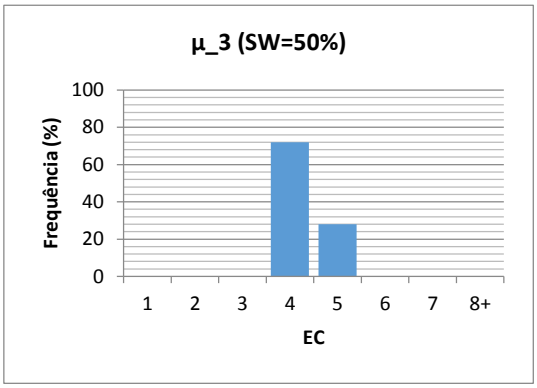


Figura 115: Histograma de μ_3 (assíncrono) *com EC=1000 µs e SW de 50%*

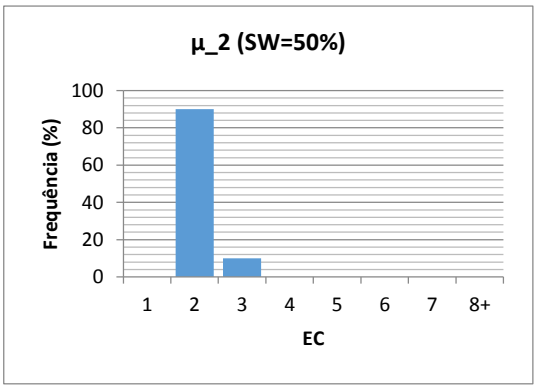


Figura 114: Histograma de μ_2 (assíncrono) *com EC=1000 µs e SW de 50%*

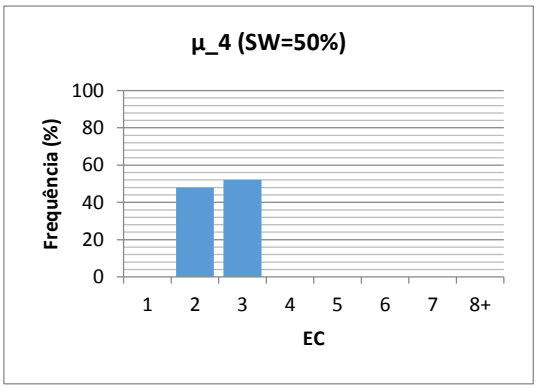


Figura 116: Histograma de μ_4 (assíncrono) *com EC=1000 µs e SW de 50%*

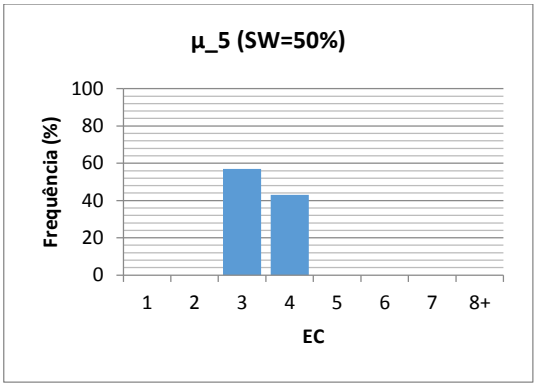


Figura 117: Histograma de μ_5 (assíncrono) *com EC=1000 µs e SW de 50%*

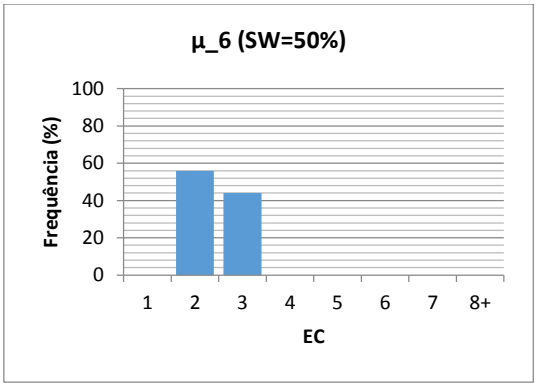


Figura 118: Histograma de μ_6 (assíncrono) *com EC=1000 µs e SW de 50%*

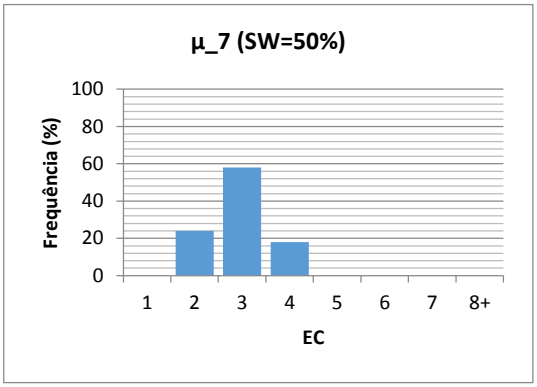


Figura 119: Histograma de μ_7 (assíncrono) *com EC=1000 µs e SW de 50%*

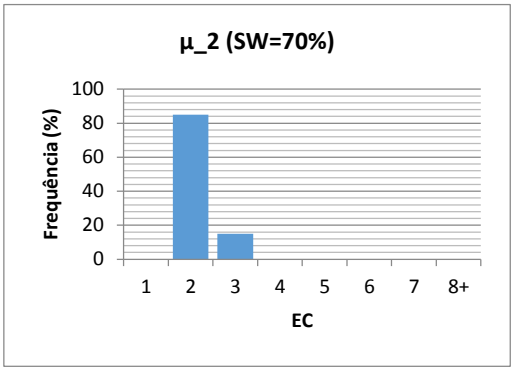


Figura 121: Histograma de μ_2 (assíncrono) *com EC=1000 µs e SW de 70%*

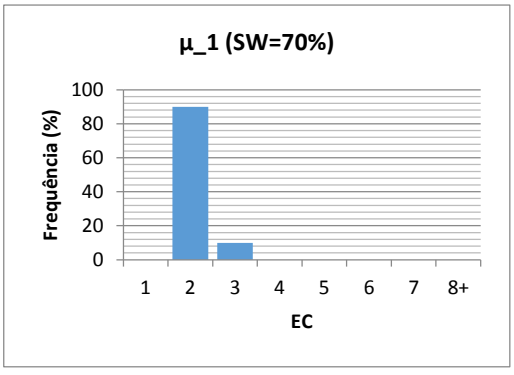


Figura 120: Histograma de μ_1 (assíncrono) *com EC=1000 µs e SW de 70%*

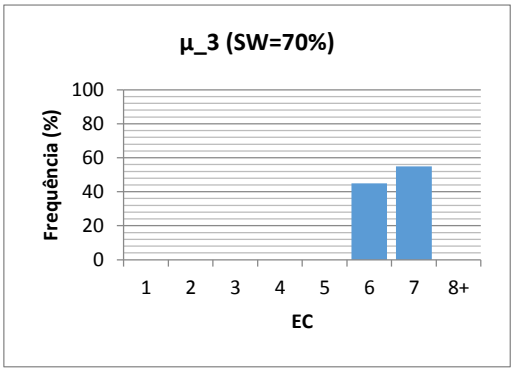


Figura 122: Histograma de μ_3 (assíncrono) *com EC=1000 µs e SW de 70%*

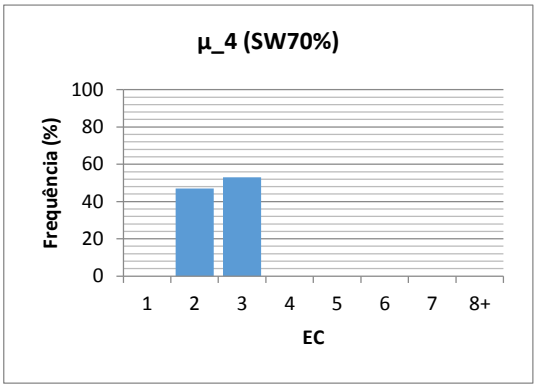


Figura 123: Histograma de μ_4 (assíncrono) *com EC=1000 μs e SW de 70%*

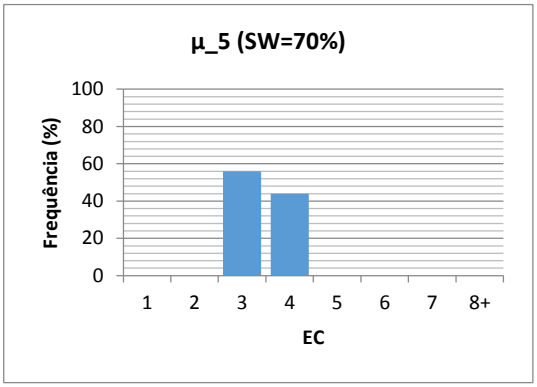


Figura 124: Histograma de μ_5 (assíncrono) *com EC=1000 μs e SW de 70%*

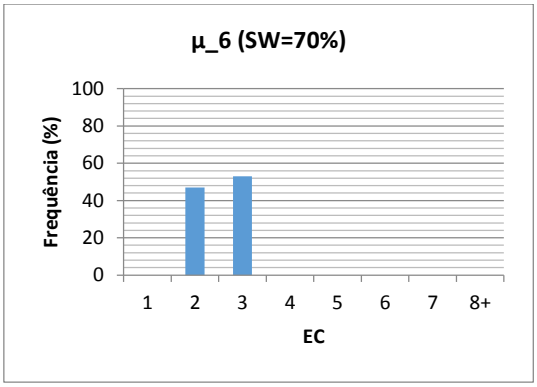


Figura 125: Histograma de μ_6 (assíncrono) *com EC=1000 μs e SW de 70%*

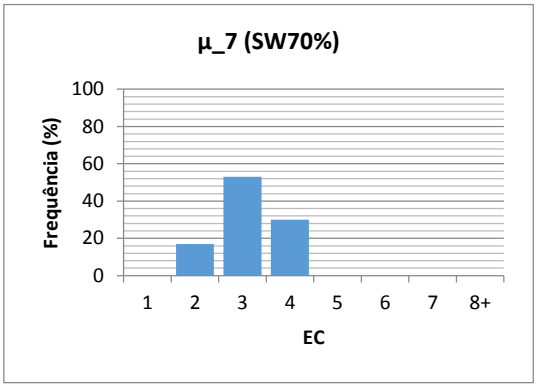


Figura 126: Histograma de μ_7 (assíncrono) *com EC=1000 μs e SW de 70%*

Os resultados apresentados nos histogramas desta experiência revelam que a diferença mais significativa comparando o cenário com SW a 50% e com 70% está nos RT obtidos na aplicação μ_3. Reduzir a largura de banda para o tráfego assíncrono tem um impacto significativo nos RTs das aplicações com grandes quantidades de tráfego e com prioridades baixas, uma vez que o RT tende a aumentar devido à menor largura de banda disponível para a transmissão de mensagens assíncronas por EC. Desta forma, para esta experiência, a configuração de rede com 50% de SW revelou melhores resultados do que a simulação com 70% de SW.

6.2 Fork-Join Parallel/Distributed

Neste capítulo são apresentadas simulações definidas segundo o paradigma *Fork-Join Parallel/Distributed* de forma a analisar o impacto que este paradigma oferece nos tempos de resposta às tarefas dos nós da rede.

6.2.1 Configuração da rede

Como, no capítulo 6.1, já foram analisadas as diferenças entre a utilização de várias topologias de rede, as experiências deste capítulo apenas se vão centrar na topologia em árvore, uma vez que resulta de uma combinação entre a topologia em estrela e a topologia *daisy chain*, revelando um equilíbrio entre as vantagens e desvantagens de ambas as topologias, e também por ser uma topologia que oferece adaptabilidade à estrutura físicas das aplicações e baixos custos. Todas as ligações entre os vários nós da rede suportam a velocidade de 100 Mbits/s. (ver Figura 127) Além disso, é importante relembrar que as ligações entre os nós da rede são *full-duplex*, pelo que permitem a transmissão e receção de mensagens em simultâneo.

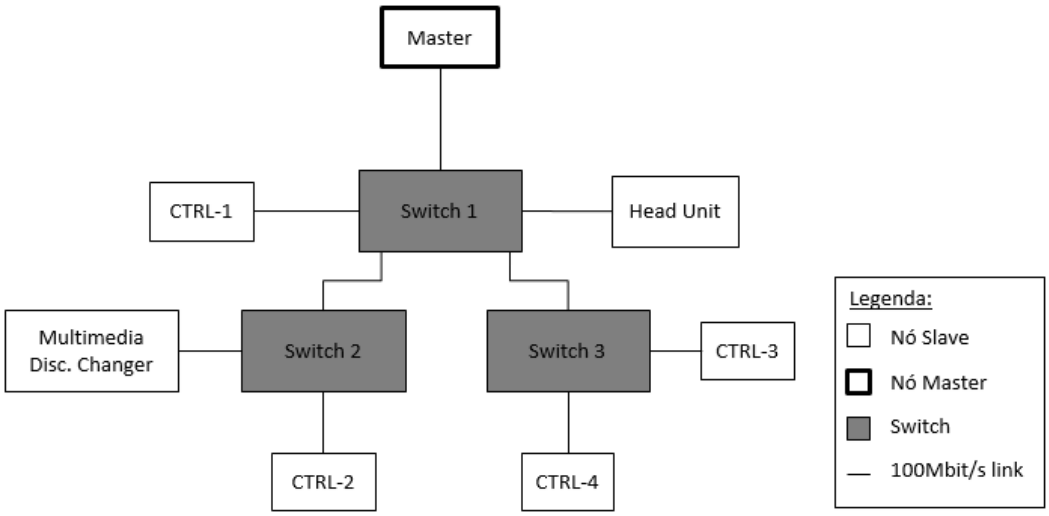


Figura 127: Topologia de rede utilizada na simulação de comunicações Fork-Join Parallel Distributed

As experiências a seguir apresentadas envolvem a combinação de aplicações que produzem mensagens em comunicações sequenciais e em comunicações paralelas/distribuídas. As aplicações sequenciais compreendem o envio de uma mensagem, seguido da computação remota de uma *thread* durante um determinado período de tempo. Já as aplicações paralelas/distribuídas (P/D) cumprem um procedimento mais complexo. Aplicações P/D iniciam com a execução sequencial de uma *thread*, à qual é seguido de uma operação de *fork*

(*D-Fork*) que resulta na definição de vários segmentos chamados *Distributed Execution Paths* (DEP). No final é executada uma operação de *join* (*D-Join*) que agrega os resultados obtidos, na *thread* do nó local. (ver capítulo 3.6)

Cada segmento (DEP) implica o envio de uma mensagem (μ) para um nó remoto, que procede à execução remota de uma *thread* (θ) durante uma determinada quantidade de tempo, seguido do envio de uma outra mensagem desde o nó remoto até ao nó local.

Neste capítulo o *response time* avaliado não envolve apenas o tempo de transmissão da mensagem na rede. Nas experiências a seguir apresentadas, para além da transmissão na rede, também existe uma execução remota de tarefas que concorrem entre si pelo processador. Desta forma, para efeitos de representação, no capítulo 6.2 cada aplicação em análise é apresentada como τ_i.

6.2.2 Aplicações P/D síncronas

Para as experiências das secções 6.2.2.1 e 6.2.2.2, foi definido com uma duração de 1500 µs de *Elementary Cycle*, sendo reservados 50% do EC (700 µs) para a *Synchronous Window* e 100 µs são reservados para a *Signalling Window*. Todas as *streams* utilizadas nestas simulações são do tipo síncrono. As características da rede FTT-SE utilizada nestas simulações encontram-se reunidas na Tabela 14.

Tabela 14: Características da rede FTT-SE em simulações com aplicações P/D síncronas

Elementary Cycle	**1500 µs**
Synchronous Window	50% (700 µs)
Asynchronous Window	50% (700 µs)
Singnalling Window	100 µs

6.2.2.1 Simulação de aplicações P/D síncronas com 8 threads

Nesta simulação existem mensagens de controlo, que são produzidas pelas aplicações sequenciais τ_1, τ_2 e τ_3 e têm origem no nó "Head-Unit", sendo produzidas mensagens com tamanho de 350 *bytes*, equivalente a um WCML de 30 µs, com um período de 5 ECs. Porém, a aplicação τ_1 tem como destino uma aplicação do nó "CTRL-1", a τ_2 tem como destino uma aplicação do nó "CTRL-2" e a aplicação τ_3 envia para uma aplicação do nó "CTRL-3". Cada uma das três aplicações referidas anteriormente realiza uma comunicação sequencial e tem associado um WCET remoto de 80 µs.

Já no caso da tarefa associada à aplicação τ_4, é realizada uma comunicação paralela/distribuída, sendo geradas em 8 *threads*, cada uma relacionada com um DEP, que compreendem o envio de mensagens de tamanho de 1400 *bytes* (equivalente a um WCML de 114 µs) a cada 10 ECs. As 8 *threads* são divididas por 4 destinatários diferentes (aplicações nos nós "CTRL-1", "CTRL-2", "CTRL-3" e "CTRL-4"), sendo associadas a cada um deles 2 *threads*. A cada *thread* está associada uma execução remota de 800 µs. Como as operações de *D-Fork* e *D-Join* não são livres de custos e provocam atrasos, são considerados 150 µs como o custo destas operações.

Por fim, é considerada uma comunicação sequencial realizada pela aplicação τ_5, que compreende o envio de uma mensagem de 2000 *bytes* (equivalente a um WCML de 163 µs) a cada 30 ECs e tem associado um WCET remoto de 350 µs, enviando para uma aplicação instalada no nó "CTRL-1". AS características de todas as aplicações desta simulação encontram-se resumidas na Tabela 15.

Tabela 15: Características das aplicações síncronas na simulação com 8 P/D threads

App	Categ.	Período (ECs)	WCET (F/J)	WCET (Rem.)	Tamanho (bytes)	WCML	Origem	Destino
τ_1	Controlo	5	-	80 µs	350	30 µs	Head Unit	CTRL-1
τ_2	Controlo	5	-	80 µs	350	30 µs	Head Unit	CTRL-2
τ_3	Controlo	5	-	80 µs	350	30 µs	Head-Unit	CTRL-3
τ_4	MM Vídeo	10	150 µs	800 µs	1400	114 µs	Mult. Disc	CTRL-1 – CTRL-4
τ_5	MM Áudio	30	-	350 µs	2000	163 µs	Mult. Disc	CTRL-1

A Figura 128 mostra os resultados obtidos pela simulação realizada. Verifica-se que as aplicações sequenciais de controlo (τ_1, τ_2 e τ_3) obtiveram uma ART de 228 µs, de 288 µs e de 319 µs, respetivamente, sendo que os seus resultados se revelam inferiores ao tempo de 1 EC (1500 µs). Já a aplicação P/D τ_4 registou ART de 8303 µs, inferior ao valor de 6 ECs (9000 µs). Por fim, a aplicação sequencial τ_5 obteve uma ART de 2481, valor inferior ao tempo de 2 ECs (3000 µs).

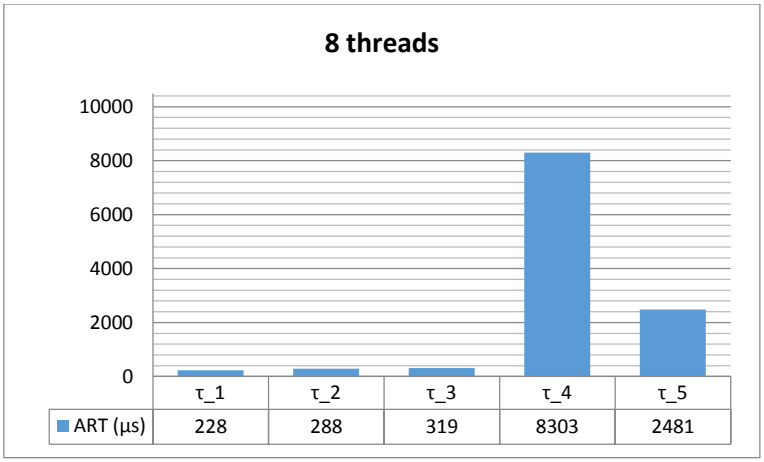


Figura 128: Resultados da simulação com 8 P/D threads síncronas

Relativamente aos resultados da aplicação P/D τ_4, é importante referir que, para a definição do seu *response time*, é considerado o pior RT obtido entre as 8 *threads* que lhes estão associadas. Só é possível definir o *response time* da tarefa de τ_4 quando todas as suas *threads* cumprirem a sua execução.

Porém, cada uma das 8 *threads* tem o seu próprio *response time*. A Figura 129 apresenta a média dos *response times* obtidos para cada uma das *threads*. As *threads* $DP_{4,2,1}$ *e* $DP_{4,2,2}$ tinham como nó remoto o "CTRL-1" e as ARTs obtidas foram de 3575 µs e de 5076 µs, respetivamente. Já $DP_{4,2,3}$ *e* $DP_{4,2,4}$ tinham como nó remoto o "CTRL-2", obtendo, respetivamente, ARTs de 4961 µs e de 6461 µs. As *threads* $DP_{4,2,5}$ *e* $DP_{4,2,6}$ tiham o "CTRL-3" como nó remoto e registaram uma ART de 5190 µs e de 8188 µs. Por fim restam as *threads* $DP_{4,2,7}$ *e* $DP_{4,2,8}$ que tinham em comum o nó remoto "CTRL-4", obtendo resultados de 6688 µs e de 8303 µs.

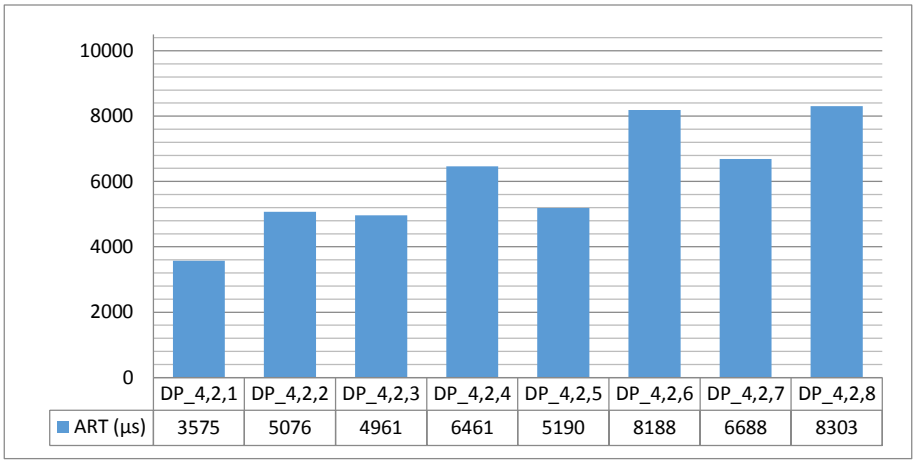


Figura 129: Resultados obtidos para cada uma das 8 P/D threads síncronas

6.2.2.2 Simulação de aplicações P/D síncronas com 16 threads

Nesta simulação são utilizadas as mesmas aplicações que integraram a simulação apresentada no capítulo 6.2.2.1. τ_1, τ_2 e τ_3 representam aplicações sequenciais, com origem no nó "Head-Unit", produtoras de mensagens de 350 *bytes*, equivalente a um WCML de 30 µs, com um período de 5 ECs. A aplicação τ_1 tem como destino uma aplicação do nó "CTRL-1", a τ_2 tem como destino uma aplicação instalada no nó "CTRL-2" e a aplicação τ_3 envia para uma aplicação do nó "CTRL-3". Cada uma das três aplicações referidas anteriormente realiza uma comunicação sequencial e tem associado um WCET remoto de 80 µs.

Contudo, relativamente à aplicação P/D τ_4, nesta experiência é duplicado o número de *threads* geradas, compreendendo, portanto, 16 *threads.* Como o número de *threads* foi duplicado, no sentido de manter as características da aplicação, o tamanho das suas mensagens, o WCML e o WCET remoto foram reduzidos para metade. Desta forma, cada *thread*, relacionada com um PED, envia mensagens de 700 *bytes* (equivalente a um WCML de 57 µs) a cada 10 ECs. São considerados os mesmos nós remotos ("CTRL-1", "CTRL-2", "CTRL-3" e "CTRL-4"), sendo que cada um deles está associado a 4 das 16 *threads*. Para cada *thread* está definida uma execução remota de 400 µs. As operações de *D-Fork* e *D-Join* mantêm um custo de 150 µs.

Por fim, é considerada uma comunicação sequencial realizada pela aplicação τ_5, que compreende o envio de uma mensagem de 2000 *bytes* (equivalente a um WCML de 163 µs) a cada 30 ECs e tem associado um WCET remoto de 350 µs, enviando para uma aplicação instalada no nó "CTRL-1". As características de todas as aplicações desta simulação encontram-se reunidas na Tabela 16.

Tabela 16: Características das aplicações síncronas na simulação com 16 P/D threads

App	Categ.	Período (ECs)	WCET (F/J)	WCET (Rem.)	Tamanho (bytes)	WCML	Origem	Destino
τ_1	Controlo	5	-	80 µs	350	30 µs	Head Unit	CTRL-1
τ_2	Controlo	5	-	80 µs	350	30 µs	Head Unit	CTRL-2
τ_3	Controlo	5	-	80 µs	350	30 µs	Head-Unit	CTRL-3
τ_4	MM Vídeo	10	150 µs	400 µs	700	57 µs	Mult. Disc	CTRL-1 – CTRL-4
τ_5	MM Áudio	30	-	350 µs	2000	163 µs	Mult. Disc	CTRL-1

A Figura 130 mostra os resultados obtidos pela simulação realizada. Verifica-se que as aplicações sequenciais de controlo (τ_1, τ_2 e τ_3) obtiveram uma ART de 228 µs, de 288 µs e de 319 µs, respetivamente, sendo que os seus resultados se revelam inferiores ao tempo de 1 EC (1500 µs). Já a aplicação P/D τ_4 registou ART de 8017 µs, inferior ao valor de 6 ECs (9000 µs). Por fim, a aplicação sequencial τ_5 obteve uma ART de 3564 µs, valor inferior ao tempo de 3 ECs (4500 µs).

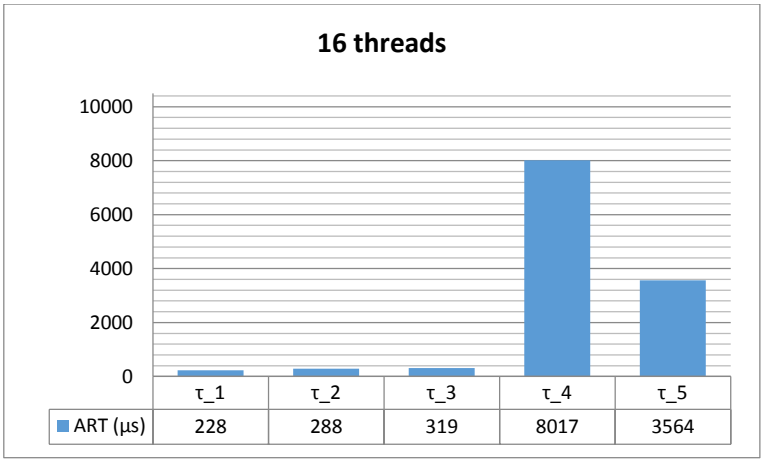
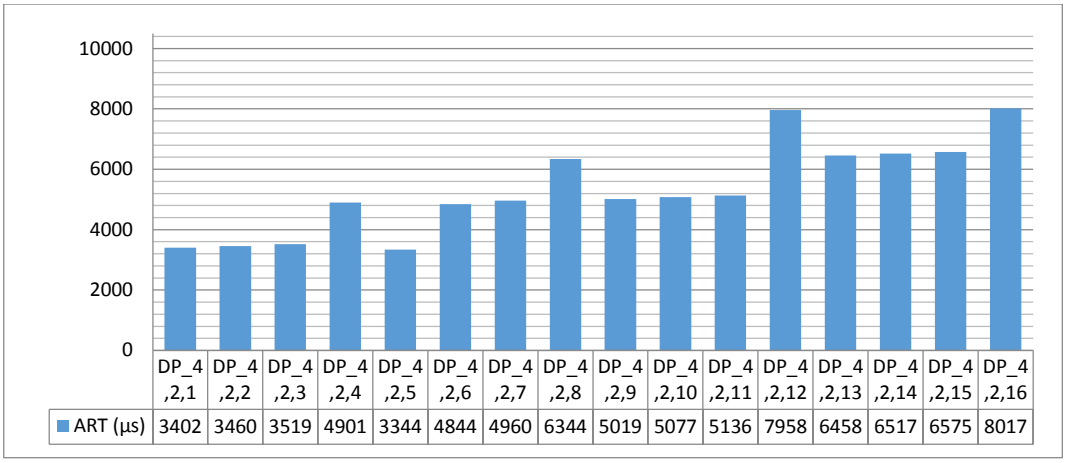


Figura 130: Resultados da simulação com 16 P/D threads síncronas

Tal como referido no capítulo 6.2.2.1, a definição do *response time* de uma tarefa P/D implica uma análise a todos os *response times* obtidos pelas várias *threads* geradas. Apenas quando todas as *threads* tiverem sido executadas é que a execução da tarefa principal termina. Desta forma, o resultado de τ_4 compreendeu o pior RT obtido entre as 16 *threads* que lhes estão associadas.

A Figura 126 apresenta a média dos *response times* obtidos para cada uma das 16 *threads* consideradas na aplicação τ_4. As *threads* $DP_{4,2,1}$, $DP_{4,2,2}$, $DP_{4,2,3}$ e $DP_{4,2,4}$ tinham como nó remoto o "CTRL-1" e registaram ARTs de 3402 µs, de 3460 µs, de 3519 µs e de 4901 µs, respetivamente. Já $DP_{4,2,5}$, $DP_{4,2,6}$, $DP_{4,2,7}$ e $DP_{4,2,8}$ tinham como nó remoto o "CTRL-2", obtendo ARTs de 3344 µs, de 4844 µs, de 4960 µs e de 6344 µs, respetivamente. As *threads* $DP_{4,2,9}$, $DP_{4,2,10}$, $DP_{4,2,11}$ e $DP_{4,2,12}$ tinham o "CTRL-3" como nó remoto e registaram ARTs de 5019 µs, de 5077 µs, de 5136 µs e de 7958 µs, respetivamente. Por fim, as *threads* $DP_{4,2,13}$, $DP_{4,2,14}$, $DP_{4,2,15}$ e $DP_{4,2,16}$, que tinham em comum o nó remoto "CTRL-4", obtiveram resultados de 6458 µs, de 6517 µs, de 6575 µs e de 8017 µs, respetivamente .



	DP_4,2,1	DP_4,2,2	DP_4,2,3	DP_4,2,4	DP_4,2,5	DP_4,2,6	DP_4,2,7	DP_4,2,8	DP_4,2,9	DP_4,2,10	DP_4,2,11	DP_4,2,12	DP_4,2,13	DP_4,2,14	DP_4,2,15	DP_4,2,16
ART (µs)	3402	3460	3519	4901	3344	4844	4960	6344	5019	5077	5136	7958	6458	6517	6575	8017

Figura 131: Resultados obtidos para cada uma das 16 P/D threads síncronas

6.2.2.3 Análise aos resultados das aplicações P/D síncronas

Neste capítulo, é realizada uma análise aos resultados apresentados nas secções 6.2.2.1 e 6.2.2.2, onde foram realizadas simulações com aplicações P/D síncronas. O principal foco das simulações é perceber o impacto que tem o número de *threads* associada a uma aplicação P/D no seu *response time*. No caso da simulação da secção 6.2.2.1, foi utilizada uma aplicação P/D que gerou 8 *threads*, e no caso da secção 6.2.2.2, o número de *threads* foi duplicado, havendo, portanto, uma aplicação P/D síncrona que gerou 16 *threads*.

Comparando e analisando os resultados de ambas simulações (Figura 129 e Figura 131), verifica-se que, relativamente às aplicações sequenciais τ_1, τ_2 e τ_3, não existiu qualquer variação. Este facto justifica-se pela prioridade das mensagens destas aplicações, uma vez que são as que têm prioridade mais elevada e, por isso, tendem a ser escalonadas mais cedo. Desta forma, as alterações no número de *threads* da aplicação τ_3 não evidenciou qualquer efeito nas aplicações τ_1, τ_2 e τ_3.

Relativamente à aplicação P/D τ_4, analisando os resultados em ambas as experiências, verifica-se que o aumento do número de *threads* geradas pela aplicação resultou numa ligeira descida nos resultados registados para a ART. A descida do *response time* está relacionada com a diminuição do tamanho das mensagens associadas a cada *thread*. Ao diminuir o tamanho de cada mensagem, aumentam as possibilidades para que mais mensagens possam ser escalonadas em cada EC, uma vez que a granularidade em relação às mensagens aumenta.

No entanto, um efeito diferente foi verificado no *response time* de τ_5. Com o aumento do número de *threads* na aplicação τ_4, o seu *response time* aumentou. Este aumento é causado pela interferência com o tráfego gerado pela aplicação τ_4, uma vez que partilham ligações no percurso das suas mensagens, que com o aumento do número de *threads,* aumentou também

o número de mensagens produzidas. Como a aplicação τ_5 tem menos prioridade e, não havendo largura de banda suficiente para ambos transmitirem, o escalonamento das suas mensagens sofre um atraso.

Os resultados também registam algumas subidas abruptas nos resultados obtidos, tanto na simulação com 8 *threads*, nomeadamente nas *threads* $DP_{4,2,2}$, $DP_{4,2,4}$, $DP_{4,2,6}$ e $DP_{4,2,8}$, como na simulação com 16 *threads*, nas *threads* $DP_{4,2,4}$, $DP_{4,2,8}$, $DP_{4,2,12}$ e $DP_{4,2,16}$. É necessário relembrar que, em cada simulação, as *threads* são divididas em subconjuntos estando cada um deles associado a um nó de destino diferente. Desta forma, a subida nos resultados destas *threads* está relacionado com o facto de serem a últimas a transmitirem as suas mensagens na rede, uma vez que não existe largura de banda suficiente para a transmissão de todas as mensagens de cada subconjunto no mesmo EC. Consequentemente, pelo atraso na transmissão da mensagem associada a cada uma das *threads* identificadas, os seus RTs são mais elevados.

Deste modo, pode-se concluir que, para este exemplo, que aumentar o número de *threads* mantendo as características de uma aplicação, isto é, dividir os dados em mais mensagens, mas mais pequenas, revela uma ligeira diminuição do RT da aplicação P/D síncronas.

6.2.3 Aplicações P/D assíncronas

Nas experiências das secções 6.2.3.1 e 6.2.3.2, foram utilizadas as mesmas características da rede que as definidas no capítulo 6.2.2. O *Elementary Cycle* tem uma duração de 1500 µs de *Elementary Cycle*, sendo reservados 50% do EC (700 µs) para a *Synchronous Window* e 100 µs são reservados para a *Signalling Window*. Todas as *streams* utilizadas nestas simulações são do tipo assíncrono. As características da rede FTT-SE utilizada nestas simulações encontram-se reunidas na Tabela 17.

Tabela 17: Características da rede FTT-SE em simulações com aplicações P/D assíncronas

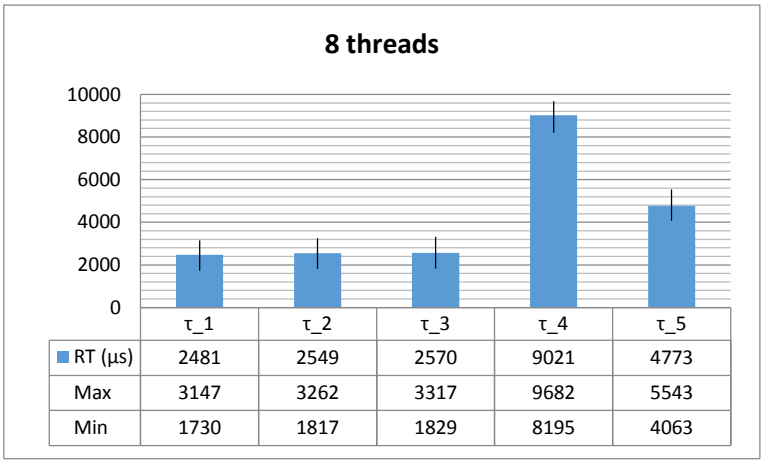
Elementary Cycle	**1500 µs**
Synchronous Window	50% (700 µs)
Asynchronous Window	50% (700 µs)
Singnalling Window	100 µs

6.2.3.1 Simulação de aplicações P/D assíncronas com 8 threads

Nesta simulação foram utilizadas as mesmas aplicações que na secção 6.2.2.1, onde as características das mesmas se encontram sumarizadas na Tabela 5. As aplicações sequenciais τ_1, τ_2 e τ_3 geram mensagens de controlo com origem no nó "Head-Unit", tendo um tamanho de 350 *bytes* cada mensagem (equivalente a um WCML de 30 µs), com um período de 5 ECs. A aplicação τ_1 tem como destino uma aplicação no nó "CTRL-1", a τ_2 tem como destino uma aplicação no nó "CTRL-2" e a aplicação τ_3 tem como destino uma aplicação no nó "CTRL-3". Cada uma das três aplicações referidas anteriormente realiza uma comunicação sequencial e tem associado um WCET remoto de 80 µs.

A aplicação P/D τ_4, gera 8 *threads*, cada uma relacionada com um DEP, e enviam, a cada 10 ECs, mensagens de 1400 *bytes* (correspondente a um WCML de 114 µs). As 8 *threads* são divididas por 4 destinatários diferentes (aplicações dos nós "CTRL-1", "CTRL-2", "CTRL-3" e "CTRL-4"), sendo associadas a cada um deles 2 *threads*. A cada *thread* está associada uma execução remota de 800 µs. As operações de *D-Fork* e *D-Join* tem um custo de 150 µs associado. A aplicação sequencial τ_5, envia uma mensagem de 2000 *bytes* (equivalente a um WCML de 163 µs) a cada 30 ECs e tem associado um WCET remoto de 350 µs enviando para uma aplicação instalada no nó "CTRL-1".

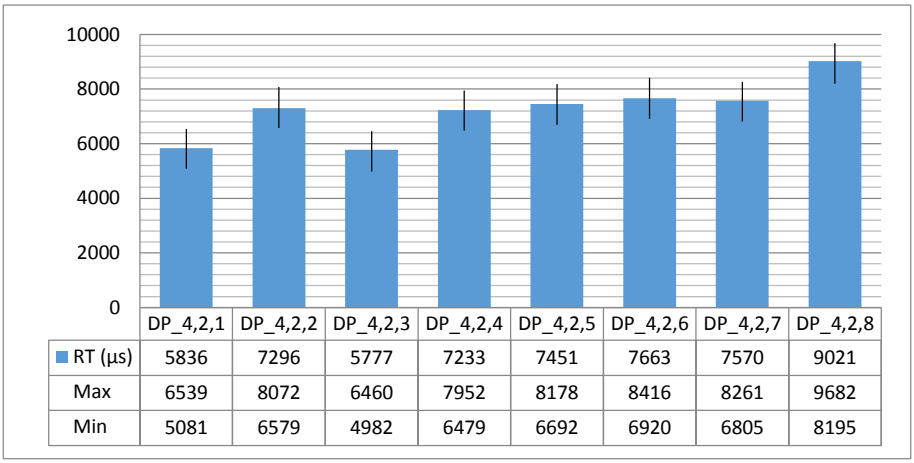
A Figura 132 apresenta a média dos *response times* obtidos para cada uma das 8 *threads*. As aplicações sequenciais de controlo (τ_1, τ_2 e τ_3) registaram uma ART de 2481 µs, de 2549 µs e de 2570 µs, respetivamente, sendo os seus resultados inferiores ao tempo de 2 ECs (3000 µs). Já a aplicação P/D τ_4 obteve uma ART de 9021 µs, inferior ao valor de 7 ECs (105000 µs). Por fim, a aplicação sequencial τ_5 obteve uma ART de 4773, valor inferior ao tempo de 4 ECs (6000 µs).



	τ_1	τ_2	τ_3	τ_4	τ_5
RT (µs)	2481	2549	2570	9021	4773
Max	3147	3262	3317	9682	5543
Min	1730	1817	1829	8195	4063

Figura 132: Resultados da simulação com 8 P/D threads assíncronas

A Figura 133 apresenta as ARTs obtidas para cada uma das 8 *threads* da aplicação P/D τ_4. As *threads* $DP_{4,2,1}$ *e* $DP_{4,2,2}$ tinham como nó remoto o "CTRL-1" e as ARTs obtidas foram de 5836 µs e de 7296 µs, respetivamente. Já $DP_{4,2,3}$ *e* $DP_{4,2,4}$ tinham como nó remoto o "CTRL-2", obtendo, respetivamente, ARTs de 5777 µs e de 7233 µs. As *threads* $DP_{4,2,5}$ *e* $DP_{4,2,6}$ tiham o "CTRL-3" como nó remoto e registaram uma ART de 7451 µs e de 7663 µs. Por fim restam as *threads* $DP_{4,2,7}$ *e* $DP_{4,2,8}$ que tinham em comum o nó remoto "CTRL-4", obtendo resultados de 7570 µs e de 9021 µs.



	DP_4,2,1	DP_4,2,2	DP_4,2,3	DP_4,2,4	DP_4,2,5	DP_4,2,6	DP_4,2,7	DP_4,2,8
RT (µs)	5836	7296	5777	7233	7451	7663	7570	9021
Max	6539	8072	6460	7952	8178	8416	8261	9682
Min	5081	6579	4982	6479	6692	6920	6805	8195

Figura 133: Resultados obtidos para cada uma das 8 P/D threads assíncronas

6.2.3.2 Simulação de aplicações P/D assíncronas com 16 threads

Nesta simulação são utilizadas as mesmas aplicações que integraram a simulação apresentada na secção 6.2.3.1. τ_1, τ_2 e τ_3 são aplicações sequenciais com origem em "Head-Unit" e enviam mensagens de 350 *bytes* de tamanho, equivalente a um WCML de 30 µs, com um período de 5 ECs. A aplicação τ_1 tem como destino uma aplicação instalada no o nó "CTRL-1", a τ_2 tem como destino uma aplicação no nó "CTRL-2" e a aplicação τ_3 envia para uma aplicação no nó "CTRL-3". Cada uma das três aplicações referidas anteriormente realiza uma comunicação sequencial e tem associado um WCET remoto de 80 µs.

Tal como na simulação da secção 6.2.2.2, na aplicação P/D τ_4 é duplicado o número de *threads* geradas, compreendendo, portanto, 16 *threads.* De forma a manter as características da aplicação, o tamanho das suas mensagens, o WCML e o WCET remoto foram reduzidos para metade. Em cada *thread* é enviada uma mensagem com 700 *bytes* de tamanho (equivalente a um WCML de 57 µs) a cada 10 ECs. São considerados os mesmos nós remotos ("CTRL-1", "CTRL-2", "CTRL-3" e "CTRL-4"), sendo que cada um deles está associado a 4 das 16 *threads*. Cada *thread* tem uma execução remota de 400 µs. As operações de *D-Fork* e *D-Join* mantêm um custo de 150 µs.

Na aplicação sequencial τ_5, é enviada uma mensagem de 2000 *bytes* (equivalente a um WCML de 163 µs) a cada 30 ECs para uma aplicação instalada no nó "CTRL-1" e tem associado um WCET remoto de 350 µs. A Tabela 18 reúne todas as características das aplicações que integram esta simulação.

Tabela 18: Características das aplicações assíncronas na simulação com 16 P/D threads

App	Cat.	Período (ECs)	WCET (F/J)	WCET (Rem.)	Tamanho (bytes)	WCML	Origem	Destino
τ_1	Controlo	5	-	80 µs	350	30 µs	Head Unit	CTRL-1
τ_2	Controlo	5	-	80 µs	350	30 µs	Head Unit	CTRL-2
τ_3	Controlo	5	-	80 µs	350	30 µs	Head-Unit	CTRL-3
τ_4	MM Vídeo	10	150 µs	400 µs	700	57 µs	Mult. Disc	CTRL-1 – CTRL-4
τ_5	MM Áudio	30	-	350 µs	2000	163 µs	Mult. Disc	CTRL-1

A Figura 134 mostra os resultados obtidos pela simulação realizada. Verifica-se que as aplicações sequenciais de controlo (τ_1, τ_2 e τ_3) obtiveram uma ART de 2481 µs, de 2489 µs e de 2559 µs, respetivamente, sendo que os seus resultados se revelam inferiores ao tempo de 2 EC (3000 µs). Já a aplicação P/D τ_4 registou ART de 8838 µs, inferior ao valor de 6 ECs (9000 µs). Por fim, a aplicação sequencial τ_5 obteve uma ART de 4830 µs, valor inferior ao tempo de 4 ECs (6000 µs).

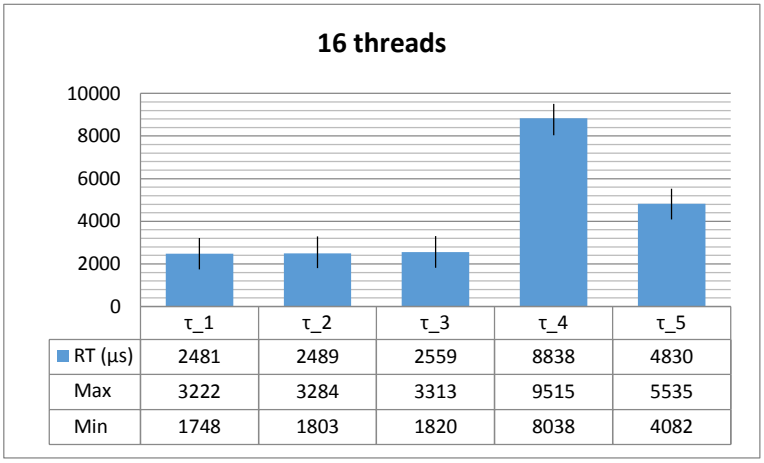
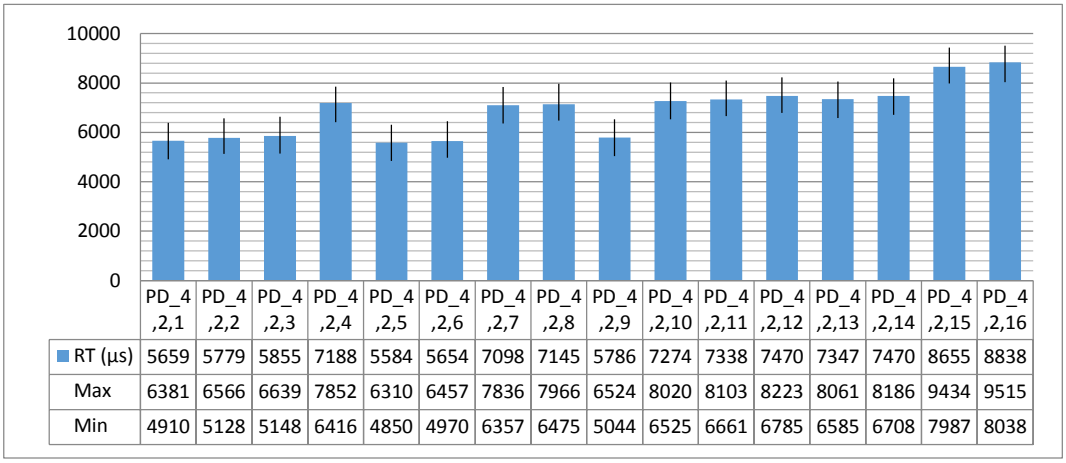


Figura 134: Resultados da simulação com 16 P/D threads assíncronas

A Figura 135 apresenta a ART de cada uma das 16 *threads* consideradas na aplicação τ_4. As *threads* $DP_{4,2,1}$, $DP_{4,2,2}$, $DP_{4,2,3}$ e $DP_{4,2,4}$ tinham como nó remoto o "CTRL-1", obtendo ARTs de 5659 µs, de 5779 µs, de 5855 µs e de 7188 µs, respetivamente. Já $DP_{4,2,5}$, $DP_{4,2,6}$, $DP_{4,2,7}$ e $DP_{4,2,8}$ tinham como nó remoto o "CTRL-2", registando ARTs de 5584 µs, de 5654 µs, de 7098 µs e de 7145 µs, respetivamente. As *threads* $DP_{4,2,9}$, $DP_{4,2,10}$, $DP_{4,2,11}$ e $DP_{4,2,12}$ tinham o "CTRL-3" como nó remoto e registaram ARTs de 5786 µs, de 7274 µs, de 7338 µs e de 7470

µs, respetivamente. Por fim, as *threads* $DP_{4,2,13}$, $DP_{4,2,14}$, $DP_{4,2,15}$ e $DP_{4,2,16}$ que tinham em comum o nó remoto "CTRL-4", obtiveram resultados de 7347 µs, de 7470 µs, de 8655 µs e de 8838 µs, respetivamente .



	PD_4,2,1	PD_4,2,2	PD_4,2,3	PD_4,2,4	PD_4,2,5	PD_4,2,6	PD_4,2,7	PD_4,2,8	PD_4,2,9	PD_4,2,10	PD_4,2,11	PD_4,2,12	PD_4,2,13	PD_4,2,14	PD_4,2,15	PD_4,2,16
RT (µs)	5659	5779	5855	7188	5584	5654	7098	7145	5786	7274	7338	7470	7347	7470	8655	8838
Max	6381	6566	6639	7852	6310	6457	7836	7966	6524	8020	8103	8223	8061	8186	9434	9515
Min	4910	5128	5148	6416	4850	4970	6357	6475	5044	6525	6661	6785	6585	6708	7987	8038

Figura 135: Resultados obtidos para cada uma das 16 P/D threads assíncronas

6.2.3.3 Análise aos resultados das aplicações P/D assíncronas

Neste capítulo é realizada uma análise aos resultados apresentados nas secções 6.2.3.1 e 6.2.3.2, onde foram realizadas simulações com aplicações P/D assíncronas. O objetivo desta análise é perceber impacto que tem o número de *threads* de a uma aplicação P/D assíncrona P/D no seu *response time*. Foram feitas simulações com 8 *threads* (secção 6.2.3.1) e com 16 *threads* (secção 6.2.3.2).

Comparando e analisando os resultados de ambas simulações (Figura 132 e Figura 134), verifica-se que, relativamente às aplicações sequenciais τ_1, τ_2 e τ_3, não existiram variações significativas, uma vez que que são as aplicações com mais prioridade na rede. Conclui-se que as alterações no número de *threads* da aplicação τ_3 não provocaram variações significativas nas aplicações sequenciais τ_1, τ_2 e τ_3.

No que diz respeito à aplicação P/D τ_4, o aumento do número de *threads* geradas pela aplicação teve como resultado uma ligeira descida na média do *response time* da aplicação. Devido à existência de mensagens mais pequenas em cada *thread*, aumentando a granularidade das mesmas, as possibilidades para que mais mensagens possam ser escalonadas em cada EC aumentam.

Em relação à aplicação sequencial τ_5, o aumento do número de *threads* na aplicação τ_4 não causou grandes alterações no seu RT. Porém, é importante referir que em ambas as simulações o RT desta aplicação sofreu interferência com o tráfego proveniente de τ_1 e τ_4, uma vez que

têm ligações em comum no percurso das suas mensagens e τ_5 tem menos prioridade, causando um atraso no escalonamento das suas mensagens.

Tal como na experiência da secção 6.2.2 pode-se concluir que, para este exemplo, que aumentar o número de *threads* mantendo as características de uma aplicação tem como consequência uma ligeira diminuição do RT da aplicação P/D assíncronas.

7 Conclusões

Neste capítulo são apresentadas as conclusões finais relativamente a todo o trabalho desenvolvido neste projeto, mencionando o sucesso ou insucesso no que diz respeito aos objetivos propostos. Além disso, são também relatados outros trabalhos realizados durante o projeto e identificadas as limitações do trabalho desenvolvido e possíveis trabalhos futuros. Este capítulo termina com uma apreciação final ao projeto de estágio.

7.1 Objetivos realizados

O objetivo principal deste projeto era a implementação da rede FTT-SE no simulador NS-3. Para alcançar esse objetivo havia que implementar uma aplicação que simulasse o comportamento de um *Master* FTT-SE da rede, que tinha como funções o escalonamento do tráfego trocado da rede. O resultado desse escalonamento é integrado na *Trigger Message*, que é enviada para todos os restantes nós da rede no início de cada EC, sendo esta mensagem também utilizada para marcar o início do mesmo.

Além disso, era também objetivo do trabalho implementar uma aplicação que simulasse o comportamento de *Slaves*, produtores ou consumidores de tráfego. Cada aplicação *Slave* tinha de estar associada a um tipo de tráfego, síncrono ou assíncrono, sendo que os produtores tinham de gerar tráfego de acordo com a natureza da aplicação, isto é, *time-triggered*, se fosse tráfego síncrono, e *event-triggered*, se o tráfego fosse de natureza assíncrona. Teria de ser

possível que uma aplicação *Slave* realizasse comunicações sequenciais, onde a mensagem gerada é enviada desde o nó local até ao nó remoto, com possibilidade de execução de uma *thread* no nó remoto durante um determinado período de tempo e associada a uma prioridade. Também deveria realizar comunicações segundo o paradigma *Fork-Join Parallel/Distributed*, onde uma mensagem é enviada desde um nó local até um nó remoto, havendo um processamento remoto durante um determinado tempo, seguido do envio de volta de uma mensagem desde o nó remoto até ao nó local. Neste tipo de comunicação, utilizando aplicações síncronas teria de ser calculado o seu *Offset* de sincronização, entre a aplicação do nó local e a do nó remoto, considerando sempre o *Worst-Case Response Time*.

Uma vez que o NS-3 não integra a funcionalidade de processamento de tarefas, era também objetivo deste trabalho simular a concorrência deste tipo de processamento em sistemas monoprocessadores, segundo a política *Rate Monotonic*, sendo uma implementação única no simulador.

Teria ainda de ser implementado um mecanismo de exportação de resultados, de forma a facilitar o tratamento dos mesmos.

Um último objetivo deste projeto era realizar simulações a partir da implementação realizada, criando vários cenários e analisar os resultados obtidos, procurando sempre justifica-los.

Pode-se concluir, após revistos os objetivos inicialmente propostos, que todos os objetivos foram cumpridos com sucesso, estando todas as funcionalidades pretendidas a funcionar corretamente e devidamente validadas.

7.2 Outros trabalhos realizados

Durante este projeto, houve tempo para a participação, em paralelo, noutro trabalho do colega de curso Roberto Duarte, que envolvia a implementação do paradigma de comunicação *Parallel/Distributed* sobre a implementação real do protocolo FTT-SE.

Esta participação revelou-se significativamente positiva para o meu projeto, uma vez que: i) permitiu analisar de perto a implementação real do protocolo; ii) validar alguns resultados do simulador com os resultados práticos; iii) esclarecer de dúvidas com o autor da implementação do protocolo, Ricardo Marau, facilitando a compreensão de alguns conceitos relacionados com o funcionamento do protocolo.

Infelizmente o trabalho do meu colega Roberto Duarte não foi totalmente concluído e consequentemente não foi possível validar mais profundamente os resultados do simulador, comparando com os resultados práticos.

7.3 Limitações e trabalho futuro

Apesar de todos os objetivos propostos para o projeto terem sido cumpridos com sucesso, existe ainda muito espaço para a adição de novas funcionalidades à implementação realizada. O protocolo FTT-SE é muito complexo e muitos outros aspetos podem ser explorados e integrados com a implementação deste projeto.

Possíveis trabalhos futuros podem ser:

- Implementação de comunicações segundo o padrão *publish*/*subcribe*;
- Integração de um mecanismo de controlo de admissão de *streams*;
- Computação do tamanho da *Signalling Window* proporcional ao tamanho da rede;
- Integrar diferentes políticas de escalonamento;
- Integrar os módulos implementados como parte da distribuição oficial do NS-3;
- Comparação com resultados da implementação real.

7.4 Apreciação final

A realização deste projeto de estágio foi, sem dúvida uma experiência muito positiva e enriquecedora. Permitiu o desenvolvimento de competências e a aplicação dos conhecimentos adquiridos ao longo da Licenciatura em Engenharia Informática. Porém, um dos fatores que mais me motivou em integrar este projeto foi a possibilidade de aprender algo de novo relacionado com uma das áreas que mais aprecio, Redes. Deste modo, foi um projeto bastante exigente, porque exigiu de mim um constante estudo relativamente ao protocolo FTT-SE, do qual, à partida, não tinha conhecimento algum mas que se revelou bastante proveitoso, uma vez que esse estudo me permitiu adquirir bastantes conhecimentos.

Para finalizar, quero também deixar o meu agradecimento ao CISTER, que me deu todas as condições para a realização de um bom projeto, integrando um ambiente de investigação onde a aprendizagem é constante e onde existe uma grande interajuda entre os membros da organização. Avaliando o resultado final, fico muito satisfeito com o trabalho desenvolvido e com a certeza de que a participação neste projeto me permitiu crescer a todos os níveis.

8 Bibliografia

[1] "Research Centre in Real-Time and Embedded Computing Systems," [Online]. Available: http://www.cister.isep.ipp.pt/.

[2] "NS-3," [Online]. Available: http://www.nsnam.org/.

[3] R. R. D. Marau, Real-time communications over switched Ethernet supporting dynamic QoS management., 2009.

[4] R. Garibay-Martinez, G. Nelissen, L. Ferreira e L. Pinho, "On the scheduling of fork-join parallel/distributed real-time tasks," em *Industrial Embedded Systems (SIES), 2014 9th IEEE International Symposium on*, 2014.

[5] H. Kopetz, Real-time systems: design principles for distributed embedded applications., 1997.

[6] E. M. D. M. Tovar, Supporting Real-Time Communications with Standard Factory-Floor Networks., 2012.

[7] C. L. &. L. J. W. Liu, "Scheduling algorithms for multiprogramming in a hard-real-time environment.," *Journal of the ACM (JACM),* 1973.

[8] P. A. L. &. G. P. Pedreiras, "The FTT-Ethernet protocol: Merging flexibility, timeliness and efficiency.," 2002.

[9] P. &. L. A. Pedreiras, "The flexible time-triggered (FTT) paradigm: an approach to QoS management in distributed real-time systems.," 2003.

[10] A. &. P. J. Ballesteros, "A description of the FTT-SE protocol.," 2013.

[11] R. F. L. L. N. G. &. P. L. M. Garibay-Martinez, Towards Holistic Analysis for Fork-Join Parallel/Distributed Real-Time Tasks., 2014.

[12] M. B. M. N. T. &. A. L. Ashjaei, Performance analysis of master-slave multi-hop switched ethernet networks., 2013.

[13] R. F. L. L. N. G. &. P. L. M. Garibay-Martinez, Holistic Analysis for Fork-Join Parallel Distributed Real-Time Tasks using the FTT-SE Protocol, To be submitted to IEEE Transactions on Industrial Informatics..

[14] "Flexible Time-Triggered Communications," [Online]. Available: http://paginas.fe.up.pt/~ftt/sections/Repository/index.html.

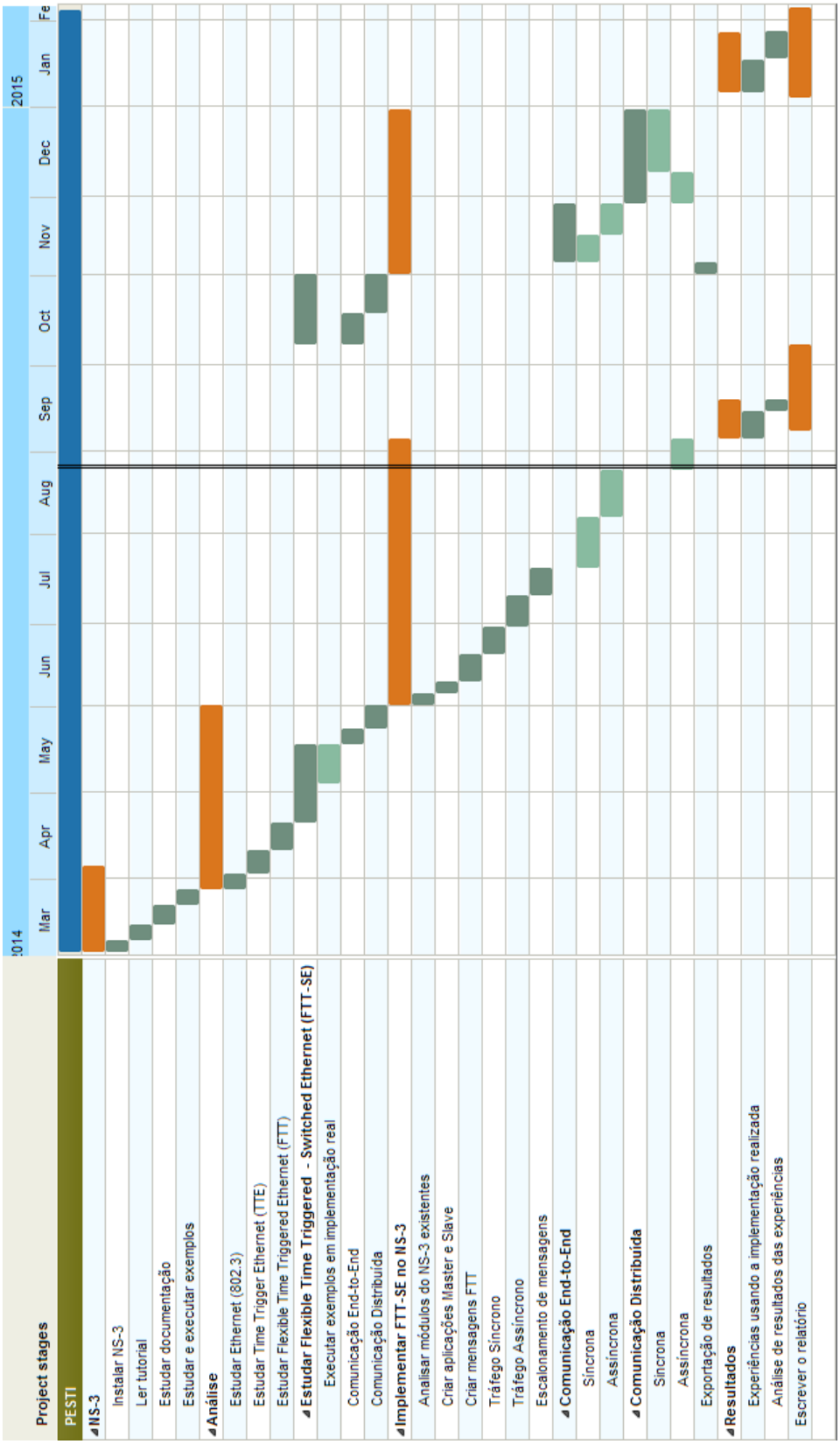
[15] NS-3, "Manual," 2014.

[16] NS-3, "Rastreamento," 2013.

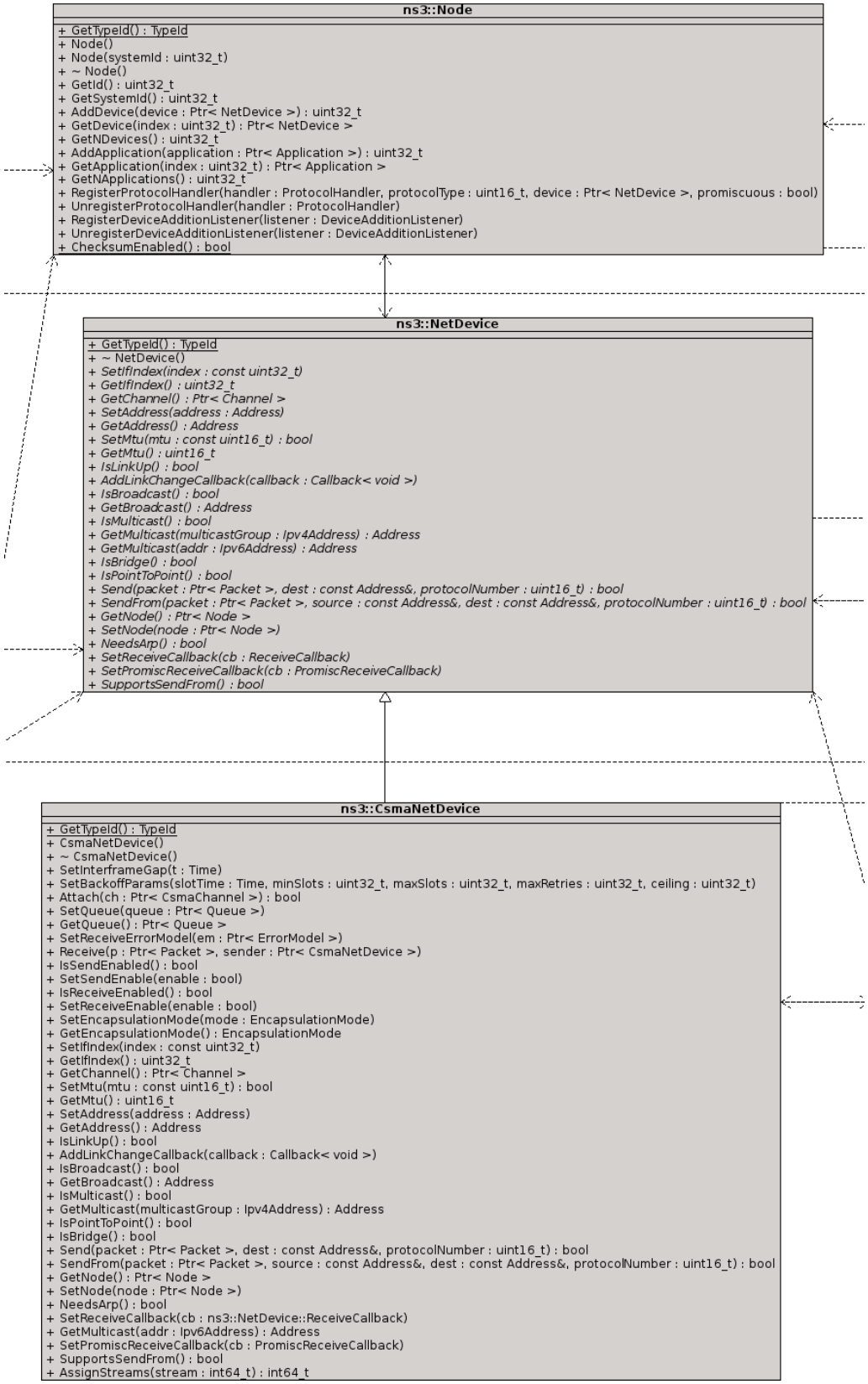
[17] E. Y. J. M. M. Laprise, "full duplex extensions for CSMA," [Online]. Available: https://codereview.appspot.com/187880044/.

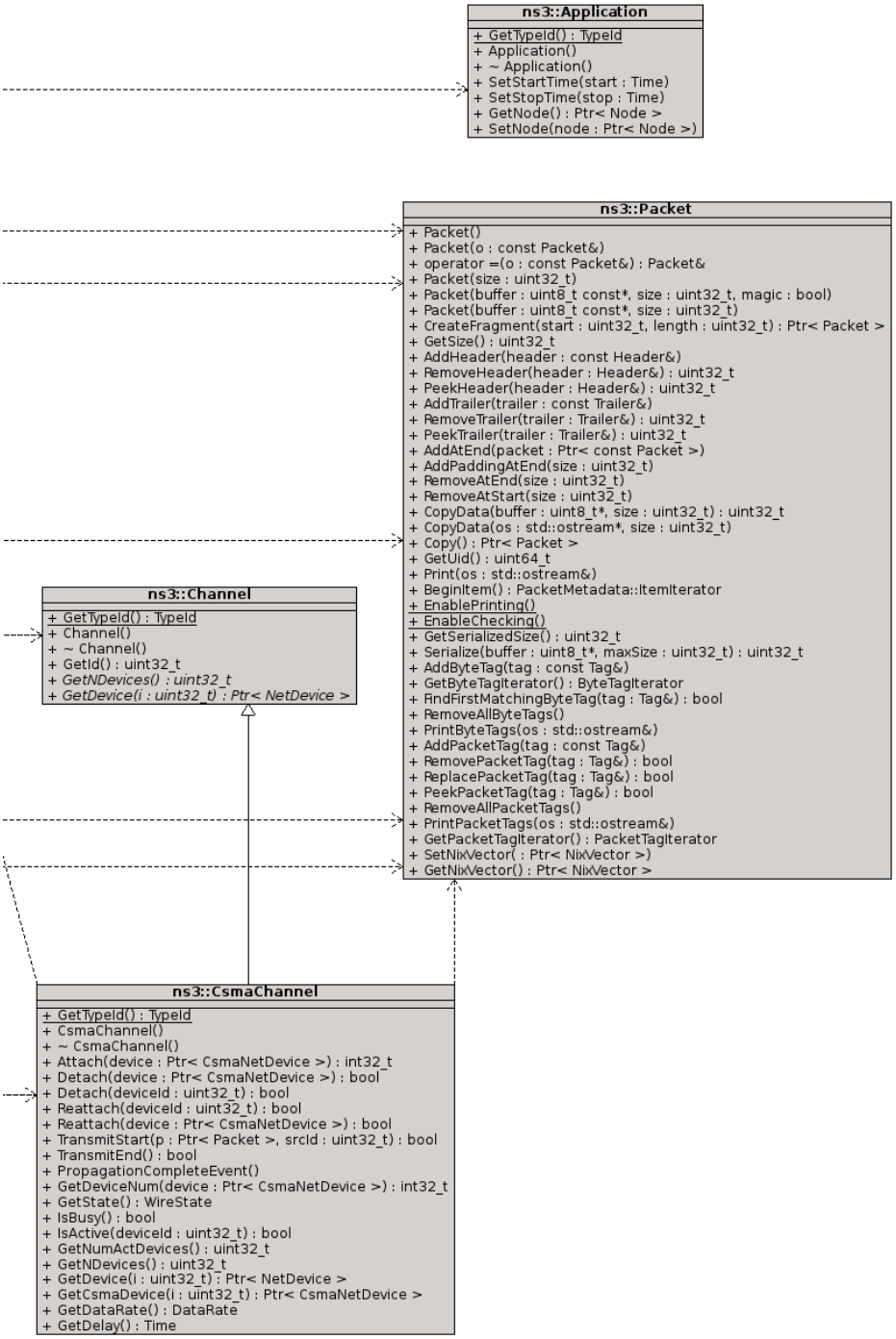
[18] H. T. V. L. &. H. D. Lim, Challenges in a future IP/Ethernet-based in-car network for real-time applications., 2011.

Anexo 1 Planeamento



Anexo 2 Diagrama de classes das classes do NS-3 utilizadas (em 3 partes)


ns3::Node
+ GetTypeId() : TypeId
+ Node()
+ Node(systemId : uint32_t)
+ ~ Node()
+ GetId() : uint32_t
+ GetSystemId() : uint32_t
+ AddDevice(device : Ptr< NetDevice >) : uint32_t
+ GetDevice(index : uint32_t) : Ptr< NetDevice >
+ GetNDevices() : uint32_t
+ AddApplication(application : Ptr< Application >) : uint32_t
+ GetApplication(index : uint32_t) : Ptr< Application >
+ GetNApplications() : uint32_t
+ RegisterProtocolHandler(handler : ProtocolHandler, protocolType : uint16_t, device : Ptr< NetDevice >, promiscuous : bool)
+ UnregisterProtocolHandler(handler : ProtocolHandler)
+ RegisterDeviceAdditionListener(listener : DeviceAdditionListener)
+ UnregisterDeviceAdditionListener(listener : DeviceAdditionListener)
+ ChecksumEnabled() : bool
ns3::NetDevice
+ GetTypeId() : TypeId
+ ~ NetDevice()
+ SetIfIndex(index : const uint32_t)
+ GetIfIndex() : uint32_t
+ GetChannel() : Ptr< Channel >
+ SetAddress(address : Address)
+ GetAddress() : Address
+ SetMtu(mtu : const uint16_t) : bool
+ GetMtu() : uint16_t
+ IsLinkUp() : bool
+ AddLinkChangeCallback(callback : Callback< void >)
+ IsBroadcast() : bool
+ GetBroadcast() : Address
+ IsMulticast() : bool
+ GetMulticast(multicastGroup : Ipv4Address) : Address
+ GetMulticast(addr : Ipv6Address) : Address
+ IsBridge() : bool
+ IsPointToPoint() : bool
+ Send(packet : Ptr< Packet >, dest : const Address&, protocolNumber : uint16_t) : bool
+ SendFrom(packet : Ptr< Packet >, source : const Address&, dest : const Address&, protocolNumber : uint16_t) : bool
+ GetNode() : Ptr< Node >
+ SetNode(node : Ptr< Node >)
+ NeedsArp() : bool
+ SetReceiveCallback(cb : ReceiveCallback)
+ SetPromiscReceiveCallback(cb : PromiscReceiveCallback)
+ SupportsSendFrom() : bool
ns3::CsmaNetDevice
+ GetTypeId() : TypeId
+ CsmaNetDevice()
+ ~ CsmaNetDevice()
+ SetInterframeGap(t : Time)
+ SetBackoffParams(slotTime : Time, minSlots : uint32_t, maxSlots : uint32_t, maxRetries : uint32_t, ceiling : uint32_t)
+ Attach(ch : Ptr< CsmaChannel >) : bool
+ SetQueue(queue : Ptr< Queue >)
+ GetQueue() : Ptr< Queue >
+ SetReceiveErrorModel(em : Ptr< ErrorModel >)
+ Receive(p : Ptr< Packet >, sender : Ptr< CsmaNetDevice >)
+ IsSendEnabled() : bool
+ SetSendEnable(enable : bool)
+ IsReceiveEnabled() : bool
+ SetReceiveEnable(enable : bool)
+ SetEncapsulationMode(mode : EncapsulationMode)
+ GetEncapsulationMode() : EncapsulationMode
+ SetIfIndex(index : const uint32_t)
+ GetIfIndex() : uint32_t
+ GetChannel() : Ptr< Channel >
+ SetMtu(mtu : const uint16_t) : bool
+ GetMtu() : uint16_t
+ SetAddress(address : Address)
+ GetAddress() : Address
+ IsLinkUp() : bool
+ AddLinkChangeCallback(callback : Callback< void >)
+ IsBroadcast() : bool
+ GetBroadcast() : Address
+ IsMulticast() : bool
+ GetMulticast(multicastGroup : Ipv4Address) : Address
+ IsPointToPoint() : bool
+ IsBridge() : bool
+ Send(packet : Ptr< Packet >, dest : const Address&, protocolNumber : uint16_t) : bool
+ SendFrom(packet : Ptr< Packet >, source : const Address&, dest : const Address&, protocolNumber : uint16_t) : bool
+ GetNode() : Ptr< Node >
+ SetNode(node : Ptr< Node >)
+ NeedsArp() : bool
+ SetReceiveCallback(cb : ns3::NetDevice::ReceiveCallback)
+ GetMulticast(addr : Ipv6Address) : Address
+ SetPromiscReceiveCallback(cb : PromiscReceiveCallback)
+ SupportsSendFrom() : bool
+ AssignStreams(stream : int64_t) : int64_t


ns3::Application
+ GetTypeId() : TypeId
+ Application()
+ ~ Application()
+ SetStartTime(start : Time)
+ SetStopTime(stop : Time)
+ GetNode() : Ptr< Node >
+ SetNode(node : Ptr< Node >)
ns3::Packet
+ Packet()
+ Packet(o : const Packet&)
+ operator =(o : const Packet&) : Packet&
+ Packet(size : uint32_t)
+ Packet(buffer : uint8_t const*, size : uint32_t, magic : bool)
+ Packet(buffer : uint8_t const*, size : uint32_t)
+ CreateFragment(start : uint32_t, length : uint32_t) : Ptr< Packet >
+ GetSize() : uint32_t
+ AddHeader(header : const Header&)
+ RemoveHeader(header : Header&) : uint32_t
+ PeekHeader(header : Header&) : uint32_t
+ AddTrailer(trailer : const Trailer&)
+ RemoveTrailer(trailer : Trailer&) : uint32_t
+ PeekTrailer(trailer : Trailer&) : uint32_t
+ AddAtEnd(packet : Ptr< const Packet >)
+ AddPaddingAtEnd(size : uint32_t)
+ RemoveAtEnd(size : uint32_t)
+ RemoveAtStart(size : uint32_t)
+ CopyData(buffer : uint8_t*, size : uint32_t) : uint32_t
+ CopyData(os : std::ostream*, size : uint32_t)
+ Copy() : Ptr< Packet >
+ GetUid() : uint64_t
+ Print(os : std::ostream&)
+ BeginItem() : PacketMetadata::ItemIterator
+ EnablePrinting()
+ EnableChecking()
+ GetSerializedSize() : uint32_t
+ Serialize(buffer : uint8_t*, maxSize : uint32_t) : uint32_t
+ AddByteTag(tag : const Tag&)
+ GetByteTagIterator() : ByteTagIterator
+ FindFirstMatchingByteTag(tag : Tag&) : bool
+ RemoveAllByteTags()
+ PrintByteTags(os : std::ostream&)
+ AddPacketTag(tag : const Tag&)
+ RemovePacketTag(tag : Tag&) : bool
+ ReplacePacketTag(tag : Tag&) : bool
+ PeekPacketTag(tag : Tag&) : bool
+ RemoveAllPacketTags()
+ PrintPacketTags(os : std::ostream&)
+ GetPacketTagIterator() : PacketTagIterator
+ SetNixVector(: Ptr< NixVector >)
+ GetNixVector() : Ptr< NixVector >
ns3::Channel
+ GetTypeId() : TypeId
+ Channel()
+ ~ Channel()
+ GetId() : uint32_t
+ GetNDevices() : uint32_t
+ GetDevice(i : uint32_t) : Ptr< NetDevice >
ns3::CsmaChannel
+ GetTypeId() : TypeId
+ CsmaChannel()
+ ~ CsmaChannel()
+ Attach(device : Ptr< CsmaNetDevice >) : int32_t
+ Detach(device : Ptr< CsmaNetDevice >) : bool
+ Detach(deviceId : uint32_t) : bool
+ Reattach(deviceId : uint32_t) : bool
+ Reattach(device : Ptr< CsmaNetDevice >) : bool
+ TransmitStart(p : Ptr< Packet >, srcId : uint32_t) : bool
+ TransmitEnd() : bool
+ PropagationCompleteEvent()
+ GetDeviceNum(device : Ptr< CsmaNetDevice >) : int32_t
+ GetState() : WireState
+ IsBusy() : bool
+ IsActive(deviceId : uint32_t) : bool
+ GetNumActDevices() : uint32_t
+ GetNDevices() : uint32_t
+ GetDevice(i : uint32_t) : Ptr< NetDevice >
+ GetCsmaDevice(i : uint32_t) : Ptr< CsmaNetDevice >
+ GetDataRate() : DataRate
+ GetDelay() : Time