CISTER
Research Centre in
Real-Time & Embedded
Computing Systems

BEng Thesis

Framework para Sistemas Distribuídos em Tempo-real

Roberto Duarte

CISTER-TR-161201

2016/10/31

Framework para Sistemas Distribuídos em Tempo-real

Roberto Duarte

*CISTER Research Centre

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: 1070485@isep.ipp.pt

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

Abstract

The development of real-time distributed systems has always been a complex, platform-specific and expensive task. The introduction of the fork-join paradigm in multicore systems allowed the distribution of computations between the cores.This work proves that it is possible to implement the fork/join parallel computing paragidm in real-time systems. This paradigm allows to relocate part of the computation to different computational nodes when it is not feasible to execute the computation locally within a given deadline. Another motivation to distribute the computation can be the need to save energy on some battery-powered nodes by means of load balancing. The framework was developed with a focus on embedded nodes with low computational capabilities, and the implementation features a great deal of low-level optimization, with the goal of meeting deadlines up to 70 ms.The work was based on an open-source implementation of the communication protocol Flexible Time Trigger-Switched Ethernet (FTT-SE), in which the distributed operations are executed in Linux nodes equipped with the real-time scheduler provided by the kernel patch PREEMPT-RT.

Framework para Sistemas Distribuídos de Tempo-real

Research Centre in Real-Time & Embedded Computing Systems (CISTER)

2015 / 2016

1070485 Roberto Daniel Alves Duarte

Framework para Sistemas Distribuídos em Tempo-real

Research Centre in Real-Time & Embedded Computing Systems (CISTER)

2015 / 2016

1070485 Roberto Daniel Alves Duarte

Licenciatura em Engenharia Informática

Outubro de 2016

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

Supervisor Externo: **Eng. Ricardo Garibay-Martinez**

Aos meus pais,

Á minha família,

À minha namorada

E aos meus amigos.

Agradecimentos

Quero em primeiro lugar endereçar um forte agradecimento ao Professor Doutor Luís Lino Ferreira e ao Engenheiro Ricardo Garibay-Martínez por todo o seu apoio, dedicação e disponibilidade que me facultaram e por me orientarem sempre no sentido de melhorar o meu trabalho.

Quero também agradecer a todos os amigos que fiz no CISTER que por toda ajuda e camaradagem ao longo do tempo que estive lá. Não esquecendo claro o agradecimento ao Departamento de Informática pela oportunidade de frequentar a Licenciatura em Engenharia Informática. Estes últimos anos de estudo, apesar de extenuantes, têm sido muito recompensadores, cheios de desafios e aprendizagem. O que aprendemos no ISEP, não só nos preparou intelectualmente para o mercado de trabalho como também nos ajudou a ganhar maturidade.

Resumo

O desenvolvimento de sistemas distribuídos de tempo-real tem sido sempre uma tarefa complexa, altamente especializada para cada plataforma e com custo muito elevados. A introdução do paradigma de computação paralela fork/join em sistemas multicore permite dividir a computação entre vários cores.

Este trabalho demostra que é possível a implementação do paradigma de computação fork/join distribuída em sistemas de tempo-real. Este paradigma permite distribuir parte de determinadas operações que não podem ser executadas localmente num nó, dentro da deadline definida para essa operação. Tal deve-se ao facto desses nós não terem capacidade de processamento suficiente. Outra razão para distribuir a computação pode também ser a necessidade de poupar energia num nó sem fios. Assim, a framework desenvolvida permite distribuir parte da computação por outros nós do mesmo sistema que tenham recursos livres. Foi especialmente desenvolvida para ser utilizada em sistemas embebidos com fraca capacidade de processamento, a operar numa rede totalmente fechada ao exterior. A implementação é por isso muito otimizada e de baixo nível de modo a que possa cumprir deadlines acima dos 70 ms.

Este trabalho foi baseado numa implementação *open-sour*ce do protocolo de comunicação Flexible Time Trigger-Switched Ethernet (FTT-SE) em que as operações distribuídas são executas em nós Linux com o patch PREEMPT-RT, que assegura o suporte a aplicações de tempo real.

Palavras Chave (Tema): Fork/Join parallel distributed, tempo-real

Palavras Chave (Tecnologias): FTT-SE, C

Índice

Índice de Figuras

Índice de Tabelas

Notação e Glossário

AW	Asynchronous Window
C	Linguagem de programação orientada a objetos
CISTER	Centro de Investigação em Sistemas Embebidos e de Tempo-Real
CSMA/CD	Carrier Sense Multiple Access With Collision Detection
EC	Elementary Cycle
FTT	Flexible Time Triggered
FTT-SE	Flexible Time Triggered - Switched Ethernet
MIT	Minimum Inter-arrival Time
MTU	Maximum Transmission Unit
NES	Network Embedded System
NRBD	Node Requirements Database
PDRTF	Parallel Distributed Real Time Framework
SRDB	System Requirements Database
SW	Synchronous Window
TM	Trigger Message
WCET	Worst-Case Execution Time
WCML	Worst-Case Message Length
WCRT	Worst-Case Response Time

1 Introdução

1.1 Apresentação do projeto/estágio

Neste capítulo, é feita a introdução do projeto realizado, onde é explicado o seu enquadramento, demonstrado o seu planeamento e reuniões de acompanhamento, descritas brevemente as tecnologias utilizadas e apresentada a organização onde ele foi desenvolvido. No seu final é descrita a organização do presente relatório.

1.1.1 Enquadramento

O presente relatório é desenvolvido no âmbito da unidade curricular Projeto/Estágio (PESTI) da Licenciatura Em Engenharia Informática (LEI) do Instituto Superior de Engenharia do Porto (ISEP). Em PESTI o principal objetivo é a consolidação das competências adquiridas durante o curso em um ambiente real de trabalho, melhor preparando o aluno para a sua integração num contexto profissional.

Este projeto de estágio foi realizado no Centro de Investigação em Sistemas Embebidos e de Tempo-Real (CISTER) e teve como objetivo a implementação de uma Framework para Sistemas Distribuídos em Tempo-real e a consequente realização de testes a fim de obter resultados que demonstrem a viabilidade deste tipo de implementação.

1.1.2 Apresentação do Projeto

Atualmente existe uma grande proliferação de sistemas embebidos integrados em objetos do nosso cotidiano tal como em meios de transporte, eletrodomésticos, edifícios, sistemas de apoio a medicina etc. Alguns destes sistemas tem requisitos temporais rigorosos e para responder a esses requisitos são divididos em partes que comunicam entre si através de uma rede para efetuar diferentes tarefas ou distribuir o trabalho, estes sistemas são denominados de *Networked Embedded Systems* (NES).

O desenvolvimento para este tipo de sistemas por norma torna se uma tarefa complexa devido a necessidade de integrar o processamento distribuído e transmissão de dados para que forneçam o grau de determinismo temporal necessário.

É com vista a responder a este tipo de dificuldade que foi criada esta *Framework* que integra num só lugar ferramentas que dão suporte a este tipo de programação como os elementos necessários para que o cumprimento destas exigências temporais.

É também importante mencionar que esta *Framework* assenta na utilização no protocolo de transmissão de dados com garantias de tempo-real FTT-SE que é baseado no paradigma Flexible Time Triggered (FTT).

1.1.3 Planeamento de projeto

O planeamento deste projeto consistiu na separação em 5 fases: análise, implementação da *Framework* inicial, introdução ao tempo-real, implementação da *Framework* com tempo-real e obtenção e análise de resultados. A extensão deste projeto deve se ao facto de ter sido necessário estudar vários conceitos e tecnologias diferentes cujo seu conhecimento seria essencial para o desenvolvimento de uma *Framework* desta natureza. Alem desse facto apesar da conclusão do trabalho estar prevista para junho de 2014 devido a problemas de saúde o projeto foi interrompido tendo sido retomado em julho de 2016 para conclusão do relatório.

A Tabela 1 demonstra o planeamento elaborado para o projeto.

% Concluída	Nome da Tarefa	Duração	Início	Conclusão	Predecessoras
100%	**Análise**	**28 dias**	**Seg 16/09/13**	**Qua 23/10/13**	
100%	Estudar OpenMP e MPI	7 dias	Seg 16/09/13	Ter 24/09/13	
100%	Estudar Pthreads	7 dias	Qua 25/09/13	Qui 03/10/13	2
100%	Estudar Sockets	7 dias	Sex 04/10/13	Seg 14/10/13	3
100%	Estudar outras bibliotecas e Frameworks relacionadas com o projeto	7 dias	Ter 15/10/13	Qua 23/10/13	4
100%	**Implementar *Framework* Inicial**	**28 dias**	Qui 24/10/13	**Seg 02/12/13**	**1**
100%	Implementar funções de paralelização	7 dias	Qui 24/10/13	Sex 01/11/13	1
100%	Implementar mecanismo de configuração	14 dias	Seg 04/11/13	Qui 21/11/13	7
100%	Implementar comunicação distribuída com Sockets	14 dias	Seg 04/11/13	Qui 21/11/13	7
100%	Testar implementação inicial	7 dias	Sex 22/11/13	Seg 02/12/13	8;7;9
100%	**Introdução ao Tempo-real**	**28 dias**	**Qui 05/12/13**	**Seg 13/01/14**	**6**
100%	Estudar programação em tempo-real	7 dias	Qui 05/12/13	Sex 13/12/13	6
100%	Estudar sistemas operativos tempo-real	7 dias	Seg 16/12/13	Ter 24/12/13	12
100%	Estudar escalonadores linux tempo-real	7 dias	Qua 25/12/13	Qui 02/01/14	13

100%	Estudar comunicação em tempo-real	7 dias	Sex 03/01/14	Seg 13/01/14	14
100%	Estudar Flexible Time Triggered - Switched Ethernet (FTT-SE)	7 dias	Sex 03/01/14	Seg 13/01/14	14
100%	**Implementar *Framework* com tempo-real**	**43 dias**	**Ter 14/01/14**	**Qui 13/03/14**	**11**
100%	Implementar nova comunicação distribuida usando FTT-SE	14 dias	Ter 14/01/14	Sex 31/01/14	11
100%	Implementar novo mecanismo de configuração para suportar FTT-SE	14 dias	Ter 14/01/14	Sex 31/01/14	11
100%	Integração com escalonador tempo-real linux	7 dias	Seg 03/02/14	Ter 11/02/14	18;19
100%	Actualização mecanismo de parelização para suportar escalonamento tempo-real	7 dias	Qua 12/02/14	Qui 20/02/14	20
100%	Testar implementação final	15 dias	Sex 21/02/14	Qui 13/03/14	21
100%	**Resultados**	**15 dias**	**Sex 14/03/14**	**Qui 03/04/14**	**17**
100%	Experiências usando a implementação realizada	7 dias	Sex 14/03/14	Seg 24/03/14	17
100%	Analise de resultados das experiências	7 dias	Ter 25/03/14	Qua 02/04/14	24
100%	Escrever relatório	35 dias	Qua 20/07/16	Ter 06/09/16	25
100%	Preparar distribuição Open Source	7 dias	Qua 07/09/16	Qui 15/09/16	26

Tabela 1: Lista de tarefas que constam do planeamento do projeto

1.1.4 Reuniões de acompanhamento

Conforme se pode verificar na tabela 2, são apresentadas as reuniões de acompanhamento realizadas ao longo do projeto que tiveram sempre lugar no CISTER, nestas reuniões foi abordado o estado do trabalho, foram esclarecidas dúvidas que iam surgindo sobre o trabalho e foram discutidas questões sobre a implementação. Além destas reuniões houveram também conversas informais que ocorreram num evento semanal da própria organização denominado “Morning Coffee” onde grande parte dos elementos da organização se juntam num ambiente descontraído para socializarem.

Data	Participantes	Assunto
9-9-2013	Prof. Luís Lino Ferreira e Eng. Ricardo Garibay Martinez	Discussão sobre objetivos do projeto
18-9-2013	Eng. Ricardo Garibay Martinez	Esclarecimento de dúvidas
26-9-2013	Prof. Luís Lino Ferreira	Progresso do trabalho
1-10-2013	Prof. Luís Lino Ferreira e Eng. Ricardo Garibay Martinez	Esclarecimento de dúvidas
12-10-2013	Prof. Luís Lino Ferreira	Progresso do trabalho
15-10-2013	Eng. Ricardo Garibay Martinez	Discussão sobre implementação
21-10-2013	Eng. Ricardo Garibay Martinez	Esclarecimento de dúvidas
28-10-2013	Prof. Luís Lino Ferreira e Eng. Ricardo Garibay Martinez	Progresso do trabalho
13-11-2013	Prof. Luís Lino Ferreira e Eng. Ricardo Garibay Martinez	Discussão sobre implementação
29-11-2013	Prof. Luís Lino Ferreira	Progresso do trabalho
12-12-2013	Prof. Luís Lino Ferreira e Eng. Ricardo Garibay Martinez	Progresso do trabalho e Discussão sobre implementação
8-1-2014	Prof. Luís Lino Ferreira	Progresso do trabalho
24-1-2014	Eng. Ricardo Garibay Martinez	Esclarecimento de dúvidas
11-2-2014	Eng. Ricardo Garibay Martinez	Esclarecimento de dúvidas e Discussão sobre implementação
17-2-2014	Prof. Luís Lino Ferreira e Eng. Ricardo Garibay Martinez	Progresso do trabalho e Discussão sobre implementação
7-3-2014	Prof. Luís Lino Ferreira	Progresso do trabalho

23-3-2014	Eng. Ricardo Garibay Martinez	Esclarecimento de dúvidas e Discussão sobre implementação
10-4-2014	Prof. Luís Lino Ferreira	Progresso do trabalho
29-4-2014	Prof. Luís Lino Ferreira	Progresso do trabalho
18-7-2016	Prof. Luís Lino Ferreira	Discussão sobre o relatório
26-7-2016	Prof. Luís Lino Ferreira	Discussão sobre o relatório
4-8-2016	Prof. Luís Lino Ferreira	Discussão sobre o relatório
12-8-2016	Prof. Luís Lino Ferreira	Discussão sobre o relatório

Tabela 2: Lista de tarefas que constam do planeamento do projeto

1.2 Tecnologias utilizadas

Para este projeto foi necessário o uso de várias tecnologias tanto para o seu desenvolvimento como para suporte ao desenvolvimento. Estas tecnologias são descritas na seguinte lista que está organizada por escalões:

- Sistemas Operativos
    - Linux Debian 7.8 com Kernel de tempo-real RT-PREEMPT
- Programas utilizados
    - Netbeans IDE
    - Microsoft Word
    - Microsoft Excel
    - Microsoft Visio
    - Microsoft Project
- Controlo de Versões
    - Github
- Linguagens de programação
    - C
- Bibliotecas usadas

- FTT-SE
- Libconfig

1.3 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.

A informação contida nesta secção foi elaborada com base na referência: [1].

1.4 Contributos deste trabalho

Apesar e existirem algumas soluções para desenvolvimento de aplicações distribuídas de tempo-real estas são ofertas que por norma requerem *hardware e software* dispendioso.

Com este projeto é possível colmatar estas problemáticas pois é possível desenvolver um sistema distribuído com *hardware* comum e relativamente barato, para além disso o código fonte é disponibilizado para a comunidade *open source* e foi desenvolvido a pensar na facilidade de utilização não obstante da necessidade de alguns conhecimentos da linguagem C e de boas práticas de programação.

Parte deste projeto nomeadamente o componente "*FTT-SE Wrapper*" foi desenvolvido a para facilitar utilização da biblioteca FTT-SE de Ricardo Marau fornecendo uma interface mais simplificada e de fácil aprendizagem. Por esta razão este módulo foi utilizado no projeto Arrowhead [11] que também fez uso do protocolo FTT-SE [12].

1.5 Organização do relatório

O presente relatório tem uma estrutura organizada por objetivos específicos os quais serão apresentados a seguir:

Capítulo 1 – Introdução: Este é o capítulo atual e pretende apresentar o projeto realizado e fornecer uma contextualização generalizada do problema proposto.

Capítulo 2 – Contexto: Neste capítulo, as tecnologias utilizadas ao longo do projeto para o desenvolvimento da *Framework* são apresentadas de forma a fornecer noções teóricas fundamentas para a compreensão de como o projeto foi elaborado.

Capítulo 3 - Descrição Técnica: Neste capítulo é apresentada a descrição da implementação de maneira a fundamentar as decisões tomadas no decurso do desenvolvimento para corresponder aos requisitos do trabalho.

Capítulo 4 -Resultados e validação: Neste capítulo são realizadas experiências utilizando a *Framework* de forma a validar os objetivos propostos.

Capítulo 5 Conclusão: Neste capítulo final são apresentadas as conclusões do trabalho analisando os objetivos concretizados e possível trabalho futuro.

2 Contexto

Neste capítulo é fornecida uma contextualização das tecnologias e conceitos utilizados ao longo do projeto. A secção 2.1 irá descrever o que é um sistema de tempo-real, assentando caminho para a secção 2.2 que irá caracterizar os sistemas operativos de tempo-real fornecendo exemplos de alguns destes e detalhando o que foi usado neste projeto. Seguidamente na secção 2.3 será descrita a comunicação distribuída de tempo-real, finalizando na secção 2.4 com a descrição do protocolo FTT-SE que foi utilizado neste projeto.

2.1 Sistemas de tempo-real

De acordo com [2] um sistema de tempo-real pode ser definido como um sistema em que o resultado de um computação não depende apenas do resultado lógico da mesma, mas também do tempo em que os resultados são produzidos.

Consequentemente, estes sistemas devem responder a eventos (temporais, físicos ou outros) dentro de um determinado intervalo de tempo sendo o seu limite designado por deadline. O tipo de consequência e utilidade do resultado após uma falha de cumprimento deste deadline é o que caracteriza o tipo de sistema de tempo-real. Se um resultado tiver utilidade mesmo depois de a deadline ter passado, é classificado como soft, senão é classificado como firm, se o não cumprimento der origem a uma falha catastrófica (p.e. uma acidente), então o sistema é classificado como hard.

Exemplos de sistemas de tempo-real [3] incluem o controlo de centrais de energia nuclear, sistemas de automação industrial, controlo de experiencias laboratoriais, controlo de motores automóveis, sistemas de telecomunicações, entre muitos outros. Um exemplo de um sistema de tempo-real hard é o airbag de um automóvel, se o airbag for acionado tarde demais (e também neste caso cedo demais) em caso de acidente, o condutor poderá correr perigo de vida pois não será protegido do impacto.

2.2 Sistemas operativos de tempo-real

A maioria dos sistemas operativos permite executar várias tarefas simultaneamente apesar de na verdade cada núcleo do processador apenas pode correr uma única thread de cada vez. O escalonador é a parte do sistema operativo responsável por decidir que tarefa será executada num dado momento criando a ilusão da execução simultânea através da

alternância entre threads no processador. De forma a se obter determinismo, num sistema operativo de tempo-real, o escalonador é concebido de maneira a permitir atribuição de prioridades às tarefas impedindo desta forma que as tarefas de maior prioridade não serão interrompidas pelas de menor prioridade [4]. Existe também muitos outros modelos de escalonamento, tal como o Earliest Deadline First, por exemplo.

Alguns exemplos de sistemas operativos de tempo-real incluem o FreeRTOS, o RTLinux, o VxWorks, o QNX, o eCos e o Linux com patch PREEMPT-RT. Sendo que este ultimo foi o escolhido para este projeto, nas próximas secções será brevemente descrito o escalonamento em Linux e em que consiste a patch PREEMPT-RT.

2.2.1 Escalonamento em Linux

Atualmente o escalonamento em Linux [5] está dividido em 3 políticas principais, uma para tarefas normais do utilizador, designada por SCHED_OTHER que é a utilizada por omissão e duas políticas de escalonamento de tempo-real compatíveis com o norma POSIX, designadas por SCHED_FIFO e SCHED_OTHER, as tarefas escalonadas por estas políticas de tempo-real nunca serão interrompidas pela tarefas escalonadas pela política SCHED_OTHER.

A política SCHED_OTHER atualmente é implementada seguindo o algoritmo de escalonamento chamado CFS (Completely Fair Scheduler), este algoritmo tenta maximizar a utilização do processador por todas as tarefas de maneira a manter o sistema responsivo ao utilizador.

Na política SCHED_FIFO, são atribuídas prioridades ás tarefas, sendo as de mesma prioridade colocadas em filas FIFO (First In First Out), as tarefas de mesma prioridade são executadas por ordem de chegada sem nunca saírem do processador a não ser que estejam à espera de um recurso ou sejam explicitamente interrompidas a nível de utilizador.

O escalonamento de tarefas na política SCHED_RR difere da SCHED_FIFO na medida que, em vez de as tarefas com mesma prioridade em vez de serem colocadas numa fila, são alternadas entre si de forma a todas terem hipótese de obter acesso ao processador diminuindo a interferência das tarefas de longa duração sobre as de curta duração.

2.2.2 Patch PREEMPT-RT

Apesar da existência das políticas de escalonamento tempo-real para o Linux, devido à maneira como o *kernel* está implementado, existem partes deste que não permitem a preempção, por causa desta situação, apesar de as tarefas de tempo-real não poderem ser interrompidas por tarefas normais, elas podem acabar por ser interrompidas por tarefas do

kernel, isto significa que este tipo de escalonamento seria apenas adequado para tarefas soft real-time pois não existe garantia de determinismo necessário para tarefas firm e hard real time.

Com o objetivo de resolver este tipo de limitações que a *patch* RT-PREEMPT [6] foi desenvolvida, as partes do *kernel* que não permitiam preempção foram alteradas para o permitir e foi adicionado suporte a um relógio de alta resolução fornecendo um nível de precisão temporal superior, com estas alterações o kernel passa a ganhar capacidades hard real-time.

2.3 Comunicação distribuída de tempo-real

Atualmente, devido a sua disponibilidade, baixo custo e potencial para expansibilidade, a tecnologia mais utilizada para comunicações em rede cablada é a tecnologia *Ethernet,* no entanto, quando esta foi desenvolvida não foi tido em conta o cumprimento de requisitos de tempo-real sendo por isso geralmente considerada inapropriada para utilização neste tipo de aplicações. Parte deste problema deve-se ao facto de que o acesso à rede pelos nós ser feito através do mecanismo CSMA/CD (Carrier Sence Multiple Access with Colision Detection) para evitar colisões em meio partilhado, recorrendo ao mecanismo BEB (Binary Exeponential Back-Off) para resolver as colisões. Estes algoritmos funcionam de maneira não determinística, o que os torna inapropriados para comunicação em tempo-real.

A introdução de *switches* e ligações *full-duplex* na rede permite resolver este problema pois deixa de haver um único domínio de colisão, deixando de ser necessário resolver colisões. No entanto, continua a existir o problema de não ser possível priorizar a transmissão de mensagens o que por sua vez torna impossível obter determinismo, um fator essencial para comunicação em tempo-real.

A Ethernet e outros protocolos que não são de tempo real, focam-se mais na consistência dos dados transmitidos entre diferentes nós do que o tempo que estes demoram do nó origem ao nó destino. Aplicações como o email, para um funcionamento correto não necessitam de uma resposta rápida para o consumo das suas funcionalidades. Se um utilizador envia um email, este não está preocupado se o email foi recebido no destino em dois ou mais segundos. Contudo, já em sistemas de tempo-real, especialmente em sistemas críticos, um qualquer atraso de transmissão de dados pode implicar uma falha do sistema, podendo ocorrer uma catástrofe com perda de vidas.

Exemplos de tecnologias de comunicação distribuída de tempo real incluem: *Asynchronous Transfer Mode* (ATM), Controlled Area Network (CAN) e o protocolo FTT-SE usado neste projeto.

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

O protocolo Flexible Time Triggered – Switched Ethernet (FTT-SE) foi desenvolvido por Ricardo Marau e é baseado no protocolo FTT-Ethernet [5], sendo que ambos seguem o paradigma Flexible Time Triggered FTT [6]. Nas próximas secções será brevemente descrito o paradigma FTT, a diferença entre o FTT-SE e o FTT-Ethernet, a arquitetura interna do FTT-SE, em que é que consiste o Elementary Cycle e por fim expostos os tipos e trafego existentes no FTT-SE.

2.4.1 Paradigma Flexible Time Triggered (FTT)

O paradigma FTT é baseado no modelo master/slave em que a entidade master é o nó da rede que coordena todo o trafego trocado pelos restantes nós slaves. Com esta metodologia é possível introduzir políticas de escalonamento de mensagens e implementar um mecanismo de controlo de admissão oferecendo assim garantias de cumprimento de requisitos temporais.

O funcionamento dos protocolos FTT assenta na existência de intervalos regulares de duração pré fixa, chamados de Elementary Cycles (ECs), dentro dos quais são trocadas as mensagens previamente escalonadas pelo master. A informação sobre o escalonamento é comunicada pelo master, a todos os slaves e no início de cada EC, através da transmissão em *broadcast* de uma mensagem designada de Trigger Message (TM). Por sua vez a quando a TM é recebida pelos slaves estes verificam se são transmissores de alguma das mensagens escalonadas procedendo então ao envio das mesmas.

2.4.2 FTT-SE vs FTT-Ethernet

Enquanto o FTT-Ethernet foi desenvolvido para funcionar sobre uma rede *Ethernet* partilhada, o FTT-SE foi desenvolvido para funcionar sobre uma rede conectada por *switches* com ligações *full duplex*. Desta maneira a rede torna-se micro-segmentada, onde cada segmento contem o seu domínio de colisão privado com um único nó na sua extremidade. Como entre cada nó e *switch* existem dois *links:* um *uplink* do *switch* para o nó e um *downlink* do nó para o *switch,* a comunicação bidirecional ocorre sem interferência, o que não é o caso com o FTT-Ethernet pois existe um único domínio de colisão. Este facto implica que o FTT-Ethernet não poderá fornecer o mesmo nível de determinismo que o FTT-SE, dado que o seu mecanismo de

resolução de colisões, apesar de ser mais eficiente que o mecanismo CSMA/CD, não é completamente determinístico.

2.4.3 Arquitetura interna do FTT-SE

Como se pode observar na Figura 1, a arquitetura do FTT-SE divide se em 3 camadas principais designadas por Interface, Management e Core. Como parte da lógica é partilhada por Masters e Slaves, as camadas Interface e Core encontram-se em ambos enquanto a camada Management encontra-se apenas no Master.

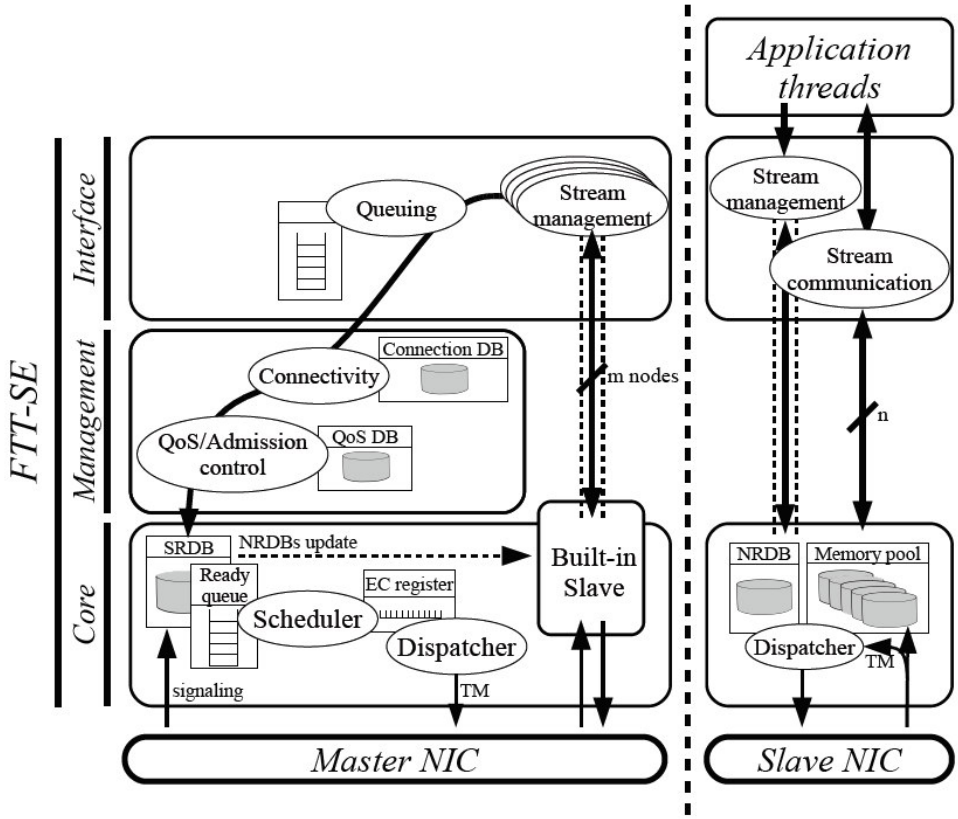

Figura 1: Arquitetura interna do FTT-SE. [7]

A camada Interface disponibiliza as funções de gestão e de comunicação do protocolo a nível aplicacional.

A camada Management, exclusiva do Master é onde se encontra a lógica relacionada com a gestão de Qualidade de Serviço e controlo de admissão de mensagens, sendo portanto nesta camada que os recursos das streams (ligação de dados entre dois nós) são geridos, bem como tomadas decisões face ao registo de novas streams com base na possibilidade de serem ou não escalonáveis.

A camada Core é onde é gerida a comunicação, nela existe, do lado do Master, uma base de dados designada por System Requirements Database (SRDB) onde é guardada a informação sobre as streams que se encontram registadas. As mensagens pendentes que se encontram na SRDB são guardadas pelo Scheduler numa fila chamada de Ready Queue, a qual é usada em cada EC para se efetuar o escalonamento de acordo com a política de escalonamento configurada. Depois de efetuado o escalonamento, as mensagens são colocadas no EC Register que por usa vez será usado pelo Dispatcher para construir e enviar a TM.

Já do lado do Slave, a camada Core possui uma base de dados designada por Node Requirements Database (NRDB), esta similarmente a SRDB contem a informação das streams registadas mas apenas aquelas que dizem respeito ao Slave em questão. Esta informação é usada pelo Dispatacher em conjunto com a TM para verificar se alguma das mensagens escalonadas corresponde, no caso de ser produtor ou consumidor de alguma delas o Dispatcher irá iniciar o envio ou receção das mensagens de ou para a Memory Pool respetivamente.

2.4.4 Elementary Cycle

Tal como foi falado na secção 2.4.1, o Elementary Cycle é o intervalo de duração pré fixa dentro do qual são trocadas as mensagens previamente escalonadas pelo master, sendo que é no início dele que é enviada a TM que contem a informação de escalonamento a ser processada pelos Slaves.

O Elementary Cycle é dividido em várias partes as quais podem ser analisadas mais facilmente através da Figura 2.

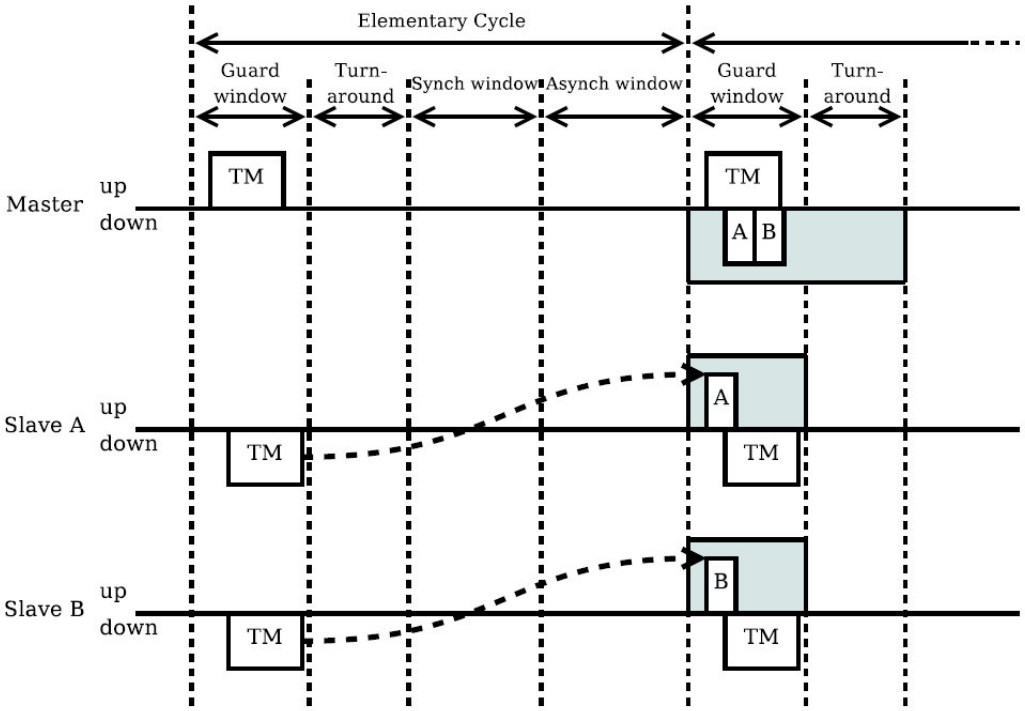

Figura 2: Demonstração da composição do Elementary Cycle. [7]

A primeira parte designada por Guard Window representa a largura de banda necessária para a transmissão da TM, a segunda parte designada por Turn-around Window representa a largura de banda reservada para permitir o processamento da TM pelos Slaves. Existe ainda a Signalling Window, que é composta pelas duas partes anteriores e onde os Slaves podem enviar para o master sinalizações de eventos como o registo de streams ou pedidos de envio de mensagens, estas mensagens de sinalização são designadas por Signalling Messages.

As partes seguintes designadas por Synchronous Window e Assynchronous Window reservam a largura de banda para a transmissão de trafego síncrono e assíncrono respetivamente, o seu tamanho pode variar de acordo com a largura de banda necessária para a transmissão de todas as mensagens síncronas e assíncronas que foram escalonadas naquele EC sem, no entanto, ultrapassar o limite máximo configurado no FTT-SE. Estes dois tipos de trafego serão descritos em maior detalhe na próxima secção.

2.4.5 Tipos de tráfego

No protocolo FTT-SE existem dois tipos de tráfego distintos, síncrono e assíncrono, a maneira como estes são escalonados é influenciada não só pela política de escalonamento configurada como também pelas suas especificidades particulares que serão descritas a seguir:

Tráfego síncrono

O trafego síncrono no FTT-SE segue o modelo comunicações Time-Triggered (TT), isto significa que as mensagens são trocadas em instantes de tempo periódicos específicos fornecendo assim um nível elevado de determinismo. Como foi mencionado na secção 2.4.3 a informação sobre as streams é colocada na SRDB, no caso das mensagens síncronas estas são colocadas na Synchronous Requirements Table (SRT) que é uma parte integrante da SRDB. O conjunto das streams síncronas constituinte da SRT pode ser definido de acordo com a Equação 1:

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

Equação 2: Equação da Synchronous Requirements Table (SRT). [7]

Nesta equação cada stream é caracterizada pelo Worst-Case Message Length (WCML) C_i, que é o tempo máximo de transmissão da mensagem, o deadline D_i, o período T_i e o offset O_i que são representados em numero inteiro de ECs, a prioridade Pr_i, o ID do nó produtor S_i e o conjunto de IDs dos nós consumidores $\left\{R_i^1, \dots, R_i^{k_i}\right\}$. [7]

É com base nestas propriedades e a política de escalonamento configurada no FTT-SE que as mensagens síncronas pendentes são colocadas no EC Register para posteriormente ser construída a Trigger Message.

Tráfego assíncrono

O trafego assíncrono segue o modelo Event Triggered (ET), isto significa que a transmissão das mensagens não está diretamente relacionada com um período como no modelo TT mas com a ocorrência de um evento, este modelo de transmissão não fornece o mesmo grau de determinismo que o modelo TT devido a imprevisibilidade da ocorrência dos eventos [8,9]. Desta forma ao alocar recursos para este tipo de mensagens é preciso considerar sempre o pior caso tendo como base para esse calculo o intervalo mínimo em que as mensagens são esperadas para serem transmitidas o *minimum inter arrival time* (Tmit). Tal como no trafego síncrono como a informação sobre a streams é colocada na SRDB com a diferença que neste caso as mensagens assíncronas são colocadas na Assynchronous Requirements Table (ART), que também é uma parte integrante da SRDB. O conjunto das streams assíncronas constituinte da ART pode ser definido como na Equação 3 [7]:

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

Equação 4: Assynchronous Requirements Table (ART).

Nesta equação as streams são caracterizadas similarmente às síncronas na SRT, com a diferença que em lugar do período e deadline é considerado o minimum inter-arrival time $Tmit_i$, que é o intervalo mínimo entre a transmissão de mensagens assíncronas consecutivas, não sendo também considerado o offset.

Ao contrário das mensagens síncronas que seguem o modelo TT, as mensagens assíncronas, seguindo o modelo ET, fazem uso de um mecanismo de Signalling como mencionado na secção 2.4.4, desta maneira para quando um evento de uma transmissão de uma mensagem assíncrona é gerado no slave produtor, este envia uma SM para o master com a informação das mensagens pendentes na sua NRDB. A quando da receção das SM dos slaves, o master colocará essa informação na sua SRDB e procederá ao escalonamento como explicado na secção 2.4.3. Por causa deste processo de signalling, que implica informar a master do envio e esperar pela sua resposta, o tempo de resposta de uma mensagem assíncrona nunca será inferior a dois ECs.

2.5 Paradigma de programação paralela fork/join distribuída

Hoje em dia há uma grande disponibilidade de plataformas multiprocessador que beneficiam de modelos de execução paralela para aumentar a sua capacidade de computação, naturalmente este tipo de modelos também trazem vantagens para aplicações de tempo-real. Com eles é possível cumprir requisitos temporais mais restritos que de outra forma não seriam possíveis com plataformas uniprocessador. Um dos modelos mais comuns é o modelo *fork*/*join*, o qual consiste na execução sequencial seguida de uma divisão de trabalho (*fork*) para ser executado em paralelo, quando as operações paralelas tiverem terminado, os resultados são agregados através da operação *join*.

Em sistemas distribuídos que possuem várias unidades de processamento ligadas em rede, é possível utilizar o modelo *fork/join* para aproveitar a capacidade de processamento disponível em todos os nós. Atendendo ao facto de que a operação de transmissão tem um custo temporal, é preciso ter em consideração que nem sempre poderá ser benéfico o tempo que se ganha com distribuição do trabalho face ao tempo que se gasta com as comunicações.

A Figura 3 representa um exemplo da aplicação do modelo *fork/join* distribuído que neste caso foi realizado usando a tecnologia MPI.

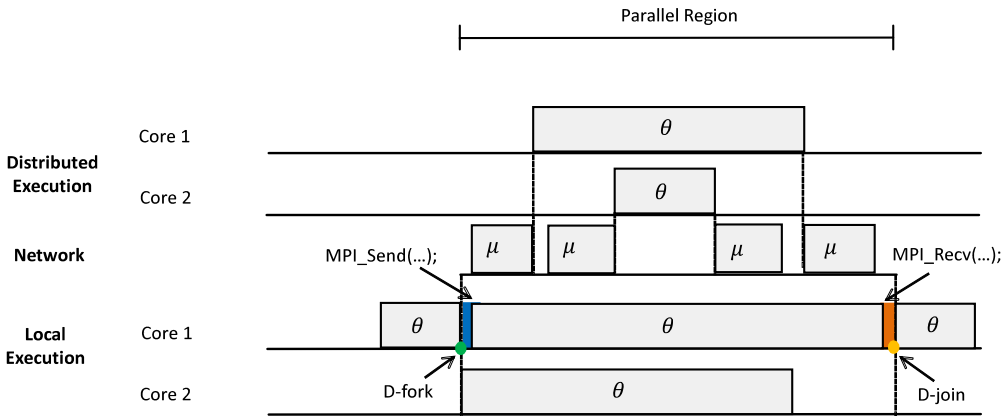

Figura 3: Execução de uma tarefa usando o modelo de execução paralelo distribuído. [10]

A mesma aproximação pode ser aplicada aos sistemas distribuídos de tempo-real, tendo em conta porém que, a utilização de um protocolo de transmissão com garantias tempo-real será necessária para fornecer as garantias temporais necessárias face ao grau de determinismo requerido por este tipo de aplicações.

3 Descrição Técnica

Neste capítulo da descrição técnica será abordado o levantamento de requisitos funcionais e não funcionais bem como a modelação da solução implementada.

3.1 Levantamento de Requisitos

O processo de levantamento de requisitos é um passo fundamental para a análise de um projeto, é com base nele que e é escolhida a metodologia de desenvolvimento mais apropriada para elaborar a solução para o problema. Por esta razão nas próximas secções serão enumerados e descritos os requisitos funcionais e não funcionais.

3.1.1 Requisitos Funcionais

Os requisitos funcionais, que descrevem explicitamente as funcionalidades e serviços do sistema, são enumerados nesta secção.

Fork-Join Parallel-Distributed

Devem ser criadas rotinas que permitam o envio, receção e processamento dados em nós distribuídos, de acordo com o paradigma *Fork-Join Parallel/Distributed, tal* como descrito na secção 2.5.

Tarefas de Tempo-real

Devem ser criadas rotinas que permitam a criação de tarefas de tempo-real com atribuição de prioridades e que se encarreguem da respetiva gestão de periodicidade e verificação de cumprimento de deadlines.

Integrar Protocolo FTT-SE

O protocolo FTT-SE deverá ser integrado na *Framework* de forma a se obter determinismo e garantias de tempo-real na transmissão de dados entre os nós distribuídos.

3.1.2 Requisitos Não Funcionais

Os requisitos não funcionais, que estão relacionados com as qualidades globais ou atributos do sistema, são enumerados nesta secção.

Usabilidade

A interface da *Framework* deve ser simplificada e fácil de usar não obstante da necessidade de alguns conhecimentos da linguagem c e de boas práticas de programação.

A *Framework* deve ser de alta performance

A *Framework* deverá garantir *overheads* temporais pequenos e grande previsibilidade temporal.

3.2 Modelação e desenvolvimento da solução

Nesta secção será descrita a abordagem escolhida para a implementação dos requisitos identificados na secção anterior. Começando por uma introdução à utilização da Framework, como ponto de partida para uma caracterização em maior detalhe dos pormenores de funcionamento interno das suas funções.

3.2.1 Introdução à Parallel Distributed Real-Time Framework (PDRTF)

A *Framework* PDRTF é fornecida como uma biblioteca de linguagem C que permite, através da sua interface, criar e gerir tarefas de tempo-real, que poderão ser executadas de forma distribuída num *cluster* de computadores, por sua vez este cluster será configurado através de um ficheiro de configuração.

O código criado fazendo uso desta *Framework*, será exatamente o mesmo a ser executado em todos os nós do *cluster,* comportando-se de maneira distinta dependendo do nó onde é executado. Devido a esta característica, o uso de um ficheiro de configuração para identificação dos nós, permite que sejam adicionados ou removidos nós ao *cluster* sem que seja necessário recompilar novamente o código.

Devido às exigências de tempo-real, é utilizado o protocolo FTT-SE, descrito em maior detalhe na secção 2.4, para a comunicação entre os nós, esta tecnologia implica a existência no *cluster* de um nó dedicado designado de *Master,* que se encarregará de gerir a comunicação dos restantes nós que neste contexto são designados de *Slaves*.

A fim de melhor compreender a interligação entre todos os elementos descritos nesta secção, na Figura 4 encontra-se representado o diagrama de *Deployment* da solução:

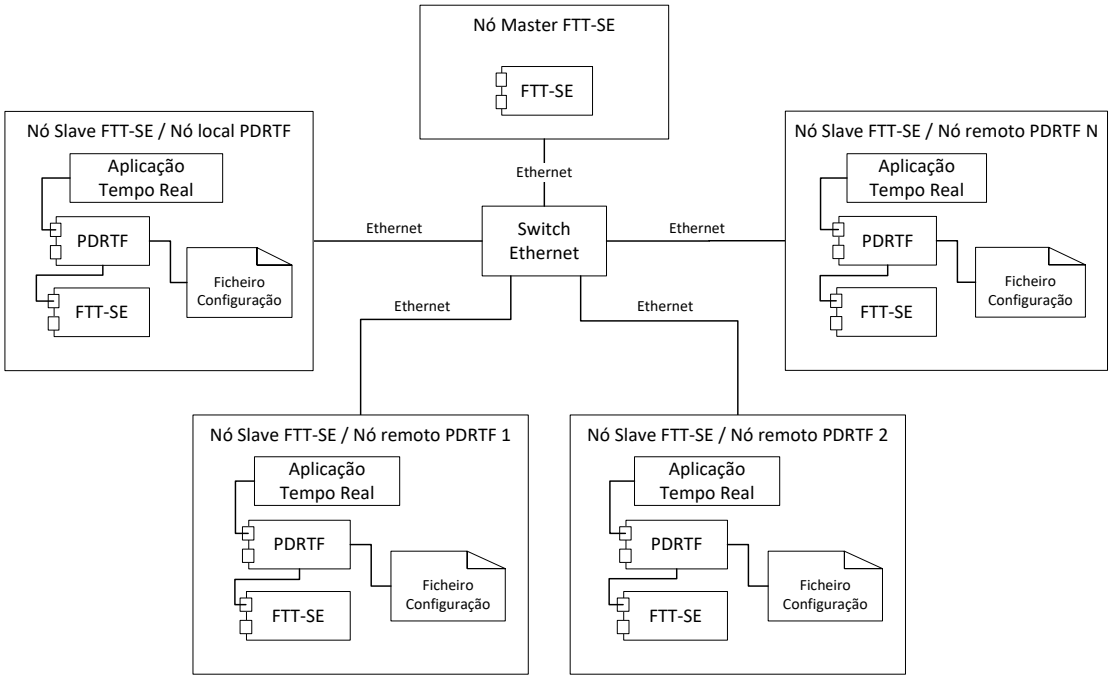

Figura 4: Diagrama de Deployment PDRTF.

Neste diagrama de *Deployment* o elemento "Aplicação Tempo-real" representa a programação das tarefas de tempo-real que serão realizadas com a PDRTF, também ela representada por um componente. A tecnologia FTT-SE encontra-se representada como um componente que é utilizado pela PDRTF. O ficheiro de configuração, tal como descrito anteriormente, utilizado pela PDRTF também se encontra neste diagrama.

3.2.2 Como utilizar a *Framework* PDRTF.

Nesta secção será demonstrada a aplicação prática da utilização da Interface C e do ficheiro de configuração da *Framework* PDRTF na programação de um conjunto de tarefas distribuídas de tempo-real.

Para programar tarefas distribuídas de tempo-real utilizando a PDRTF é necessário seguir 3 passos importantes:

1. Adicionar o nome da placa de rede e endereço Mac de cada nó do *cluster* ao ficheiro de configuração.
2. Programar no ponto de entrada `main()` do programa C o mapeamento das tarefas com as respetivas propriedades de tempo-real e atribuir aos nós remotos do cluster as partes paralelas/remotas de cada tarefa.
3. Programar para cada tarefa que foi adicionada, como descrito no ponto anterior, as funções que representam a parte local e remota e serão executadas como *callbacks* pela PDRTF.

Antes de continuar, será importante mencionar que, devido a existência de vários conceitos diferentes, a explicação de todos eles ao mesmo tempo pode-se tornar uma tarefa complexa, podendo dificultar a sua compreensão. Assim, de forma a melhor os explicar, o segundo e terceiro passos serão considerados como ações separadas, apesar de na prática acabarem por ser aplicados em conjunto como um único só.

Primeiro passo: Configuração do *cluster.*

Como foi descrito anteriormente a PDRTF identifica o nó onde está a executar através da informação contida no ficheiro de configuração, este ficheiro de configuração contem, para cada um dos nós, o nome e endereço Mac da placa de rede *Ethernet* que vai ser utilizada. Esta estrutura permite também, no caso de existirem várias placas de rede no nó, especificar qual delas se pretende utilizar. No Excerto de Código 1 encontra-se o ficheiro de configuração com a configuração utilizada durante o desenvolvimento e testes da PDRTF:

```
# config.cfg
# Ficheiro de configuração PDRTF.

# Endereço mac do nó local e nome da placa de rede usada.
local_node_mac_address = "e4:11:5b:56:8f:eb";
local_node_device_name = "eth0";

# Conjunto dos nós remotos.
remote_nodes = (
  {
    # Nó remoto 1.
    mac_address = "e4:11:5b:56:8f:86";
    device_name = "eth0";
  },
  {
    # Nó remoto 2.
    mac_address = "e4:11:5b:56:8f:66";
    device_name = "eth0";
  },
  {
    # Nó remoto 3.
    mac_address = "e4:11:5b:56:8f:bc";
    device_name = "eth0";
  });
```

Excerto de Código 1: Exemplo de ficheiro de configuração

Como se pode observar existe uma distinção clara entre o nó local e os nós remotos no ficheiro de configuração, os nós remotos são agrupados em conjunto sendo que a sua posição corresponde ao número que os identificará. Os comentários presentes no ficheiro de configuração, identificados através da presença do caracter # no seu início, foram colocados de maneira a melhor elucidar e complementar esta explicação. A próxima Figura 5 fornecerá uma representação gráfica da configuração descrita neste exemplo.

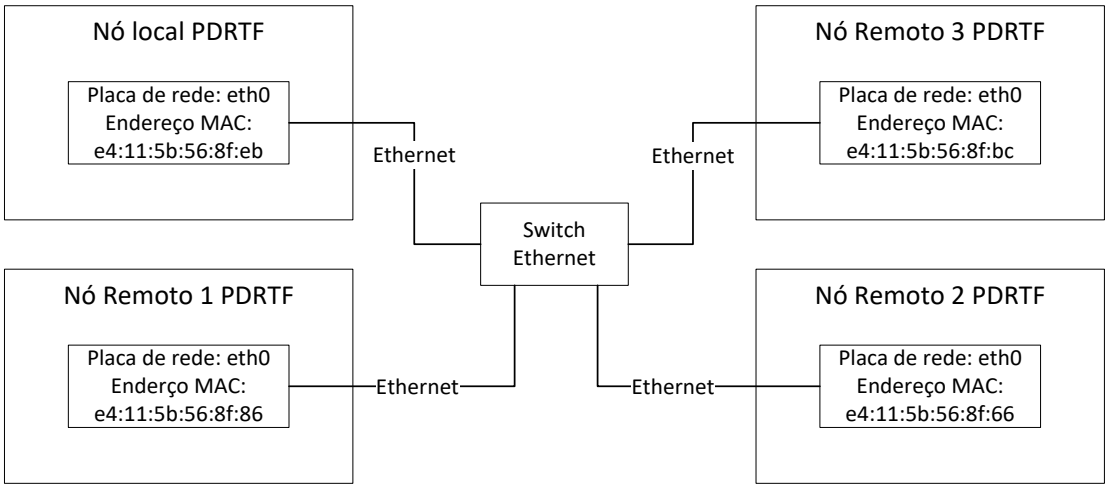

Figura 5: Configuração utilizada durante o desenvolvimento e testes da PDRTF.

Segundo passo: Mapear tarefas com a interface C PDRTF

Depois de adicionados os dados dos nós ao ficheiro de configuração, o próximo será a programação, no ponto de entrada `main()` do programa C, do mapeamento das tarefas com as respetivas propriedades de tempo-real.

A programação das tarefas deverá seguir o modelo *fork/join* distribuído, como explicado na secção 2.5, onde as tarefas são divididas em parte local e parte remota. Devido a natureza do sistema distribuído, não existe memória partilhada entre elas, isto significa que a parte remota tem de receber todos os dados partilhados que necessitar como parâmetro de entrada, retornando por sua vez à parte local todos os dados necessários como parâmetros de saída. Por esta razão, a PDRTF precisa que toda esta interação seja "mapeada" de forma a poder preparar todas as condições necessárias para a sua execução. De forma a melhor explicar este conceito de mapeamento das tarefas, encontra-se no Excerto de Código 2 um exemplo que contempla a programação de duas tarefas, cada uma com duas partes remotas/paralelas:

```
void main() {

	pdrtf_start();

	int id_tarefa_x = new_pdrtf_task(tarefa_x_parte_local, 150,
	48);
	set_pdrtf_task_sub_task(id_tarefa_x, 1,
	tarefa_x_parte_remota_1,
	sizeof(int), sizeof(char));
	set_pdrtf_task_sub_task(id_tarefa_x, 2,
	tarefa_x_parte_remota_2,
	sizeof(double), sizeof(float));
```

```
        int id_tarefa_y = new_pdrtf_task(tarefa_y_parte_local, 300,
        47);
        set_pdrtf_task_sub_task(id_tarefa_y, 1,
        tarefa_y_parte_remota_1,
        200, 100);
        set_pdrtf_task_sub_task(id_tarefa_y, 3,
        tarefa_y_parte_remota_2, 1000, 2000);

        pdrtf_launch_tasks();
}
```

Excerto de Código 2: Exemplo de utilização PDRTF no ponto de entrada main().

Neste exemplo é possível observar a sequência de instruções a seguir durante o mapeamento das tarefas, o significado de cada uma delas será explicado de seguida:

A função `pdrtf_start()` é a primeira a ser executada e não faz uso de qualquer parâmetro de entrada, esta é responsável por acionar a leitura do ficheiro de configuração e inicialização da biblioteca FTT-SE,

A função `new_pdrtf_task()` tem como função adicionar à lista interna da PDRTF uma nova tarefa a ser executada, após a sua execução esta função retorna um valor inteiro que representa o ID da tarefa criada, este ID será utilizado posteriormente para mapear nos nós remotos as respetivas partes remotas dessa tarefa. Os parâmetros desta função são enumerados e descritos em seguida:

- `void * function_ptr` Este é o primeiro parâmetro da função, ele contem o apontador para a função com a especificação da execução local da tarefa. Esta especificação será descrita mais a frente na terceira fase.
- `unsigned short period_in_ms` Este é o segundo parâmetro da função, ele contem o período/deadline em milissegundos da tarefa.
- `unsigned short linux_priority` Este é o terceiro e ultimo parâmetro da função, ele contem a prioridade Linux de tempo-real que será atribuída à tarefa. Este parâmetro pode tomar um valor entre 1 e 48. Um número maior corresponde a uma prioridade superior a um número inferior.

A função `set_pdrtf_task_sub_task()` tem como função atribuir a uma tarefa previamente adicionada pela função `new_pdrtf_task()` a parte remota a executar num dos nós disponíveis. De forma a melhor explicar esta operação, em seguida será apresentada uma figura que representa a disposição no *cluster* do mapeamento das tarefas caracterizado na Figura 8:

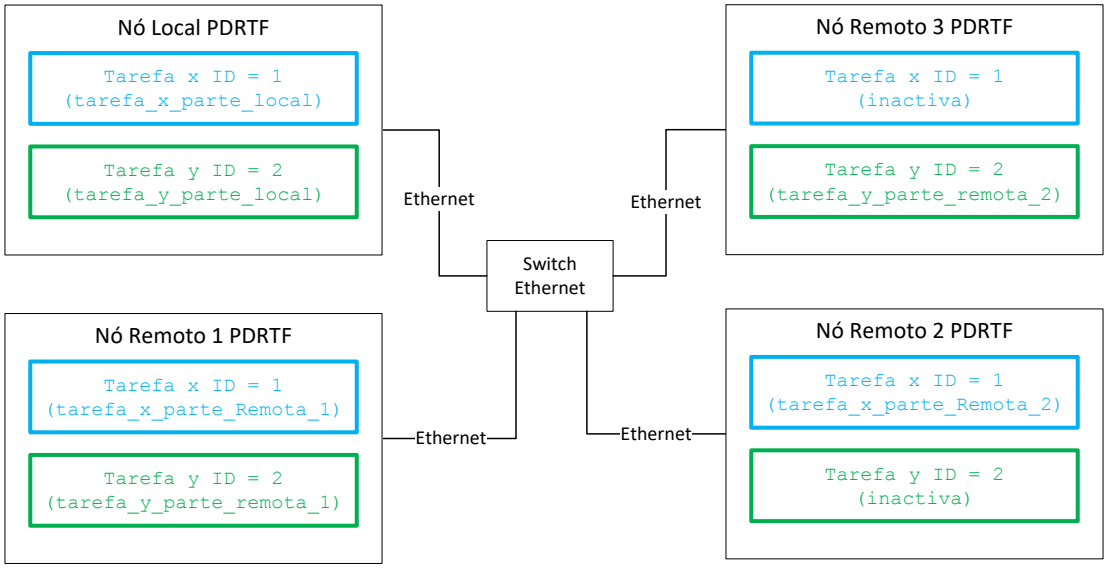

Figura 6: Disposição das partes locais e remotas das tarefas no cluster.

Cada tarefa tem a sua disposição todos os nós remotos do *cluster* e pode atribuir uma parte remota de execução a cada um deles, não tendo obrigatoriamente de o fazer para todos. Neste exemplo, para demonstrar esta possibilidade, foram apenas mapeadas partes remotas nos nós 1 e 2 para a tarefa x, enquanto para a tarefa y, apenas foram mapeadas partes remotas nos nós 1 e 3. Os parâmetros desta função são enumerados e descritos em seguida:

- `unsigned short pdrtf_task_id` Este é o primeiro parâmetro da função , contem o ID da tarefa à qual se pretende mapear uma parte remota. Este id terá sido retornado pela função `new_pdrtf_task()`, a quando da criação da tarefa.
- `unsigned short node_id` Este é o segundo parâmetro da função, ele contem o ID do nó remoto ao qual se pretende mapear a parte remota. Para cada tarefa só é possível mapear uma parte remota por nó, não sendo obrigatório que o seja feito.
- `void * function` Este é o terceiro parâmetro da função, ele contem o apontador para a função com a especificação da execução remota da tarefa. Esta especificação será descrita mais a frente.
- `unsigned short input_data_size` Este é o quarto parâmetro da função, ele contem o tamanho em bytes dos parâmetros de entrada da parte remota.
- `unsigned short output_data_size` Este é o quinto e ultimo parâmetro da função, ele contem o tamanho em bytes dos parâmetros de saída da parte remota.

A função `pdrtf_launch_tasks()` é a ultima a ser executada e não faz uso de qualquer parâmetro de entrada, esta encarregar-se-á de iniciar a abertura das *Streams* FTT-SE entre os nós do *cluster*, que serão usadas na comunicação tempo-real entre os nós, e de criar e lançar as threads que executarão as tarefas de tempo-real.

Terceiro passo: Programar parte local e remota de cada tarefa

É neste terceiro passo que serão programadas a parte local e as partes remotas de cada tarefa. De forma a melhor compreender como programar estas partes de uma tarefa, começaremos por fornecer um exemplo simplificado do código local e remoto, necessário para programar a tarefa x do exemplo anterior, que possui execução distribuída em 2 nós remotos. Como o processo será o mesmo para qualquer tarefa, a programação da tarefa y não será incluída neste exemplo.

```
void * tarefa_x_parte_local (void * param_pdrtf){

      /* execução sequencial */
      (…)

      int param_entrada_1;
      double param_entrada_1;
      pdrtf_launch_remote_execution(param_pdrtf, &param_entrada_1,
      sizeof (int), 1);
      pdrtf_launch_remote_execution(param_pdrtf, &param_entrada_2,
      sizeof (double), 2);

      /* execução local */
      (…)


      char param_saida_1;
      float param_saida_2;
      pdrtf_get_remote_execution_result(param_pdrtf, &param_saida_1,
      sizeof (char), 1);
      pdrtf_get_remote_execution_result(param_pdrtf, &param_saida_2,
      sizeof (float), 2)

      /* processar dados recebidos */
      (…)

}

void * tarefa_x_parte_remota_1 (void * param_pdrtf){

      int param_entrada_1;
      pdrtf_get_input_parameters(param_pdrtf, &param_entrada_1,
      sizeof (int));

      /* execução remota */
      /* processar dados recebidos e retornar resultados */
      (…)
```

```
        char param_saida_1;
        pdrtf_set_output_parameters(param_pdrtf, &param_saida_1, sizeof
        (char));
}

void * tarefa_x_parte_remota_2 (void * param_pdrtf){

        double param_entrada_2;
        pdrtf_get_input_parameters(param_pdrtf, &param_entrada_2,
        sizeof (double));

        /* execução remota */
        /* processar dados recebidos e retornar resultados */
        (…)


        float param_saida_2;
        pdrtf_set_output_parameters(param_pdrtf, &param_saida_2, sizeof
        (float));
}
```

Excerto de Código 3: Exemplo simplificado do código local e remoto

De forma a ajudar à compreensão deste exemplo de código, é apresentada uma representação visual de como se processa a interação entre a parte local e remota através do diagrama de sequência da Figura 7 e diagramas de estado da Figura 8 e 9:

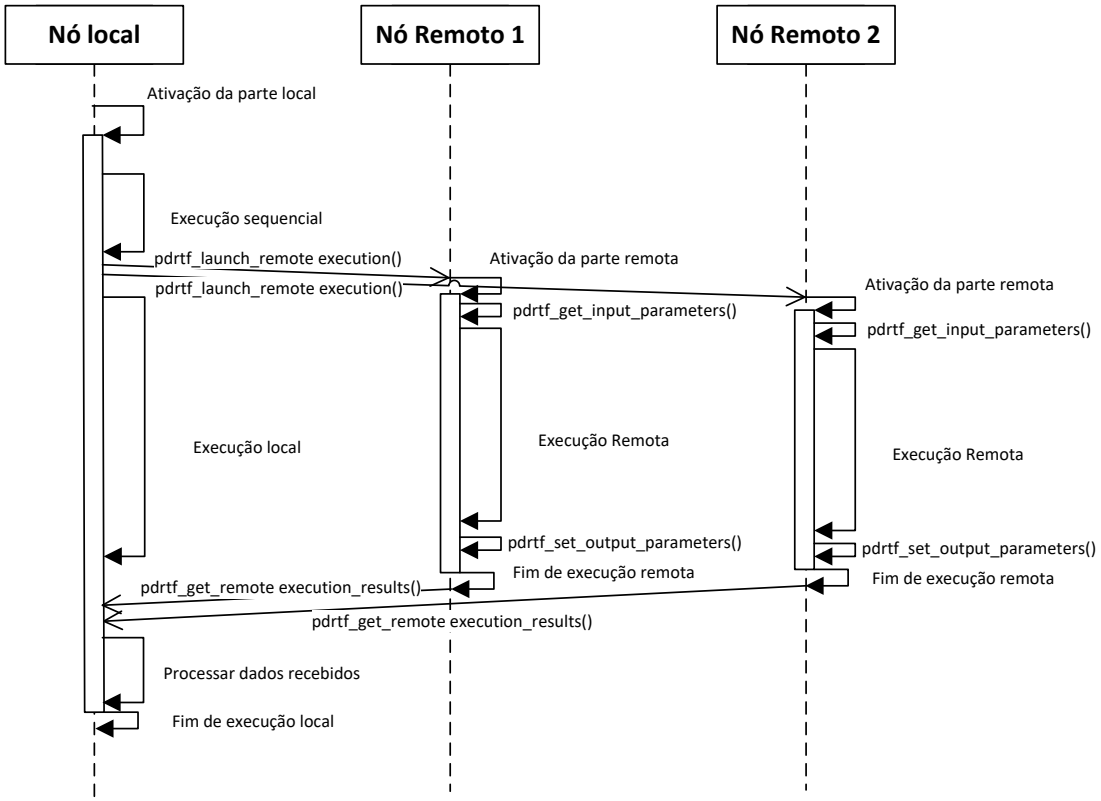

Figura 7: Diagrama de sequência da Interação entre parte local e remota da tarefa.

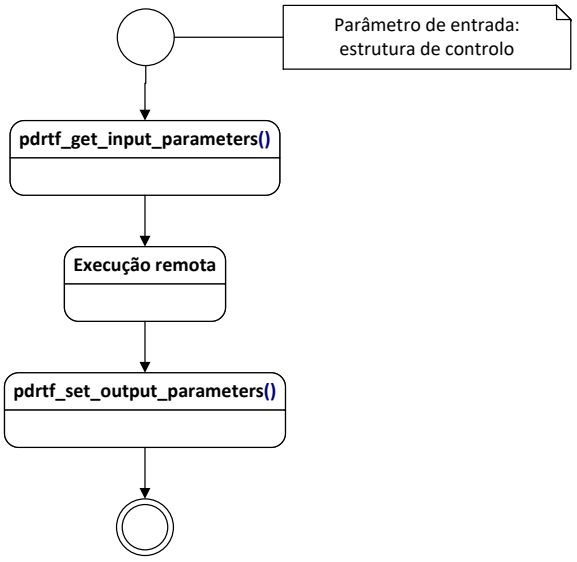

Figura 8: Diagrama de estados da parte remota da tarefa.

Será importante mencionar que este exemplo foca-se apenas na troca de informação entre a parte local e remota, que é parte relevante para implementar o modelo *fork/join,* os restantes detalhes da lógica da tarefa que serão deixados a cargo do programador estão representados nó código através de comentários.

A assinatura da função local e remota deverá ser do tipo `void *` porque será executada como uma *callback* pela *Framework* e deverá ter um único parâmetro de entrada também do tipo `void *`, nomeado neste exemplo de `param_pdrtf`, que será fornecido pela *Framework* durante a execução. Este parâmetro de entrada contem uma estrutura de controlo que será utilizada em todas as chamadas à *Framework* de forma a identificar a tarefa.

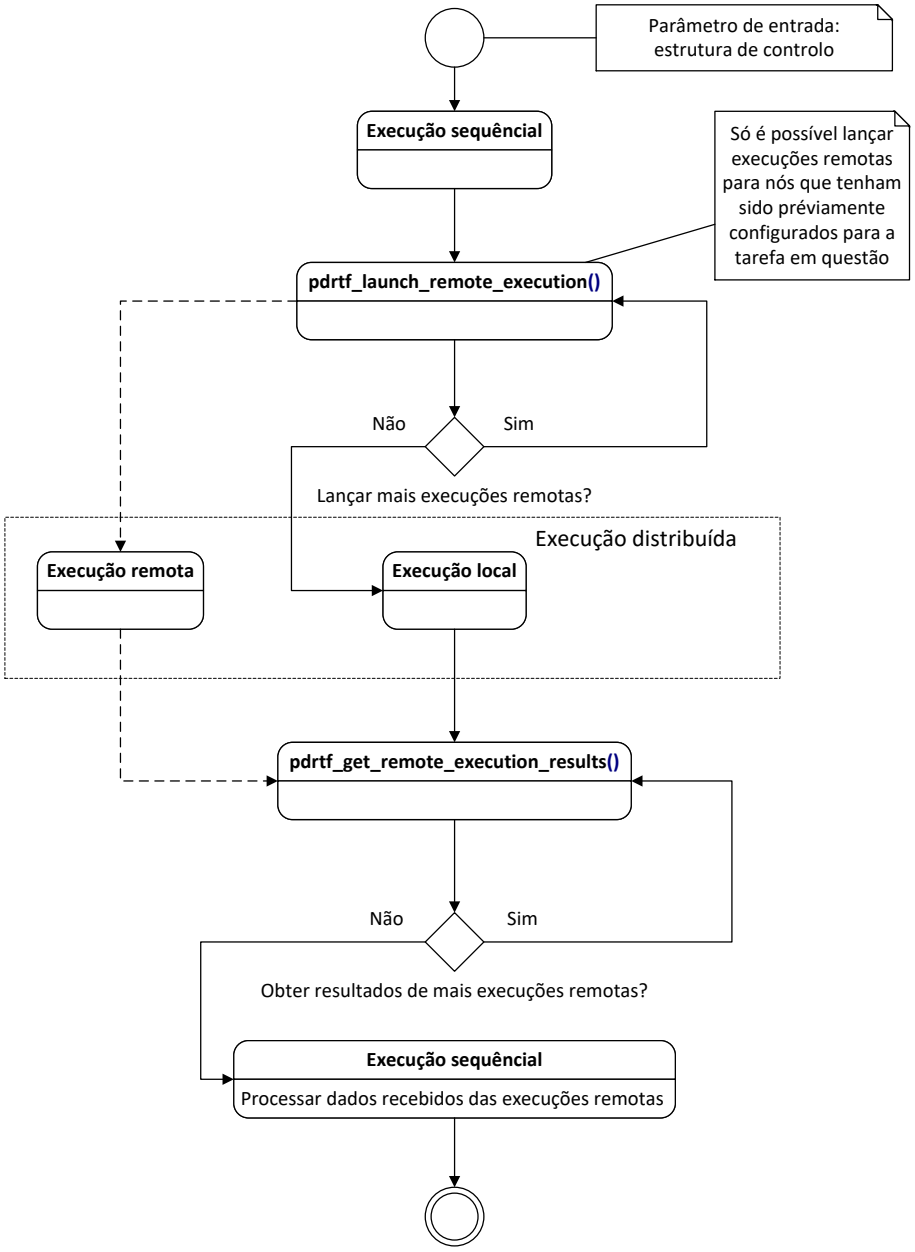

Figura 9: Diagrama de estados da parte local da tarefa.

A função `pdrtf_launch_remote_execution()` é utilizada apenas na parte local da tarefa e é responsável por iniciar o envio dos parâmetros de entrada da função remota para o nó onde ela estiver configurada para executar. Como se pôde observar no diagrama de sequência anterior este envio também é responsável por despoletar a execução da parte remota. Os parâmetros desta função são agora enumerados e descritos:

- `void * task_data`: Este é o primeiro parâmetro da função, e como já foi descrito anteriormente, ele contêm a estrutura de controlo que permite à *Framework* identificar a tarefa, este parâmetro será fornecido através do parâmetro de entrada da função local ou remota.
- `void * input_parameters`: Este é o segundo parâmetro da função e corresponde a um apontador C de tipo indefinido para posição de memória onde se encontram os dados dos parâmetros da função remota a serem enviados.
- `unsigned short input_parameters_size`: Este é o terceiro parâmetro da função e corresponde ao tamanho dos dados que são apontados pelo parâmetro `input_parameters`. Esta informação é necessária porque a *Framework* permite enviar qualquer tipo de dados entre os nós, prevendo a utilização de estruturas de dados criadas pelo programador de tamanho desconhecido. Este facto também explica a razão do uso de um apontador de tipo indefinido `void *`. Além disso também serve como verificação do cumprimento do tamanho fixo dos parâmetros de entrada, pré-estabelecido durante a fase de mapeamento de tarefas.
- `unsigned short pdrtf_node_id`: Este é o quarto e último parâmetro da função e contem a identificação do nó para o qual estão a ser enviados os parâmetros de entrada da função remota. Este parâmetro deverá corresponder a um nó que exista no *cluster* e tenha sido configurado durante a fase de mapeamento de tarefas.

A função `pdrtf_get_remote_execution_results()` é utilizada apenas na parte local da tarefa e é responsável por iniciar a receção dos parâmetros de saída da função remota do nó onde ela estiver configurada para executar. Caso, a quando da chamada desta função, o processamento remoto não estiver concluído, a execução local bloqueará até que a execução remota termine e a mensagem seja recebida. Os parâmetros desta função são agora enumerados e descritos:

- `void * task_data`: Este é o primeiro parâmetro da função, e como já foi descrito anteriormente, ele contêm a estrutura de controlo que permite à *Framework* identificar a tarefa, este parâmetro será fornecido através do parâmetro de entrada da função local ou remota.
- `void * output_parameters`: Este é o segundo parâmetro da função e corresponde a um apontador C de tipo indefinido para posição de memória onde serão copiados os dados dos parâmetros de saída da função remota que forem retornados.

- `unsigned short output_parameters_size`: Este é o terceiro parâmetro da função e corresponde ao tamanho dos dados que são apontados pelo parâmetro `output_parameters`. Esta informação é necessária porque a *Framework* permite enviar qualquer tipo de dados entre os nós, prevendo a utilização de estruturas de dados criadas pelo programador de tamanho desconhecido. Este facto também explica a razão do uso de um apontador de tipo indefinido `void *`. Além disso também serve como verificação do cumprimento do tamanho fixo dos parâmetros de saída, pré-estabelecido durante a fase de mapeamento de tarefas.
- `unsigned short pdrtf_node_id`: Este é o quarto e último parâmetro da função e contem a identificação do nó a partir do qual serão recebidos os parâmetros de saída da função remota. Este parâmetro deverá corresponder a um nó que exista no *cluster* e tenha sido configurado durante a fase de mapeamento de tarefas.

A função `pdrtf_get_input_parameters()` é utilizada apenas na parte remota da tarefa e é responsável por obter os parâmetros de entrada que foram recebidos a quando da sua ativação. Os parâmetros desta função são agora enumerados e descritos:

- `void * task_data`: Este é o primeiro parâmetro da função, e como já foi descrito anteriormente, ele contêm a estrutura de controlo que permite à *Framework* identificar a tarefa, este parâmetro será fornecido através do parâmetro de entrada da função local ou remota.
- `void * data`: Este é o segundo parâmetro da função e corresponde a um apontador C de tipo indefinido para posição de memória onde serão copiados os dados dos parâmetros de entrada da função remota que foram recebidos.
- `unsigned short data_size`: Este é o terceiro e ultimo parâmetro da função e corresponde ao tamanho dos dados que são apontados pelo parâmetro `data`. Esta informação é necessária porque a *Framework* permite enviar qualquer tipo de dados entre os nós, prevendo a utilização de estruturas de dados criadas pelo programador de tamanho desconhecido. Este facto também explica a razão do uso de um apontador de tipo indefinido `void *`. Além disso também serve como verificação do cumprimento do tamanho fixo dos parâmetros de saída, pré-estabelecido durante a fase de mapeamento de tarefas.

A função `pdrtf_set_output_parameters()` é utilizada apenas na parte remota da tarefa e é responsável por guardar os parâmetros de saída que forem produzidos a quando da sua

execução, sendo posteriormente enviados de volta para a parte local da tarefa. Os parâmetros desta função são agora enumerados e descritos:

- `void * task_data`: Este é o primeiro parâmetro da função, e como já foi descrito anteriormente, ele contêm a estrutura de controlo que permite à *Framework* identificar a tarefa, este parâmetro será fornecido através do parâmetro de entrada da função local ou remota.
- `void * data`: Este é o segundo parâmetro da função e corresponde a um apontador C de tipo indefinido para posição de memória a partir de onde serão copiados os dados dos parâmetros de saída da função remota que foram produzidos.
- `unsigned short data_size`: Este é o terceiro e ultimo parâmetro da função e corresponde ao tamanho dos dados que são apontados pelo parâmetro `data`. Esta informação é necessária porque a *Framework* permite enviar qualquer tipo de dados entre os nós, prevendo a utilização de estruturas de dados criadas pelo programador de tamanho desconhecido. Este facto também explica a razão do uso de um apontador de tipo indefinido `void *`. Além disso também serve como verificação do cumprimento do tamanho fixo dos parâmetros de saída, pré-estabelecido durante a fase de mapeamento de tarefas.

3.2.3 Funcionamento interno das funções da API

Esta secção irá descrever o funcionamento interno da API da PDRTF descrevendo as funções pela ordem que apareceram na secção anterior. Internamente a PDRTF faz uso de um componente designado de FTT-SE Wrapper para lidar com a biblioteca do FTT-SE. As chamadas internas ao FTT-SE Wrapper serão demonstradas nesta secção quando necessário, no entanto a descrição detalhada do seu funcionamento será deixado para a próxima secção, a qual será dedicada ao componente FTT-SE Wrapper.

3.2.3.1 pdrtf_start()

Esta função é a primeira ser executada, entre outras operações, ela altera a política de escalonamento da thread em execução para a política de tempo-real SCHED_FIFO e define a prioridade da tarefa como 49. Este facto merece especial atenção e está diretamente relacionado com o uso da biblioteca FTT-SE, sendo a seguir explicada a sua razão.

A biblioteca FTT-SE foi desenvolvida para trabalhar em Linux e RTLinux que é um sistema operativo tempo-real. No nosso caso, como o RTLinux não é uma distribuição open-source foi colocada de parte a sua utilização, no entanto, devido a necessidade de um sistema operativo de tempo-real, o Linux foi aqui utilizado com recurso a patch Preempt RT como descrito na

secção 2.2.2. Durante a integração da PDRTF com a tecnologia FTT-SE foram encontradas as seguintes condicionantes:

- Ao correr threads de tempo-real em concorrência com as threads da biblioteca FTT-SE, se as ultimas não tiverem prioridade sobre todas as outras, serão objeto de preempção. Como do lado dos *Slaves* é necessária constante verificação da *Trigger Message*, se este processo for interrompido o FTT-SE falhará.
- Se as threads em execução tiverem uma prioridade igual ou superior as threads do *kernel*, que possuem prioridade de nível 50, o mesmo poderá eventualmente suceder. A observação deste facto leva a crer que o FTT-SE depende de chamadas ao sistema, atendendo ao facto que no caso da patch Preempt RT o *kernel* do sistema operativo passa a ser completamente preemptível, a existência de chamadas ao sistema em outras threads poderá levar a que o acesso a estes recursos pelo FTT-SE seja interrompido, causado o problema aqui descrito.

Apesar destas condicionantes as restantes qualidades da patch Preempt RT justificam ainda assim a sua utilização. Alem do mais, apesar desta situação significar que garantias de tempo-real *Hard* não poderão ser garantidas, a utilidade do desenvolvimento e estudo da PDRTF mantem-se. Desta maneira, como a resolução mais aprofundada destas condicionantes se encontra fora do âmbito deste projeto, para este caso de estudo, assumiu-se que as threads FTT-SE correriam com uma prioridade 49, que se encontra abaixo da prioridade das tarefas da *kernel* do Linux, e que as tarefas criadas e geridas pela PDRTF teriam uma prioridade inferior a 49 de forma a não interromperem a execução das threads do FTT-SE.

Voltando a explicação da função pdrtf_start(), a definição da thread corrente para a prioridade 49 garante que as threads do FTT-SE criadas durante a sua inicialização partilharão a mesma prioridade.

A próxima ação a ser executada é a leitura do ficheiro de configuração, para determinar em que nó se encontra a executar, após o seu término será despoletado por intermédio do FTT-SE Wrapper a inicialização da biblioteca FTT-SE na placa de rede configurada. Caso termine com sucesso retornará 1 ou retornará um número negativo em caso de erro. O processo aqui explicado encontra-se representado na Figura 15 por um diagrama de estados:

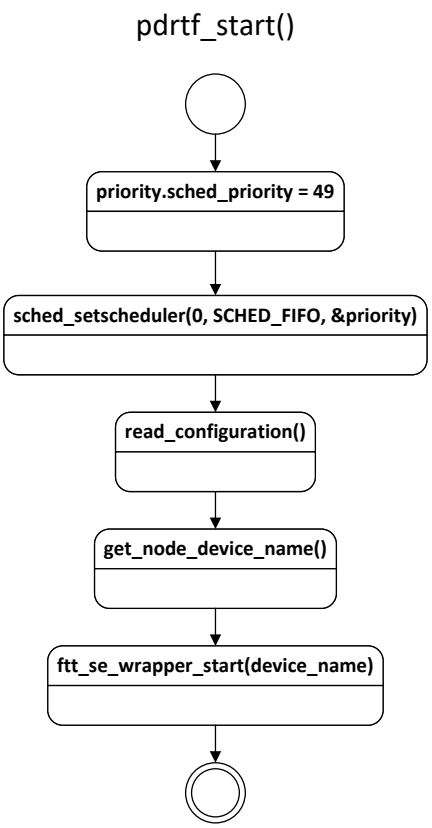

Figura 10: Máquina de estados da função pdrtf_start()

De forma a melhor se compreender o processo da leitura do ficheiro de configuração, encontra-se na Figura 17 um diagrama de estados detalhando o funcionamento da função read_configuration():

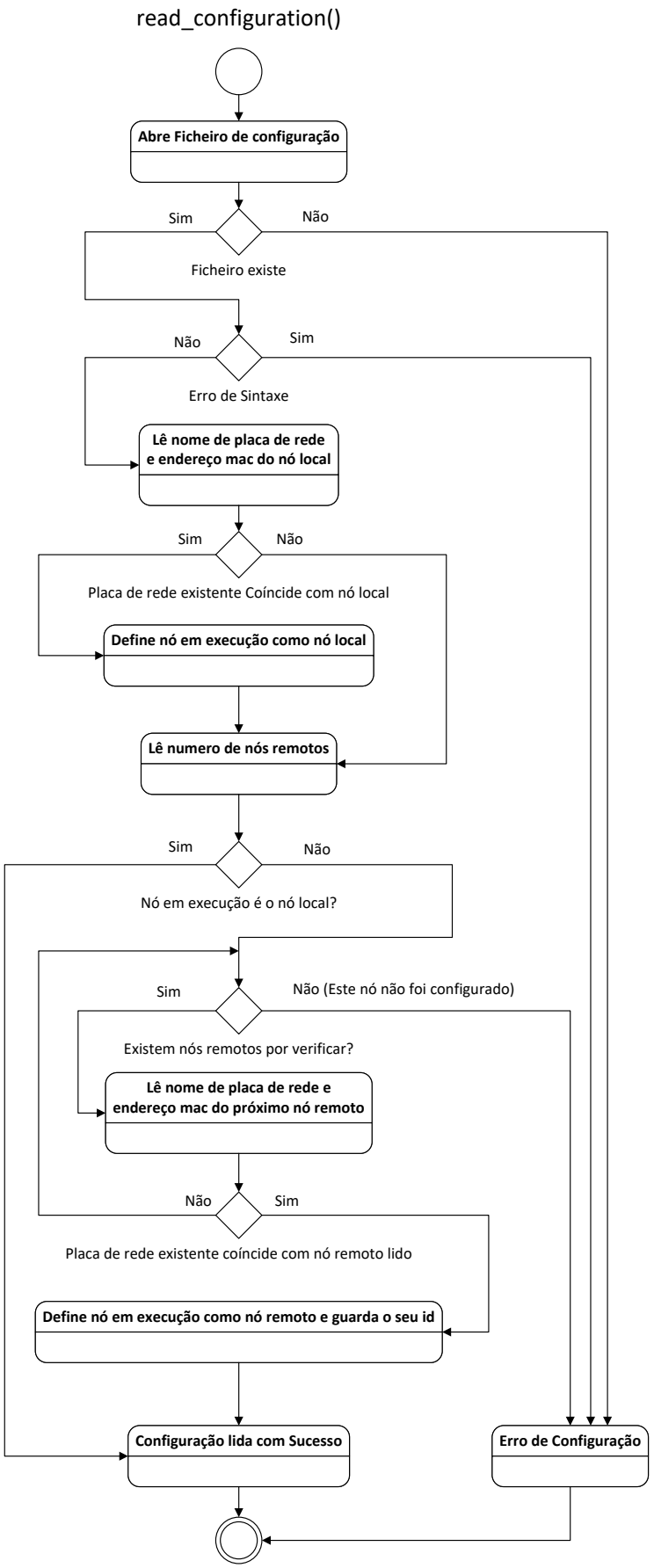

Figura 11: Diagrama de estados do processo de Leitura do ficheiro de configuração.

3.2.3.2 new_pdrtf_task()

A função `new_pdrtf_task()` adiciona uma nova tarefa à lista interna do PDRTF. Ela começa por verificar se o apontador para a função da parte local `function_ptr` é válido e se a prioridade da tarefa `linux_priority` está dentro do intervalo entre 1 e 48. Verificando-se estas condições, ela alocará memória para uma nova posição da lista interna de tarefas e guardará os parâmetros introduzidos. Caso termine com sucesso retornará um número positivo com a identificação da tarefa senão retornará um número negativo representativo do erro.

3.2.3.3 set_pdrtf_task_sub_task()

A função `set_pdrtf_task_sub_task()` atribuí a um nó disponível de uma tarefa, uma parte remota de execução. Ela começa por verificar se o id da tarefa `pdrtf_task_id` e o id do nó remoto `node_id` existem e se o apontador para a função da parte remota `function` é válido. Após essas verificações irá guardar, para a tarefa existente na lista interna, no nó escolhido, o apontador da função da parte remota de execução e respetivos tamanhos dos parâmetros de entrada `input_data_size` e saída `output_data_size`. A condição do nó perante a tarefa passará para ativo. Caso termine com sucesso retornará 1, senão retornará um número negativo representativo do erro.

3.2.3.4 pdrtf_launch_tasks()

A função `pdrtf_launch_tasks()` irá percorrer as tarefas guardadas na lista interna de tarefas e com essa informação tratará de primeiro criar as streams que serão necessárias a comunicação entre os nós e a seguir de criar e lançar as threads que irão executar as respetivas partes locais e remotas das tarefas. Na Figura 12 encontra-se um diagrama de estados representativo da criação das streams para o nó local:

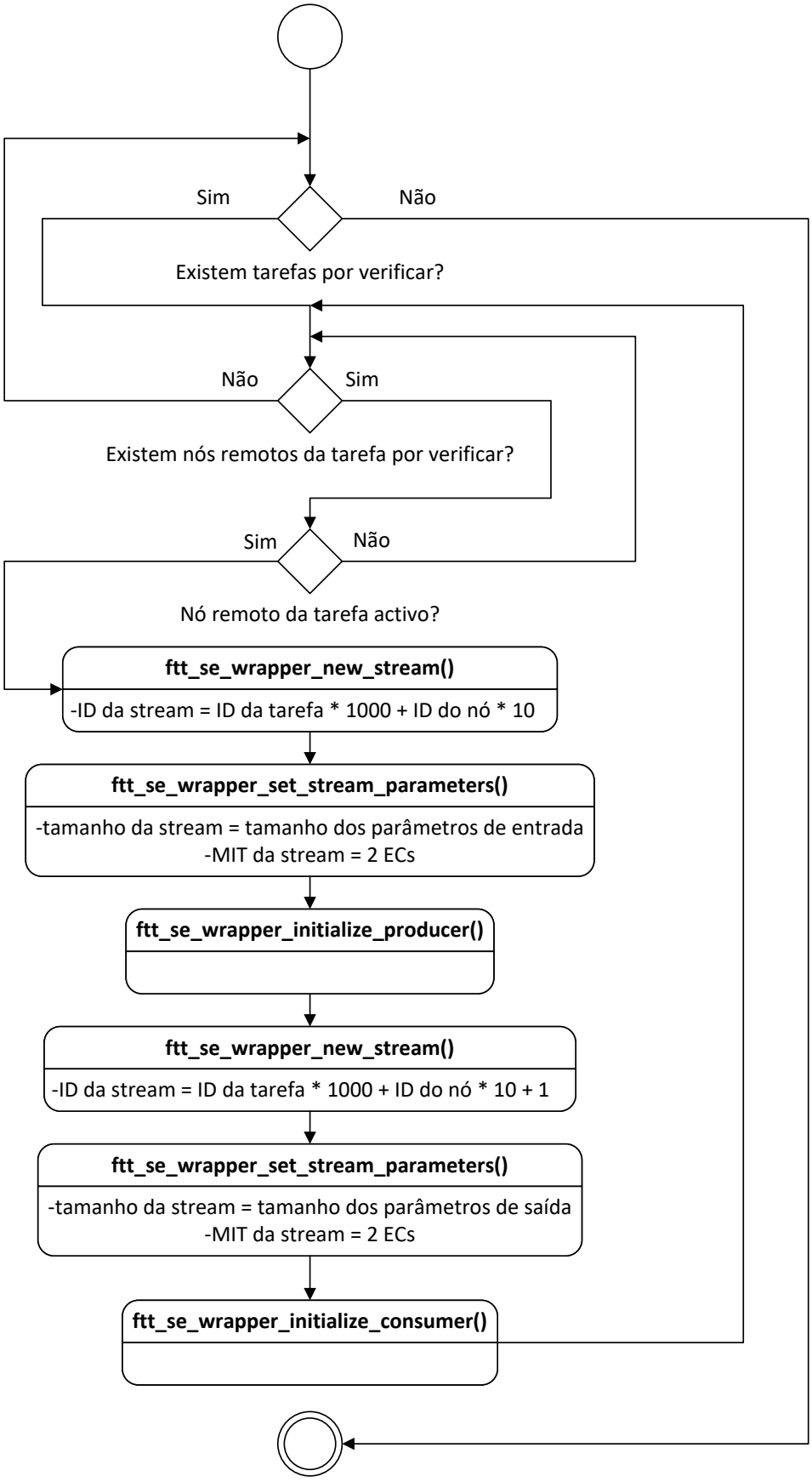

Figura 12: Diagrama de estados da criação de streams no nó local

Neste diagrama, é possível observar que, para cada parte remota, serão criadas duas streams, correspondentes à entrada e saída dos seus parâmetros, e apenas para os nós que estiverem ativos, ou seja, os nós que foram mapeados através da função `set_pdrtf_task_sub_task()`.

Primeiro, será chamada a função `ftt_se_wrapper_new_stream()` que irá alocar a posição de memoria onde guardará as definições da stream, correspondentes ao seu id, tipo (síncrona ou assíncrona) e numero de posições do buffer. Como a primeira stream a ser definida é a de entrada para o nó, o seu id será dado pela fórmula (ID da tarefa * 1000 + ID do nó * 10) como demonstrado no diagrama. A stream será do tipo assíncrono, pois a natureza das mensagens síncronas FTT-SE impossibilita a sua utilização neste contexto. O buffer terá apenas uma posição, pois nunca será enviada uma nova mensagem na mesma stream sem que a anterior tenha sido consumida.

A segunda função chamada será a função `ftt_se_wrapper_set_stream_parameters()` que irá definir as restantes propriedades da stream, nomeadamente o seu tamanho, o tamanho do MTU a usar e o seu período/MIT. Neste caso o seu tamanho corresponde ao tamanho dos parâmetros de entrada da função remota, o tamanho de MTU escolhido é de 1450 bytes, que será o mesmo a usar em todas as streams, por fim o Período/MIT escolhido será de 2 Elementary Cycles, que é o mínimo exigível de uma stream FTT-SE do tipo assíncrono como descrito na secção 2.4.5. A escolha do valor mínimo baseia-se na necessidade das aplicações de tempo-real distribuídas dependerem não só do determinismo, como do desempenho máximo possível da transmissão de forma a reduzir o overhead associado, maximizando o ganho face à paralelização.

A terceira função chamada será a função `ftt_se_wrapper_initialize_producer()` que irá efetivamente abrir a stream criada sendo o nó o produtor da mesma.

Para a stream correspondente a saída da parte remota, as chamadas as funções `ftt_se_wrapper_new_stream()` e `ftt_se_wrapper_set_stream_parameters()` serão iguais as anteriores com a diferença que o id da stream será dado pela fórmula (ID da tarefa * 1000 + ID do nó * 10 + 1) e o tamanho da stream corresponderá ao tamanho dos parâmetros de saída da função remota.

Por fim será chamada a função `ftt_se_wrapper_initialize_consumer()` que irá efetivamente abrir a stream criada sendo o nó o consumidor da mesma.

Na Figura 13 encontra-se um diagrama de estados representativo da criação das streams para o nó remoto:

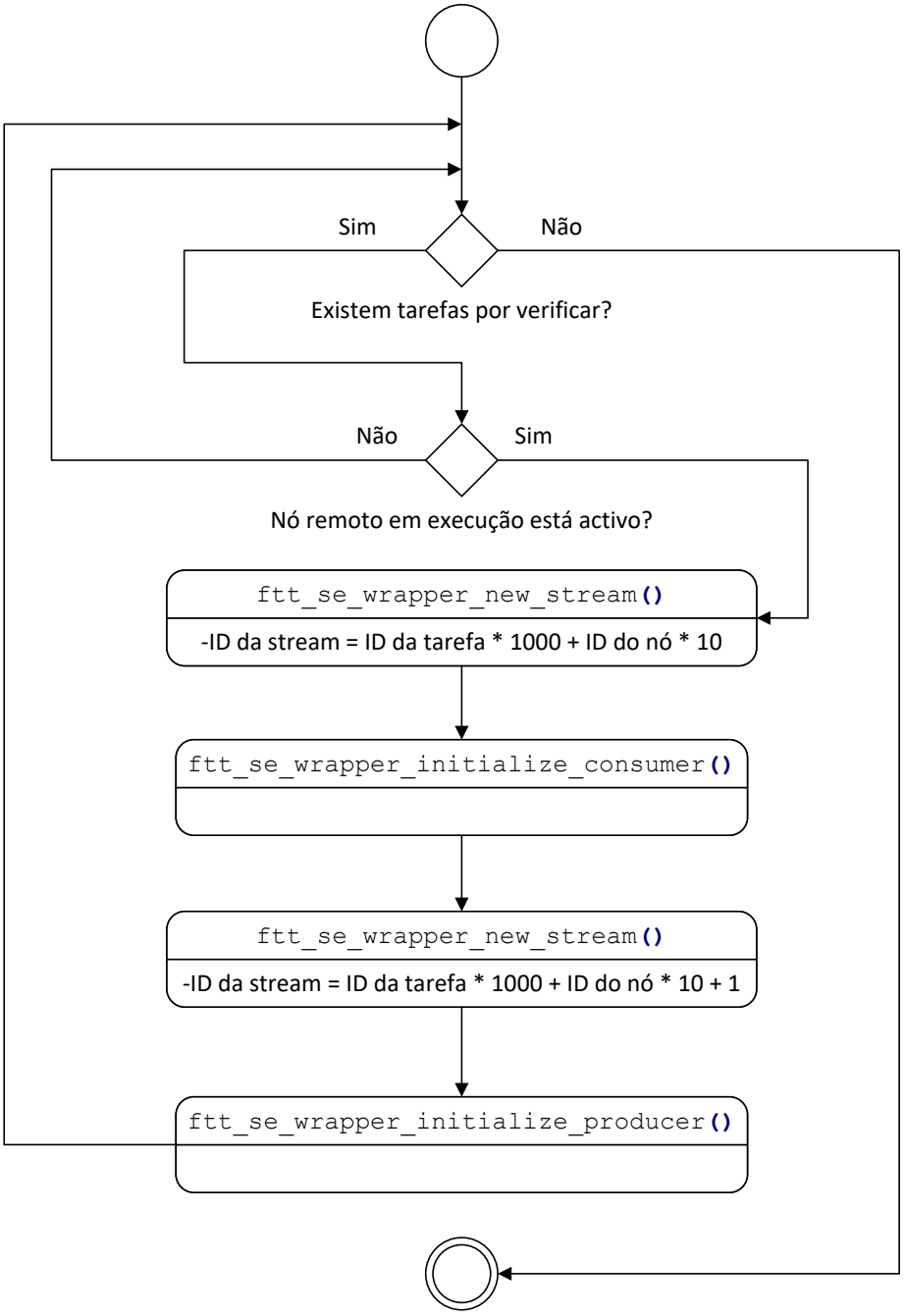

Figura 13: Diagrama de estados da criação de streams no nó remoto

Neste diagrama é possível observar o processo de criação das streams do lado do nó remoto, este processo assemelha-se ao do nó local com algumas diferenças: apenas são abertas streams referentes ao próprio nó, a função `ftt_se_wrapper_set_stream_parameters()` não é chamada pois só precisa de ser

chamada no nó local. Em vez de produtor é consumidor da stream de entrada, e em vez de consumidor é produtor da stream de saída.

De forma a complementar a explicação anteriorna Figura 14 encontra-se uma representação gráfica de um exemplo de atribuição dos ids às streams para duas tarefas, cada uma com partes remotas em dois nós.

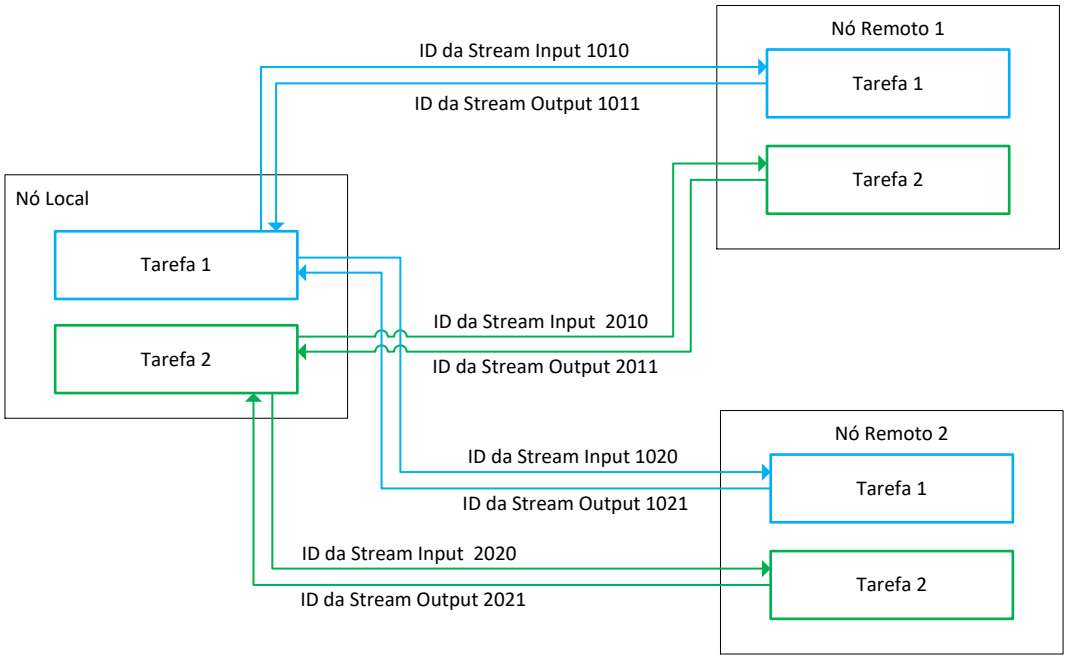

Figura 14: Exemplo de atribuição de ids às streams para duas tarefas com dois nós remotos.

Depois de criadas as streams será então a vez de serem criadas as threads que executaram as partes locais e remotas das tarefas. De forma a melhor explicar este processo na Figura 15 e 16 encontram se os diagramas de estado que descrevem este processo tanto do lado do nó local como do lado do nó remoto:

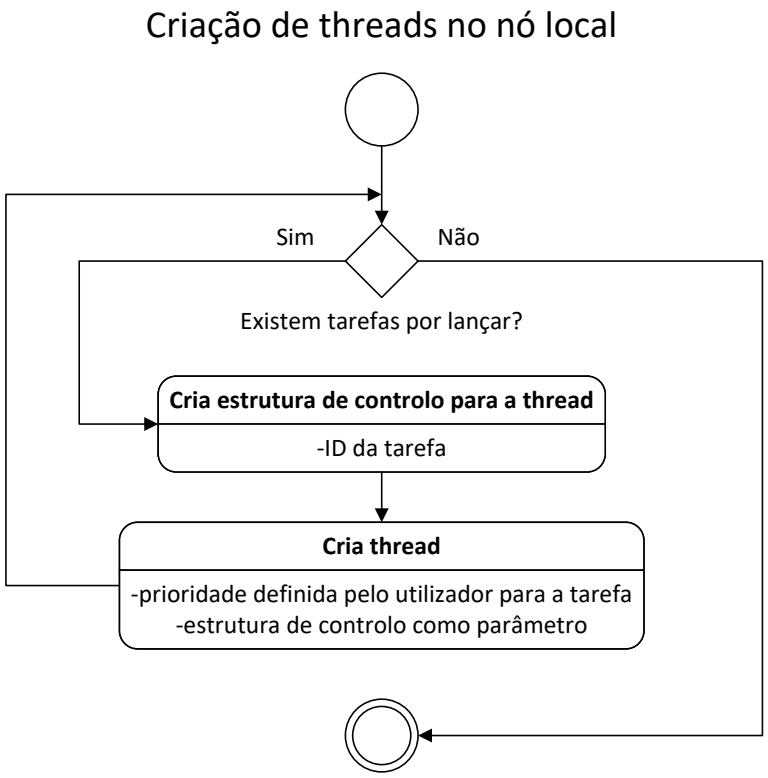

Figura 15: Diagrama de estados da criação de threads no nó local

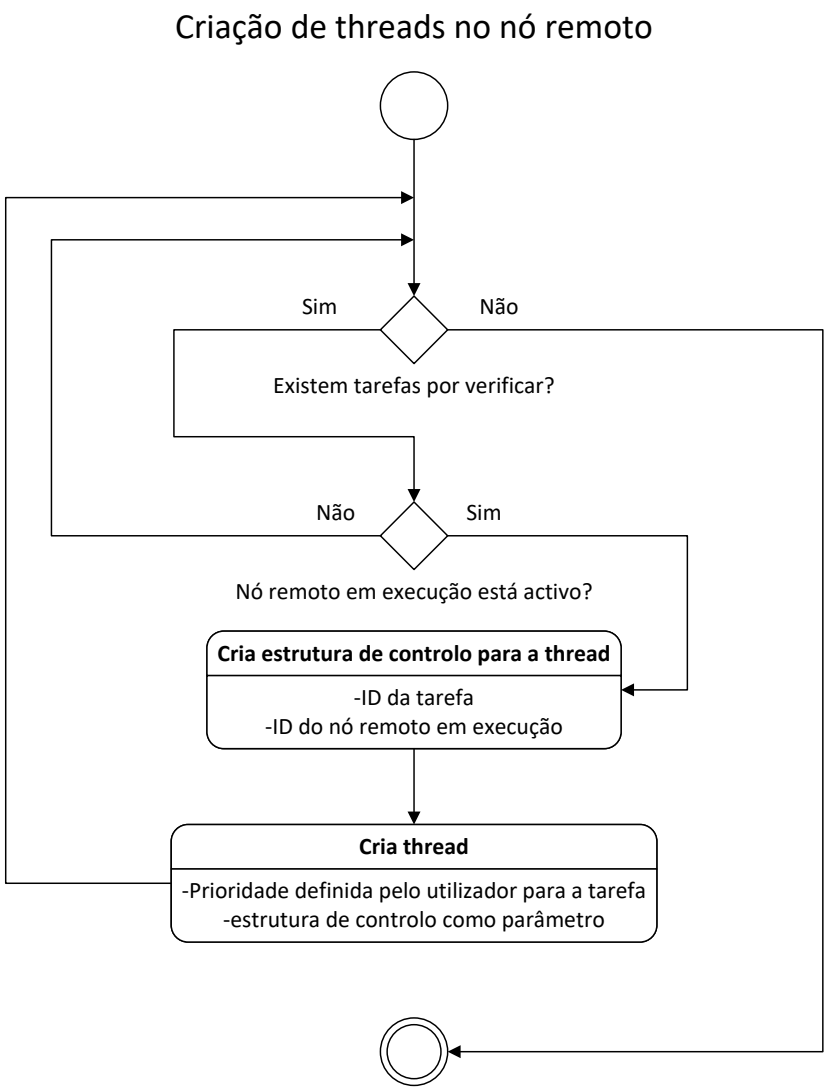

Figura 16: Diagrama de estados da criação de threads no nó remoto

Depois de criadas as threads estas encarregar-se-ão de chamar as funções especificadas para as partes locais e remotas das tarefas como se pode observar nos próximos diagramas de estado representados na Figura 17 e 18:

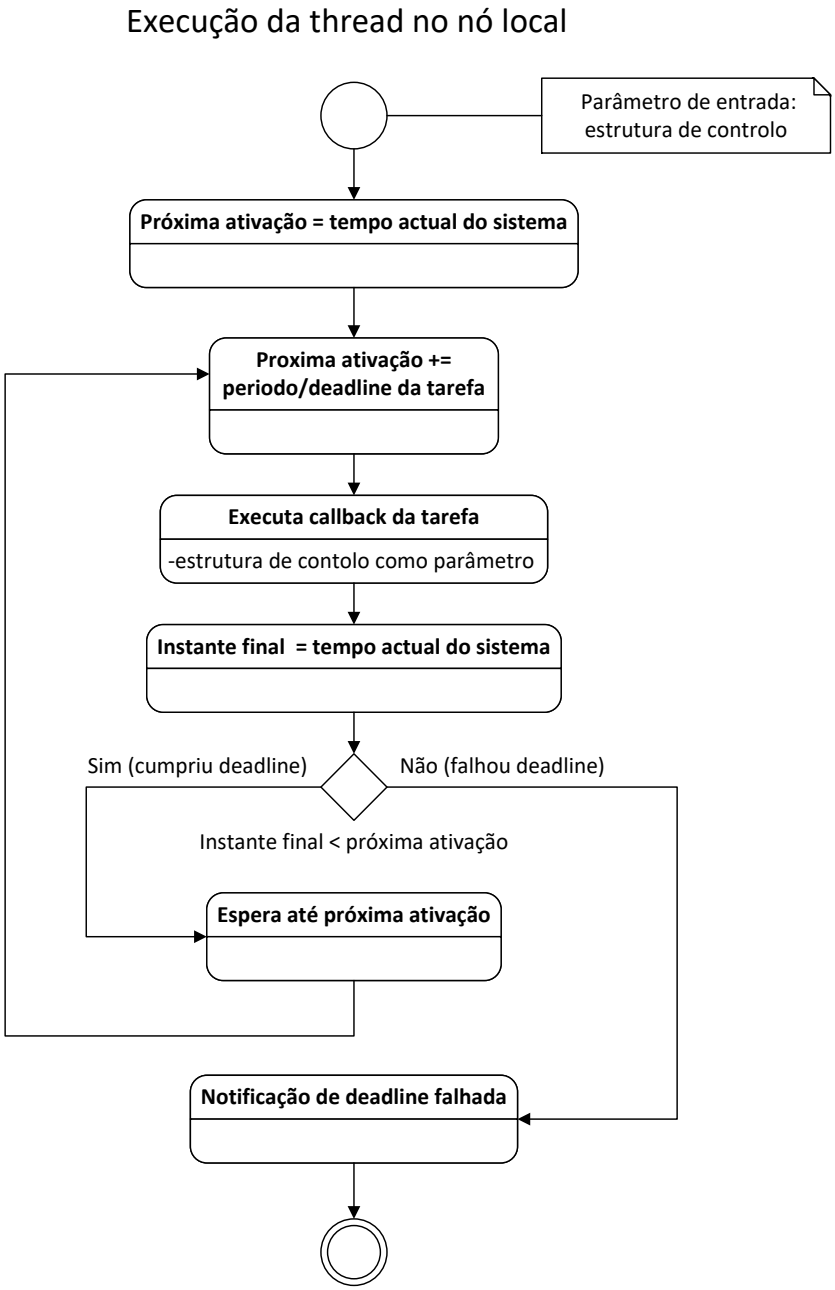

Figura 17: Diagrama de estados da execução da thread no nó local.

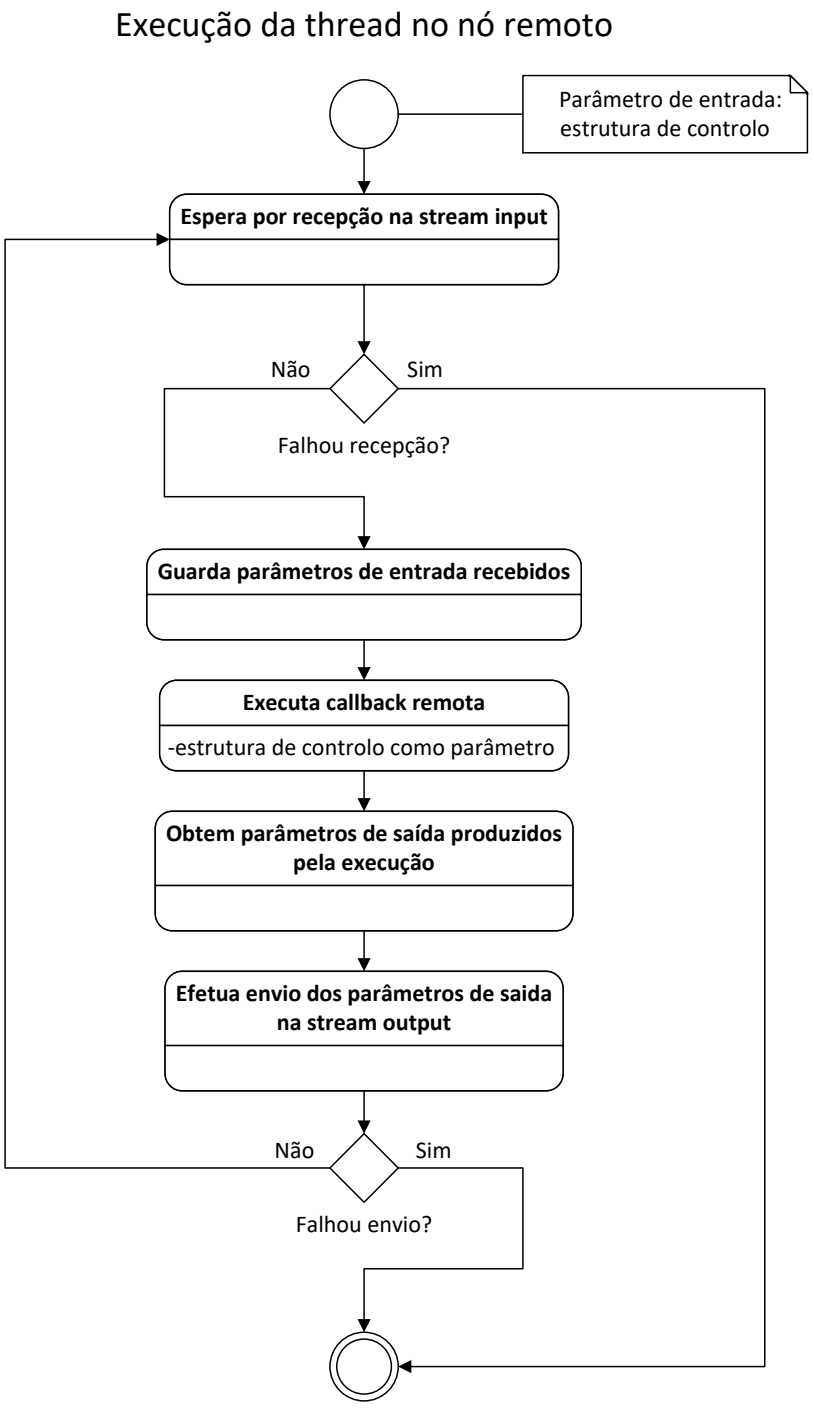

Figura 18: Diagrama de estados da execução da thread no nó remoto.

3.2.3.5 pdrtf_launch_remote_execution()

A função `pdrtf_launch_remote_execution()` é responsável por executar o envio dos parâmetros de entrada da função remota para o nó onde ela se encontra a executar. Esta função começa por verificar se a estrutura de controlo `task_data` é válida e o id do nó `pdrtf_node_id` existe e está ativo, verifica se o apontador da mensagem a enviar `input_parameters` é válido e se o tamanho da mensagem `input_parameters_size`

coincide com o tamanho especificado pela função `set_pdrtf_task_sub_task()` durante a fase de mapeamento de tarefas.

Depois de verificados os parâmetros de entrada ela chamará a função `ftt_se_wrapper_send()` que acionará o envio da mensagem para o nó escolhido, através da stream de entrada da parte remota da tarefa.

3.2.3.6 pdrtf_get_remote_execution_result()

A função `pdrtf_get_remote_execution_results()` é responsável por acionar a recção dos parâmetros de saída da função remota a partir do nó onde ela se encontra a executar. Esta função começa por verificar se a estrutura de controlo `task_data` é válida e o id do nó `pdrtf_node_id` existe e está ativo, verifica se o apontador da mensagem a receber `output_parameters` é válido e se o tamanho da mensagem `output_parameters_size` coincide com o tamanho especificado pela função `set_pdrtf_task_sub_task()` durante a fase de mapeamento de tarefas.

Depois de verificados os parâmetros de entrada ela chamará a função `ftt_se_wrapper_receive()` que acionará a receção da mensagem a partir do nó escolhido, através da stream de saida da parte remota da tarefa.

3.2.3.7 pdrtf_get_input_parameters()

A função `pdrtf_get_input_parameters()` é responsável por fornecer os parâmetros de entrada à função da parte remota da tarefa. Esta função começa por verificar se a estrutura de controlo `task_data` é válida, se o apontador `data` para a posição de memoria onde os parâmetros serão copiados também é válida e se o tamanho dessa posição `data_size` coincide com o tamanho especificado pela função `set_pdrtf_task_sub_task()` durante a fase de mapeamento de tarefas.

Depois de verificados os parâmetros de entrada ela copiará para a posição em questão os parâmetros de entrada.

3.2.3.8 pdrtf_set_output_parameters()

A função `pdrtf_set_otput_parameters()` é responsável por guardar os parâmetros de saída produzidos pela função da parte remota da tarefa. Esta função começa por verificar se a estrutura de controlo `task_data` é válida, se o apontador para posição de memoria `data` a partir de onde os parâmetros serão copiados também é válida e se o tamanho dessa posição `data_size` coincide com o tamanho especificado pela função `set_pdrtf_task_sub_task()` durante a fase de mapeamento de tarefas.

Depois de verificados os parâmetros de entrada ela copiará dessa posição para o *buffer* interno para posteriormente serem enviados de volta para o nó local.

3.2.4 Descrição da API do módulo FTT-SE Wrapper

Devido a complexidade da interface da biblioteca do FTT-SE, decidiu-se implementar um pequeno wrapper de forma a simplificar a sua utilização. A sua implementação foi desenhada propositadamente modular de forma à sua utilidade se estender além do âmbito deste projeto. De facto pode-se constatar tal utilidade pois o FTT-SE Wrapper foi já utilizado em outro projeto relacionado com a Arrowhead Framework que também fez uso do FTT-SE.

A descrição das funções será demonstrada a seguir:

3.2.4.1 ftt_se_wrapper_start()

A função `ftt_se_wrapper_start()` é obrigatoriamente a primeira função do FTT-SE Wrapper a ser chamada, é nesta função que são executadas as inicializações internas da biblioteca FTT-SE e que tem de ser executadas em primeiro lugar, de maneira a ser possível usar as suas funcionalidades. Esta função tem como único parâmetro `char *device_name` este parâmetro representa o nome da placa de rede na qual se pretende que seja utilizado o FTT-SE.

3.2.4.2 ftt_se_wrapper_stop()

A função `ftt_se_wrapper_stop()` executa a finalização da execução do FTT-SE e não leva usa nenhum parâmetro. Se no entanto existirem `streams` abertas quando esta função for executada, de forma a evitar fugas de memória, ela retornará um erro a informar desta situação. E só retornará sucesso se no contador interno não existir registo de `streams` abertas.

3.2.4.3 stream

A estrutura de dados do FTT-SE Wrapper chamada de `stream` contem todos os parâmetros necessários para gerir e usar `streams` FTT-SE. Desta maneira através do uso de apontadores para `stream` é possível efetuar as várias operações disponíveis sem necessitar de gerir qualquer outro tipo de dados. A sua definição encontra-se escondida protegendo desta maneira o acesso inadvertido ao seu conteúdo.

3.2.4.4 ftt_se_wrapper_new_stream()

A função `ftt_se_wrapper_new_stream()` aloca memoria para uma `stream` e inicializa a com os seus parâmetros básicos. Esta operação é obrigatória pois para além da `stream` ser

usada como parâmetro em todas as restantes funções do FTT-SE Wrapper, a sua definição encontra-se escondida como proteção de acesso ao seu conteúdo, sendo apenas possível ao próprio FTT-SE Wrapper usa-la internamente. Esta função faz uso dos seguintes parâmetros:

- `stream ** stream_ptr`: Este parâmetro é um apontador para apontador de `stream` e é através dele que será devolvida a posição de memória alocada e inicializada que será utilizada pelas outras funções.
- `unsigned short msg_id`: Este parâmetro é o id que a `stream` criada usará, no FTT-SE este id é muito importante pois não existem endereços de partida e de chegada de mensagens, apenas existem produtores e consumidores de `streams`, por essa rasão, este id terá de ser único.
- `unsigned short sync_type`: Este parâmetro estabelece o tipo de sincronismo da `stream` e pode tomar os seguintes valores: 0 para síncrona, 1 para assíncrona *hard real-time*, 2 para assíncrona *soft real-time* e 3 para assíncrona *best effort*.
- `unsigned char fifo_size`: Este parâmetro é o tamanho da fila FIFO do FTT-SE para a `stream` criada, na prática esta fila age como um buffer de mensagens que ficarão à espera de ser consumidas pela aplicação através do `ftt_se_wrapper_receive()`, se a `stream` for do tipo síncrono este parâmetro será ignorado pois o FTT-SE necessita de uma fila de tamanho fixo de 3 para este tipo de `stream` e é esse o valor que é utilizado pelo FTT-SE Wrapper.

3.2.4.5 ftt_se_wrapper_close_stream()

A função `ftt_se_wrapper_close_stream()` leva como parâmetro um apontador de `streams` e tem como funcionalidade fechar a `stream` e libertar a memória para ela alocada.

3.2.4.6 ftt_se_wrapper_set_stream_parameters()

A função `ftt_se_wrapper_set_stream_parameters()` tem como funcionalidade definir as propriedades da `stream` FTT-SE, e pode ser chamada mais do que uma vez se for necessário alterar estas propriedades dinamicamente. Esta função faz uso dos seguintes parâmetros:

- `stream * stream_ptr`: Este parametro é o apontador para uma `stream` previamente criada pela função `ftt_se_wrapper_new_stream()`.
- `unsigned int msg_size`: Este parâmetro é o tamanho em bytes que a `stream` terá.
- `unsigned short mtu_size`: Este parâmetro é o tamanho em bytes do MTU em que a `stream` se dividirá.

- `unsigned short period_in_ecs`: Este parâmetro será considerado como período caso seja síncrona e MIT caso seja assíncrona.

3.2.4.7 ftt_se_wrapper_initialize_producer()

A função `ftt_se_wrapper_initialize_producer()` leva como parâmetro um apontador de `streams` e tem como funcionalidade abrir a `stream` como produtora no nó onde for chamada.

3.2.4.8 ftt_se_wrapper_initialize_consumer()

A função `ftt_se_wrapper_initialize_consumer()` leva como parâmetro um apontador de `streams` e tem como funcionalidade abrir a `stream` como consumidora no nó onde for chamada.

3.2.4.9 ftt_se_wrapper_send()

A função `ftt_se_wrapper_send()` tem como funcionalidade despoletar um envio numa determinada `stream`, antes de se poder chamar esta função é necessário usar a função `ftt_se_wrapper_initialize_producer()` de forma a definir o nó como produtor da `stream`. Esta função faz uso dos seguintes parâmetros:

- `stream * stream_ptr`: Este parametro é o apontador para uma `stream` previamente criada pela função `ftt_se_wrapper_new_stream()`.
- `void * msg_ptr`: Este parâmetro é o apontador para a posição de memória onde se encontra a menagem que se pretende transmitir.
- `unsigned int msg_size`: Este parâmetro representa o tamanho da memória da mensagem que se pretende transmitir. Se este tamanho for superior ao tamanho definido para a `stream` na função `ftt_se_wrapper_set_stream_parameters()` será retornado um código de erro indicativo dessa situação.
- `unsigned short blocking_flag`: Este parâmetro define se o envio deverá ser bloqueante "1" ou não bloqueante "0". Isto significa que se for bloqueante a execução irá bloquear até que a mensagem chegue ao outro lado.

3.2.4.10 ftt_se_wrapper_receive()

A função `ftt_se_wrapper_receive()` tem como funcionalidade obter uma mensagem da fila das mensagens recebidas de uma determinada `stream`, antes de se poder chamar esta função é necessário usar a função `ftt_se_wrapper_initialize_consumer()` de forma a definir o nó como consumidor da `stream`. Esta função faz uso dos seguintes parâmetros:

- `stream * stream_ptr`: Este parâmetro é o apontador para uma `stream` previamente criada pela função `ftt_se_wrapper_new_stream()`.
- `void * msg_ptr`: Este parâmetro é o apontador para a posição de memória onde será copiada a mensagem recebida.
- `unsigned int msg_size`: Este parâmetro representa o tamanho da memória da mensagem que se pretende receber. Se este tamanho for superior ao tamanho definido para a `stream` na função `ftt_se_wrapper_set_stream_parameters()` será retornado um código de erro indicativo dessa situação.

4 Resultados e Validação

4.1 Experiencias Realizadas

De forma a comprovar a viabilidade da utilização da PDRTF, foram executadas uma serie de experiências que comparam os tempos de execução de um conjunto de tarefas de tempo-real quando são executadas sequencialmente e em paralelo usando o modelo fork/join distribuído através da PDRTF.

Iniciaremos pela configuração dos parâmetros utilizados nas experiencias que serão posteriormente descritas.

4.1.1 Configuração das experiencias

O hardware utilizado nas experiencias foi o seguinte:

- Nó FTT-SE Master (que arbitrará a transmissão de dados na rede FTT-SE):
  - Tipo de computador: Desktop.
  - Sistema Operativo: Debian 7.8, kernel 3.2.0-4-rt-686-pae com patch PREEMPT RT 3.2.68-1+deb7u1 i686.
  - Processador: AMD Sempron 2800+, 1600 MHz.
  - RAM: 1 GB.
- Quatro nós PDRTF (que também serão nós Slave FTT-SE):
  - Tipo de computador: Laptop (modelo Laptop HP ProBook 6460b).
  - Sistema Operativo: Debian 7.8, kernel 3.2.0-4-rt-686-pae com patch PREEMPT RT 3.2.68-1+deb7u1 i686.
  - Processador: Intel Celeron B840 1895 MHz.
  - RAM: 2 GB.
- Ethernet Switch (TPLINK TL-SF1008D):
  - Velocidade: 100 Mbps.
  - Número de portas: 8.

De forma a obter o máximo nível de determinismo foi necessário configurar os seguintes parâmetros nos nós:

- Desativar funções ACPI e APM que introduzem latência ao sistema operativo.
- Desabilitar reserva de processamento do kernel Linux tempo-real para tarefas não tempo real.

4.1.2 Características das tarefas tempo-real a testar

Após devida consideração o caso estudo a ser aplicado às experiências escolhido foi o seguinte:

- As prioridades das tarefas a seguir representadas serão atribuídas com base no modelo Rate Monotonic, em que a tarefa com maior frequência e por consequência o período mais curto tem maior prioridade.
- As tarefas terão um deadline igual ao período.

Relativamente ás características da PDRTF será assumido o seguinte:

- O mínimo tamanho de EC estável é de 10ms e as streams FTT-SE usadas são assíncronas com o MIT de 2 ECs. O tempo estimado de Round-trip nestas condições será de cerca de 40ms.
- A PDRTF usa o escalonador de tempo real SCHED_FIFO de prioridades fixas, portanto de forma a simular o escalonamento Rate Monotonic será atribuída uma prioridade a tarefa com o período mais curto.
- O *cluster* de testes DPRTF dispõe de 4 nós, portanto o WCET atribuído a cada nó será ¼ do WECT sequencial ou total, que corresponde a divisão do trabalho em partes iguais pelo número de nós.

Como se pode observar na Tabela 3 este conjunto de tarefas não será escalonável em ambiente uniprocessador pelas seguintes razões:

- A tarefa 2 apesar de ter o deadline igual ao seu WCET tem prioridade inferior a tarefa 1, desta forma será interrompida não podendo cumprir o seu deadline.
- A tarefa 1 apesar de ter prioridade superior à tarefa 2 tem um WCET superior ao seu período, sendo por consequência impossível cumprir o seu deadline.

Usando a PDRTF o WCET estimado para a tarefa 1 sem interferência da tarefa 2, será a soma do round-trip FTT-SE estimado de 40ms mais o WECT distribuído pelos 4 nós do *cluster,* que é 50ms resultando em um total de 90ms (estimado). Como a tarefa 1 tem prioridade superior a tarefa 2 esta não deverá ser interrompida sendo esperado que este valor se mantenha.

Analogamente ao que foi considerado para a tarefa 2, a soma do round-trip FTT-SE estimado de 40ms mais o WECT distribuído pelos 4 nós do *cluster* que é 75ms resulta em um total de 115ms (estimado). Como a tarefa 1 tem prioridade superior a tarefa 2 esta poderá ser interrompida. Deverá se considerar o tempo de execução da tarefa 1 na determinação do

WCET da tarefa dois, desta forma será adicionando aos 115ms previstos da tarefa 2 o WCET da tarefa 1 de 90ms é esperado que o WCET total da tarefa 2 ronde os 195ms (estimado).

Tendo estes factos em consideração será expectável que este conjunto de tarefas seja escalonável utilizando a PDRTF, uma vez que estes tempos estimados se encontram abaixo do período/deadline.

	Período/ Deadline	WCET Sequencial/ Total	Round-trip FTT-SE (estimado)	WCET / Nó = WCET Sequencial /4	WCET PDRTF (estimado)
Tarefa 1	150ms	200ms	40ms (4 ECs de 10ms)	50ms	40+50=90ms
Tarefa 2	300ms	300ms	40ms (4 ECs de 10ms)	75ms	40+75=115ms 90+115=195ms

Tabela 3: Características das tarefas tempo-real a testar

4.1.3 Resultados das experiencias

De forma a simular a ocupação do Processador, foi necessário criar uma função que ocupasse o processador durante WCET estimado para as tarefas, após o desenvolvimento dessa função foram medidos os tempos de 1000 execuções de forma a comprovar o determinismo obtendo-se os resultados presentes no Gráfico 1.

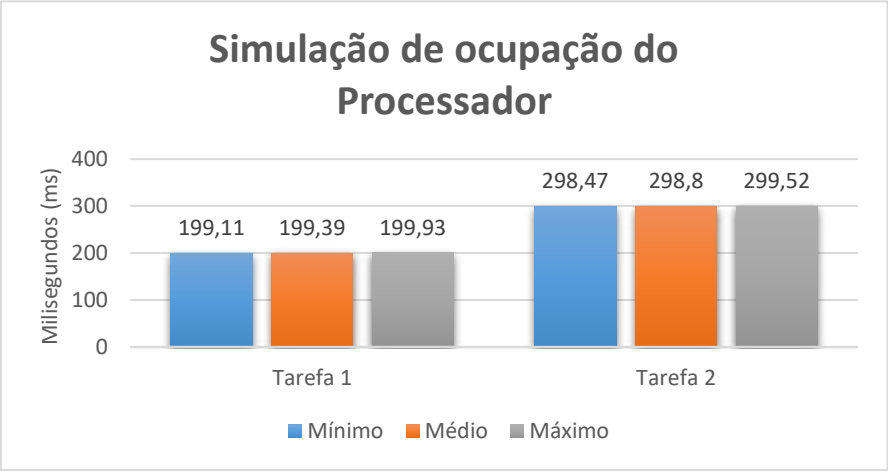

Gráfico 1: Simulação de ocupação do processador.

De forma a obter os tempos reais de roud-trip FTT-SE, foram medidos os tempos de 1000 execuções das tarefas 1 e 2 utilizando a PDRTF sem utilizar a função que simula ocupação do processador. Os resultados obtidos nesta experiencia foram os demonstrados no Gráfico 2.

Finalmente, para a obter os tempos de execução totais foram medidos os tempos de 1000 execuções das tarefas 1 e 2, utilizando a PDRTF com a função que simula ocupação do processador com ¼ do WCET sequencial por nó. Foram também medidos os tempos de 1000 execuções das tarefas 1 e 2 isoladamente para obter uma comparação com a execução das duas concorrentemente. Os resultados obtidos nesta experiencia foram os demonstrados no Gráfico 3.

Nesta secção foram demonstrados os resultados obtidos através das experiencias realizadas de forma a validar a utilidade da framework PDRTF, comparando os resultados obtidos com os resultados esperados/estimados. Conclui-se que a framework é eficiente na distribuição de tarefas tempo-real em vários nós num cluster de computadores distribuídos, permitindo assim o cumprimento de deadlines que não seriam possíveis num sistema uniprocessador.

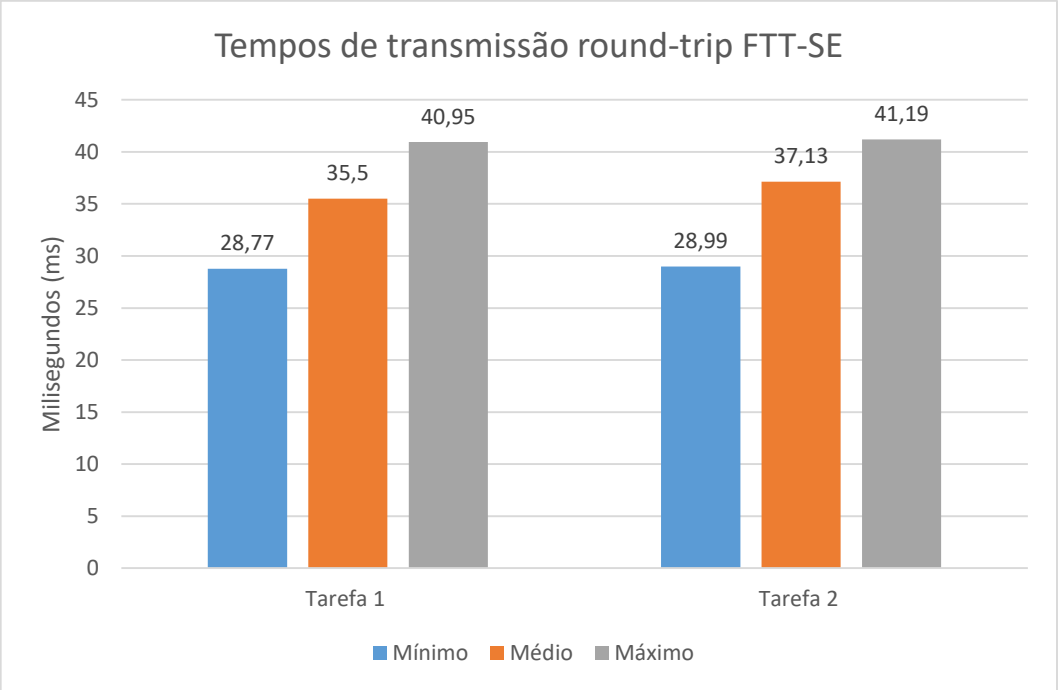

Gráfico 2: Tempos de transmissão round-trip FTT-SE.

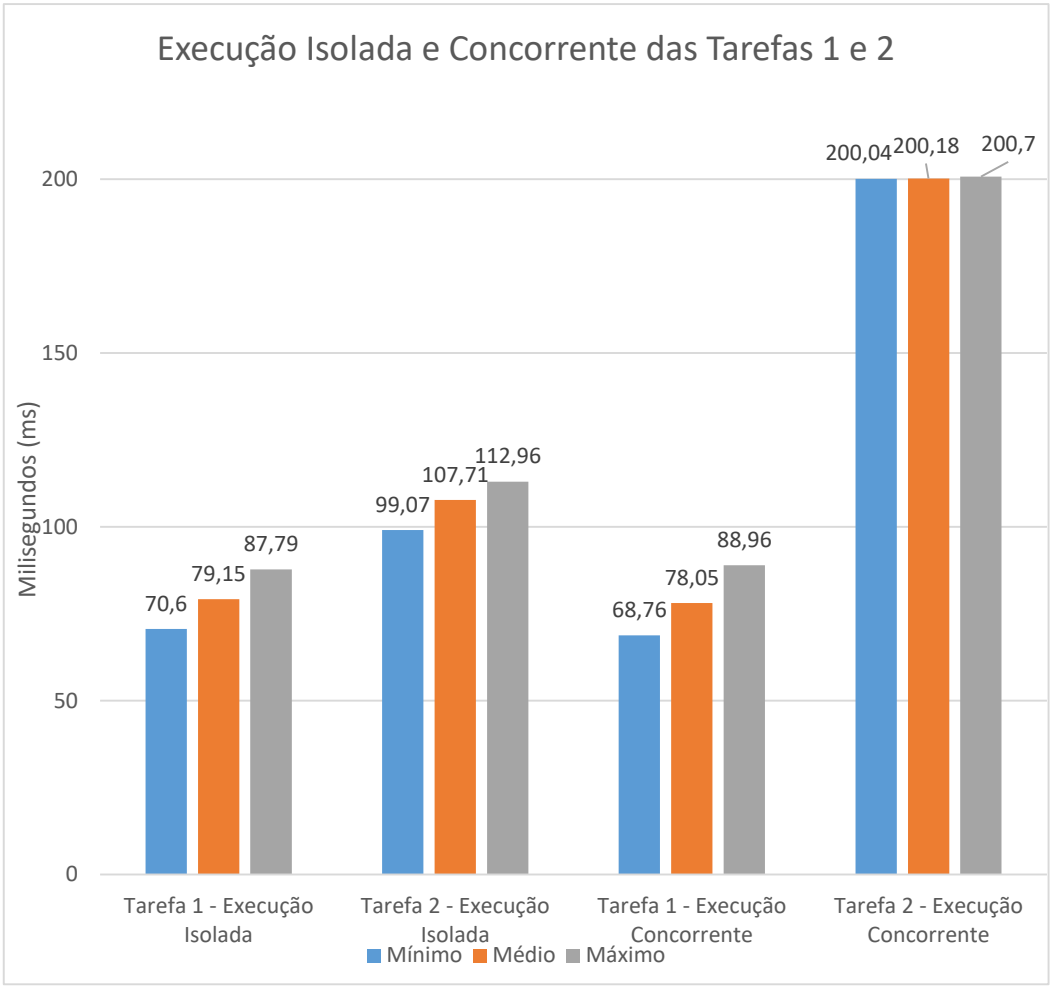

Gráfico 3: Demonstração da execução isolada e concorrente das tarefas 1 e 2.

5 Conclusões

Este trabalho demostra que é possível a implementação do paradigma de computação fork/join distribuída em sistemas de tempo-real. Este paradigma permite distribuir parte de determinadas operações que não podem ser executadas localmente num nó, dentro da deadline definida para essa operação. Tal deve-se ao facto desses nós não terem capacidade de processamento suficiente. Outra razão para distribuir a computação pode também ser a necessidade de poupar energia num nó sem fios. Assim, a framework desenvolvida permite distribuir parte da computação por outros nós do mesmo sistema que tenham recursos livres. Foi especialmente desenvolvida para ser utilizada em sistemas embebidos com fraca capacidade de processamento, a operar numa rede totalmente fechada ao exterior. A implementação é por isso muito otimizada e de baixo nível de modo a que possa cumprir deadlines acima dos 70 ms.

Este trabalho foi baseado numa implementação *open-sour*ce do protocolo de comunicação Flexible Time Trigger-Switched Ethernet (FTT-SE) em que as operações distribuídas são executas em nós Linux com o patch PREEMPT-RT, que assegura o suporte a aplicações de tempo real.

Durante o trabalho foi ainda possível detetar e resolver problemas com esta implementação, o que obviamente causou alguns atrasos, mas a versão disponível publicamente engloba agora as correções efetuadas. Foi possível, não só, interagir com os orientadores da organização, mas também com os elementos da equipa que desenvolveu a implementação do protocolo FTT-SE.

As principais dificuldades sentidas foram relacionadas com a falta de documentação do código utilizado, assim como devido à necessidade de adquirir conhecimentos em sistemas de tempo-real, especificamente no FTT-SE e na utilização de patches de tempo-real para o Linux.

5.1 Objetivos realizados

O objetivo principal deste projeto era a criação de uma framework que possibilitasse a fácil aplicação do modelo fork/ join parallel distributed a tarefas de tempo-real num *cluster* de computadores distribuídos foi considerado realizado com sucesso.

Outros objetivos inerentes a este principal que incluem:

- Devem ser criadas rotinas que permitam a criação de tarefas de tempo-real com atribuição de prioridades e que se encarreguem da respetiva gestão de periodicidade e verificação de cumprimento de deadlines.
- Integrar o protocolo FTT-SE na Framework de forma a se obter determinismo e garantias de tempo-real na transmissão de dados entre os nós distribuídos.
- Criar uma interface simplificada e fácil de usar não obstante da necessidade de alguns conhecimentos da linguagem c e de boas práticas de programação.
- Implementar a framework de forma a garantir *overheads* temporais pequenos e grande previsibilidade temporal.

Estes objetivos foram também considerados realizados com sucesso.

5.2 Outros trabalhos realizados

Durante o ano de 2016 foi também possível participar de forma ativa como membro da equipa que esteve a desenvolver o sistema de suporte a qualidade de serviços para aplicações internet das coisas em meio industrial [11] [12]. Este trabalho consistiu na elaboração de uma interface simplificada para aceder ao FTT-Se, descrita na secção 3.2.4.

5.3 Trabalho futuro

A framework implementada foi baseada numa implementação do FTT-SE em user space, com algumas limitações em relação à sua revisibilidade temporal, dado que o seu comportamento é afetado pelas rotinas to kernel, com maior prioridade. Consequentemente, a utilização de períodos muito pequenos (menores que 10 ms) no escalonamento do FTT-SE pode levar a alguma instabilidade temporal.

Como evolução futura para esta framework seria interessante a sua extensão para suportar a implementação desde paradigma no kernel do Linux, que garante uma maior robustez a previsibilidade temporal. Outras extensões futuras seria a sua implementação utilizando switches HARTES, que já implemtam as funcionalidade dos FTT-masters em hardware, assim como também a sua extensão para outras redes de tempo-real.

5.4 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.

Quero 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.

6 Bibliografia

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

[2] John A. Stankovic. “Misconceptions about real-time computing: A serious problem for next generation systems” em Computer, 21(10):10–19, October 1988. ISSN 0018-9162.

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

[4] “What is An RTOS?” [Online]. Disponível: http://www.freertos.org/about-RTOS.html

[5] “Real-Time Linux Kernel Scheduler“, [Online]. Disponível: http://www.linuxjournal.com/magazine/real-time-linux-kernel-scheduler.

[6] “RT PREEMPT HOWTO”, [Online] Disponível: https://rt.wiki.kernel.org/index.php/RT_PREEMPT_HOWTO.

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

[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] R. Garibay-Martinez, “A Framework for the Development of Parallel and Distributed Real Time Embedded Systems”, 2016.

[11] LL. Ferreira, M-Albano, QoS-as-a-Service in the local cloud, 2016.

[12] Renato Ayres, Paulo Barbosa, "QoS for High Performance IoT Systems", 2016