D 2020

U.PORTO
FEUP FACULDADE DE ENGENHARIA
UNIVERSIDADE DO PORTO

REPLICAÇÃO EM SISTEMAS DISTRIBUÍDOS UTILIZANDO A NORMA IEC 61499

ADRIANO MANUEL DE ALMEIDA SANTOS

TESE DE DOUTORAMENTO APRESENTADA
À FACULDADE DE ENGENHARIA DA UNIVERSIDADE DO PORTO EM
ENGENHARIA MECÂNICA

Replicação em Sistemas Distribuídos Utilizando a Norma IEC 61499

Adriano Manuel de Almeida Santos

Faculdade de Engenharia da Universidade do Porto

Departamento de Engenharia Mecânica

Rua Roberto Frias s/n, 4200-465 Porto
Portugal

Porto, maio de 2020

U.PORTO

FEUP FACULDADE DE ENGENHARIA
UNIVERSIDADE DO PORTO

Replicação em Sistemas Distribuídos Utilizando a Norma IEC 61499

Adriano Manuel de Almeida Santos

Mestre em Manutenção Industrial
Faculdade de Engenharia da Universidade do Porto

Tese submetida para a obtenção do grau de
Doutor em Engenharia Mecânica

Programa Doutoral em Engenharia Mecânica

Tese realizada sob orientação de

Professor Doutor Mário Jorge Rodrigues de Sousa
Departamento de Engenharia Eletrotécnica e de Computadores da FEUP

e coorientação de

Professor Doutor António José Pessoa de Magalhães
Departamento de Engenharia Mecânica da FEUP

Professor Doutor António José de Sousa Ferreira da Silva
Departamento de Engenharia Mecânica do ISEP (PPorto)

Porto, junho de 2020

Presidente do Júri

Doutor Armando Carlos Figueiredo Coelho de Oliveira, Professor Catedrático da Faculdade de Engenharia da Universidade do Porto;

Vogais

Doutor Jaime Francisco Cruz Fonseca, Professor Associado com Agregação do Departamento de Eletrónica Industrial da Escola de Engenharia da Universidade do Minho (Arguente);

Doutor José Mendes Machado, Professor Auxiliar do Departamento de Engenharia Mecânica da Universidade do Minho (Arguente);

Doutor Francisco Jorge Teixeira de Freitas, Professor Associado do Departamento de Engenharia Mecânica da Faculdade de Engenharia da Universidade do Porto (Arguente);

Doutor Mário Jorge Rodrigues de Sousa, Professor Auxiliar do Departamento de Engenharia Eletrotécnica e de Computadores da Faculdade de Engenharia da Universidade do Porto (Orientador).

Porto, maio de 2020

"Nada dura e, todavia, nada acaba também.
E nada acaba precisamente porque nada dura."

In "Mancha Humana"
Philip Roth

Para a Sandra e
para os meus Filhos,

Resumo

A evolução dos computadores ocorrida nas últimas décadas conduziu à proliferação de sistemas de controlo que se encontram disseminados por um grande domínio de aplicações quer ao nível dos sistemas de automação industrial e do controlo de processos quer ao nível dos sistemas automotivos e dos ambientes domésticos, etc. A dependência destes sistemas tornar-nos-á cada vez mais vulneráveis à ocorrência de falhas que poderão tornar-se catastróficas, aquando da ocorrência de erros, quer ao nível do domínio dos valores quer do domínio temporal. Por outro lado, as aplicações, cada vez mais complexas e com necessidade de requisitos de tolerância a falhas bem como de tempo real, encontram-se distribuídas e suportadas por minicomputadores ou microcomputadores que normalmente não suportam mecanismos de tolerância a falhas e, como tal, apresentam dificuldades de gestão do determinismo dos componentes replicados.

O principal objetivo de investigação desta tese consiste no desenvolvimento de uma *framework* transparente e de uso genérico capaz de suportar a replicação em ambiente distribuído com recurso à *Open Source PLC Framework for Industrial Automation & Control – Eclipse 4diac™*. Desenvolvida em concordância com a arquitetura IEC 61499 e suportada por componentes de baixo custo, a *framework* possui como fim último o desenvolvimento da aplicação focando-nos nos requisitos de controlo do sistema, abstraindo-nos dos mecanismos de replicação e de distribuição.

A *framework* de desenvolvimento de aplicações tolerantes a falhas e de tempo real proposta nesta tese centra-se no uso das linguagens de programação referidas na IEC 61499, bem como na utilização dos objetos standard disponibilizados pelo repositório do *Eclipse 4diac*. Esta disponibiliza, para além dos objetos do repositório, um novo conjunto de objetos genéricos possíveis de utilização no desenvolvimento de sistemas determinísticos. Os objetos desenvolvidos permitem definir os mecanismos de interação entre sistemas distribuídos e sistemas replicados.

O suporte de comunicações entre réplicas, quer sejam ao nível do *software* quer do *hardware*, é realizado por um par de blocos função standards e por um conjunto de protolocos *atomic multicast* capazes de garantir a tolerância a falhas em tempo real trabalhando sobre uma rede Ethernet. Assim sendo são usadas mensagens temporizadas como garantia do sincronismo das réplicas suportadas por um protocolo de sincronização dos relógios, do tipo NTP, da rede local.

Foi desenvolvido um protótipo de testes que permitisse testar o determinismo dos blocos de comunicação standards (*Publish/Subscribe*) em aplicações distribuídas e tolerantes a falhas, bem como a implementação do modelo de determinismo genérico e transparente proposto.

Palavras-chave: *Blocos Função, Confiabilidade, Eclipse 4diac, IEC 61499, Redundância, Replicação, Sistemas Distribuídos, Tolerância a Falhas.*

Abstract

The evolution of computers in recent decades has led to the proliferation of control systems are used in a wide range of application domains, such as industrial automation systems and process control as well as automotive systems and home environments, etc. Dependence on these systems will make us increasingly vulnerable to the occurrence of failures that could become catastrophic when errors occur, either in the value domain or in the time domain. On the other hand, increasingly complex applications that require fault tolerance as well as real-time requirements are distributed and supported by minicomputers or microcomputers that typically do not support fault tolerance mechanisms and as such, present difficulties in managing the determinism of replicated components.

The main research objective of this thesis is to develop a transparent and generic *framework* capable of supporting replication in distributed environment using the *Open Source PLC Framework for Industrial Automation & Control - Eclipse 4diac™*. Developed in accordance with the IEC 61499 architecture and supported by low cost components, the framework has the ultimate purpose of application development focusing on the system control requirements and abstracting from the replication and distribution mechanisms.

The real-time, fault-tolerant application development *framework* proposed in this thesis focuses on the use of the programming languages referred to the IEC 61499 as well as the use of the standard objects provided by the *4diac* repository. In addition to the repository objects, it makes available a new set of generic objects that can be used in the development of deterministic systems. The developed objects allow defining the mechanisms of interaction between distributed systems and replicated systems.

Support for communication between replicas, both *software* and *hardware*, is provided by a pair of standard function blocks and a set of *atomic multicast* protocols capable of real-time fault tolerance in an Ethernet network. Timed messages are used to ensure replica timing supported by a local network clock synchronization protocol.

A prototype was also developed to assess the determinism of standard communication blocks (*Publish/Subscribe*) in distributed and fault tolerant applications, as well as the implementation of the proposed generic and transparent determinism approach.

Keywords: *Function Blocks, Reliability, Eclipse 4diac, IEC 61499, Replication, Distributed Systems, Fault Tolerance.*

Agradecimentos

Em primeiro lugar, gostaria de agradecer ao meu orientador, Professor Doutor Mário Jorge Rodrigues de Sousa, pela sua dedicação, apoio e pelo inestimável incentivo, sem o qual esta etapa nunca teria chegado a bom porto.

Aos meus dois coorientadores Professor Doutor António José Pessoa de Magalhães e ao Professor Doutor António José de Sousa Ferreira da Silva pela acutilância, incentivo e inestimável empenho e dedicação

Agradeço também ao Instituto Superior de Engenharia do Porto (ISEP) e ao Politécnico do Porto (PPorto) pelos apoios concedidos e incentivo, importantíssimos para a realização deste trabalho.

Não poderia deixar de agradecer ao Centro de Investigação e Desenvolvimento em Engenharia Mecânica (CIDEM) e ao Departamento de Engenharia Mecânica (DEM) do ISEP, pelo apoio e por todos os incentivos recebidos.

Um agradecimento especial à minha família por estar sempre presente.

Por último, e não menos importantes, todos os amigos e todas as pessoas que de algum modo contribuíram, me incentivaram e apoiaram ao longo deste projeto.

Porto, maio 2020

Adriano Manuel de Almeida Santos

Índice

Índice de Figuras

Índice de Tabelas

Listas de Abreviaturas e Símbolos

A	- Disponibilidade (*Availability*)
AADL	- Architecture Analysis and Design Language
ACK	- Acknowledgement
ADU	- Application Data Unit
ANSI	- American National Standards Institute
AOP	- Aspect Oriented Paradigm
ASCII	- American Standard Code for Information Interchange
ASIC	- Application Specific Integrated Circuit
AT	- Acceptance Test
BA	- Behavioral Annex
BCRT	- Best Case Response Time
BFB	- Basic Function Block
BFT	- Byzantine Fault-Tolerant
BHS	- Baggage Handling Systems
C	- Linguagem de programação
C++	- Linguagem de programação
CAN	- Controller Area Network
CAN	- Campus Area Network
CFB	- Composit Function Block
CIDEM	- Centro de Investigação e Desenvolvimento em Engenharia Mecânica
CIFB	- Communication Interface Function Blocks
CLP	- Controlador Lógico Programável
CNF	- Confirm event
COTS	- Commercial Off-The-Self
CPU	- Central Processing Unit
CRC	- Cyclic Redundancy Chec
Debian	- Sistema operativo livre com base no Kernel Linux
DEM	- Departamento de Engenharia Mecânica
DIPMCS	- Distributed Industrial-Process Measurement and Control Systems
DNS	- Domain Name System
DRA	- Data Re-expression Algorithm
DSM	- Distributed Shared Memory
DSPN	- Deterministic and Stochastic Petri Nets
ECC	- Execution Control Chart
EIP	- Extended Internet Protocol
EMB_RES	- Embedded Resource
EPL	- Ethernet Private Line

FB	- Function Block
FB’	- Function Block Replication
FBD	- Function Block Diagram
FBDK	- Function Block Development Kit
FEUP	- Faculdade de Engenharia da Universidade do Porto
FIFO	- Frist-In-First-Out
FIPA-ACL	- Foundation for Intelligent Physical Agents - Agent Communication Language
FT	- Fault Tolerance
GPS	- Global Positioning System
GUI	- Graphical User Interface
HMI	- Interface Homem-Máquina
HRTS	- Hard Real-Time Subsystem
HS	- Handling Systems
HTM	- Hardware Transactional Memory
I/O	- Sinais de entrada e saída (Input/Output)
ICN	- Interchannel Communication Network
IDE	- Integrated Development Environment
IEC	- International Electrotechnical Commission
IL	- Instruction List
INIT	- Service Initialization (event input)
INITO	- Initialization Confirmation (event output)
IP	- Internet Protocol
IPMCS	- Industrial Process Measurement and Control System
IPP	- Instituto Politécnico do Porto
ISEP	- Instituto Superior de Engenharia do Porto
JAVA	- Linguagem de programação orientada a objetos.
Jitter	- Desvio ou erro nas leituras de relógio (NTP)
LAN	- Local Area Network
LD	- Ladder
Linux	- Sistema operativo (Kernel Linux)
LREAL	- Long Real, tipo de variável
MAN	- Metropolitan Area Network
MBAP	- Modbus Application Protocol
Mbps	- Megabit por segundo
MTBF	- Mean Time Between Failure
MTTF	- Mean Time To Failure
MTTR	- Mean Time To Repair
MV	- Majority Voter
NAC	- Network Attachment Controller
NIC	- Network Interface Card
NMR	- N-Modular Redundancy
NSCP	- N Self-Checking Programming
NTP	- Network Time Protocol
NVP	- N-Version Programming
Offset	- Diferença de tempo entre dois relógios
OS	- Operated System
OS X	- Operating System Ten (OS Ten), Mac
OSCAT	- Open Source Community for Automation Technology
OSI	- Open System Interconnection

PC	- Personal Computer
PDU	- Protocol Data Unit
PFD	- Probability of Failure on Demand
PFH	- Probability of Failure per Hour
PLC	- Programmable Logic Controller
PMI	- Performance Monitoring Infrastructure
Ppm	- Partes por milhão
PUBLISH	- Publicador
PTP	- Precision Time Protocol
QMR	- Quintuple Modular Redundancy (5-MR)
RAM	- Random Access Memory
Raspbian	- Variante do OS Debian baseada no ARM, usado no Raspberry Pi
RBD	- Reliability Block Diagram
RcB	- Recovery Blocks
RD	- Output data types
REQ	- Service Request
RMT_DEV	- Remotely Managed Type
RMT_FRAME	- Remotely Managed Window
RMT_RES	- Remote Resource
RPi	- Raspberry Pi
RRF	- Risk Reduction Factor
RTE	- RunTime Environment
RTR	- Recovery Time Requirement
RTT	- Recovery Time-Tiered
RTU	- Remote Terminal Unit
SD	- Input data types
SEU	- Single Event Upset
SFC	- Structured Function Chart
SIMD	- Single Instruction, Multiple Data
SIFB	- Service Interface Function Block
SIL	- Safety Integrity Level
SIS	- Safety Instrumented System
ST	- Structured Text
SUBSCRIBE	- Subscritor
TAI	- Temps Atómic International
TEEs	- Trusted Execution Environments
TCP	- Transmission Control Protocol
TMR	- Triple Modular Redundancy
TPCDC	- Tiered Placement Constrained Decreasing *with cold standby*
TR	- Tempo real
TRTI	- Tiered Recovery-Time Constraint Increasing
UDP	- User Datagram Protocol
UML	- Unified Modeling Language
UTC	- Universal Time Coordinated
V & V	- Verificação e validação
W	- Token walk-time
WAN	- Wide Area Network
WCET	- Worst-Case Execution Time
WCRT	- Worst-Case Response Time

WWW	- World Wide Web
XPA	- eXtended Performance Architecture
λ	- Taxa de falhas ou taxa de avarias
ρ	- Taxa de impulso
μ	- Taxa de manutenibilidade
α_v	- Exatidão
δ_m	- Transmissão
F(t)	- Função de repartição (probabilidade de avaria até t)
Fs	- Probabilidade de avaria até ao tempo t do sistema
g	- Granularidade taxa de impulsos (ρ)
k	- Número de erros na transmissão
M(t)	- Manutenibilidade
ns, nsec	- Nanossegundos, Nanoseconds
p	- Processo
R(t)	- Fiabilidade ou Confiabilidade
Rs	- Fiabilidade do sistema
s, sec	- Segundos, Seconds
tsa	- Timestamp to availability
tsr	- Timestamp to receiving
vc_p	- Relógio virtual do processo
π_v	- Precisão

Capítulo 1

Introdução

A evolução tecnológica ocorrida nos últimos anos conduziu à proliferação de sistemas computorizados quer ao nível industrial, base de suporte a tecnologias de controlo de sistemas (controlo de voo, centrais nucleares, etc.), quer ao nível do consumo e utilização doméstico que, de uma forma mais discreta (sistemas embebidos) se encontram presentes em diversos eletrodomésticos e, de uma forma mais visível, nos tradicionais computadores de uso diário. Esta proliferação deixa-nos, cada vez mais, dependentes dos sistemas computadorizados tornando-nos mais vulneráveis à ocorrência de falhas quer ao nível do controlo de sistemas domésticos quer ao nível dos sistemas industriais. As falhas ocorridas no sistema de controlo de um reator duma central nuclear terão consequências mais graves do que as ocorridas em sistemas domésticos. O reator quando sujeito a uma falha poderá pôr em perigo milhares de pessoas e afetar gravemente o ecossistema. Por outro lado, uma falha ocorrida num sistema embebido de uma máquina de lavar ou de uma torradeira, mantendo o equipamento com energia, poderá também, a um nível mais reduzido, pôr em risco pessoas e bens. Evidentemente que se em consequência da falha do sistema, deflagrar um incêndio, a habitação em causa e as contíguas poderão correr grandes perigos.

Do ponto de vista da confiança, sempre que adquirimos um bem depositamos confiança no funcionamento do mesmo quer ao nível da realização das tarefas, para que são concebidos, quer ao nível da segurança. Assim, ao adquirir um eletrodoméstico, com um sistema computacional embebido, esperamos que este cumpra corretamente a função para que foi construído, durante um determinado período de tempo bem definido. Então, do ponto de vista do utilizador, a confiança no funcionamento de um dado bem é a confiança que este deposita no sistema para prestar um dado serviço tendo em conta não só a produção de valores lógicos corretos (domínio do valor) mas também na capacidade para os disponibilizar dentro de um determinado período de tempo (domínio do tempo).

Na área da confiança no funcionamento de um sistema computacional a ideia subjacente é a de que o sistema se comporte de acordo com a sua especificação, perante os inúmeros problemas que podem ocorrer (acidentes naturais, erros de *software*, de *hardware*, etc.). Na verdade, o que esperamos dos sistemas computacionais é que estes funcionem corretamente pelo que, não basta usar boas técnicas de engenharia, para que o computador de controlo de um avião se mantenha em funcionamento. A tolerância a estes eventos passa por colocar a bordo vários computadores (redundantes) garantindo que a falha do sistema só ocorrerá se todos os computadores falharem – mecanismo clássico de tolerância à falha. Isto quer dizer, garantir a integridade e a disponibilidade do sistema, constituído por diversos equipamentos interligados através de uma rede de comunicações, mesmo que alguns desses equipamentos apresentem falhas.

O objetivo deste texto centra-se na apresentação e definição de alguns conceitos relacionados com os atributos de confiança no funcionamento de sistemas computacionais. Desta forma pretende-se ainda expor alguns dos mecanismos clássicos que permitem resolver os problemas de confiança no funcionamento, nomeadamente as técnicas de tolerância à falha aplicadas a sistemas distribuídos. Encontrando-se cada vez mais presentes nas nossas ações, os sistemas computacionais interligam-nos aos mais diversos locais e juntamente com estas interligações é semeada a incerteza de se obter um resultado seguro e confiável. As redes de comunicação, essenciais para a interconexão dos sistemas estão também elas sujeitas a falhas e à perda de dados quer devido a erros de transmissão, rutura das linhas de comunicação que unem os vários pontos da rede, ou mesmo pelo colapso esporádico, ou mesmo deliberado, de um ponto/computador preponderante na cadeia de transmissão. Não se pode esperar que ao interligarmos computadores entre si se obtenha um sistema confiável sem que para isso se adotem medidas de tolerância a falhas de processamento e/ou de comunicação. Por outro lado, e devido às diferenças dos relógios internos dos computadores, causadas essencialmente por ligeiros desvios da frequência de pulsação, a informação não só não será recebida no instante esperado como também poderá ser disponibilizada tardiamente levando a que os processos trabalhem com dados errados, produzindo resultados errados.

A utilização de protocolos de comunicação específicos obviou estes problemas garantindo que a informação disponibilizada pelos processos chegaria a todos os componentes da rede na mesma ordem em que foi enviada. Esta metodologia resolveu em parte o problema da transmissão da informação entre processos deixando, no entanto, em aberto as falhas de sistema quer ao nível do *software*, quer do *hardware*. A utilização de protocolos de tolerância a falhas baseada na replicação quer de *software*, quer de *hardware* e do mascaramento dos componentes em falha reduziu, consideravelmente, a falta de confiança no funcionamento dos sistemas.

Industrialmente, o controlo do processo passa por aplicações que se baseiam em Controladores Lógicos Programáveis (PLCs – *Programmable Logic Controllers*). Estes são usualmente programados usando linguagens padronizadas pela norma IEC 61131-3 (IEC 61131, 2013). Por outro lado, com a proliferação das redes de comunicação no domínio industrial, os PLCs têm sido interligados entre si resultando em aplicações de controlo distribuído em que a maioria delas são combinadas e complementadas com requisitos de tempo real, segurança e fiabilidade. No entanto, a semântica do IEC 61131 e as linguagens de programação e sua execução não se apresentam como uma boa prática para a adoção dos novos requisitos utilizados nas aplicações de automação flexível e de controlo distribuído. Foi principalmente por este motivo que a Comissão Eletrotécnica Internacional (IEC – *International Electrotechnical Commission*) desenvolveu em 2005 a norma IEC 61499-1 tendo como intuito facilitar a abordagem orientada ao desenvolvimento de sistemas de medição e controle industrial em processos distribuídos (DIPMCS – *Distributed Industrial Process Measurement and Control Systems*). Esta nova norma, reeditada em 2012 (IEC 61499, 2012), propõe a utilização de blocos função, como base de construção, para o desenvolvimento de módulos reutilizáveis para *software* de controlo de forma independente. Cada bloco de função é uma unidade funcional de *software* que encapsula os dados locais e o comportamento algorítmico dentro de uma interface de eventos/dados onde as operações, dentro de um bloco de função, são controladas por uma máquina de estados orientada a eventos, ou seja, são definidos um novo padrão e uma nova interface para o desenvolvimento e o controlo de aplicações distribuídas.

Devido à natureza das aplicações de controlo distribuído, muitas e novas questões devem ser tidas em consideração. A maioria destas surge devido ao novo modelo de erro e aos modos de falha do *hardware* onde a aplicação distribuída está a ser executada. Outras são devidas à natureza distribuída do *hardware* que permite que novas capacidades possam ser exploradas. Destas salienta-se o aumento de fiabilidade dos sistemas, pelo mascaramento das falhas e pela introdução de tolerância a falhas na arquitetura das aplicações. A tolerância a falhas é geralmente

realizada pela replicação dos componentes da aplicação. Esta abordagem garante que se um dos elementos replicados falhar, e as restantes réplicas continuarem em função, o sistema será capaz de mascarar a existência da réplica em falha perante a restante aplicação.

1.1 Motivação

Nos últimos anos foram realizados vários trabalhos envolvendo a tolerância a falhas em sistemas distribuídos dos quais resultaram várias abordagens. Utilizando mensagens temporizadas, tabelas de escalonamento, tempo real, etc., todos se dedicaram a sistemas de computação distribuída com linguagens de programação não preconizadas na norma IEC 61499. Embora a abordagem preconizada pela IEC 61499 não seja nova, utilização de modelos gráficos baseados, por exemplo, em *LabVIEW* (Johnson, 2006) e o *Simulink* (Simulink, 2019) a norma IEC 61499 oferece mais vantagens práticas do que estas abordagens bem como a possibilidade de utilização de outras metodologias no desenvolvimento de *software* de controlo industrial.

Com a publicação desta nova norma e, com a possibilidade de desenvolver sistemas distribuídos, novas funcionalidades, que até então não se encontravam disponíveis na IEC 61131, foram disponibilizadas com vista a responder aos problemas associados aos sistemas de controlo distribuído. Assim, podemos contar com níveis de abstração diferentes, modelação intuitiva com diagramas de blocos onde o comportamento interno de cada bloco pode ser definido por digramas de estado e suporte para a reutilização de algoritmos, isto é, algoritmos antigos escritos nas línguas de PLCs segundo a IEC 61131 podem agora ser encapsulados em IEC 61499, blocos de função, garantindo a sua reutilização em novos projetos e uma nova abordagem, ao nível do sistema, para a conceção de sistemas distribuídos preservando a vista global do sistema ao longo de toda a fase do desenvolvimento.

Neste sentido pretende-se com esta tese colmatar alguns problemas que não se encontram definidos/resolvidos na norma, nomeadamente, a tolerância a falhas de aplicações distribuídas utilizando componentes de uso genérico vulgarmente conhecidos como PLCs e a comunicação entre as partes do sistema distribuído. Surge assim, a oportunidade de definir ferramentas transparentes de tolerância a falhas que permitam ao programador, para além das funcionalidades já enunciadas, concentrar-se nas especificações e no comando do sistema, deixando para mais tarde a distribuição e a replicação do mesmo.

O foco motivacional para o desenvolvimento desta tese, e justificação do modelo utilizado, prende-se com a vontade que o ser humano sempre demonstrou de conhecer novos mundos. Se numa fase inicial percorria longas distância caminhando, depressa encontrou formas mais eficientes e mais rápidas para viajar, o cavalo e o barco. Com a evolução dos tempos foram surgindo outras alternativas de locomoção (o comboio e o automóvel) que para além de serem um transporte mais comodo, permitia também transportar grandes quantidades de bagagem. Nos inícios do século XX surge o avião e com ele uma nova forma de viajar.

Com o desenvolvimento da tecnologia aeronáutica foi possível construir aviões de maior porte que podiam percorrer distâncias maiores, mais rapidamente e transportar mais passageiros e carga. A eleição do avião como meio ideal de transporte ocorreu assim que foi possível fazer viagens intercontinentais tornando-se inevitável o aumentando do número de pessoas e carga a transportar. Associado a esta procura foram surgindo novos destinos e novas companhias que partilhavam os mesmos espaços e como tal surgiram novas necessidades, a gestão de passageiros e da carga. Se por um lado a gestão de passageiros se mostrou mais ou menos simples a manipulação de bagagens requereu mais atenção (BHS – *Baggage Handling Systems*).

Com efeito já muito foi feito no que se refere à dentificação e à resolução de problemas que podem surgir durante o processo de embarque de uma mala pelo que se espera, que o tempo de tratamento de bagagens seja menos de 30 minutos (Calleam, 2008). Esporadicamente haverá a

necessidade de efetuar tratamento individualizado da bagagem (por ex. instrumentos musicais de grandes dimensões, pranchas de *surf*, malas com produtos suspeitos, etc.) que em muitas situações poderá traduzir-se no aumento dos tempos de embarque e no consequente aumento dos custos. Para este aumento de custos contribuirá não só o aumento do tempo de permanência da aeronave em pista, mas também possíveis compensações a pagar a passageiros insatisfeitos e à perda de fidelidade do cliente (Vickers, 1998). Outros fatores como a não padronização dos processos (Black, 2010) e as ligações entre voos poderão contribuir para o aumento dos custos. A título de exemplo podemos citar o caso do terminal 2 do aeroporto de Munique onde o tempo mínimo requerido entre ligação é de 30 minutos (Siemens, 2011). Isto é, o tempo mínimo exigido para que se possa assegurar o embarque de passageiros e respetivas bagagens.

Por outro lado, e no que se refere ao controlo, os sistemas de manipulação de bagagens são, tradicionalmente, implementados recorrendo a controladores lógicos programáveis (PLCs) supervisionados por computadores ou por outros PLCs. Estes são, normalmente, comandados a partir de uma sala de controlo, sistema centralizado, ou por uma rede de computadores ou PLCs com responsabilidades específicas na alocação, comando e comunicação com os PLCs supervisionados (Vickers, 1998). Apesar desta prática de controlo ser a adotada em inúmeros aeroportos esta não se tem mostrado a melhor abordagem. Com efeito a centralização do controlo num único PC (*Personal Computer*) ou PLC não será a melhor abordagem dado que os sistemas de transporte possuem uma natureza modular, com elevado número de elementos a controlar, e grande dispersão (Hall, 2007). Foi com base na modularidade dos sistemas de transporte que vários pesquisadores desenvolveram aplicações de controlo distribuído suportadas quer por controladores embebidos (Hayslip, 2006) quer por agentes múltiplos (Hallenborg, 2006; Marik, 2005; Vrba, 2003) suportadas por aplicações desenvolvidas em JAVA e com agentes de comunicação FIPA-ACL (*Foundation for Intelligent Physical Agents – Agent Communication Language*) (FIPA, 2018). Com o desenvolvimento da IEC 61499 o controlo distribuído dos sistemas de transporte (HS – *Handling Systems*) poderá assumir uma nova fase de desenvolvimento pelo que o controlo destes sistemas poderá ser, efetivamente, distribuído e suportado por PLCs desde que, estes adotem uma programação tendo em conta as especificações desta nova norma, a IEC 61499.

Em termos construtivos os sistemas de manipulação de bagagens (BHS) podem-se apresentar segundo duas configurações distintas: transporte de bagagens em carros (*carts*) ou em transportadores convencionais (*conveyors*). Naturalmente que cada um destes sistemas apresentará diferentes problemas de funcionamento e possíveis falhas que, de modo semelhante, contribuem para atrasos na distribuição das bagagens e para o consequente aumento dos custos de utilização dos aeroportos. O aumento destes custos deve-se a choques entre carros com queda de bagagem, projeção de bagagens (devido a elevadas velocidades em curva), perda de conteúdos, etc., sistemas baseados em *carts* (Calleam, 2008), e problemas relacionados com a prisão de alças da bagagem e sobreposições (que provocam erros de leitura dos códigos de barras), choques de bagagens, etc., sistemas baseados em *conveyors* (Vickers, 1998), ou ainda devido a problemas nos desvios mecânicos, linhas congestionadas, etc. Se por um lado alguns destes problemas já se encontram resolvidos, quer seja pela diminuição de velocidade dos sistemas quer pelo espaçamento entre carros ou entre bagagens ou pelo reencaminhamento para outros pontos de leitura, outros permanecerão latentes surgindo esporadicamente e aleatoriamente durante o seu funcionamento. Fisicamente, e no que se refere a *conveyors*, os sistemas de transporte (HS) são normalmente constituídos por elementos modulares que, interligados pelos topos, distribuem as bagagens pelos diversos pontos de recolha, transporte de e para as aeronaves. Estas secções podem encontrar-se ligadas, pelo topo, ao *conveyor* seguinte ou a ligações de junção e de divergência conforme o esquema apresentado na Figura 1-1.

Cada secção do sistema de manipulação é considerada com um elemento modular que interage com os elementos que se encontram na sua vizinhança e como tal cada módulo poderá

ser representado por um elemento de *software*, bloco função (FB), que contemplará o tratamento de sinais provenientes dos sensores de entrada e de saída, dos atuadores, da plataforma computacional e dos canais de comunicação bem como do *software* de controlo.

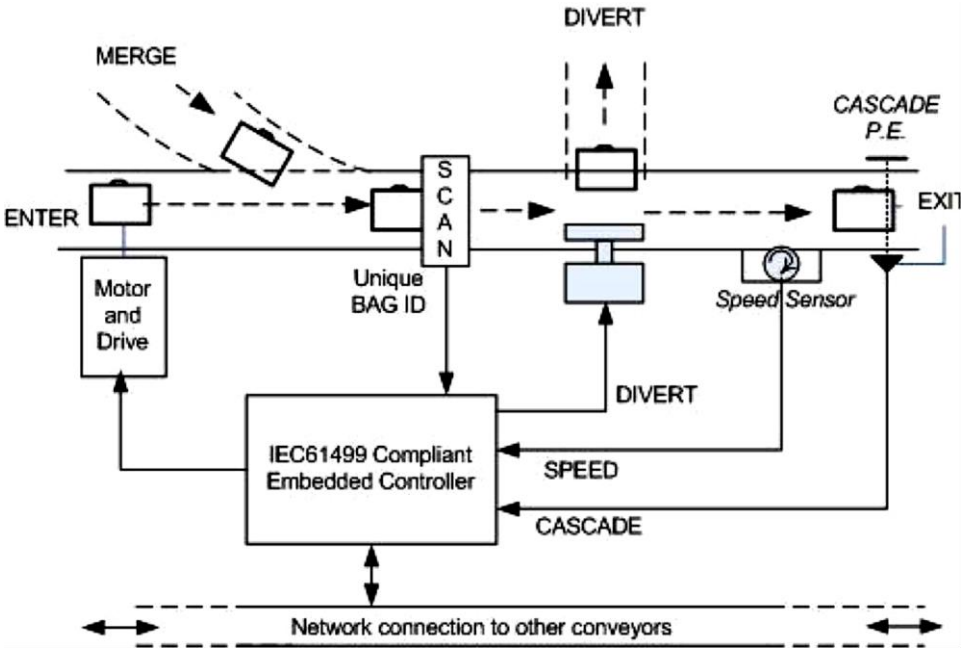


Figura 1-1. Secção do sistema de manipulação (Black, 2010)

Cada FB informará o seu vizinho do seu estado garantindo um fluxo constante de bagagens ao longo do sistema. Estes serão também responsáveis pela deteção de falhas na leitura do código de identificação da bagagem reencaminhando a bagagem assinalada com a falha de leitura para um caminho alternativo, secção redundante utilizada em caso de falha de leitura (Ribeiro *et al.*, 2008; Tambe, 2008). Por outro lado, cada secção do sistema de manipulação de bagagens deve controlar todos os movimentos das bagagens dentro da sua área de atuação tomando todas as decisões necessárias para a sua gestão, receção e entrega permitindo deste modo a execução do controlo em ambiente distribuído. Assim, na passagem de bagagem de um sector para outro, C1 para C2 (Figura 1-2), o sector que possui o controlo deve informar o sector a jusante removendo essa informação do sector a montante e do sector de controlo ativo, ou seja, transferência do controlo da bagagem para o sector seguinte. Este, por sua vez, ao assumir o controlo poderá enviar informações para a secção C3 alertando-o da chegada de bagagem, isto é, colocando-o em prontidão para a receção de bagagem.

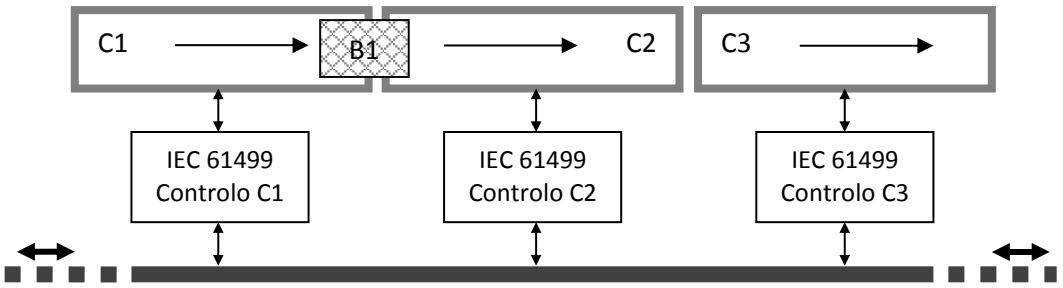


Figura 1-2. Controlo e transferência de bagagens

O controlo deve ainda possuir capacidades para alterar dinamicamente as rotas de movimentação das bagagens caso alguma das secções se encontre em falha (falha nos sensores, atuadores, controlo de velocidade, etc.). A alteração da rota será realizada em função da posição

da bagagem tendo em conta os sinais dos sensores, do posicionamento da bagagem e das possíveis divergências existentes na vizinhança da secção em falha. Na Figura 1-3 apresenta-se um esquema da simulação do sistema de tratamento de bagagens num aeroporto. O verde define o caminho normal de funcionamento do sistema, enquanto o vermelho representa o possível percurso das bagagens em caso de falha dos leitores de códigos de barras, do Rx, etc.

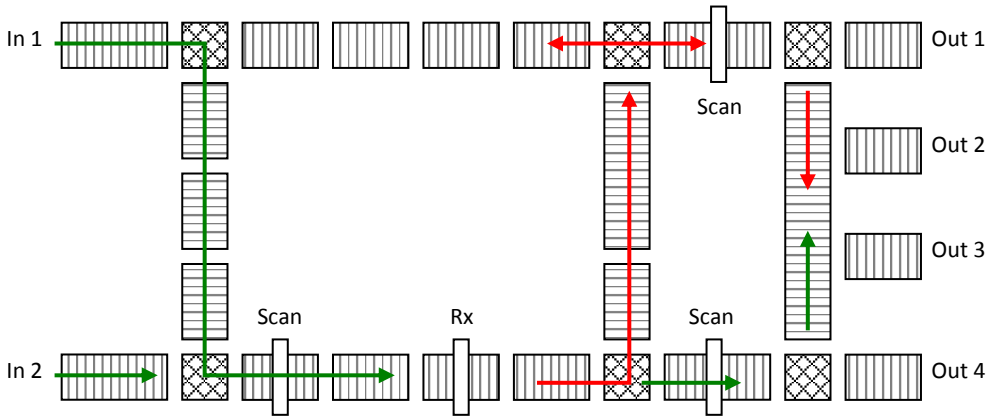


Figura 1-3. Simulador do sistema de manipulação de bagagens (Verde – principal; vermelho – alternativo)

A eficiência do sistema de controlo dependerá da sua capacidade de tolerar falhas ou da sua capacidade de resposta a situações imprevistas. Esta maior ou menor eficiência do sistema traduz-se na forma como este lida com a ocorrência de situações excecionais minimizando as consequências da falha. Nesta situação a descentralização do controlo poderá apresentar-se como uma solução vantajosa se cada elemento de *software* (IEC 61499 controlo) possuir capacidades, de forma autónoma, para lidar com a falha ou situação imprevista. Cada elemento de *software* controlará uma secção do sistema de manipulação. Assim, torna-se importante que os restantes elementos possam colmatar a falha de um dado elemento sem que o desempenho do sistema seja posto em causa. Várias poderão ser as metodologias utilizadas na tolerância a falhas em sistemas distribuídos das quais se salientam a replicação ativa (Schneider, 1990), a passiva (Budhiraja *et al.*, 1993) e a semi-ativa (Powell, 2001).

1.2 Objetivos da investigação

Considerando o presente contexto, o objetivo fundamental desta Tese de Doutoramento consiste no desenvolvimento de uma *framework* genérica e transparente para o controlo de sistemas distribuídos, baseada na arquitetura IEC 61499, tendo como fim último a replicação de sistemas. Para a percussão destes objetivos tornou-se imperativo escolher uma *framework* direcionada para a norma IEC 61499 que, apesar de relativamente recente, satisfaça o perfil definido. Neste sentido e de entre as ferramentas existentes, (FBDK - *Function Block Development Kit*) (HOLOBLOC, 2019), ISaGRAF (2019) e *Eclipse 4diac™* (2019a) foi necessário escolher uma que permitisse desenvolver a *framework* pretendida. A escolha recaiu sobre esta última, *Eclipse 4diac*, por se tratar de uma ferramenta *open-source* de origem académica com grandes potencialidades e ainda pelos resultados positivos apresentados em algumas aplicações práticas (Strasser *et al.*, 2008; Zoitl *et al.*, 2010; Wenger *et al.*, 2018).

Sendo assim, o programa desenvolvido deverá permitir a replicação/distribuição suportada por mecanismos de comunicação standards que garantam a consistência dos dados enviados para as réplicas bem como garantir a consolidação dos dados de saída de acordo com as regras predefinidas. Por outro lado, o modelo proposto nesta tese deve permitir uma programação transparente de tal modo que o designer se possa abstrair da replicação, concentrando os seus esforços no desenvolvimento do programa. Neste sentido a replicação de componentes críticos

deve ser realizada de forma transparente de modo a que a replicação de *software* ou de *hardware* possa ser, à posteriori, perfeitamente integrada no programa. Esta deve garantir que os mecanismos de comunicação, entre infraestruturas, possam ser especificados usando interfaces genéricas, que possam ser refinadas pelo designer.

1.3 Contributos da investigação

Considerando o que foi exposto anteriormente os principais contributos desta tese são:

- *Desenvolvimento de uma framework transparente para a replicação de aplicações industriais IEC 61499.*
  Propõem-se nesta tese uma *framework* genérica e transparente para o desenvolvimento de aplicações distribuídas e replicadas tendo como base de implementação a norma internacional IEC 61499. Para isso criaram-se uma série de objetos (blocos função) que permitem o desenvolvimento das aplicações sem que sejam, na fase de programação, tomados em consideração os problemas de replicação e de distribuição. Assim, na fase de configuração, os objetos criados poderão ser instanciados para que a replicação e a distribuição sejam conseguidas.
- *Avaliação da tolerância a falhas e de tempo real suportadas pela IEC 61499/4diac.*
  Nesta tese analisaram-se os blocos função de comunicações preconizadas pela norma IEC 61499 e como estes se comportavam perante um sistema distribuído e replicado desenvolvido com recurso à ferramenta *Eclipse 4diac™*. Neste sentido, analisaram-se diversos suportes e protocolos de comunicação standards constatando-se que os blocos de comunicação definidos na norma não são capazes de garantir, por si só, o determinismo dos sistemas. Assim sendo, propõem-se um conjunto de objetos desenvolvidos para a obtenção do sincronismo e a consolidação das réplicas.
  Por outro lado, com o intuito de garantir o tempo real do sistema definiram-se um conjunto de blocos função de suporte à aplicação que permitem realizar uma análise temporal da rede em *offline*. Esta análise permite definir um conjunto de tempos de pré-execução que garantem a execução da aplicação mesmo na presença de erros originados quer pelo *bus* quer pela execução das interfaces da aplicação.

Descreve-se ainda a implementação da *framework* proposta. Com o desenvolvimento de um protótipo permitiu avaliar-se a assertividade da abordagem, genérica e transparente, desenvolvida para a implementação de sistemas distribuídos tolerantes a falhas e de tempo real. O protótipo opera sobre uma rede Ethernet constituído por cinco equipamentos de baixo custo com capacidade para correm os executáveis desenvolvidos no *4diac* com os quais é possível desenvolver o sistema de replicação quer ao nível do *software*, quer do *hardware* ou de ambos.

1.4 Estrutura da tese

A presente tese está organizada em 9 capítulos. Neste capítulo faz-se o enquadramento e descreve-se a motivação, o alcance e algumas questões fundamentais para o desenvolvimento da investigação, bem como a metodologia utilizada para o desenvolvimento da mesma.

No Capítulo 2 desta tese apresenta-se uma introdução aos princípios de confiança no funcionamento quer do ponto de vista do *hardware* quer do *software*, falhas, avarias e erros. Neste são tratados com algum detalhe os impedimentos, os atributos, a quantificação (MTBF, MTTR, λ e R(t)) e os meios necessários para a obtenção de confiança no funcionamento.

No Capítulo 3 apresenta-se detalhadamente as técnicas de tolerância a falhas. Exploram-se as várias técnicas de redundância, ativa, passiva e semi-ativa quer do ponto de vista de tolerância

a falhas com base na redundância realizada por *software* quer na realizada com base no *hardware* ou por *software* e *hardware*.

No Capítulo 4 faz-se a introdução aos conceitos dos sistemas distribuídos. São apresentadas as topologias de redes mais comuns bem como as arquiteturas que normalmente são utilizadas nos sistemas distribuídos. São referidos os seus paradigmas bem como os seus modelos de computação distribuída. O protocolo de difusão *multicast* é apresentado como suporte de comunicação.

No Capítulo 5 apresentam-se alguns trabalhos relevantes referentes à tolerância a falhas. Abordam-se diversos trabalhos tolerantes a falhas tendo por base o escalonamento de tarefas bem como a prioridade de execução. A tolerância a falhas em sistema de tempo real e as mensagens temporizadas são também abordadas. Finalmente referem-se algumas abordagens de tolerância a falhas em sistemas industriais.

No Capítulo 6 apresenta-se a nova norma internacional IEC 61499 e sua implementação na ferramenta de desenvolvimento *Eclipse 4diac*. Detalham-se alguns conceitos básicos, funcionais e estruturais, de desenvolvimento e de implementação de sistemas de controlo distribuído.

No capítulo 7 faz-se a apresentação da Infraestrutura para a Replicação de Aplicações IEC 61499 e o seu enquadramento nos sistemas distribuídos replicados utilizando os conceitos preconizados na nova norma internacional IEC 61499. É apresentada a *framework* que permite de uma forma transparente a replicação de sistemas e a tolerância a falhas com base na IEC 61499.

No Capítulo 8 faz-se a validação da *framework* desenvolvida apresentando-se os resultados obtidos durante a fase de teste finalizando-se com o Capítulo 9 onde se apresentam as conclusões deste trabalho deixando bem visível a natureza desta abordagem. Faz-se abordagem à complexidade inerente à tolerância às falhas e ao muito que ainda pode ser realizado utilizando a IEC 61499 em trabalhos futuros. Em anexos apresentam-se diversos documentos, tabelas e gráficos necessários à explicação e compreensão do trabalho desenvolvido nesta tese.

Capítulo 2

Sistemas Confiáveis

2.1 Introdução

A replicação é hoje-em-dia muito utilizada nos sistemas distribuídos como mecanismo de tolerância a falhas com o intuito de manter a disponibilidade e a fiabilidade desejada. Entre outras razões para a sua utilização, destaca-se o facto de a replicação se encaixar naturalmente nas topologias dos sistemas distribuídos.

Com o desenvolvimento da norma IEC 61499 surge a possibilidade da implementação de sistemas de controlo distribuído de automação industrial e como tal é necessário a utilização de ferramentas que permitam a replicação de componentes, como forma de garantir a tolerância a falhas. No entanto, este tipo de serviço não está definido neste novo standard pelo que numa abordagem à replicação utilizando a norma IEC 61499 há a necessidade de resolver problemas de coerência das réplicas, sincronismos, determinismo bem como todos os problemas resultantes da migração de um sistema distribuído não replicado para um replicado. Sabe-se, no entanto, que a maioria destes problemas surgem devido aos novos modelos de erro e modos de falha do *hardware* distribuído no qual a aplicação distribuída está a ser executada. Por outro lado, a natureza distribuída do *hardware* permite também que novas capacidades possam ser exploradas, tais como a melhoria da confiabilidade, por meio de mascaramento de falhas, recorrendo a metodologias de tolerância a falhas geralmente obtidas através do uso da replicação.

Neste capítulo explica-se os conceitos básicos e terminologia dos sistemas confiáveis, começando por uma introdução à confiança no funcionamento com especial enfase à tolerância a falhas. Em relação a esta área seguiremos, sempre que se aplicar, a terminologia adotada para português por Veríssimo e Lemos (1989) e a adotada por Avizienis *et al.* (2004).

2.2 Confiança no funcionamento

Um sistema é constituído por um conjunto de elementos interligados, de modo a formar um todo organizado, que interagem com outros sistemas – computacionais, mecânicos, físicos e humanos – através da sua fronteira. Todos os elementos/sistemas que se encontram fora da fronteira constituem o seu ambiente. Um sistema fornece um determinado serviço, comportamento percebido pelo utilizador, que evolui com o passar do tempo, ocupando dois estados possíveis. Serviço adequado, se este é fornecido de acordo com as condições especificadas ou serviço inadequado, sempre que o serviço fornecido é diferente das condições especificadas. Os eventos que constituem as transições entre estes dois estados são a avaria e a reposição do serviço (Veríssimo e Lemos, 1989), Figura 2-1.

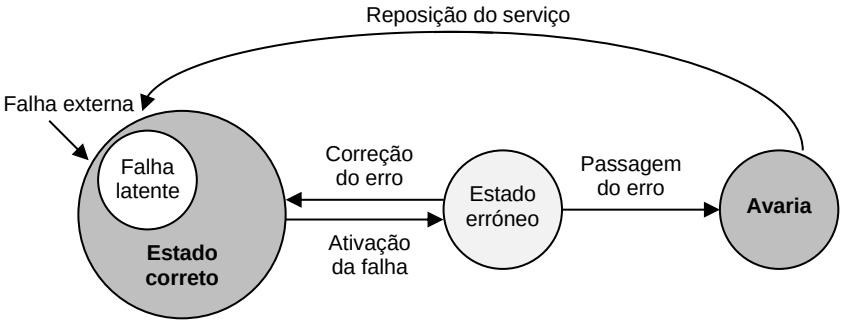


Figura 2-1. Manifestação de uma avaria e reposição do serviço (adaptada de Veríssimo e Lemos, 1989)

Assim, a confiança no funcionamento consiste em fazer com que o serviço prestado cumpra as especificações, ou seja, que não falhe. No entanto, a passagem a um estado de avaria dar-se-á sempre que um estado erróneo se propague. Este será, por sua vez, resultante da ativação de uma falha.

A falha, causa remota de uma avaria, torna-se ativa quando obriga o sistema a evoluir para um estado interno errado tornando-o efetivo quando ativado, ou seja, quando o erro produz um desvio às condições mencionadas na especificação do serviço ou quando evolui para um estado interno admissível, através de sequências de estado incorretas. A falha pode ser causada pela ocorrência de eventos internos, quando se encontra previamente dormente e é ativada pelo processo computacional, ou provocada por eventos de origem externa. O erro é a consequência da manifestação de uma falha no sistema pelo que, na sequência da resposta a determinados estímulos, poderá propagar-se conduzindo à avaria. O erro poderá encontrar-se num estado latente, sempre que seja criado por uma falta que terá existido muito antes de produzir efeitos que, quando ativado, se torna efetivo. Uma avaria é a consequência de um erro. Esta ocorre quando o serviço prestado se desvia das condições mencionadas nas especificações do serviço.

Dito em outros termos, o erro é a manifestação de uma falha no sistema enquanto a avaria é a manifestação do erro no serviço, Figura 2-2.

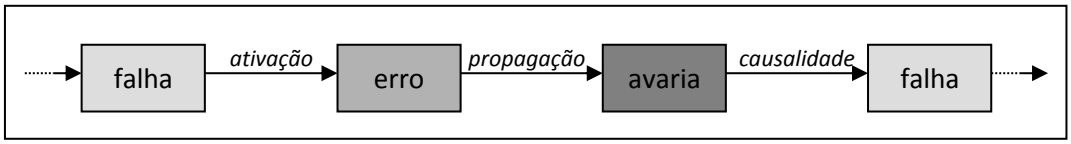


Figura 2-2. Cadeia de impedimentos à confiança no funcionamento, (adaptada de Avizienis et al., 2004)

Assim, se olharmos para um pneu de um automóvel, podemos dizer que a falha neste sistema pode ser causada por um evento externo (um prego, um arame ou um vidro) que provoca um furo ou por uma causa interna como seja o trilhar da câmara-de-ar (*causalidade*). Esta falha irá dar origem a um erro (*ativação*), esvaziamento do pneu, pondo em causa a sua utilização para os fins especificados (*propagação*). O esvaziar do pneu dá origem à avaria, impedimento do veículo continuar em marcha.

Com a confiança no funcionamento pretendemos garantir um conjunto de atributos essenciais para o correto funcionamento dos sistemas. Neste conjunto de atributos (ver Figura 2-3) estão congregados conceitos como:

- **Disponibilidade** (*Availability)* – capacidade para estar pronto a fornecer o serviço correto, em que A(t) é a fração de tempo em que o sistema se encontra disponível.
- **Fiabilidade** (*Reliability***)** – continuidade do serviço prestado, em que R(t) é a probabilidade de que o sistema se encontre em funcionamento no intervalo [0, t].

- **Segurança** (*Safety*) – não ocorrência de avarias catastróficas sobre os utilizadores e o ambiente.
- **Confidencialidade** (*Confidentiality)* – não revelação de informação não autorizada.
- **Integridade** (*Integrity*) – ausência de alterações inadequadas ao sistema.
- **Manutenibilidade** *(Maintainability)* – capacidade de sofrer, modificações e reparações.
- **Testabilidade** *(Testability)* – facilidade para testar o sistema (ponto de teste, testes automatizados, etc.).

Por outro lado, a garantia dos atributos referidos anteriormente passa pela deteção, identificação e aplicação de diversos meios que permitam resolver os impedimentos ao funcionamento de modo a atingir a confiança no mesmo. Nos últimos anos vários meios foram desenvolvidos para a obtenção da confiança no funcionamento (Figura 2-3). Estes podem-se agrupar em quatro grandes categorias:

- **Prevenção de falhas** (*Fault prevention*) – meios para prevenir a ocorrência ou introdução de falhas.
- **Tolerância a falhas** (*Fault tolerance*) – meios para garantir, por redundância, o fornecimento do serviço especificado perante a ocorrência de falhas. Os meios desta categoria podem ainda ser divididos em duas subcategorias:
    - *Mascaramento de falhas* – usar redundância para garantir que as falhas não causem uma avaria no sistema;
    - *Deteção e processamento* – meios para detetar a ocorrência de erros e processá-los de forma a neutralizá-los.
- **Supressão de falhas** (*Fault removal*) – meios para reduzir o número e/ou a gravidade das avarias.
- **Previsão de falhas** (*Fault forecasting)* – meios para estimar o número de falhas no sistema, e prever o número e consequências das falhas futuras.

O esquema completo da taxonomia de computação para confiança no funcionamento é apresentado na Figura 2-3.

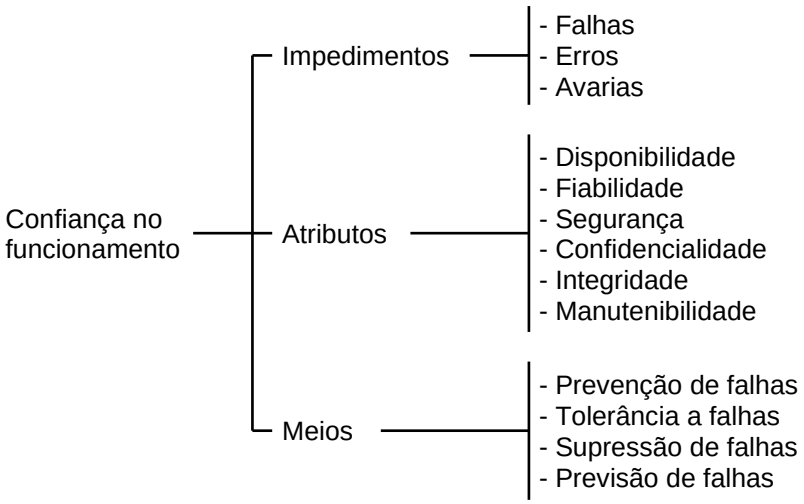


Figura 2-3. Conceitos da confiança no funcionamento, (adaptada de Avizienis et al., 2004)

2.2.1 Relação entre falha, erro e avaria

Um serviço só será executado corretamente quando este cumprir as condições mencionadas nas especificações. Neste sentido, a existência de um erro, ver Figura 2-1, dever-se-á à ocorrência de uma falha num componente provocada por fenómenos físicos de origem mecânica ou elétrica, interna ou externa. Este poderá propagar-se manifestando-se pela degradação do serviço como seja, por exemplo a limitação de serviços, a diminuição da velocidade de prestação dos mesmos, etc., dando origem a uma avaria. Em *software* as falhas decorrentes de uma programação errada traduzir-se-ão na existência de uma falha latente que quando ativada, chamada das instruções erradas ou utilização dos dados errados, dará origem a um erro que por sua vez se propagará até produzir uma avaria (Bloomfield e Lala, 2013).

Myers (1976) refere que, em termos comuns, ocorre um erro de *software* sempre que o *software* não é executado de acordo com as especificações. Este considera ainda que ao aceitarmos esta definição incorremos noutra falha fundamental: assumir-se que as especificações estão corretas. No entanto, e como se tem verificado, uma das maiores fontes de erro reside precisamente na escrita das especificações. Por outro lado, o facto de o *software* não trabalhar de acordo com as especificações coloca-nos, provavelmente, na presença de uma avaria, embora, se este trabalhar segundo as especificações, não possamos dizer que o produto não tem falhas.

Uma segunda definição de erro de *software* (Myers, 1976) é que o erro ocorre quando se verifica que a programação não é executada de acordo com as suas especificações supondo-se que este trabalha dentro dos seus limites. No entanto, se o sistema é usado acidentalmente para além dos seus limites, este terá de responder de uma forma razoável às solicitações da programação e, se isto não se verificar, então produzirá uma avaria.

Uma terceira definição apresentada por Myers (1976) prende-se com o facto de o erro ocorrer quando o *software* não se comportar de acordo com a documentação oficial ou com as publicações fornecidas ao utilizador (manuais).

Esta última definição define erro ou falha do *software* como a performance para trabalhar de acordo com o contrato original, ou a documentação de exigências do utilizador (NP EN 29000, 1994). Esta definição apresenta uma melhoria em relação às três anteriores, mas também possui várias falhas, isto é, se os requisitos do utilizador preveem um erro de *software* com um MTBF (*Mean Time Between Failure*) de 100 horas e o sistema atual provar ter um MTBF de 150 horas, o sistema ainda tem falhas embora exceda as expectativas do cliente. Myers (1976) define erro de *software* do seguinte modo:

> *Ocorre um erro de software sempre que o software não trabalha de acordo com o que o utilizador espera que este faça.*

Neste sentido, e como forma de eliminar a existência de falhas em *software* realizam-se vários testes que são parte essencial do processo de desenvolvimento. O teste do *software* deve ser planeado e realizado em condições perfeitamente definidas e disciplinadamente, de tal modo que seja possível detetar o máximo, se não todas as falhas.

O teste de *software* possui limites aceitáveis dentro dos quais se torna viável desempenhar esta tarefa. Poderão, no entanto, estes testes tornarem-se impraticáveis em determinados programas cuja complexidade é extrema, não pela impossibilidade de os realizar, mas sim pelo facto de que o tempo necessário para a realização dos mesmos seria exorbitante e extremamente dispendioso, Figura 2-4. Os custos de desenvolvimento do *software*, de manutenção (April *et al.*, 2005) e de teste do *software* estão estimados em aproximadamente 75% do custo total do mesmo como se mostra na Figura 2-4.

A este respeito Kitechenham e Neumann (1991) referem que estes valores estão indissociavelmente ligados à medição e à estimativa de desenvolvimento do projeto de controlo

pelo que, não será possível controlar corretamente o que não se consegue medir. Estes acrescentam ainda que não se pode planear o que não se consegue estimar.

Figura 2-4. Custos do desenvolvimento de software, (Myers, 1976)

Assim, e numa tentativa de diminuir as falhas de *software* será conveniente que durante o processo de desenvolvimento do mesmo sejam feitos vários testes iterativos de modo a detetar rapidamente possíveis erros e, sempre que possível, testar as ligações aos módulos independentes (entrada e saída de variáveis). A correção e deteção de erros tornam-se assim menos onerosa, mais fiável e mais rápida. Saliente-se ainda que, sempre que seja efetuada uma correção ao programa, este deve ser novamente testado com o fim de verificar se a correção efetuada resultou efetivamente e se não se introduziu uma outra falha qualquer.

O teste a grandes e complexos sistemas é normalmente uma tarefa que envolve um grande número de técnicos, trabalhando individualmente ou em equipa. Este tipo de testes, designados de testes de integração, deverão cobrir os módulos de interface, bem como demonstrar uma perfeita integridade e satisfação dos requisitos. Concluído o desenvolvimento do sistema, e uma vez que este aparenta funcionar corretamente, proceder-se-á à verificação e validação (V&V) do mesmo. Segundo Storey (1996) estes termos referem-se a:

- **Verificação** (*Verification*) – processo utilizado para determinar que um sistema ou módulo cumpre a sua especificação.
- **Validação** (*Validation*) – processo utilizado para determinar que um sistema é adequado para o seu fim.

No entanto, a fiabilidade de um programa não depende só da existência de erros, mas também da probabilidade de estes existirem, bem como do modo como os erros afetam as saídas e mais do que tudo, da natureza do seu efeito. Os erros que pela sua natureza se manifestam, normalmente na fase de teste do *software*, são facilmente detetáveis e corrigidos durante a fase de desenvolvimento. Os erros que só se manifestam em condições de trabalho muito especiais, falha que se encontra num estado latente onde o erro só ocorrerá numa situação em que o setor do código seja solicitado, poderão permanecer latentes até à degradação ou substituição do *software*. Se a avaria provocar a perda de vidas humanas ou de equipamentos muito caros, por exemplo, uma nave espacial ou um satélite, esta avaria é na realidade um desastre e terá de ser tratada exaustivamente de tal modo que a sua probabilidade de ocorrência seja praticamente igual a zero.

A Figura 2-5 apresenta um esquema de tratamento de falhas de programação, que podem ser consideradas como latentes até que o conjunto de instruções seja utilizado (Marques e Guedes, 2003). A estas avarias há ainda que acrescentar os erros resultantes do fator humano os quais podem ter a mais variada forma. A geração de erros, deteção e correção de falhas assume-

se como uma função da capacidade e da organização humana que, por si só, se torna ela mesmo numa fonte de novas falhas.

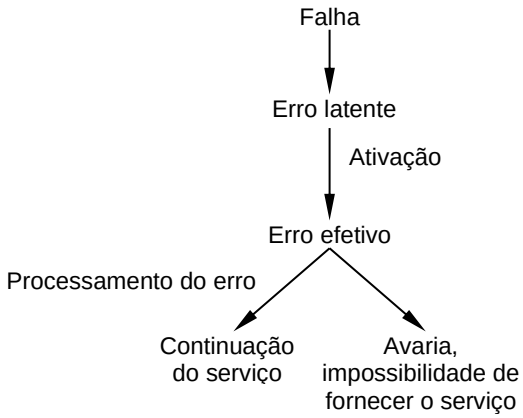


Figura 2-5. Evolução de um erro latente em software, (Marques e Guedes, 2003)

Por outro lado, os sistemas computacionais falham também devido a problemas de *hardware*, normalmente, associados à fiabilidade dos componentes eletrónicos (soldaduras, contactos, órgãos mecânicos, etc.) sendo que as falhas de *software* são, normalmente, atribuídas a problemas resultantes do desenvolvimento dos programas pelo que, estas podem ser estimadas entre 60 a 90% (Abdulhameed *et al.*, 2019). A diminuição das falhas nos sistemas computacionais passa pela melhoria da fiabilidade da componente de *hardware* nomeadamente dos menos fiáveis como por exemplo: os discos rígidos (Santos, 2001). O *software* permanece ainda hoje como a maior fonte de problemas apesar de todas as técnicas para melhoria da fiabilidade (programação estruturada, inspeção de código, testes, metodologias, qualidade, manutenção, etc.) serem conhecidas, ninguém acredita ser possível produzir *software* sem erros (Marques e Guedes, 2003).

Assim, para a maioria dos sistemas e componentes eletrónicos *(hardware)* o fator determinante para a elevada fiabilidade dos equipamentos deve-se ao rigoroso controlo de qualidade realizado ao longo de todo o processo produtivo. A constante miniaturização dos componentes eletrónicos, bem como as elevadas taxas de produção originam variações de produção que, por si só, darão origem a defeitos graves de conceção facilmente detetáveis. Não serão porventura os defeitos de fácil deteção a causa de falta de fiabilidade dos equipamentos, mas sim, aqueles que se encontram num estado latente, isto é, que não afetam imediatamente o comportamento do sistema.

Nos sistemas eletrónicos podem ser encontrados vários defeitos e várias causas possíveis para os mesmos ocorrerem, mas o mais corrente será sem dúvida, a falta de solda nas suas ligações. Este tipo de defeito é um dos mais importantes dos sistemas eletrónicos e, não sendo de fácil deteção, dá origem a um defeito em estado latente. Estes poderão mais tarde provocar uma avaria do sistema devido a fenómenos de fadiga, vibrações e sobreaquecimento, provocado por correntes elevadas, e variações da temperatura ambiente. Segundo O'Connor (2012) os componentes eletrónicos hermeticamente selados, só deverão falhar em condições extremas de trabalho, nomeadamente em situações de sobreaquecimento ou grandes variações de temperatura ou se existir um defeito que cause o enfraquecimento imediato ou progressivo.

Na Figura 2-6 apresentam-se as curvas de densidade de avarias correspondentes aos componentes eletrónicos, obtidas durante um processo normal de produção, isto é, peças boas, produzidas segundo as especificações, peças que apresentam pequenas deficiências e por isso quando submetidas aos primeiros testes são removidas dos lotes e, por fim, os componentes que embora defeituosos passam nos testes, apresentando-se como uma falha potencial. Neste último grupo de componentes, normalmente designados de "aberrações", do inglês *freaks*, podem ser

encontrados vários tipos de defeitos tais como: fraca soldadura das ligações dos componentes, óxidos, impurezas, inclusões, bem como falta de estanquicidade do encapsulamento dos componentes (O'Connor, 2012).

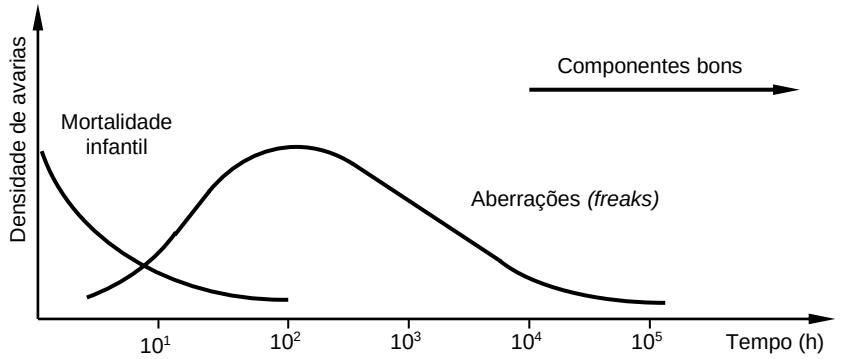


Figura 2-6. Função de densidade de avarias dos componentes eletrónicos, (adaptada de O'Connor, 2012)

Neste sentido e em termos de conclusão poderemos dizer que o *hardware* poderá ficar indisponível devido a três causas possíveis, ou seja, erros de conceção, erros de construção e avaria.

Erro de conceção – é um erro que se encontra inicialmente em todas as reproduções do produto.
Erro de construção – é um erro que se encontra inicialmente presente em uma ou mais cópias do produto devido a erros de produção (erros industriais), que ocorrem devido à falta ou fraca soldadura dos componentes eletrónicos. Estas falhas são problemas que poderão não ser detetados inicialmente no produto, mas que surgem durante o seu tempo de funcionamento devido a fenómenos físicos, bem como deteriorações causadas por falhas moleculares, aquecimento, humidades, fricções, radiações, etc.

2.2.2 Classe de falhas

Um serviço é executado corretamente quando este cumpre as condições mencionadas nas especificações do sistema e por isso, uma avaria ocorrerá sempre que o serviço prestado se desvie das especificações, quer devido a causas internas, quer externas. Neste sentido podemos dizer que todas as falhas que poderão afetar o sistema durante o seu ciclo de vida podem ser classificadas segundo três classes elementares, Figura 2-7.

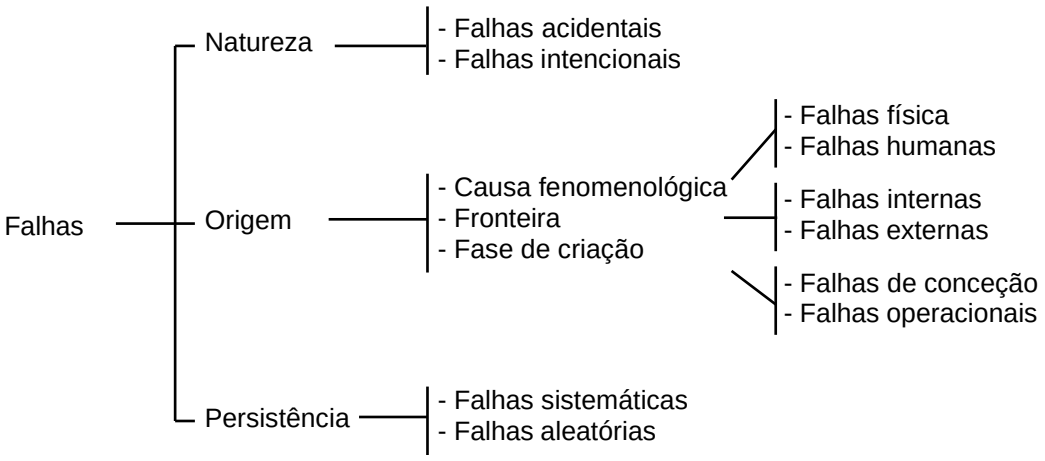


Figura 2-7. Classes de falhas, (adaptada de Avizienis et al., 2004)

- **Natureza da falha** (*Nature of fault*) – a natureza da falha (Cristian, 1991) pode ser de origem física (*hardware*) quando causada por fenómenos naturais sem intervenção humana. Estas podem surgir de modo acidental durante a fase de desenvolvimento ou mesmo na fase de operacionalidade (Adelsbach *et al*., 2003). A natureza da falha divide-se em:
  - **Falhas acidentais** – que surgem ou são criadas fortuitamente;
  - **Falhas intencionais** – que são criadas deliberadamente e de má-fé durante o processo de desenvolvimento.
- **Origem da falha** (O*rigin of fault*) – a origem das falhas pode ser causada por defeitos físicos que, por sua vez, podem ser de origem humana. Estas podem ainda surgir na fase de operacionalidade devido a falhas internas, que provocam deterioração física dos componentes ou de origem externas, devido a processos com origem fora do sistema (radiações, etc.). A origem das falhas divide-se em:
  - **Causas fenomenológicas** – *falhas físicas* que são devidas a fenómenos físicos adversos e *falhas humanas* provocadas deliberadamente, por exemplo intrusão, ou sem fins maliciosos tal como erros de programação ou provocados por decisões erradas.
  - **Fronteira** – *falhas internas* devidas a processos naturais de deterioração do *hardware*, que, por sua vez, é uma consequência de um ou mais erros elétrico ou eletrónico e *falhas externas* resultantes de interferências ou de interações com o ambiente externo, físico e humano.
  - **Fase de criação** – *falhas de conceção* que resultam de erros ocorridos durante a fase de desenvolvimento do sistema (conceção) ou durante as fases de modificação e estabelecimento de procedimentos para operar ou manter o sistema (são difíceis de tolerar, necessidade de conceções independentes e quando corrigidas não voltam a ocorrer) e *falhas operacionais* que ocorrem durante a exploração (mesmo após correção podem voltar a ocorrer, as falhas físicas são fáceis de tolerar através de redundância).
- **Persistência** – *falhas sistemáticas* reprodutíveis com determinadas condições de funcionamento e *falhas aleatórias* que não possuem relação aparente com o funcionamento do sistema. Este item divide-se em:
  - **Falhas permanentes** – falhas que após a ocorrência não desaparecem espontaneamente.
  - **Falhas transitórias** – falhas que estão presentes durante um período de tempo limitado.
  - **Falhas intermitentes** – falhas que ocorrem de forma aleatória e sem continuidade.

2.2.3 Modos de avaria

As avarias podem manifestar-se de vários modos de acordo com o modo como o sistema lida com a sua ocorrência. Neste sentido, um erro conduzirá ou não a uma avaria, dependendo da composição do sistema. Assim, o tipo de redundância associada à composição do sistema irá condicionar o comportamento do mesmo perante a ocorrência de um erro, redundância intencional ou involuntária. Se a redundância intencional se destina explicitamente a evitar que um erro conduza a uma avaria na redundância involuntária poderá ou não ter o mesmo resultado. Por outro lado, e do ponto de vista dos utilizadores do sistema, um desvio das especificações pode ser considerado uma avaria para um dado utilizador enquanto a mesma avaria, para um outro utilizador, pode ser considerada insignificante. Isto alerta-nos para o facto de que deve-se ter em conta os diferentes critérios de aceitação da avaria pois estas estão subjacentes à perspetiva humana, ou seja, diferentes taxas de erro aceitáveis.

Neste sentido podemos dizer que esta subjetividade prende-se com o que esperamos do sistema, ou seja, até que ponto é que o valor do serviço prestado não está de acordo com as especificações bem como a quantificação do tempo de prestação de um dado serviço se encontra desviado do esperado, avarias de valor ou temporais.

Os vários modos de manifestação de avarias que podemos encontrar nos sistemas computadorizados são (Avizienis *et al*., 2004):

- **Avaria por paragem** *(Stopping ou halt failure)* – quando o sistema simplesmente para, interrompendo o serviço. A atividade do sistema, se houver, não é mais percetível para os utilizadores, passando a ser prestado um serviço de valor constante. Um sistema no qual as avarias são unicamente por paragem designa-se por Sistema Para em caso de Avaria *(Fail-Stop)*.
- **Avaria por omissão** (*Omission failure*) – caso particular da avaria por paragem, no qual não é prestado qualquer serviço.
- **Avaria por omissão persistente** (*Crash failure*) – um sistema no qual as avarias são unicamente por omissão persistente designa-se por Sistema Silencioso em caso de Avaria (*Fail-Silent*).
- **Avarias coerentes** (*Consistent failures*) – todos os utilizadores do sistema possuem a mesma perceção das avarias.
- **Avarias incoerentes ou Bizantinas** (*Byzantines failures*) – todos ou alguns dos utilizadores do sistema poderão ter diferentes perceções de uma dada avaria (Lamport *et al.*, 1982).
- **Avarias benignas** (*Minor failures*) – quando as consequências da avaria são da mesma ordem de grandeza, em termos de custos, do benefício proporcionado por um serviço correto. Um sistema no qual as avarias são unicamente avarias benignas designa-se por Sistema Seguro em caso de Avaria (*Fail-Safe*)
- **Avarias catastróficas** (*Catastrophic failure*s) – quando os custos das consequências são incomensuravelmente superiores aos benefícios proporcionado por um serviço correto.

Na Figura 2-8 apresenta-se um esquema dos modos de avaria.

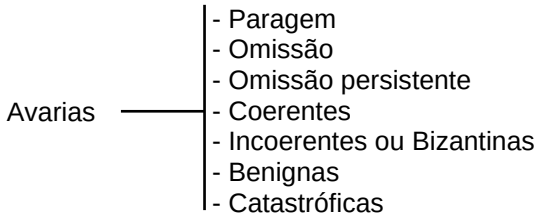


Figura 2-8. Modos de avaria

Em resumo os modos de avaria mais correntes podem ser designados como:

Avarias por paragem (*Fail-Stop*),

Avarias por omissão persistente (*Fail-Silent*),

Avarias benignas (*Fail-Safe*),

2.3 Atributos de confiança no funcionamento

A confiança no funcionamento é considerada como a capacidade do sistema prestar um dado serviço com níveis de confiança aceitáveis. Neste sentido espera-se que um sistema confiável seja capaz de evitar falhas de serviço que são mais frequentes e mais graves do que é normalmente aceitável para o utilizador. O conceito de confiança no funcionamento integra atualmente os atributos que seguidamente se expõem.

2.3.1 Disponibilidade (Availability, A(t))

A disponibilidade *(Availability)* representa a probabilidade de um componente, ou um sistema encontrar-se a funcionar corretamente no instante *t*, sabendo que no instante $t = 0$ ele encontrava-se operacional (EN 13306, 2010). Quantitativamente a disponibilidade (*A*) é a percentagem de tempo durante o qual o sistema está a funcionar em conformidade com a sua especificação. Se exprimirmos a disponibilidade em função dos períodos de funcionamento bem como em função dos tempos em que o sistema não está em condições de funcionar teremos (NP EN 15341, 2009):

$$A = \frac{UT}{UT + DT} \tag{2.1}$$

onde:

UT (*up-time*) – período de tempo em que o equipamento está em condições de ser utilizado;
DT (*down-time*) – período de tempo em que o equipamento não está em condições de ser utilizado.

Se utilizarmos como variável métrica o tempo e se nos referirmos ao tempo entre avarias consecutivas e o tempo de reparações, a disponibilidade poderá ser escrita como:

$$A = \frac{MTBF}{MTBF + MTTR} \tag{2.2}$$

Em que MTBF (*Mean Time Between Failure*) representa o tempo médio entre avarias ou tempo médio de bom funcionamento e, MTTR (*Mean Time To Repair*) representa o tempo médio de reparação ou média dos tempos técnicos de reparação de uma avaria (EN 13306, 2010).

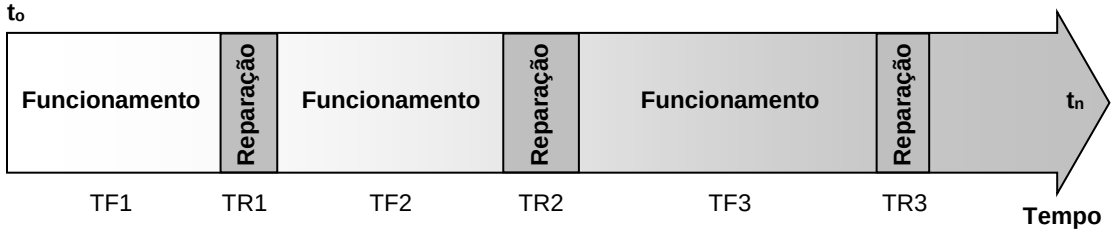


Figura 2-9. Tempos de funcionamento e de reparação (Santos, 2001)

A abreviatura MTBF é um acrónimo anglo-saxónico para *Mean Time Between or Before Failure*, esta é definida como o inverso da taxa de avarias. MTBF exprime o tempo que decorre, em média, entre duas avarias consecutivas num sistema reparável. O esquema da Figura 2-9 traduz o que se acabou de afirmar.

Neste trabalho considera-se que o MTBF é o tempo que decorre entre o fim da última reparação e o início da próxima, ou seja, por exemplo TF2 (tempo de funcionamento 2) *Mean Time Between Failure*.

O termo MTBF poderá também ser utilizado em sistemas não reparáveis, no entanto, este não terá o mesmo significado numa e noutra situação. Assim, nos sistemas não reparáveis MTBF assume a designação de *Mean Time Before Failures*, ou seja, tempo médio antes da avaria. Em sistemas não reparáveis esta terminologia poderá ser substituída pelo termo MTTF – Tempo Médio Até à Avaria, do inglês (*Mean Time To Failure*), que representa o tempo até à ocorrência de avaria (Pereira, 1996).

A abreviatura MTTR é também um acrónimo anglo-saxónico para *Mean Time To Repair*. Exprime o tempo médio necessário para reparar uma avaria, ou seja, o tempo durante o qual, pelo menos, um funcionário da manutenção executa o trabalho de reparação. Este tempo de reparação agregará, naturalmente, todo o tempo necessário para efetuar o diagnóstico da avaria, reunir todos os recursos (ferramentas, peças de substituição, desenhos técnicos, etc.), efetuar a reparação, testar e entregar o equipamento. Este indicador é, primordialmente, uma medida da manutenibilidade dos sistemas, ou seja, é o indicador da aptidão do sistema ser restaurado para uma condição de bom funcionamento (Cabral, 2006; NP EN 15341, 2009).

Podemos considerar que um sistema ou subsistema reparável é um conjunto de elementos em que a ocorrência de uma avaria não significa o fim da operacionalidade do mesmo, mas somente, uma interrupção dessa mesma operacionalidade. Do ponto de vista fiabilístico, poderá dizer-se que a fiabilidade do sistema representa a probabilidade de não se verificar qualquer falha durante um dado período de tempo, no qual se considera possível a ocorrência de mais que uma avaria (O'Connor, 2012). Recordemos que uma avaria corresponde a uma alteração ou à cessação da capacidade de um sistema para realizar a sua função específica.

A taxa de avarias será então um indicador de fiabilidade do sistema. Esta representa a proporção de componentes ou equipamentos que devem sobreviver num instante *t*. A taxa de avarias λ será dada pela seguinte expressão (NP EN 15341, 2009):

$$\lambda = \frac{N^o\ de\ avarias}{Duração\ de\ utilização} \quad ou \quad \lambda = \frac{N^o\ de\ avarias}{n^o\ items \times Duração\ de\ utilização} \tag{2.3}$$

A taxa de avarias representa a "velocidade" a que surgem as avarias. Assim, quanto menor for a variação da taxa de avarias ao longo do tempo, maior será a duração do sistema. Esta pode também exprimir-se em função do MTBF onde a taxa de avarias será dada por:

$$\lambda = \frac{1}{MTBF} \tag{2.4}$$

A relação entre a degradação do sistema ao longo do tempo de funcionamento e a taxa de avarias representa a evolução deste ao longo do seu ciclo de vida. Esta relação é normalmente representada pela curva conhecida como "Curva da banheira", ver Figura 2-10.

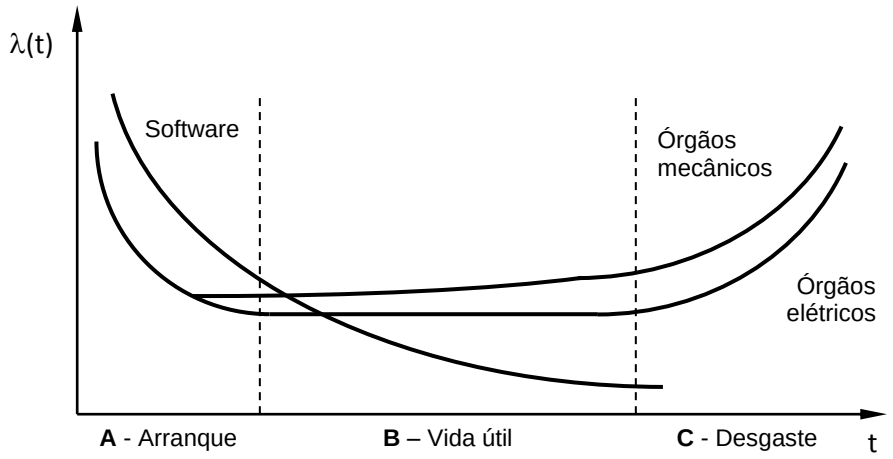


Figura 2-10. Evolução da taxa de avarias com o tempo, "Curva da banheira", (Santos, 2001)

Ao longo desta tese considera-se que as avarias ocorrem segundo um processo homogéneo de Poisson e que a taxa de avaria (λ) é constante pelo que a fiabilidade do sistema pode-se exprimir em função da distribuição exponencial negativa.

2.3.2 Manutenibilidade (Manutibility, M(t))

A manutenibilidade é definida, de acordo com as normas (EN 13306, 2010), como a:

> *Aptidão de um bem nas condições de uso especificadas para ser mantido ou restaurado de tal modo que possa realizar as funções que lhe são exigidas, quando a manutenção é realizada em condições definidas utilizando procedimentos e recursos prescritos.*

A manutenibilidade é essencialmente uma característica de conceção, fabricação e desenvolvimento dos sistemas. Pode ser avaliada pelo tempo necessário para a sua manutenção caracterizando-se pelo grau de facilidade de acesso a inspeções, verificações, substituição ou reparação de componentes. A manutenibilidade traduz a capacidade de um sistema ser mantido em boas condições de funcionamento.

Do ponto de vista matemático, a manutenibilidade define-se como a probabilidade de recuperar o sistema e repô-lo em funcionamento num dado intervalo de tempo TTR (*Time To Repair*), isto é:

$$M(t) = \Pr(TTR < t) \tag{2.5}$$

Então a probabilidade de uma operação durar até um certo limite de tempo *t* será dada por:

$$M(t) = 1 - e^{-\mu t} \tag{2.6}$$

Onde μ é a taxa de manutenibilidade. O tempo médio despendido nas operações de manutenção será:

$$MTTR = \frac{1}{\mu} \quad onde \quad \mu = \frac{n^o \ de \ operações \ de \ manutenção}{tempo \ total \ de \ reparações} \tag{2.7}$$

ou

$$MTTR = \frac{\sum f_i \times TTR_i}{\sum f_i} \tag{2.8}$$

em que:

f_i - frequência das operações de manutenção;

TTR_i - duração das operações de manutenção.

2.3.3 Fiabilidade (Reliability, R(t))

Quando se adquire um bem, espera-se que ele corresponda às expectativas. No entanto é possível que ocorra uma avaria de funcionamento em qualquer momento da sua vida útil, ou seja, no período em que o bem está em condições económicas e tecnológicas para desempenhar a sua função. À relação do estado de funcionamento de um bem com o tempo dá-se o nome de Fiabilidade (R – *Reliability*). Esta pode ser definida como a (O'Connor, 2012):

> *Capacidade de um bem desempenhar a sua função específica em condições definidas e por um período de tempo determinado.*

Pode exprimir-se a fiabilidade através da probabilidade do bem funcionar corretamente nas condições e no período de tempo referido. Esta noção temporal pode, no entanto, ser substituída por um outro tipo de unidade de contagem (quilómetros, ciclos, rotações, etc.).

Quantitativamente a fiabilidade R(*t*) é definida como a probabilidade de um sistema funcionar em conformidade com a sua especificação durante um período de duração *t*. Matematicamente a função fiabilidade exprime-se por:

$$R(t) = e^{-\lambda t} \tag{2.9}$$

com:

e – número de Neper;
t – tempo decorrido desde a última reparação;
λ – taxa de avarias constante.

Por outro lado, o conceito de fiabilidade R(*t*) apresentar-se-á como uma relação íntima, que é ela mesma, dependente dos vários subconjuntos de que o sistema possa ser constituído. A determinação da fiabilidade do sistema completo é, portanto, um cálculo probabilístico mais ou menos complexo, que poderá ser representado de maneira sistemática por diagramas de blocos de confiabilidade (RBD – *Reliability Block Diagram*) (IEC 61078, 2006).

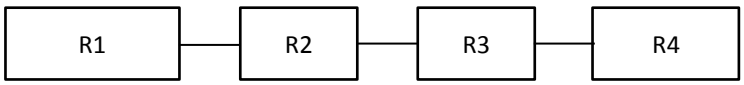


Figura 2-11. Esquema da aplicação inicial da aplicação desenvolvida

Consideremos então os blocos função usados inicialmente no desenvolvimento do teste de replicação apresentados na Figura 2-11.

Nesta primeira implementação o sistema encontra-se totalmente dependente da fiabilidade de cada um dos FB que constituem a aplicação. Neste caso a fiabilidade do sistema é obtida em função da fiabilidade de cada componente considerando a taxa de avarias *λ(n)* constante. Então a fiabilidade do sistema, num dado período de funcionamento *(Rs)* será dada por:

$$Rs = \prod_{i=1}^{n} Ri \tag{2.10}$$

Para que a fiabilidade de sistemas críticos seja melhorada é habitualmente utilizada a redundância ativa, colocando em paralelo componentes do sistema, *hardware* e/ou *software*, que assegurem as mesmas funções trabalhando em simultâneo. Nesta situação, desde que um dos componentes ou elemento de *software* funcione, o sistema realizará a sua função. No exemplo apresentado a redundância é realizada pela triplicação dos elementos centrais do sistema distribuídos por vários dispositivos e/ou recursos. A representação desta redundância é apresentada, sob a forma do RBD apresentado na Figura 2-12.

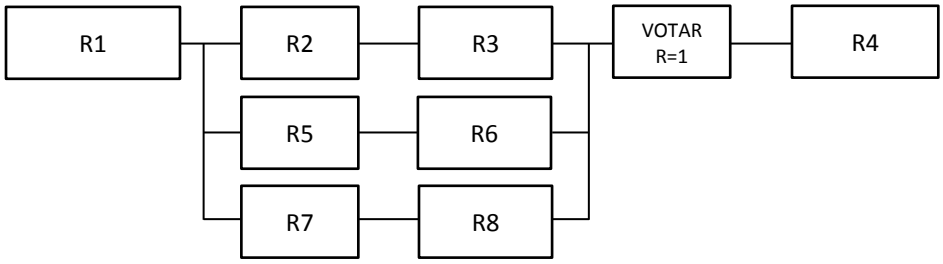


Figura 2-12. Esquema da redundância tripla

Mantendo as condições definidas anteriormente, poderemos quantificar a fiabilidade do sistema tendo em conta que pelo menos uma das réplicas [1º (R2 e R3), 2º (R5 e R6) ou 3º (R7 e R8) paralelo] funcione. Note-se que agora estamos na presença de um sistema misto constituído por uma série e vários paralelos. Para o sistema em paralelo, isolando-se os componentes, a fiabilidade do sistema, num dado período de funcionamento *(Rs)* será dada por:

$$Rs = 1 - \prod_{i=1}^{n} (1 - Ri) \tag{2.11}$$

onde o valor da fiabilidade do sistema, apresentado na Figura 2-12, será dado por:

$$Rs = R1.\{1 - [(1 - R2.R3)(1 - R5.R6)(1 - R7.R8)]\}.R4 \tag{2.12}$$

Note-se que com esta construção a fiabilidade do sistema sofrerá um acréscimo. É de salientar que o elemento VOTAR deve ser considerado como um elemento que não falha, como tal, possuir fiabilidade igual a 1 (100%) pelo que este requererá apenas *k+1* réplicas em funcionamento, falha silenciosa, para que o sistema continue operacional.

Por outro lado, se o algoritmo do VOTAR for desenvolvido para uma votação maioritária o sistema só funcionará se pelo menos *k*, ou mais de *n*, itens em paralelo produzirem saídas (*k-out-of-n*, *k*oo*n*). A função de uma estrutura *k-out-of-n* pode ser escrita como (Rausand e Høyland, 2004):

$$\phi(x) = \begin{cases} 1 \; if \; \sum_{i=1}^{n} x_i \geq k \\ 0 \; if \; \sum_{i=1}^{n} x_i < k \end{cases} \tag{2.13}$$

Nesse caso, somente a falha de apenas um componente será tolerada, enquanto duas ou mais falhas de componentes levam à falha do sistema, produção de valores erróneos ou valor produzido pelo votador. O diagrama de blocos de fiabilidade da estrutura *2-out-of-3* (votador maioritário) também pode ser representado de acordo com o esquema apresentado na Figura 2-13 (indicação do número de falhas admissíveis no votador (2/3), ou seja, uma única falha em três componentes replicados). Esta é a representação esquemática preconizada pela IEC 61078 (2006).

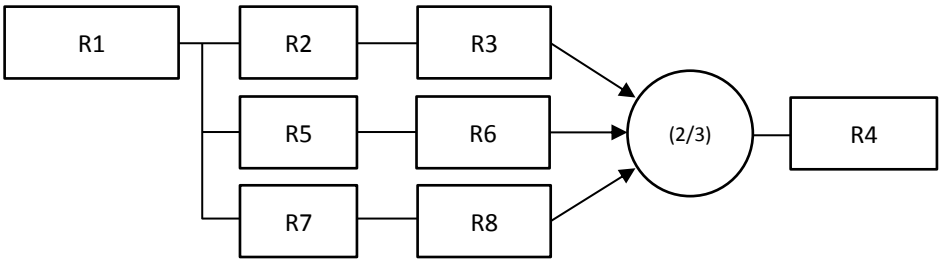


Figura 2-13. Esquema da redundância 2-em-3 (2/3, 2-out-of-3)

A equação de fiabilidade de um sistema *2-em-3* (2/3), num dado período de funcionamento, *(Rs)* será dada por:

$$Rs = \sum_{r=0}^{n-k} \binom{n}{r} . R^{n-r} . (1-R)^r = (3R_M^2 - 2R_M^3) . R_V \tag{2.14}$$

Ou calculada em função do valor da probabilidade de avaria do sistema (*Fs*) que será obtida, em função dos valores de *R(t)*, por:

$$Fs = R1.\begin{bmatrix} (1-Rs_{2,3})(1-Rs_{5,6})(1-Rs_{7,8}) + \\ (1-Rs_{2,3})(1-Rs_{5,6})Rs_{7,8} + \\ (1-Rs_{2,3})Rs_{5,6}(1-Rs_{7,8}) + \\ Rs_{2,3}(1-Rs_{5,6})(1-Rs_{7,8}) \end{bmatrix}.R4 \quad (2.15)$$

Onde a fiabilidade do sistema será dada por $Rs = 1 - Fs$. A fiabilidade do sistema sofrerá um decréscimo ligeiro, relativamente a um sistema redundante dito de "simples", uma vez que são impostas condições de funcionamento, pelo menos dois paralelos em funcionamento.

Note-se ainda, em forma de conclusão deste item, que a fiabilidade do sistema dependerá única e simplesmente da fiabilidade dos equipamentos em que a aplicação se encontra a correr. Isto será tão ou mais verdade uma vez que os sucessivos testes e as alterações ao *software* conduzem a uma taxa de avarias (λ) que tenderá, de forma exponencial, para zero e como tal, a fiabilidade do *software* tenderá para 100%, como se pode depreender da Figura 2-10.

2.4 Segurança (Safety, S(t))

A segurança é definida como a não ocorrência de falhas catastróficas para o utilizador ou para o ambiente. Esta deve ser encarada como uma componente para todo o sistema, incluindo o *hardware* e o *software*, e para todo o seu ciclo de vida. Por outro lado, estas considerações, ao afetarem todos os estágios do ciclo de vida do sistema desde a sua conceção, instalação, utilização e manutenção, irão afetar diretamente ou indiretamente, todos os que a ele se encontrarem ligados. Deste modo, e ao abordar os problemas da segurança, devemos ter em conta um outro atributo, a confidencialidade, ou seja, a necessidade de não divulgar informação não autorizada. Assim, a segurança nada mais será do que um conjunto dos atributos para a qual terá de se verificar ao mesmo tempo a existência de: 1) disponibilidade para ações autorizadas, 2) confidencialidade e 3) integridade (Avizienis *et al.*, 2004).

Na Figura 2-14 resume-se a relação entre a confiança no funcionamento e a segurança em termos dos principais atributos.

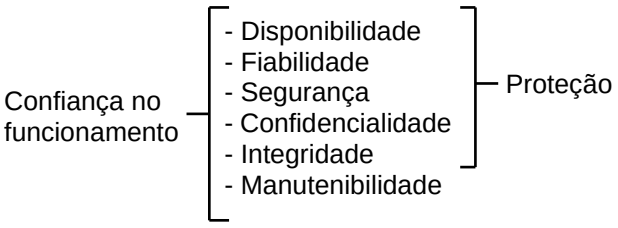


Figura 2-14. Confiança no funcionamento e atributo de segurança, (Avizienis et al., 2004)

Depreende-se que a segurança de um sistema está dependente, essencialmente, de uma conceção ou implementação segura considerando-o como um todo e não como um conjunto de subsistemas ou componentes. Esta deve considerar ainda a possibilidade de ocorrência de situações de risco e não unicamente a ocorrência de falhas. Para isso, não se deve sustentar a segurança dos sistemas em experiências anteriores, mas sim fundamentá-la através de análises de ocorrência de situações de risco e da análise de risco dos mesmos. A aplicação desta metodologia conduz ao desenvolvimento de um Sistema de Segurança Crítico, sistema onde as avarias catastróficas não ocorrem, dado que a segurança do mesmo é garantida durante a conceção ou implementação.

Podemos ainda dividir os aspetos da segurança em várias classes onde a primeira classe representa aquilo que poderia ser denominado como **Segurança Primária** do sistema dado que se encontra relacionada com os riscos causados diretamente pelo próprio sistema (Storey, 1996). A esta são associados os perigos de electrocução ou choque elétricos causados pelo *hardware*, por exemplo. A segunda classe representa a **Segurança Funcional** do sistema. Esta cobre aspetos relacionados com o equipamento diretamente controlado pelo sistema computacional e está relacionado com o correto funcionamento do sistema computacional e do *software*. Por fim, a terceira classe a **Segurança Indireta** que se encontra relacionada com as consequências indiretas da avaria computacional ou com a produção de informação incorreta.

2.4.1 Nível de Integridade de Segurança (SIL)

O Nível de Integridade de Segurança (SIL – *Safety Integrity Level*) é definido como a medida da probabilidade de um sistema de segurança desempenhar corretamente e de forma satisfatória, as funções de segurança exigidas, para as condições estabelecidas, durante um período de tempo especificado (IEC 61508, 2010).

A determinação do SIL de um sistema está relacionada com o nível de risco do mesmo. Este é determinado com base numa série de fatores quantitativos, em combinação com fatores qualitativos, como sejam as taxas de avarias definidas para cada nível e as taxas de avaria máximas toleráveis resultantes da análise de risco (Hristov e Bo, 2015). Sendo o SIL uma representação estatística da fiabilidade dos sistemas de segurança encontra-se dividido em quatro categorias distribuídas por SIL1 a SIL4 em que, o mais alto e confiável é o SIL4, risco tolerável, e o mais baixo e menos confiável o SIL1 é considerado como risco intolerável (IEC 61508, 2010).

Os níveis do SIL são definidos em termos de intervalos para a probabilidade média de falha na procura de um modo de baixa solicitação de operações (PFD – *Probability of Failure on Demand*), ou a probabilidade de uma falha perigosa por hora para uma alta solicitação ou modo contínuo de operação (PFH – *Probability of Failure per Hour*) (Hennell *et al.*, 2006). A classificação é baseada na quantidade de redução de risco que é necessário para mitigar os riscos associados com o processo a um nível aceitável, ou seja, a probabilidade dos instrumentos de segurança do sistema (SIS – *Safety Instrumented System)* falharem quando solicitados.

A questão fundamental é determinar a frequência com que as avarias, de qualquer tipo, conduzem a acidentes. Neste sentido duas abordagens podem ser realizadas (Gulland, 2004). A primeira visa analisar funções com uma taxa baixa de procura onde a taxa de acidentes é a combinação da frequência de procura e a probabilidade de falha na procura PFD. A segunda abordagem refere-se a funções que possuem uma taxa de solicitações elevadas ou que operam continuamente.

Na primeira situação o medidor do desempenho é o PFD, ou a sua reciprocidade, ou seja, o Fator de Redução de Risco (RRF – *Risk Reduction Factor*). Na segunda situação a taxa de acidentes é a taxa de falhas (λ) que é a medida de desempenho do sistema. Uma alternativa a esta análise é a medida do tempo médio até ou entre falhas (MTTF). Estas medidas de desempenho estão relacionadas na sua forma mais simples desde que se faça um teste de prova e se relacione com uma frequência que será maior do que a taxa de solicitação. Esta pode ser expressa como:

$$PFD = \lambda T/2 \quad ou \quad PFD = T/(2 \times MTTF) \tag{2.16}$$

ou

$$RRF = 2/(\lambda T) \quad ou \quad PFD = (2 \times MTTF)/T \tag{2.17}$$

em que:

T - intervalo do teste de prova

2.5 Conclusão

Neste capítulo apresentam-se uma série de indicadores bem como metodologias que nos permitem verificas e obter a confiança no funcionamento. Se por um lado são apresentadas metodologias e custos associados à deteção, verificação e estimativa/previsão de falhas, também foram referidos os meios usados para o aumento da fiabilidade dos sistemas R(*t*).

Assim sendo, o aumento de confiança dos sistemas passará não só pela análise dos KPIs de manutenção (MTBF, disponibilidade, taxa de avarias, etc.), mas também pelo aumento da fiabilidade quer seja por redundância simples ou tripla bem como por todos os fatores que possam contribuir para a segurança do sistema nomeadamente no que se refere às taxas de risco e de avaria.

Capítulo 3

Tolerância a Falhas

3.1 Introdução

Os sistemas de controlo por computador são fundamentais para um largo espectro de aplicações, que vão desde a automação industrial e o controlo de processos até à robótica e às aplicações críticas em sistemas móveis. Em todas estas áreas são utilizados computadores para controlar o ambiente envolvente sendo esperado que estes reajam a estímulos externos de acordo com os requisitos impostos pelo ambiente a controlar. Por outro lado, espera-se que os sistemas de controlo computacionais funcionem corretamente mesmo na presença de componentes defeituosos. Assim, os sistemas computacionais deverão ser capazes de fornecer à aplicação de controlo um serviço de acordo com as especificações, mesmo na presença de falhas, de forma a garantir a tolerância a falhas dos mesmos.

As falhas nos sistemas computacionais são inevitáveis, mas as consequências das mesmas, ou seja, a interrupção do fornecimento do serviço ou paragem do equipamento e a perda de dados, podem ser evitadas recorrendo a várias técnicas de fácil compreensão. É durante a fase de conceção do sistema que o programador deverá efetuar a integração destes requisitos de forma a garantir que o sistema, através de cópias de segurança dos dados (*backups*) e/ou da redundância de equipamentos, garanta a tolerância a falhas, dentro das expetativas dos utilizadores. Naturalmente que não deveremos estar à espera que um sistema tolerante a falhas tolere toda e qualquer falha em qualquer situação. Por outro lado, conhecer as consequências das falhas e as soluções existentes para evitar ou recuperar o sistema após a sua manifestação é imprescindível para que os programadores, que desenvolvem sistemas de controlo, possam fornecer serviços computacionais de qualidade aos seus clientes. A utilização destas metodologias, com elevados custos associados, não garante que as falhas associadas ao projeto de um qualquer sistema de controlo não possam subsistir na fase de implementação ou durante a fase de exploração, mesmo que em estado latente. Há ainda que ter em conta que a redundância de equipamentos aumenta consideravelmente os custos do *hardware*.

No âmbito do controlo de processos industriais é dominante a utilização dos chamados Autómatos Programáveis (PLCs) para a execução dos algoritmos de controlo. Sendo utilizados na implementação de sistemas de controlo industrial, torna-se necessário que também estes equipamentos sejam capazes de permitir a execução de programas com restrições de tempo real e tolerância a falhas com elevada fiabilidade.

Com o aumento da dependência da sociedade dos sistemas automatizados e informatizados que controlam no dia-a-dia o tráfego terrestre e aéreo, a produção de energia, a manutenção de

dados comerciais e financeiros das empresas e dos cidadãos e as transações comerciais internacionais de todo o tipo, é fácil imaginar a dimensão das catástrofes que falhas não detetadas e não reparadas poderiam causar. Por outro lado, o *software* e os procedimentos, de projeto estão a tornar-se cada vez mais complexos e apresentam cada vez mais problemas. Daí que, por si só, a confiança no funcionamento dos componentes de *hardware* não garanta mais as necessidades de grande disponibilidade, de qualidade, de segurança e de tolerância a falhas dos sistemas computacionais.

O termo tolerância a falhas foi definido por Avizienis em 1967. Desde então tem sido amplamente utilizado pela comunidade académica para designar toda a área de pesquisa ocupada com o comportamento de sistemas computacionais sujeitos a ocorrência de falhas. Na indústria o termo mais usado é *sistema redundante*. Na comercialização de sistemas computacionais como mainframes e servidores de rede, o termo usual é *alta disponibilidade*. Sistemas redundantes e sistemas de alta disponibilidade apresentam técnicas comuns, mas alcançam resultados diferentes, uns visam alta disponibilidade e outros a continuidade do serviço, ou seja, fiabilidade (Weber, 2009).

A prevenção e remoção de falhas não são suficientes quando se pretende garantir que o sistema trabalhe com elevada confiança ou alta disponibilidade. Por este facto, estes devem ser construídos usando técnicas de tolerância a falhas que garantam o correto funcionamento do sistema mesmo aquando da ocorrência de falhas. Estas técnicas de prevenção são, normalmente, baseadas em redundância, exigindo componentes adicionais ou algoritmos especiais. A tolerância a falhas não dispensa as técnicas de prevenção e de remoção pelo que os sistemas construídos com componentes frágeis e técnicas inadequadas de projeto não conseguem ser confiáveis pela simples aplicação da tolerância a falhas.

A tolerância a falhas, um dos meios de obtenção de confiança no funcionamento, só poderá ser alcançada colocando as réplicas a correr em diferentes e distintos *hardwares* e, também, pela execução de diferentes implementações do mesmo algoritmo. Neste sentido a replicação deverá ser realizada ao nível dos dispositivos críticos, juntamente com o *software* que neles está a ser executado, mantendo uma única cópia dos dispositivos não críticos onde a tolerância a falhas não será necessária (Pullum, 2001; Yang *et al.*, 2017).

A área de tolerância a falhas em sistemas distribuídos é vasta e excitante. Garantir a confiança no funcionamento envolve solucionar problemas de consenso, ordenação e atomicidade na troca de mensagens entre grupos de processos, sincronizar relógios quando necessário, implementar réplicas consistentes de objetos, garantir resiliência de dados e processos num ambiente sujeito a falhas de estações clientes e servidoras, particionamento de redes, perda e atrasos de mensagens e, eventualmente, comportamento arbitrário dos componentes do sistema.

Várias técnicas de tolerância a falhas foram abordadas na literatura existente: a tolerância ativa (Schneider, 1990), a tolerância passiva (Budhiraja *et al.*, 1993) e a replicação semi-ativa (Powell, 1991). Por outro lado, e segundo Lee e Anderson (1990), as aplicações destas técnicas de tolerância a falhas deverão ser divididas em quatro fases de aplicação: a deteção da falha, o confinamento, a recuperação e tratamento da falha, excluindo-se o mascaramento das falhas pois esta é uma técnica usada em complemento às demais. Neste sentido, e qualquer que seja a abordagem, o propósito principal é garantir que se uma réplica falhar as restantes continuam em funcionamento, mascarando a existência da réplica em falha perante toda a restante aplicação. Note-se, no entanto, que se a réplica simplesmente parar de gerar valores de saída (semântica da falha-silenciosa) qualquer um dos métodos expostos anteriormente pode ser utilizado para detetar a falha. No entanto, e se o componente em falha continuar a gerar valores de saída erróneos (erros bizantinos) só a replicação ativa é capaz de detetar a falha (Chen e Avizienis, 1978). Adicionalmente, a replicação ativa permite ainda mascarar mais rapidamente o erro pelo que não será necessário utilizar nenhum sistema auxiliar de reposição de informação (*backup*).

Várias abordagens relacionadas com a replicação e a tolerância a falhas em sistemas distribuídos foram já publicadas das quais se destaca o protocolo *atomic multicast* (Hadzilacos, 1993), protocolo *consensus agreement* (Hadzilacos, 1993), mensagens temporizadas (Poledna *et al.*, 2000) e a estrutura *DEAR-COTS* (Pinho *et al.*, 2004) usando exclusivamente a linguagem de programação Ada95. Outras metodologias de tolerância a falhas podem ser encontradas em (Carzaniga *et al*., 2009; Xie *et al.*, 2009).

Pelo exposto, poder-se-á argumentar que a redundância é o termo encontrado para definir tolerância a falhas. Todas as técnicas de tolerância a falhas utilizam de alguma forma redundância (Storey, 1996). Podemos dizer que a redundância se encontra de tal modo relacionada com a tolerância a falhas que o termo mais usado para um sistema tolerante a falhas é designação de sistema redundante.

Teremos, no entanto, que ter em conta que a aplicação desta técnica passa normalmente pela duplicação de recursos, *software* e *hardware*, de forma que a falha de um módulo não resulta na falha do sistema. Assim, a aplicação da redundância para implementação de técnicas de tolerância a falhas pode ser conseguida segundo várias formas básicas (Storey, 1996; Yang *et al.*, 2017):

- Redundância de hardware,
- Redundância de software,
- Redundância de informação,
- Redundância temporal.

Temos ainda de ter em conta que todas estas formas de redundância influenciam de alguma forma o sistema quer seja no custo, no desempenho, no tamanho ou no peso pelo que a utilização de redundância deve ser bem ponderada (Yang *et al.*, 2017).

3.2 Redundância de hardware

A redundância de *hardware* baseia-se na replicação de componentes físicos para além do que seria necessário para implementar um sistema, com ausência de defeitos. Esta metodologia caracteriza-se por três tipos de redundância: redundância passiva ou estática, redundância ativa ou dinâmica e redundância híbrida ou semi-ativa (Yang *et al.*, 2017).

No primeiro caso, redundância passiva, a metodologia é normalmente utilizada para o mascaramento de falhas não requerendo ações do sistema pelo que não dá indicação de falha. No segundo, redundância ativa, verifica-se a deteção, localização e recuperação do sistema substituindo o ou os módulos, prolongamento do ciclo de vida dos sistemas. Por fim, no terceiro caso, redundância híbrida, verifica-se a combinação da metodologia passiva com a ativa. Esta é usada para garantir o mascaramento e prolongamento do ciclo de vida do sistema, geralmente com altos custos.

3.2.1 Redundância estática/passiva mascaramento

Na redundância de hardware estática ou passiva os elementos redundantes são usados para mascarar as falhas. Estes elementos redundantes executam as mesmas tarefas onde o resultado é determinado por um mecanismo de votação que compara as saídas dos vários módulos usados no mecanismo de mascaramento das falhas ocorridas. Exemplos destes mecanismos são a Redundância Modular Tripla (TMR - *Triple Modular Redundancy*) (Habinc, 2002; Yang *et al.*, 2017) e a Redundância Modular Múltipla (NMR - *N-Modular Redundancy*) (Takaesu, 2004).

A redundância modular tripla é a técnica mais utilizada para tolerância a falhas. Esta máscara as falhas em componentes de *hardware* triplicando o componente. Estes elementos recebem o mesmo valor de entrada e deverão produzir o mesmo valor de saída. O elemento de votação compara as saídas dos três módulos, verificando a sua concordância, produzindo uma saída que traduza a sua análise, Figura 3-1. A votação pode ser obtida por maioria, 2 em 3 (2oo3 – *Two-out-of-three*) (Yang *et al.*, 2017), ou votação por seleção do valor médio (Storey, 1996).

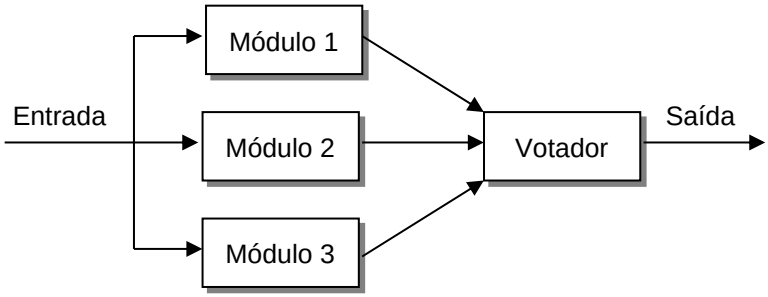


Figura 3-1. Redundância modular tripla (adaptada de Storey, 1996)

O votador realiza uma função simples de validação do resultado não possuindo a função de determinar qual o módulo de *hardware* que discorda da maioria. Este pode ainda encontrar-se associado, em série, com outros módulos de *hardware* que recebam o resultado da sua votação. Neste caso, se o votador apresentar baixa confiança no funcionamento, todo o sistema vai tornar-se tão pouco fiável quanto o votador. O votador passa a ser um ponto crítico da implementação da TMR. Outra área de vulnerabilidade desta metodologia centra-se no facto da existência de um único ponto de falha que, ao enviar dados errados para os módulos produz valores de saída erróneos que irão ser tornados consistentes pelo votador, isto é, todos os módulos enviam os mesmos dados errados para validação e, por isso, o votador torná-los-á consistentes segundo os critérios definidos para o votador.

A solução para contornar este problema passa pela construção de um votador com componentes de alta confiança no funcionamento e pela triplicação do votador ou pelo desenvolvimento de sistema com múltiplas fases de validação. A triplicação do votador resolve o problema da existência de um único ponto de falha produzindo três saídas independentes que são conduzidas aos próximos têS módulo de *hardware* produzindo resultados corretos, desde que não haja mais que um modo de falha. Na Figura 3-2 mostra-se um esquema de um votador triplicado.

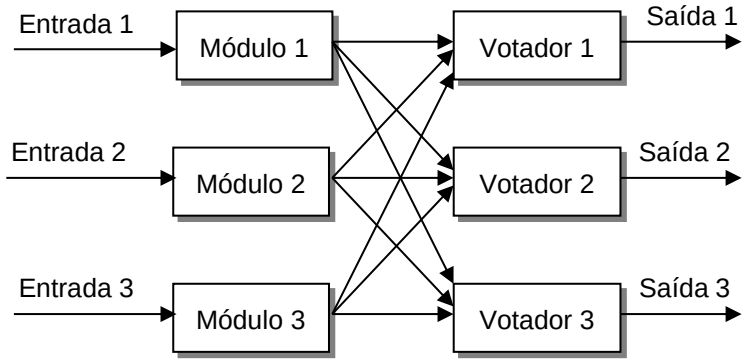


Figura 3-2. TMR com votador triplicado (adaptada de Storey, 1996)

Por outro lado, e quando estamos na presença de sistemas de elevada criticidade e pretendemos garantir a confiança no funcionamento dos mesmos, vários TMR podem ser ligados em cascata ligando as saídas de um TMR às entradas do próximo TMR, Figura 3-3. Uma única falha

votada resulta numa única saída em falha pelo que, este acontecimento é removido pela próxima redundância modular tripla.

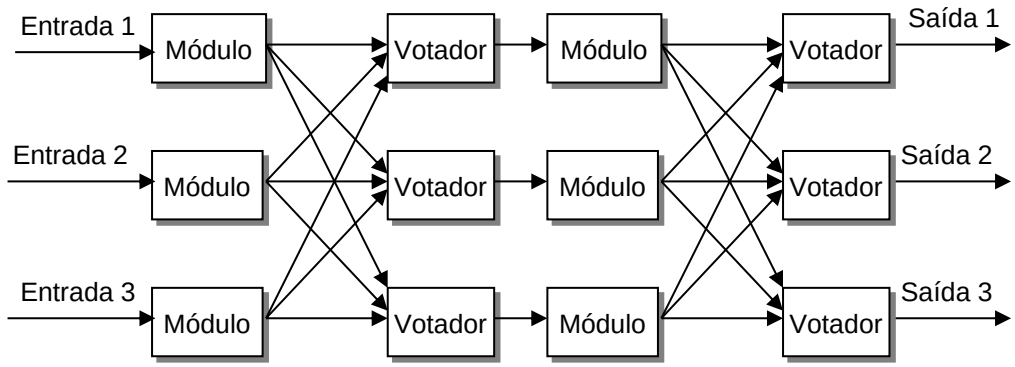


Figura 3-3. TMR com votador triplicado em cascata (adaptada de Storey, 1996)

Como se pode verificar no esquema apresentado na figura seguinte a TMR apresentará maior fiabilidade do que um sistema com um único componente após a ocorrência da primeira falha permanente, Figura 3-4. Depois disso, perde a capacidade de mascarar as falhas apresentando uma confiança menor do que um sistema de um único componente. Com o tempo e com o aumento da probabilidade dos componentes falharem (aumento a taxa de avarias) o TMR apresenta uma fiabilidade pior do que um sistema não redundante (Weber, 2009). Este é ideal para falhas temporárias que ocorram uma de cada vez durante períodos de funcionamento curtos.

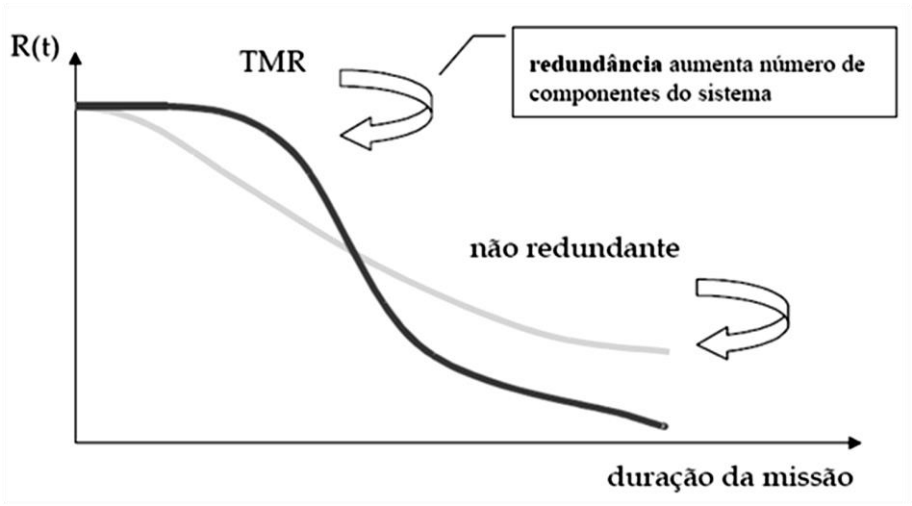


Figura 3-4. Fiabilidade do TMR versus sistema não redundante (Weber, 2009)

A Redundância Modular Múltipla (NMR – *N-modular Redundancy*) é a generalização do conceito TMR. Este consiste num arranjo com *n* módulos que votam no mesmo elemento.

3.2.2 Redundância dinâmica/ativa de hardware

Na redundância dinâmica ou ativa, a tolerância a falhas é realizada pela utilização de técnicas de deteção, localização e recuperação. A redundância ativa utiliza a deteção de falhas em vez do mascaramento da falha. O sistema redundante é constituído por um ou mais sistema em *standby* que entram em funcionamento assim que o elemento principal falhar. Assim, os erros produzidos por uma falha devem ser detetados dentro do sistema de modo que o sistema dinâmico possa reconhecer a falha e tomar as medidas adequadas à sua recuperação. A redundância dinâmica é

adequada a aplicações que podem tolerar, temporariamente, erros no seu funcionamento. Este tempo será o necessário para a deteção do erro e recuperação do sistema para um estado livre de falhas. Os estados de um sistema redundante dinâmico podem ser representados de acordo com o esquema seguinte (Weber, 2009), Figura 3-5.

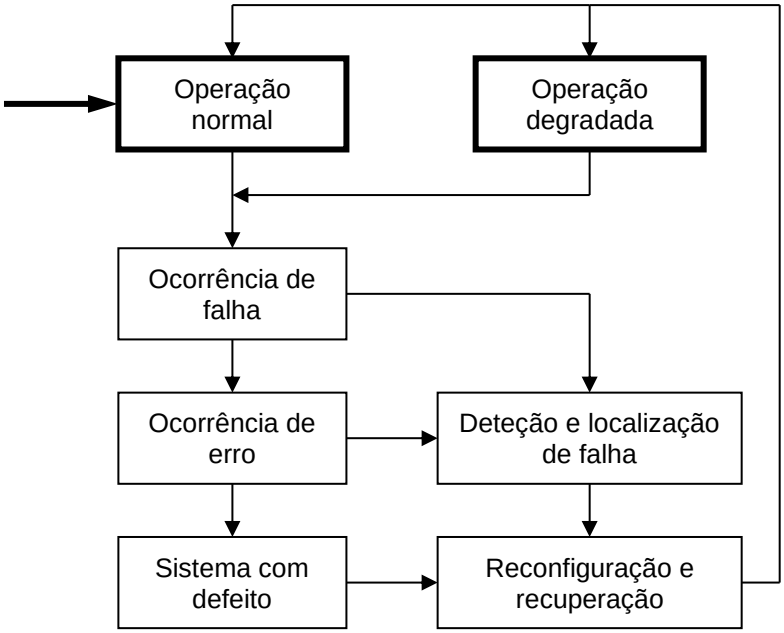


Figura 3-5. Estados de um sistema com redundância dinâmica (Weber, 2009)

3.2.3 Sistemas em standby (Standby spares)

Um dos sistemas mais comuns de redundância dinâmica são os sistemas em *standby*, Figura 3-6. Nestes os componentes alternativos, em *standby*, encontram-se a trabalhar em paralelo com o módulo principal comunicando, também eles, com o elemento detetor de falhas. Na ausência de falhas o sistema processa a informação proveniente do módulo principal (módulo 1) enviando-a para o comutador o qual é controlado pelo módulo de deteção de falhas. No caso de ser detetada uma falha o comutador reconfigura o sistema recebendo os sinais do ou dos módulos alternativos.

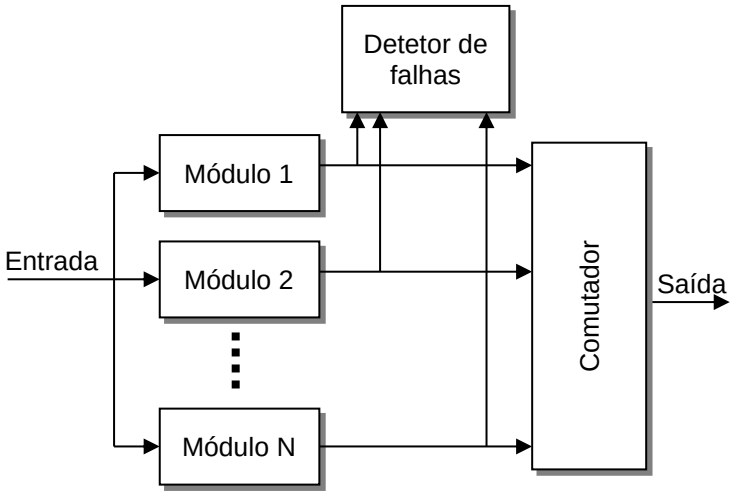


Figura 3-6. Sistema standby com N módulos

Detetada a falha o sistema remove o módulo em falha do sistema causando uma pequena descontinuidade do serviço, enquanto as saídas são comutadas. Para minimizar este efeito podemos implementar sistemas chamados de *hot standby,* ou seja, módulos que trabalham continuamente e em paralelo com o módulo principal, módulos que se encontram continuamente alimentados. Esta abordagem garante a minimização dos tempos de comutação, mas requer do módulo alternativo uma disponibilidade maior e permanente encontrando-se sujeito às mesmas condições de trabalho do módulo principal.

Como alternativa a esta metodologia podemos usar o sistema *cold standby*, módulos que não se encontram continuamente alimentados. Estes só entrarão em funcionamento quando for fornecida a alimentação ao módulo aumentando-se assim a sua vida útil, isto é, aumento do seu ciclo de vida.

3.2.4 Pares de autoavaliação (Self-checking pairs)

Os pares de autoavaliação são outro exemplo da redundância dinâmica. Dois módulos idênticos recebem a mesma entrada comparando-se as saídas, ver Figura 3-7. Um dos módulos envia diretamente para o módulo seguinte o resultado do seu processamento enquanto o resultado do comparador é utilizado para detetar uma falha (Storey, 1996).

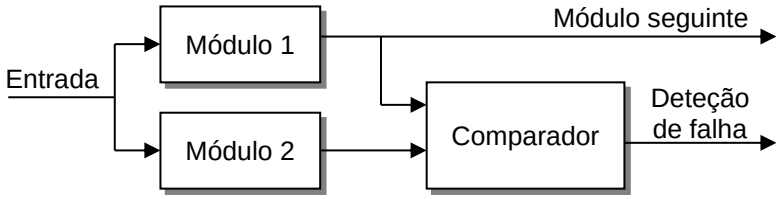


Figura 3-7. Par de autoavaliação (Self-checking pairs) (adaptada de Storey, 1996)

O comparador pode ser implementado quer em *software* quer em *hardware*. Uma simples operação de comparação, realizada por *hardware*, utilizando portas lógicas tais como XOR e OR pode ser utilizada para deteção de falhas.

3.3 Redundância de software

O desenvolvimento de *software* é um processo que, normalmente, conduz à produção de erros. Por isso, deve-se assumir que qualquer programa, independentemente da sua dimensão, possui falhas. Se, por um lado, a simples replicação de componentes idênticos é uma estratégia de deteção e mascaramento de falhas em *hardware* e em *software* esta torna-se inútil. Evidentemente que a duplicação de módulos de *hardware* promove a redundância do sistema, no entanto, duplica também os programas associados aos mesmos. A duplicação de módulos de *software* idênticos promove a difusão de falhas para todas as cópias idênticas, pelo que não basta copiar um programa e executá-lo em diferentes módulos ou em diferentes tempos, pois as falhas ocorrerão de forma idêntica para os mesmos dados de entrada.

Para combater este problema é comum recorrer a estratégias de diversidade com vários níveis de decomposição. Estas podem ser aplicadas em diversos níveis do sistema tais como – *hardware*, aplicações de *software*, sistema de *software*, operadores bem como as interfaces entre componentes. Em sistemas de segurança crítica (*Safty-critical systems*) é comum utilizar-se uma abordagem com multi-sistemas em que a replicação se faz ao nível do *hardware* e do *software* – sistemas computadorizados de controlo de voo do Boeing 777 ou 787 (Rierson, 2017) e Airbus A-310 entre outros.

As técnicas de diversidade na conceção mais utilizadas na tolerância a falhas para o desenvolvimento de *software* são baseadas na Programação N-Versões (NVP – *N-Version Programming*) (Avizienis, 1997), nos Blocos de Recuperação (RcB – *Recovery Blocks*) (Horning, 1974) e na Programação *n* Autocontrolos (NSCP – *N Self-Checking Programming*) (Laprie *et al.*, 1995).

3.3.1 Diversidade na conceção

A diversidade de conceção ou implementação consiste no desenvolvimento de diferentes algoritmos de implementação da mesma função, com o fim específico, de evitar que uma dada falha possa estar presente em diferentes implementações onde a resposta do sistema é determinada por votação (Xie *et al.*, 2014).

3.3.1.1 Programação n-versões

A diversidade de conceção pode ser utilizada em todas as fases do desenvolvimento de um programa, desde a especificação até ao teste, dependendo do tipo de erro que se pretende detetar, Programação N-Versões (Avizienis, 1995; Chen e Avizienis, 1978). Esta metodologia poderá também ser utilizada com o fim específico da prevenção de falhas na fase de teste, nesta situação é escolhida a versão do *software* em que foi detetada menor ocorrência de erros integrando-se esta última no sistema. Os passos para a implementação da metodologia NVP são apresentados por Tian (2005).

A redundância obtida por esta metodologia assenta no facto de que os módulos redundantes são desenvolvidos, implementados, verificados e validados por diferentes equipas com o fim de melhorar a tolerância a falhas do sistema considerando-o como um todo (Hosek e Cadar, 2015), Figura 3-8. As diferentes versões correm em paralelo e terão de ser sincronizadas por um mecanismo de decisão.

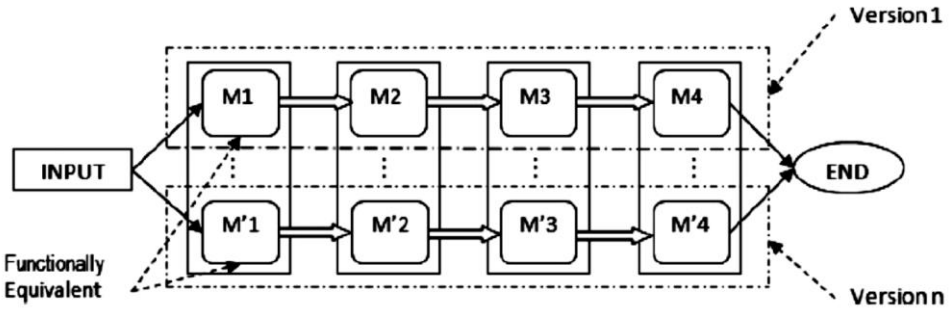


Figura 3-8. Framework de modelação NVP (Xie, 2014)

Note-se, no entanto, que uma abordagem aos problemas de tolerância a falhas baseada na diversidade é uma solução cara devido ao aumento dos custos de desenvolvimento e manutenção, à complexidade de sincronização das diversas versões bem como aos problemas associados à obtenção da correlação das fontes de erro (Weber, 2009) pelo que esta metodologia deverá ser apenas utilizada em sistemas críticos.

Por outro lado, enquanto que pode ser provado formalmente que a redundância de elementos de *hardware* aumenta a fiabilidade do sistema, tal prova não existe para a diversidade em *software*. Vários fatores influenciam a eficácia da programação diversificada: as equipas podem trocar algoritmos entre si, os membros das equipas podem, por formação, tender a adotar os mesmos métodos de desenvolvimento, ou as equipas podem procurar suporte nas mesmas fontes. Qualquer uma dessas correlações imprevisíveis constitui uma fonte potencial de erros.

A juntar a todas estas questões temos ainda de considerar a fiabilidade ou confiança a depositar no elemento votador/decisor pois este pode ser o único elemento com elevada probabilidade de falha em todo o sistema. O sistema fica totalmente dependente da fiabilidade do votador (Hu *et al.*,2017).

3.3.1.2 Blocos de Recuperação

Os Blocos de Recuperação (RcB), introduzidos em 1974 por Horning (1974), são categorizados como uma técnica dinâmica de redundância a falhas em que a escolha da variável de saída é realizada durante a execução do programa com base no resultado do teste de aceitação (AT – *Acceptance Test*). Nesta opção o *hardware* tolerante a falhas é relacionado com a arquitetura RcB segundo uma estrutura em *standby* (Zheng e Lyu, 2015).

O RcB (Randell e Xu, 1994; Kim e Welch, 1989) usa um AT com recuperação para trás pelo que a maioria das funções do programa podem ser realizadas usando diferentes algoritmos. Daí que a implementação de variantes do algoritmo apresente diferentes graus de eficiência e de gestão da memória, diferentes tempos de execução e confiança no funcionamento. O módulo de maior confiança encontra-se localizado na primeira série, versão primária, enquanto os menos eficientes, versões secundárias, são colocados após a primeira série. Nesta abordagem os módulos secundários só serão chamados aquando da deteção de erros no programa principal. Estes são executados e testados um a um até que o módulo principal passe no teste de aceitação tolerando *n-1* falhas independentes nas *n* versões, Figura 3-9.

Nas aplicações de tempo real os blocos de recuperação incluem um sistema do tipo *watchdog* temporizado.

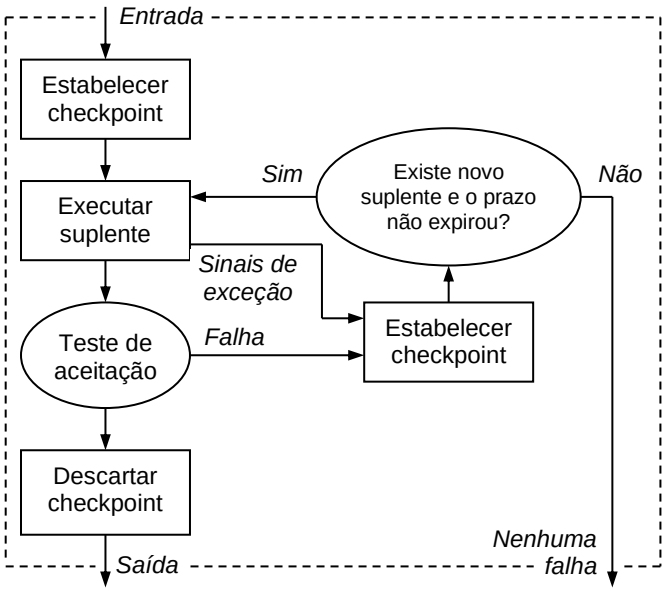


Figura 3-9. Funcionamento dos blocos de recuperação (adaptada de Randell e Xu, 1994)

3.3.1.3 Programação n Autocontrolos

A programação NSCP (Laprie *et al.*, 1995) consiste na utilização de várias versões de *software* combinadas com variações estruturais de Blocos de Recuperação e de Programação N-Versões com redundância em *standby*. Na Figura 3-10 apresenta-se a programação *n* Autocontrolos utilizando testes de aceitação. Nesta metodologia as versões e os testes são desenvolvidos independentemente dos requisitos de aceitação. A utilização dos testes de aceitação separados para cada uma das versões evidencia uma grande diferença entre a abordagem do modelo *n* Autocontrolos e a abordagem dos blocos de recuperação. À semelhante dos blocos de

recuperação, a execução das versões e dos seus testes de aceitação podem ser feitos sequencialmente ou em paralelo, mas o valor de saída é tomado a partir da versão com classificação mais alta que passe no teste de aceitação (Laprie *et al.*, 1990). Sequencialmente a execução requer a utilização de pontos de verificação enquanto paralelamente a execução requer a utilização de entradas e a coerência do estado dos algoritmos.

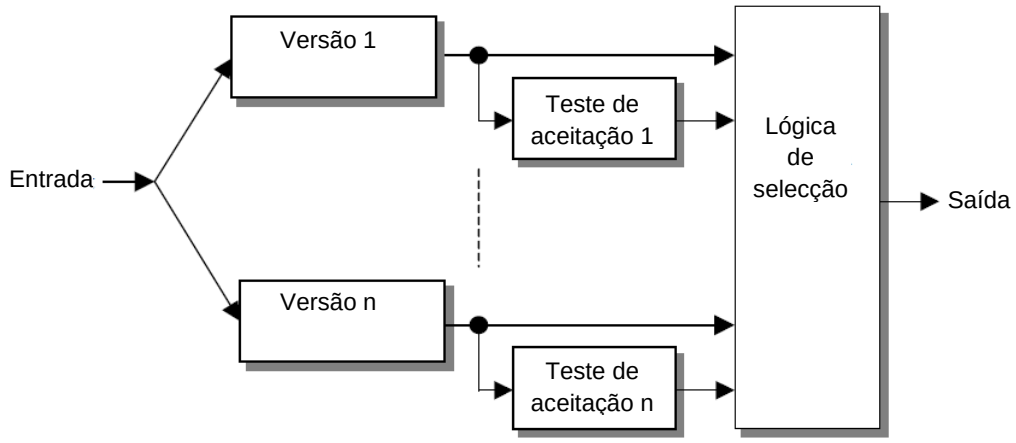


Figura 3-10. Programação n Autocontrolos utilizando testes de aceitação (adaptada de Pomales, 2000)

Uma outra abordagem à tolerância a falhas baseada neste método de programação consiste na utilização de sistemas de comparação para a deteção do erro de acordo com o esquema mostrado na figura seguinte, Figura 3-11.

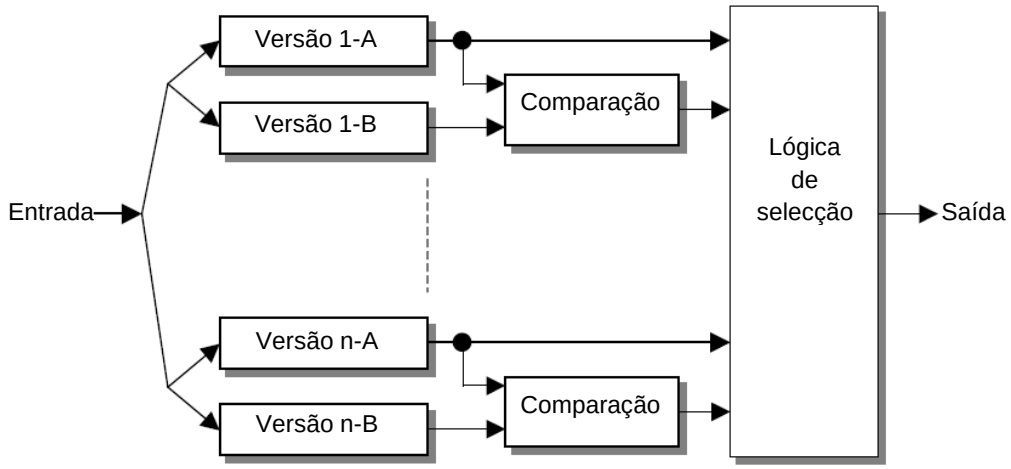


Figura 3-11. Programação n Autocontrolos utilizando Comparação (adaptada de Pomales, 2000)

Esta metodologia possui também, à semelhante da Programação N-Versões, um modelo que tira partido da utilização de um algoritmo de decisão independente para selecionar a saída correta. Esta variação teórica de programação de autocontrolos tem a vulnerabilidade de encontrar situações em que vários pares possam passar nas suas comparações, cada um com diferentes resultados. Este facto deve ser sempre considerado pelo que deve ser definida uma política de decisão, lógica de decisão, adequada à resolução deste problema durante o desenvolvimento do projeto.

3.3.2 Diversidade dos dados

A técnica da diversidade de dados é utilizada como complemento, e não como substituto, da diversidade de conceção/implementação (Ammann e Knight, 1988). A utilização da diversidade de dados passa pela introdução de um conjunto de dados, de alguma forma relacionados com o código/programa/algoritmo, e pela utilização de um decisor que determina o resultado final. Desta forma a diversidade de dados pode ser conseguida através da diversidade de sensores, ao nível do *hardware*, ou por meio de algoritmos de Reexpressão dos dados (DRA – *Data Re-expression Algorithm*) ao nível do *software*.

3.3.2.1 Algoritmos de Re-expressão dos Dados

Os algoritmos de Re-expressão dos dados são utilizados para obter dados de entrada diversificados pela geração de conjuntos de dados de entrada logicamente equivalentes. Isto é, a capacidade de tolerar falhas, dependendo da capacidade do DRA, de produzir dados fora da região de falhas. Um algoritmo de Re-expressão, *R*, transforma a entrada *x* original numa nova entrada *y = R(x)*. A entrada *y* pode ser próxima de *x* ou conter, de uma forma diferente, informações sobre *x*. O programa *P* e *R* determinam a relação entre *P(x)* e *P(y)*.

Na Figura 3-12 apresentam-se os três conjuntos de saídas possíveis no espaço de saída resultante do valor atribuído a *x* (Ammann e Knight, 1988).

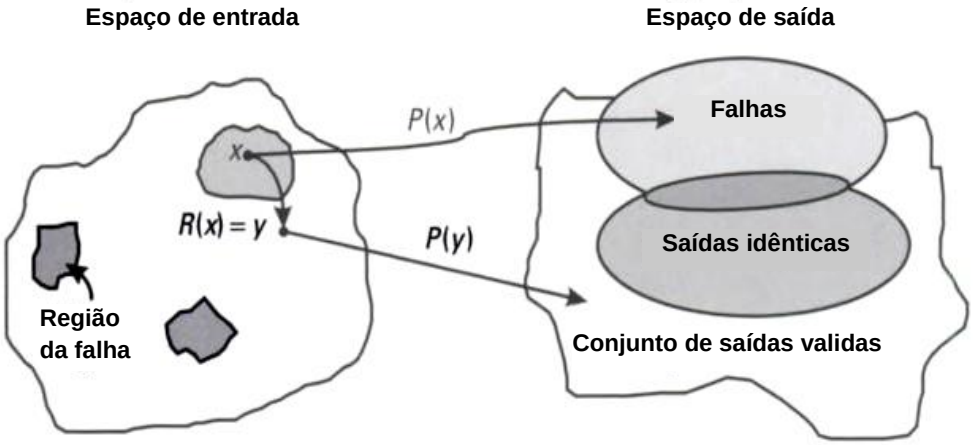


Figura 3-12. Re-expressão de Dados. Conjunto de saídas no espaço definido por x (adaptada de Pullum, 2001)

Na Figura 3-13 apresenta-se um Algoritmo de Re-expressão dos Dados básico. Este encontra-se intimamente dependente da aplicação.

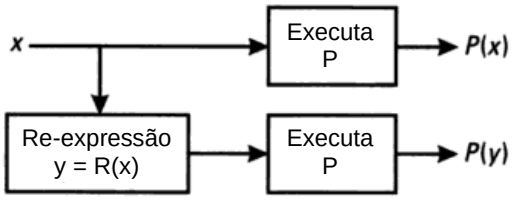


Figura 3-13. Método de Re-expressão dos Dados básico (adaptada de Ammann e Knight, 1988)

Uma outra abordagem à re-expressão dos dados passa pela adoção de um sistema de ajustamento pós execução, ver Figura 3-14. Este DRA permite produzir maior diversidade de entradas do que as produzidas quando se utiliza uma estrutura básica. Assim, a correção *A* é

realizada em *P(y)* de modo a compensar a distorção produzida pelo algoritmo de re-expressão *R*. Se a distorção induzida por *R* poder ser removida após execução, então esta abordagem permite grandes alterações nas saídas e permite que as cópias do programa operem em regiões amplamente separadas do espaço de entrada.

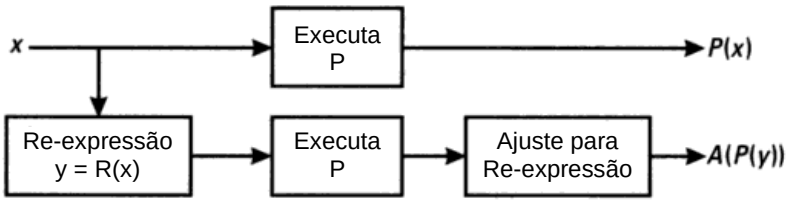


Figura 3-14. Re-expressão dos dados com ajustamento pós execução (adaptada de Ammann e Knight, 1988)

Uma terceira variante desta metodologia pode ser vista na Figura 3-15. Esta é realizada através da decomposição e recombinação do programa. Uma entrada *x* é decomposta num conjunto de fatores relacionados com o programa e, em seguida, é executada em cada um destes fatores relacionados. Os resultados são depois recombinados.

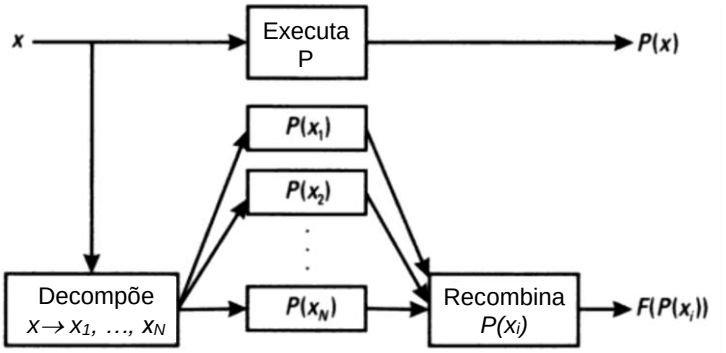


Figura 3-15. Re-expressão de dados via decomposição e recombinação (adaptada de Ammann e Knight, 1988)

Um dos exemplos de aplicação da metodologia DRA refere-se ao processamento e correção dos valores de entrada de sensores utilizados em sistemas de controlo devido, essencialmente, à introdução de ruídos de baixa intensidade nos sinais lidos. Uma outra aplicação desta metodologia prende-se com os pares (*x, y*) representando as faixas de radar. Nesta assume-se que os valores obtidos pelo radar estão limitados na sua precisão. A DRA move todos os pontos (*x, y*) para um local aleatório na linha de circunferência de um circulo centrado em (*x, y*) e para alguns pequenos círculos, com raio fixo, como se mostra na Figura 3-16.

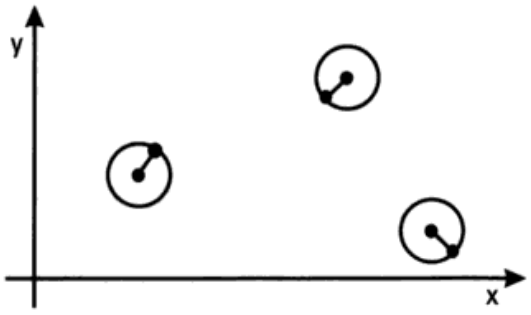


Figura 3-16. Re-expressão de três pontos de radar (adaptada de Ammann e Knight, 1988)

3.3.3 Diversidade temporal

A diversidade temporal é realizada através da produção de eventos em diferentes instantes. Esta pode ser implementada através do desfasamento do tempo de início ou pela utilização de dados produzidos ou lidos em diferentes instantes. Esta abordagem pode ser útil para ultrapassar falhas transientes dado que os problemas causados pelas condições temporais numa execução podem desaparecer nas execuções seguintes. Na Figura 3-17 apresentam-se eventos calendarizados para um sistema simples de diversidade temporal.

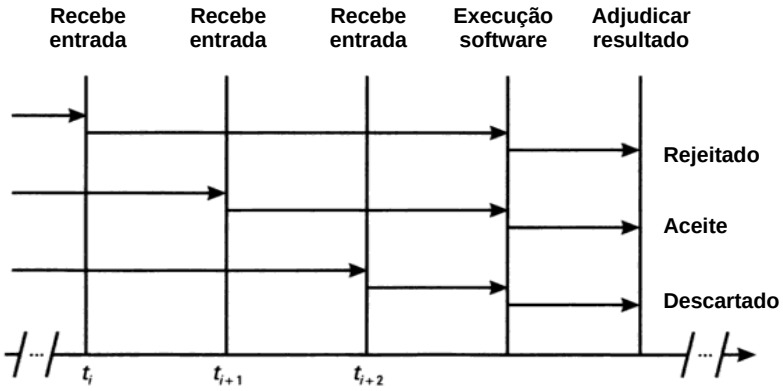


Figura 3-17. Diversidade temporal (adaptada de Pullum, 2001)

3.3.4 Redundância de informação

A redundância de informação consiste na utilização de *bits* de informação adicionais à efetivamente requerida com o objetivo final de armazenar ou transmitir informação necessária para a deteção de erros ou para mascarar falhas. Um exemplo desta utilização é o *bit* de paridade, onde para cada *n bits* são armazenados *n+1 bits*. Estes são utilizados na deteção de falhas simples. Outro exemplo de código usado para a correção e deteção de falhas é o *checksums*, isto é, a transmissão de todas as palavras juntamente com o resultado da sua soma binária.

O mascaramento de erros usando redundância de informação sustenta a sua ação de correção de erros num código do tipo ECC (*Error Correction Code*) que é usado, normalmente, na transferência de informação entre memórias e processadores. Exemplos destas ações são os códigos de Hamming (Fiedler, 2004) que combinam os *bits* de paridade tanto para deteção como para correção do erro.

3.3.5 Redundância temporal

A redundância temporal usa o tempo de computação, para além do exigido, para executar uma determinada função com o objetivo de detetar ou tolerar falhas, ou seja, repete a computação recalculando e comparando os resultados obtidos. A utilização desta metodologia evita o custo de utilização de *hardware* adicional, aumentando-se, no entanto, o tempo necessário para realizar uma computação. Esta metodologia só poderá ser usada em sistemas onde o tempo não é crítico. As aplicações mais usuais da redundância temporal são:

- **Deteção de falhas transitórias:** é realizada repetindo a computação. A obtenção de resultados diferentes é uma forte indicação de uma falha transitória. Depreende-se desta situação que a utilização desta metodologia para falhas permanentes não é adequada pois, os resultados obtidos com a repetição do cálculo serão sempre iguais.

- **Deteção de falhas permanentes:** repete-se a computação com dados codificados e descodifica-se o resultado antes da comparação com o resultado anterior. Esta metodologia permite identificar falhas permanentes.

3.4 Conclusão

Neste capítulo faz-se uma compilação das técnicas mais utilizadas na tolerância a falhas em sistemas computacionais. Neste são abordadas diversas técnica de tolerância a falhas uma vez que estas são necessárias à sua deteção e eliminação pois, é sabido que todos os sistemas computacionais poderão conter falhas uma vez que estas são intrínsecas ao processo de desenvolvimento de *software* e à construção dos diversos equipamentos (*hardware*). Assim, a tolerância a falhas, ou redundância como é normalmente designada pela indústria, assenta o seu campo de atuação sobre todos os paços de desenvolvimento e de implementação de um sistema de controlo, ou seja, ao nível do *software* e do *hardware*.

Neste sentido a redundância pode ser implementada ao nível do *hardware*, redundância passiva ou mascaramento das falhas, normalmente realizada com recurso à redundância modular tripla (TMR), com ou sem votadores triplos. A redundância de *hardware* poderá ainda ser realizada dinamicamente, modo ativo, centrando a sua ação na recuperação e na localização da falha tendo por base, na sua maioria, os sistemas em *standby*. A redundância, ou tolerância a falhas, é também realizada ao nível do *software* tendo como base de aplicação a diversidade de conceção, diferentes algoritmos para a mesma função e/ou por blocos de recuperação (RcB), pela diversidade de dados ou temporal. Assim, qualquer que seja a metodologia utilizada, independentemente dos custos associados e dos tempos de execução, tem como objetivo final a eliminação de falhas e, como tal, o aumento da confiança no funcionamento.

Capítulo 4

Sistemas Distribuídos

4.1 Introdução

Com o advento da microeletrónica e a evolução da informática é possível produzir-se computadores cada vez mais potentes e a preços mais acessíveis. Foi sem dúvida por este facto que se verificou um grande impulso na utilização massiva dos sistemas informáticos. Os computadores saíram das salas isoladas, às quais muito poucas pessoas tinham acesso e onde a execução de um qualquer programa estava condicionada a pequenas tarefas sem interação do seu autor, passando a ocupar um espaço bem definido no nosso dia-a-dia e na comunicação global.

Várias foram as tentativas desenvolvidas ao longo dos anos para aumentar a interação dos utilizadores dos sistemas e os computadores ditos centrais. Assim, no início dos anos 60 com a tecnologia "*time-sharing*", começou-se por ligar, por meio de linhas de comunicação de baixa velocidade, um conjunto de terminais a um computador central. Este avanço tecnológico permitiu que os operadores dos terminais pudessem interagir com os seus programas. Nos anos 70, com a utilização do modelo de referência OSI (*Open System Interconnection*) (Zimmermann, 1980) na arquitetura de comunicação, obtinha-se uma capacidade de processamento e de partilha dos periféricos muito superiores à disponibilizada pelos *mainframes*. Outras vantagens que se advinham da distribuição de recursos diziam respeito à facilidade de crescimento da rede, estendidas em função das necessidades de processamento, e à modularidade natural das mesmas, onda a falha dum equipamento tinha efeitos bastante limitados no processamento global.

Atualmente, as vantagens dos sistemas distribuídos, sistemas compostos por vários computadores autónomos, que comunicam entre si através de uma rede de computadores apresentando-se como um sistema único, apresentam uma evidência inegável onde as aplicações mais diversas, desde a automação, o controlo de processo, passando pela gestão bancária, reservas aéreas, etc., se encontram interligadas globalmente num mundo que, em termos de informação, se torna um gigantesco sistema distribuído.

Por outro lado, a utilização de redes de comunicação para a interligação de computadores contribui para o aumento da confiança no funcionamento do sistema como um todo. Neste sentido, a distribuição dos recursos por vários equipamentos contribui, de forma significativa, para a continuidade do funcionamento do sistema mesmo na presença de elementos em falha. As redes de comunicação, dada a sua topologia e dimensões de implantação, devem ser objeto de classificação de acordo com a sua dimensão. Assim, uma pequena extensão de rede que

interliga diversos computadores cingidos a uma sala ou um prédio é classificada como uma Rede de Área Local (LAN – *Local Area Network*). Se a interligação dos sistemas computadorizados se estende a toda a área de um Campus a rede recebe a denominação de Rede de Área de Campus (CAN – *Campus Area Network*). Por outro lado, se estendermos a nossa rede a toda a cidade podemos dizer que estamos na presença de uma Rede de Área Metropolitana (MAN – *Metropolitan Area Network*). Alargando ainda mais o alcance da nossa rede, e desta vez alargando-a a várias cidades de uma dada região, a rede denominar-se-á de Rede de Longa Distância (WAN – *Wide Area Network*) (Marques e Guedes, 2003; Stemmer, 2001). Por fim, não se pode deixar de referir a rede mundial de computadores que atualmente nos interliga a todos, ou seja a WWW – *World Wide Web* (Berners-Lee e Cailliau, 1990) também designada de *Web*, que em português assume o nome de Rede de Alcance Mundial, interligando milhares de computadores, partilhando recursos e disponibilizando conteúdos, via Internet.

4.2 Redes industriais

Nos últimos anos assistimos a um esforço considerável das empresas no sentido de implementar arquiteturas de comunicação que permitam controlar e monitorizar os elementos que compõem a estrutura industrial bem como proporcionar ferramentas para a tomada de decisão. Por outro lado, a competitividade requerida ao setor produtivo exige que os sensores mais simples ou os processadores industriais mais complexos e inteligentes façam parte de uma cadeia funcional na qual conceitos como o controlo descentralizado, a supervisão, a gestão de informação, a integração das pessoas bem como a gestão e as estratégias de manutenção tenham cada vez maior importância. Esta integração conduzirá a uma melhoria dos recursos disponíveis para a tomada de decisões quer ao nível da gestão, por meio do acesso à informação em tempo real, quer dos operadores (Koenig *et al.*, 2019).

As redes industriais permitem-nos colmatar as necessidades de comunicação, entre sistemas mais flexíveis e cada vez mais rápidos, compartilhando e interligando os diversos recursos de campo com os elementos de controlo. Há ainda, ao nível do processo, que garantir que os tempos de resposta do sistema respeitam os patamares definidos, que a informação, a confiança no funcionamento, a confiança na informação e a flexibilidade dos mesmos nos permita subir na pirâmide de automação (Figura 4-1) onde os controladores se integram com outros equipamentos e computadores (IEC 62264, 2013). A interligação destes meios proporcionará uma ligação transparente entre os vários níveis de hierarquia do sistema, facilitando a supervisão e a coordenação de processos paralelos ou independentes. No topo desta organização, pirâmide de automatização, a informação converte-se num dos valores mais importantes para otimização do processo global proporcionando-nos ferramentas para a tomada de decisões com vista à melhoria de produção.

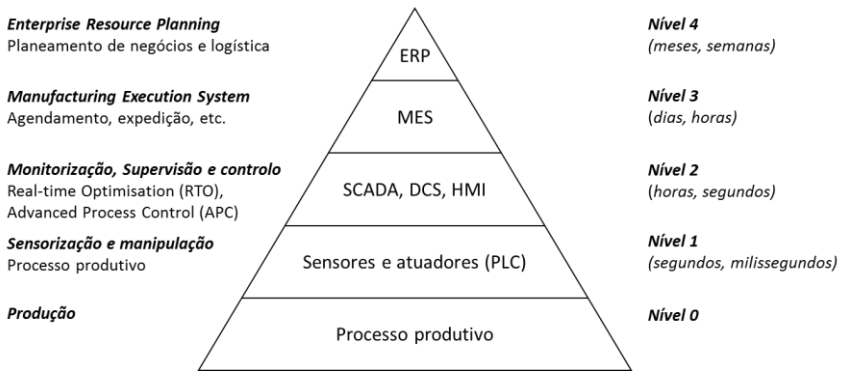


Figura 4-1. Pirâmide de automação, hierarquização

Nos últimos anos tem-se realizado um esforço considerável no sentido de definir uma arquitetura de comunicação industrial que possa responder às necessidades, características e requisitos de integração entre as redes industriais e as ditas redes de escritório. Assim, em 1980, com o desenvolvimento dos sensores inteligentes começaram a surgir novas arquiteturas baseadas em barramento de campo *Fieldbus*, como por exemplo: ProfiBus, WorldFIP e P-Net, (Koubias e Papadopoulos, 1995), Interbus-S e DeviceNet (Mahalik e Lee, 2002), CANopen (Pfeiffer *et al.*, 2008), dedicado ao nível de campo, integrando vários *tipos* de instrumentação digital, visando a otimização a performance dos sistemas. Mais recentemente surgiu o protocolo de comunicação Profinet (Neumann, 2007) permitindo a ligação dos elementos de campo com o nível de decisão, ou seja, a ligação de todos os elementos de campo com o nível de informação onde as decisões são tomadas. A utilização destes meios permite a comunicação rápida e fiável entre os equipamentos e o uso de mecanismos padronizados como: Asi, Modbus, CANopen, Profibus, Device Net, Ethernet TCP/IP, RFID entre outros, são hoje-em-dia, fatores indispensáveis no conceito de produtividade industrial. Com esta proliferação de redes de comunicação, nomeadamente os sistemas WIFI e a Ethernet, abriram-se as portas para a mais recente revolução industrial, Indústria 4.0, tendo como base os conceitos IoT – *Internet of Things*, IIoT – *Industrial Internet of Things* ou IoTS - *Internet of Things and Services* e o M2M – *Machine to Machine* suportados por redes de Internet industrial que associam as redes comerciais a todos os aspetos das fábricas inteligentes e aos ciclos de vida dos produtos inteligentes, Figura 4-2. Os serviços fornecidos por essas plataformas irão conectar pessoas, objetos e sistemas uns aos outros (Kagermann *et al.*, 2013; Beatrice *et al.*, 2018).

Figura 4-2. Conceito de Indústria 4.0 (Ribas, 2017)

As redes apresentam aspetos interessantes que as tornam soluções muito adequadas às comunicações industriais. No entanto, estas possuem alguns problemas importantes normalmente associados à difusão e ao método de acesso ao meio, uma vez que vários equipamentos deverão trocar informação num dado instante. A decisão de quem vai ter o direito de usar o meio para o envio de uma mensagem não é uma tarefa fácil. Neste sentido os protocolos de acesso ao meio assumem um papel preponderante nos tempos de entrega de uma dada mensagem via rede. Na verdade, as aplicações industriais necessitam frequentemente de sistemas de controlo e de supervisão onde a resposta temporal é essencial. Isto torna-se mais evidente nos sistemas com requisitos de Tempo Real (TR) para os quais é requerida uma reação a estímulos, físicos ou lógicos, dentro de intervalos de tempo impostos pelo ambiente a controlar. Por outro lado, quanto maior for o número de estações e quanto maior for o tráfego médio de mensagens na rede, maior a probabilidade de ocorrência de colisões de forma que o tempo de reação aumenta consideravelmente e, por este facto, não poderá ser determinado com precisão. Para

muitas aplicações industriais, especialmente aquelas com requisitos de tempo real, é importante utilizar redes com protocolos determinísticos para garantir o acesso ao meio.

4.3 Paradigmas dos sistemas distribuídos

Nos sistemas distribuídos, de igual modo que entre os humanos, é necessário atribuir nomes ou outra forma de identificação a cada posto de trabalho identificando-o, normalmente, com o seu endereço. Os endereços utilizados nos sistemas distribuídos são nomes preponderantes para garantir a comunicação entre postos. Estes podem ser referenciados como, por exemplo, um endereço de *email* ou um identificador IP (*Internet Protocolo*) vulgarmente utilizado no protocolo de comunicação TCP (*Transmission Control Protocol*). Neste sentido são dados nomes a todos os equipamentos que constituem o sistema onde, as impressoras, as caixas de correio, os computadores, os servidores, etc., todos possuem um nome de modo que as mensagens possam ser executadas e transmitidas entre eles.

4.3.1 Envio e receção de mensagens

Nos sistemas distribuídos a forma mais comum de interação passa pela troca de mensagens. Para que se possa trocar mensagens entre dois componentes do mesmo sistema devem ser selecionados os protocolos de comunicação e obter-se os endereços de cada componente. Assim, para que se possa garantir a troca de mensagens entre dois componentes duas primitivas devem ser tidas em conta, o envio e a receção. A primitiva envio é utilizada para requerer a transmissão da mensagem enquanto a primitiva receção é utilizada para identificar a receção da mensagem enviada. Poderá ser utilizada uma outra primitiva relacionada com o reconhecimento da correta receção duma mensagem (*acknowledeged-send, ack*).

Neste sentido a comunicação unidirecional poderá ser utilizada sempre que um nó precisa enviar informação para um componente e não está interessado em receber uma resposta, ou seja, envio de um evento notificando o componente. Esta abordagem à passagem de mensagens baseada na notificação tem um papel importante na programação baseada em eventos e por isso, é hoje em dia um estilo muito utilizado na programação de sistemas distribuídos. Assim sendo, nos sistemas distribuídos as mensagens são recebidas, de modo geral, pela ordem em que são enviadas, ou seja, as primeiras a chegar são as primeiras a sair, sistema FIFO (*Frist-In-First-Out*).

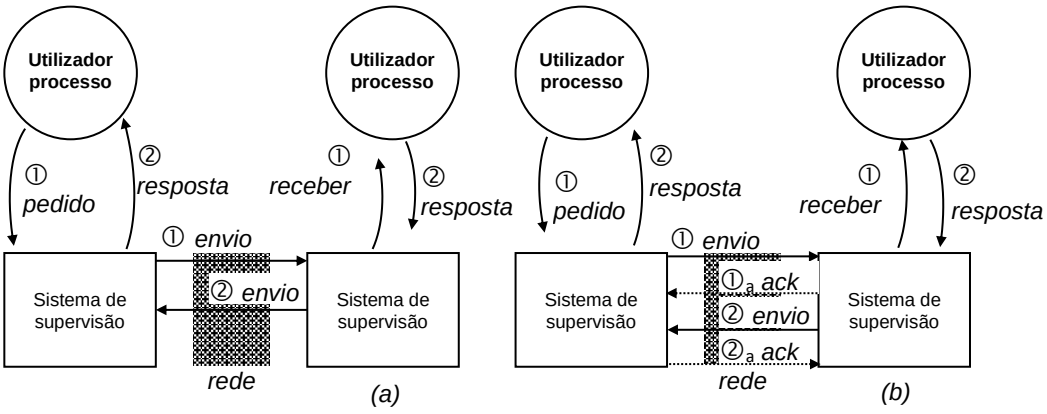


Figura 4-3. Operações remotas: (a) Pedido-Resposta; (b) Conhecimento (ack), (Veríssimo e Rodrigues, 2001)

Por outro lado, nos sistemas Cliente-Servidor, o cliente solicita uma operação remota, fazendo um pedido ao servidor remoto, ficando à espera de receber a confirmação do seu pedido de operação. Este confirmará a receção do pedido bem como o início do processamento. No

entanto, em casos em que o processamento possa demorar muito tempo, o servidor deve informar o cliente de que recebeu a mensagem, distinguindo a situação de processamento demorado de um pedido, duma eventual perda da mensagem. Esta informação torna-se vital para o cliente dado que não necessita de reenviar a mensagem para o servidor remoto e, por outro lado, este informa o cliente de que se encontra ativo e a processar o seu pedido, Figura 4-3.

4.3.2 Comunicações

As comunicações ponto-a-ponto são um caso particular das comunicações multiponto. Estas beneficiam da possibilidade de poderem entregar informações a múltiplos destinatários, simultaneamente, usufruindo dos benefícios de suporte da comunicação multiponto também chamada de *multicast*. Este tipo de comunicação utiliza um único ponto de ligação pelo que só poderá multiplicar a informação e enviá-la aos destinatários quando as ligações para os mesmos se dividirem em várias direções.

Um exemplo da aplicação deste protocolo é a distribuição de vídeo e de áudio enviada em paralelo para um grupo de recetores. Outra possível aplicação desta metodologia passa pelas aplicações cooperativas tal como as ferramentas de suporte à decisão onde a informação referente aos elementos do grupo é importante. Note-se, no entanto, que estas aplicações requerem não só um suporte *multicast,* mas também uma atualização da informação relativa aos elementos que constituem o grupo recetor, isto é, há a necessidade de conhecer os elementos do grupo, visibilidade do grupo. O protocolo é também muito usado na comunicação entre réplicas que executam as mesmas ações, ou seja, a implementação de um sistema de tolerância a falhas onde a aplicação, com a ajuda do sistema de comunicações de grupo, aborde de um modo transparente a existência das réplicas. Nesta situação diremos que estamos na presença de um grupo invisível. Esta abordagem levanta, no entanto, outros problemas dado que cada vez que se verifica uma alteração da vista do grupo esta tem de ser difundida para todos os membros e estes têm de concordar com o novo estado. Assim, o serviço prestado pelo grupo deve garantir duas propriedades fundamentais que, embora necessárias, são muito difíceis de alcançar na presença de falhas (Veríssimo e Rodrigues, 2001), esta são:

- **Exatidão** (*Accuracy*) – a informação disponibilizada deve refletir o cenário físico do sistema.
- **Consistência** (*Consistency*) – a informação disponibilizada terá de ser consistente para todo o processo.

Os grupos podem, por sua vez, ser fechados ou abertos. Num grupo fechado todos os elementos que constituem o grupo são pares pelo que cada membro pode enviar e receber mensagens. Num grupo aberto o elemento que envia mensagens não tem necessariamente de pertencer ao grupo para o qual está a enviar mensagens. Assim, estaremos na presença de *pares*, sempre que todos os elementos que constituem a vista do grupo enviam e recebem mensagens. Por outro lado, estaremos na presença de *expedidores* quando estes podem enviar mensagens para o grupo embora não recebam informações do mesmo.

4.3.2.1 Protocolo Multicast

O protocolo *multicast* é responsável pela difusão das mensagens para todos os elementos que constituem o grupo. A mensagem é difundida em simultâneo para todos os elementos mesmo em situações em que a mensagem necessite de se dividir ao longo do percurso, Figura 4-4.

Para que a mensagem possa chegar aos elementos recetores que constituem a vista do grupo deve-se ter em conta cinco serviços preponderantes para a ação de difusão das mesmas. Assim, e associado o serviço de difusão *multicast* temos os seguintes serviços: *routing*, *omission tolerance*

e o *flow-control*, os últimos dois serviços dizem respeito aos protocolos de ordenação e de recuperação de avarias (Veríssimo e Rodrigues, 2001).

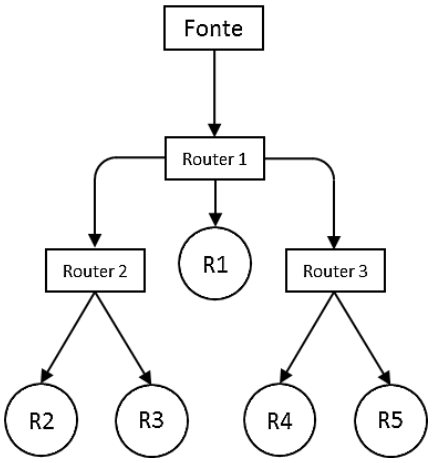


Figura 4-4. Árvore multicast

Se por um lado o *routing* deve encontrar um caminho que minimize o número de troca de mensagens bem como o tempo de processamento da informação nos vários elementos do grupo, *multicast latência* (Garcia-Molina e Spauter, 1991), a *omission tolerance* poderá ser admitida quando existir o conhecimento necessário para a deteção de erros e para o reenvio de mensagens perdidas. Neste sentido, esta informação poderá ser sempre disponibilizada e ser enviada sempre que a mensagem tenha sido recebida, conhecimento positivo, ou quando uma mensagem é perdida, conhecimento negativo. Esta última opção minimiza o tráfego de informação na rede.

4.3.3 Tempo e relógios locais

As regras do tempo estão intimamente ligadas com a ordenação, a sequência e o sincronismo pelo que podemos representar graficamente o tempo numa linha constituída por uma sequência de pontos ao longo da mesma, ou seja, um cronograma do tempo. Sendo o tempo um fator muito importante na vida quotidiana regendo cada um dos nossos passos este é também um fator importante nos sistemas computacionais. Convencionalmente o tempo é medido em segundos estando intimamente ligado à frequência de pulsação do átomo de Césio-133.

O uso do tempo nos sistemas computorizados prende-se com dois aspetos fundamentais: a gravação e observação da localização dos eventos no cronograma do tempo e o posicionamento futuro dos mesmos no referido cronograma. Nos sistemas distribuídos o primeiro diz respeito à gravação distribuída dos eventos enquanto que o segundo se refere à sincronização do sistema. Neste sentido poder-se-ia utilizar os relógios locais como forma de obtenção de sincronismo. Há, no entanto, que lembrar que os relógios locais são obtidos pela oscilação de um cristal de quartzo que controla o tempo de forma imperfeita devido, essencialmente, à granularidade (g) e à taxa de impulsos (ρ) do mesmo. Note-se também que o erro causado pelos impulsos ou incrementos do relógio (ρ) são normalmente insignificantes para tempos de execução na ordem dos $\rho_p \cong 10^{-5}$ microssegundos (ppm – partes por milhão). Embora os relógios locais apresentem alguns inconvenientes são normalmente utilizados como controladores do *timeout* e também para medir o *round-trip* da distribuição dos eventos *a* e *b (t(b)>t(a)).*

4.3.4 Relógio global

O relógio global utilizado em sistemas distribuídos é obtido pela sincronização, para o mesmo valor inicial, de todos os relógios locais, ou seja, cria-se em cada processo (p) um relógio virtual

(vc_p). Assim todos os processos são iniciados pelo vc_p (t_{init}) no entanto, e contando que todos os relógios dos processos sofrem desvios, há a necessidade de ressincronizar, periodicamente, todos os relógios.

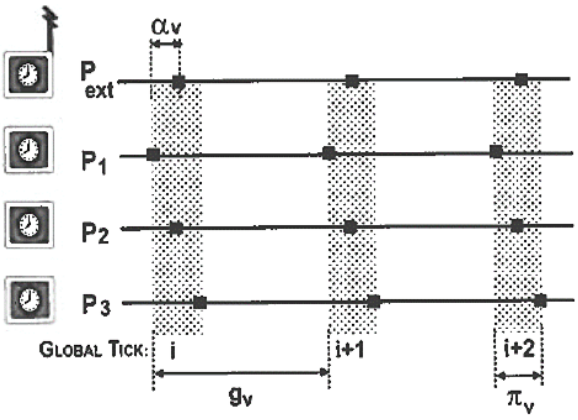


Figura 4-5. Propriedades do relógio global (Veríssimo e Rodrigues, 2001)

A sincronização pode ser realizada em função do tempo real, sincronização externa realizada em função do UTC (*Universal Time Coordinated*) ou em função do TAI (*Temps Atómic International*) recebido por difusão via sistema GPS (*Global Positioning System*). Na Figura 4-5 apresenta-se os parâmetros principais dum relógio global onde g_v é a granularidade, π_v é a precisão e α_v é a exatidão. Repare-se que neste exemplo a precisão corresponde ao máximo desvio detetado nos relógios, π_v em *i+2* por exemplo.

4.3.5 Sincronismo

A sincronização de sistemas de tempo real pode ser efetuada por protocolos que têm por base estruturas suportadas por impulsos temporizados. Embora estes protocolos possam apresentar instabilidade e ausência de firmeza continuam a delimitar o envio de informação para o sistema conforme foi previamente definido.

Os conceitos de sincronismo têm como base os tempos de transmissão δ_m, o número máximo de erros consecutivos durante a transmissão *k*, o tempo de retransmissão T_{tout} e o número de participantes *n*. Neste sentido, o desenvolvimento de algoritmos que visem obter o sincronismo, num qualquer sistema, necessitam garantir algumas características temporais, isto é, garantir a capacidade de executar ações relacionadas com o tempo ou com intervalos de tempo. Assim, os algoritmos devem conter instruções específicas de construção de modo que estes possam ser condicionados, durante a sua evolução, em função de condições temporais. Condições como "*at*", "*within*", "*until*", "*every*" ou "*after*" devem ser utilizadas para condicionar a evolução do algoritmo e, deste modo, obter-se sincronismo.

Uma forma recorrente de sincronização passa pela partilha da memória bem como da capacidade de passar a mensagem. A sincronização baseada em semáforos utiliza outros dois operadores que são enviados para os processos, estes são chamados de *espera* e de *sinal* respetivamente. O semáforo é predefinido, aquando da sua criação, para albergar um determinado número de elementos. Pela utilização da primitiva *wait (n)* o semáforo controla o número de elementos no seu interior decrementando *n* unidades, sempre que for possível realizar o pedido. No caso de o semáforo não possuir unidades suficientes para dar resposta ao pedido este ficara bloqueado indeterminadamente, não executando qualquer tarefa. A primitiva s*ignal* é uma primitiva não bloqueante utilizada para incrementar o número de unidades no interior do semáforo.

4.3.6 Ordenação

A ordenação de eventos nos sistemas distribuídos é uma prática corrente e muita das vezes necessária. Assim, se a ordenação dos eventos for realizada segundo a ordem em que estes acontecem poder-se-á implementar um sistema onde a evolução do sistema de computação será não determinístico, os eventos são ordenados após a sua ocorrência, ou seja, à posterior. Os eventos podem ainda ser escalonados tendo em conta uma ordenação predefinida na qual a ocorrência de um evento se encontra forçada à priori. Neste sentido poderíamos organizar os eventos segundo um critério FIFO onde as mensagens devem ser ordenadas antes do seu envio na mesma ordem em que estas foram enviadas, atrasadas ou perdidas.

Outra possível ordenação de eventos consiste em encontrar uma regra de precedências que satisfaça os critérios introduzidos por Lamport em 1978 (Lamport, 1978). Assim, o evento *a* precede o evento *b* se estes ocorrem em sequência dentro do mesmo processo. Esta é umas das condições para definir uma relação de precedência entre eventos a qual também pode ser representada pelo símbolo ($\rightarrow$). Neste sentido uma ordenação lógica poderá ser conseguida quando uma mensagem m_1 precede logicamente uma mensagem m_2, ($\overset{l}{\rightarrow}$), se: m_1 é enviada antes de m_2 pelo mesmo componente ou se m_1 é enviada para o componente que envia m_2 antes de este enviar m_2 ou ainda se existe uma mensagem m_3 tal que $m_1 \overset{l}{\rightarrow} m_3$ e $m_3 \overset{l}{\rightarrow} m_2$.

4.3.6.1 Ordenação total

A ordenação total é necessária em cenários em que é necessário manter o determinismo durante a execução de réplicas alocadas em diferentes nós de um qualquer sistema distribuído. A ordenação pode ainda ser necessária para assegurar que diferentes elementos do sistema tenham a mesma perceção dos estados e da evolução do mesmo.

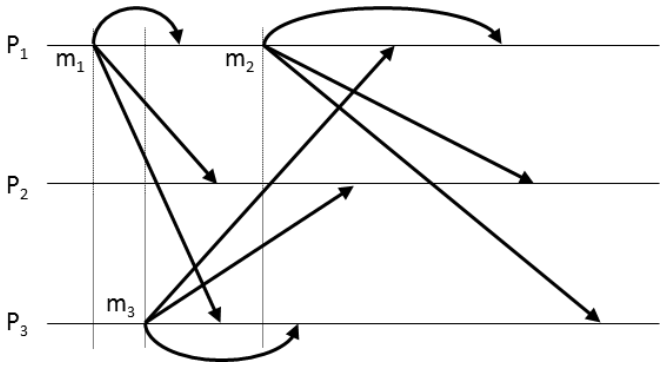


Figura 4-6. Ordenação total (Dias e Melo, 2002)

Isto quer dizer que se duas mensagens são enviadas para um qualquer elemento pertencente ao sistema elas são também enviadas, na mesma ordem, para todos os elementos do sistema, Figura 4-6. Assegura-se deste modo que os diferentes elementos tenham a mesma perceção da evolução e do estado do sistema, isto é, os diferentes elementos recebem as mesmas mensagens na mesma ordem.

Assim, quando estamos na presença de *n* processos onde é necessário trocar várias mensagens estas devem ser identificadas por um identificador, segundo uma determinada sequência, que será também enviado para todos os processos. A informação enviada é guardada num *array* chamado SENT controlado por um relógio local que poderá, também, controlar os eventos enviados. A informação trocada é representada e armazenada numa matriz designada de *matriz relógio*. Num processo em que só se trocam mensagens utilizando o protocolo de

comunicação *multicast* todos os elementos da mesma linha da matriz relógio possuem o mesmo valor pelo que a matriz reduz-se a um vetor de tamanho *n* designada de *vetor relógio*.

Várias podem ser as ferramentas utilizadas no reforço da ordenação total de dados embora, de entre estas, se possam salientar os algoritmos simétricos. No entanto, quando se pretende que uma mensagem seja recebida por todos os processos, de modo a que se possa garantir a ordenação total, esta deverá encontrar-se condicionada pela *boud rate* mais baixa de receção do processo. Assim, quando o critério de ordenação é o mesmo em todo o sistema, as mensagens serão enviadas na mesma ordem para todo o processo independentemente do tempo de entrega. Para melhorar estes resultados devemos garantir que as mensagens são enviadas em ritmos elevados e que os relógios estão sincronizados. Uma outra abordagem à ordenação total poderá passar pela seleção de um processo especial no sistema cuja função principal é ordenar todas as mensagens colocando-as segundo uma sequência. Todo os potências emissores enviam as mensagens para o sequenciador onde estas são afetadas por um número sequencial, retransmitindo-as para todos os recetores segundo a ordenação definida.

4.3.7 Consistência

O sistema diz-se consistente se não viola a integridade imposta pelas especificações. Por outro lado, num sistema computacional distribuído há, por vezes, a necessidade de garantir que determinadas propriedades globais se verificam, tal como a garantia de que o *token* continua a passar ou que não se verificou um bloqueio do sistema.

4.3.7.1 Consenso distribuído

O consenso é um dos problemas fundamentais da computação distribuída (Guerraoui e Schiper, 1996). Duma forma informal o objetivo do consenso é fazer com que um conjunto de processos concorde com um único valor que se encontra dependente do valor inicial de cada um dos nós participantes (Chen *et al.*, 2015).

4.3.7.2 Atomic broadcast

O protocolo *atomic broadcast* assegura que todas as mensagens são recebidas em todos os nós exatamente na mesma ordem em que foram enviadas. Isto é, o *atomic broadcast* combina uma transmissão confiável com uma ordenação total (Benz *et al.*, 2014). Assim, todos os nós devem concordar em primeiro lugar se a mensagem foi entregue e, em segundo lugar, na ordem da mensagem em relação às outras mensagens. As mensagens são enviadas para todos os nós que as guardam, até que uma nova vista do sistema seja novamente difundida pelos nós pertencentes ao grupo.

4.3.7.3 Replicação determinística

Informalmente o termo replicação consiste na possibilidade de manter cópias idênticas de um processo ou de dados em execução em vários locais. A replicação é a técnica utilizada por excelência em sistemas tolerantes a falhas, assim, se uma réplica falhar as outras permanecerão disponíveis para prestarem o serviço (Guerraoui e Schiper, 1997). A replicação pode não ser utilizada somente para a tolerância a falhas, esta pode também ser utilizada com o intuito de melhorar a performance dos sistemas através da colocação de réplicas de um dado serviço ou de dados perto do cliente final.

Por outro lado, há que assegurar que todas as réplicas executam o mesmo conjunto de instruções em simultâneo, isto é, sincronizadas. Deste modo, o estado de cada réplica pode ser comparado em qualquer altura e assegura-se que todas as réplicas recebem e enviam informação, aproximadamente, no mesmo instante. Para se garantir os mesmos estados para todas as réplicas é necessário que todas as réplicas recebam a mesma informação na mesma ordem e que o seu

código responda exatamente da mesma forma para as mesmas entradas. Esta metodologia deve ser utilizada em sistemas de pequena escala utilizando *hardware* que permita manter as réplicas num estado/evolução fechado.

4.3.8 Atomicidade

A atomicidade está relacionada com a indivisibilidade das operações. Dito de outra maneira, uma operação atómica não possui passos intermédios visíveis, isto é, a sequência necessária para a execução de um programa que se realiza em sequências de operações que por si só se tornam atómicas no sentido que a seguir se descreve: uma viagem do Porto (Portugal) até Orlando (USA) em que há a necessidade de comprar um bilhete até Londres, daqui até Chicago e por fim para Orlando. A necessidade de possuir todos estes bilhetes e esta sequência de viagens e não só uma parte para chegar ao destino, transforma toda esta sequência de reservas e viagens numa *sequência atómica*.

4.3.8.1 Atomicidade transacional

Atomicidade transacional é um paradigma que permite transformar sequências arbitrárias de operações em operações atómicas. Em termos de programação a sequência de operações necessita ser delimitada por um par de diretivas designadas de *início de transação* e *fim de transação*. As transações terminadas com sucesso são ditas de *commit*. As transações que não são terminadas são denominadas de *abort*.

Para conseguir estes objetivos, implementar transações atómicas, é necessário, em primeiro lugar, possuir um mecanismo de recuperação que permita manter o sistema no estado inicial sempre que a transação seja considerada como *abort*. Em segundo lugar, é necessário garantir que os resultados intermédios da transação não sejam tornados visíveis até que a transação se encontre no estado *commit*, ou seja, que a transação termine com sucesso. Os valores iniciais são guardados numa estrutura denominada de transação *log*. Sempre que se verifique a transação *abort* o estado original pode ser recuperado a partir do *log*.

4.3.8.2 Commitment distribuído atómico

Nos sistemas distribuídos, as transações podem aceder a dados em diferentes nós, estas são chamadas de *transações distribuídas*. Isto quer dizer que cada um dos nós pode estar a realizar uma operação, que pode ser a entrega de uma mensagem, até à validação de uma dada transação (Tanembaum e Steen, 2006).

Os sistemas *commits* distribuídos funcionam, geralmente, com a presença de um coordenador. Neste sentido, um dos processos participantes na transação deve ser designado como o coordenador do protocolo. Este envia uma mensagem *PREPARE* para todos os processos envolvidos (nós participantes) perguntando se todos os processos se encontram prontos para realizar o *commit*. Por outro lado, se um nó, durante o seu funcionamento, detetar um erro que impeça a execução da transação este enviará a mensagem *NOTOK (NOK)* para o coordenador abortando o *commit*. Caso contrário, o nó envia para o coordenador uma mensagem de *OK*.

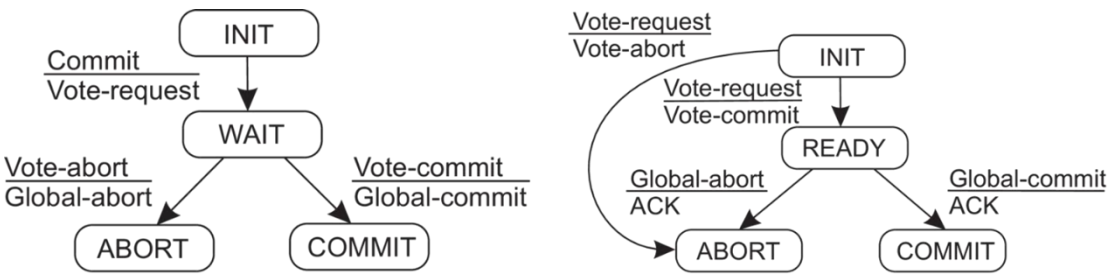


Figura 4-7. Máquina de estados do protocolo commit de duas fases (Holanda, 2007)

Na Figura 4-7 apresenta-se uma máquina de estados utilizada para o Protocolo *Commit* de Duas Fases. Visto mais em detalhe o protocolo *commit* funciona do seguinte modo:

1. O coordenador envia uma mensagem do tipo *VOTE_REQUEST* para todos os processos (nós).
2. Quando o processo recebe um *VOTE_REQUEST*, este pode reenviar duas respostas distintas. No caso em que reenvie um *VOTE_COMMIT*, este está indicando ao coordenador que está pronto para validar localmente a sua parte da transação. Por outro lado, se reenviar um *VOTE_ABORT*, significa que não poderá realizar a transação.
3. O coordenador recebe os votos de todos os processos e, se todos votaram a favor da validação da transação, o coordenador enviará uma mensagem de *GLOBAL_COMMIT*. Caso contrário, se um dos processos votar *abortar* a transação, o coordenador enviará uma mensagem de *GLOBAL_ABORT* para todos os processos.
4. Cada participante que votou pela validação espera pela reação final do coordenador. Se o participante recebe um GLOBAL_COMMIT, irá validar localmente a *commit* caso contrário, será abortada.

4.4 Modelos de interação

Os sistemas distribuídos podem ser apresentados, de acordo com a sua arquitetura de desenvolvimento e da sua distribuição, por modo a satisfazem diferentes necessidades. Assim aquando do seu desenvolvimento, o projetista poderá utilizar diversas ferramentas de desenvolvimento que se complementam de forma funcional ou de forma formal. Se por um lado tem que ter em conta as especificações da infraestrutura a desenvolver, da arquitetura e das interfaces necessárias para se obter determinadas funcionalidades por outro, teremos de ter em conta a sua implementação e validação de acordo com os paradigmas e os algoritmos de desenvolvimento adequados.

4.4.1 Infraestrutura

As infraestruturas do sistema estão relacionadas com o *hardware*, com a rede bem como com o sistema de suporte. As infraestruturas constituem a base de suporte através da qual o programador do sistema fornece ao mesmo as facilidades para a atividade de nível superior.

4.4.1.1 Infraestruturas de grande escala

Estas infraestruturas assumem grande relevância dada a grande dimensão que os sistemas distribuídos podem alcançar bem como a sua complexidade. Nestas estruturas há ainda que ter em conta o elevado número de nós bem como um sistema de comunicações demasiado complexo. Se por um lado o elevado número de nós ativos em simultâneo criam impactos na interação entre os mesmos por outro lado, as características do sistema de comunicações têm também elas um impacto importante na dimensão computacional do sistema. Este impacto traduz-se na extrema dificuldade que é transportar para um sistema de grande escala as condições de operação encontradas nos sistemas de pequena escala. Neste caso, a organização hierárquica e o agrupamento em nós parecem ser uma das técnicas estruturais mais promissoras para lidar com os problemas inerentes aos sistemas de grande escala.

4.4.1.2 Agrupamento em nós

Nesta abordagem vários processos desenvolvem interações entre os vários elementos do sistema alojados em diferentes hospedeiros, *hosts*. Os processos que constituem o sistema

encontram-se organizados segundo especialidades ligados a nós que por sua vez se interligam com todos os outros *hosts* e com os restantes processos do sistema. Os processos que pertencem a um grupo, alocado num dado nó, podem ser considerados como emissores, como recetores ou atuando como emissores/recetores interagindo com os outros nós via o nó a que se encontram ligados, Figura 4-8.

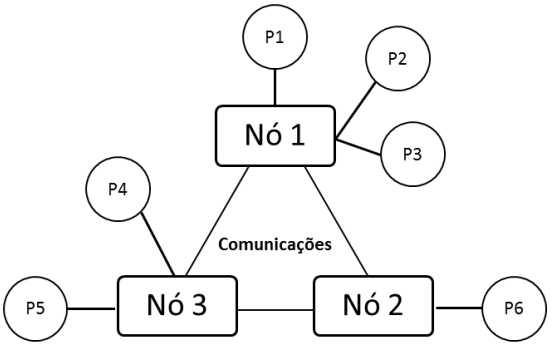


Figura 4-8. Computação distribuída, agrupamento em nós

Um aspeto importante a ter em conta nos sistemas distribuídos prende-se com as especificações da chamada linha do tempo, ou seja, do *timeliness*. Tradicionalmente, para lidar com os problemas relacionados com a semântica temporal, em sistemas de grande escala têm sido usados, tendencialmente, os chamados modelos assíncronos. Esta tendência é devida em parte ao facto dos sistemas assíncronos não contabilizarem ou demarcarem fronteiras temporais tais como a velocidade e os atrasos nas comunicações. Este modelo ignora o espaço temporal e como tal tem sido utilizado num grande número de aplicações para as quais a incerteza sobre a prestação de serviço é tolerada. No lado oposto a estes sistemas encontram-se os sistemas síncronos onde as considerações temporais são devidamente consideradas e por isso, as variáveis temporais, tal como a velocidade e os atrasos nas comunicações, são fundamentais para o correto funcionamento dos sistemas.

4.4.2 Atividades dos sistemas distribuídos

As atividades dos sistemas distribuídos estão condicionadas pela estrutura do sistema nomeadamente pelos protocolos utilizados, pelos serviços, pelos servidores dos sistemas bem como pelos acessos dos utilizadores. Neste sentido algumas atividades preponderantes para o correto funcionamento dos sistemas distribuídos devem ser referidas, assim temos:

- **Coordenação** (*Coordination*): é essencial para todas as atividades cooperativas e/ou descentralizadas tal como nos processos paralelos, nas ferramentas de engenharia concorrente, na gestão de bases de dados e na gestão de transações distribuídas.
- **Partilha** (*Sharing*): é responsável pela gestão do conjunto de nós que competem por um recurso.
- **Replicação** (*Replication*): refere-se à capacidade de controlo, de manutenção e realização da mesma sequência de ações de um conjunto de dados replicados.
- **Disseminação** (*Dissemination*): pode ser vista como uma forma de distribuição automática para todos os elementos que manifestem interesse no conteúdo da mensagem. Esta está dependente da confiança depositada no protocolo de distribuição multiponto para garantir uma eficiente disseminação da informação pelos elementos subscritores.

4.4.3 Estratégias para os sistemas distribuídos

As estratégias adequadas para o desenho de um qualquer sistema distribuído dependem de fatores subjetivos, do que se pretende em face dos requisitos, e de fatores objetivos tal como o ambiente envolvente, os custos, etc. Se por um lado uns dependem da capacidade da imaginação do programador para desenvolver o sistema outros dependem das estratégias adotadas no seu desenvolvimento. No esquema seguinte apresentam-se algumas das estratégias mais utilizadas no desenvolvimento de sistemas distribuídos, Figura 4-9.

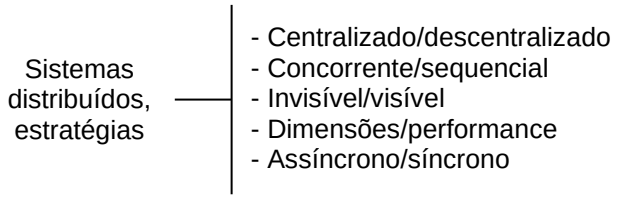


Figura 4-9. Estratégias de desenvolvimento de sistemas distribuídos

Sistemas Centralizados/descentralizados: os sistemas centralizados de controlo apresentam-se como sistemas de fácil desenvolvimento e gestão. No entanto alguns problemas de controlo seriam melhor resolvidos através da descentralização e da distribuição do sistema e das suas aplicações. Se por um lado a centralização torna fácil o seu desenvolvimento com o crescimento do processo e a necessidade de vários elementos solicitarem informações e recursos, surgirão problemas de partilha e de concorrência que só os sistemas descentralizados e distribuídos darão resposta.

Sistemas concorrentes/sequenciais: os sistemas dizem-se altamente concorrentes se o rácio de períodos de atividades/tempo decorrido para cada participante é elevado. É o caso dos modelos orientados a grupos onde ocorrem várias atividades em simultâneo em vários lugares e de forma concorrente. Caso contrário, se o rácio é baixo, pode-se dizer que o processo é não concorrente pelo que este se apresenta como um sistema de distribuição sequencial.

Sistemas invisíveis/visíveis: a distribuição é considerada invisível ou transparente quando o modelo esconde a distribuição. A visibilidade é conseguida quando os modelos permitem visualizar a troca de mensagens tal como nos modelos orientados a grupos ou nos modelos de mensagens de bus.

Dimensões/performance: as dimensões do sistema são, normalmente, determinantes na performance do mesmo. Assim, uma boa arquitetura poderá permitir que a performance do sistema não seja linearmente afetada pelo crescimento do mesmo. Por outro lado, sistemas de grandes dimensões, desenvolvidos segundo uma arquitetura aberta, apresentam, mesmo assim, menor facilidade de controlo comparando-os com os sistemas de pequena dimensão.

Assíncrono/síncrono: assíncrono quer dizer simplicidade do sistema, mas incerteza temporal. Sistema síncrono quer dizer mais complexidade com a garantia do cumprimento das especificações temporais.

Muitos outros atributos devem-se ter em conta aquando do processo de desenvolvimento de um sistema distribuído. Fatores como a disponibilidade, que passa pela garantia de que o servidor se encontra em funcionamento, ou pelo multiprocessamento ou pela replicação dos serviços em diversos servidores, no caso de falha, devem ser também considerados.

4.4.4 Modelos assíncronos

Os sistemas distribuídos assíncronos são sistemas para os quais o tempo não é uma variável que deve ser tomada em conta. As aplicações que trabalham segundo esta abordagem terão de

garantir a atividade do sistema através de pressupostos de "*a mensagem é eventualmente entregue*" ou de "*a execução está eventualmente terminada*". Isto quer dizer que o sistema não se encontra controlado temporalmente, ou seja, é um sistema livre de constantes temporais, *time-free*. Neste sentido o processamento e a troca de mensagens fazem-se arbitrariamente ao longo do tempo pelo que os relógios dos mesmos não são utilizados como tempo de referência.

Os modelos assíncronos são modelos simples que apresentam poucas garantias principalmente aquando da presença de uma falha. Assim, para sistemas totalmente assíncronos, torna-se impossível garantir o determinismo dos mesmos perante problemas básicos de consenso (Fischer *et al*, 1985). A deteção de falhas é outro dos problemas praticamente impossíveis de realizar nos sistemas assíncronos, nomeadamente no que diz respeito à identificação do elemento que falhou. No entanto, pela utilização de ferramentas de deteção específicas (*crash failure detectors*) pode ser detetada a paragem súbita de um elemento. Vários sistemas são concebidos contando com essa hipótese, como por exemplo, os sistemas assíncronos de deteção a falhas referidos por Chandra e Toueg (1996).

4.4.5 Modelos síncronos

Os sistemas distribuídos síncronos são pouco influenciados pelo ambiente pelo que completarão sempre a sua tarefa independentemente do tempo que esta possa demorar. Este é um modelo seguro, mas não muito dinâmico, na perspetiva da disponibilidade e da continuidade do serviço. A evolução deste modelo depende das fronteiras temporais definidas e por isso a sua execução, os atrasos nas trocas de mensagens, as diferenças entre relógios entre outros fatores são controláveis, o que permite a implementação de detetores de falhas perfeitos (Chandra e Toueg, 1996).

Um dos problemas inerentes aos modelos síncronos prende-se com a dificuldade de determinar uma sincronia do ambiente ou determinar os piores cenários de carga (*worst-case*). Nestes casos o sistema poderá funcionar incorretamente pelo facto das fronteiras do sistema poderem ser violadas ou, num segundo caso, devido às fronteiras se poderem tornar insuficientes.

4.4.6 Classes das atividades distribuídas

Os modelos de sistemas distribuídos desempenham algumas atividades primárias fundamentais para o seu processamento. Na Figura 4-10 apresenta-se um esquema das três principais classes primárias de atividades dos sistemas distribuídos.

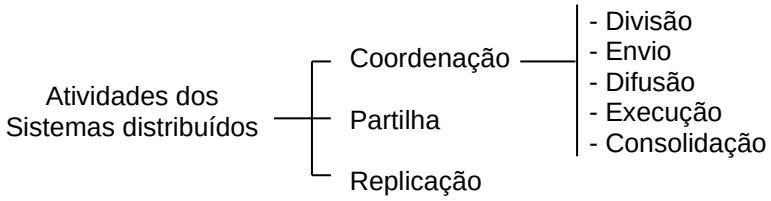


Figura 4-10. Classes das atividades primárias dos sistemas distribuídos

4.4.6.1 Coordenação

A coordenação de sistemas distribuídos diz respeito à necessidade de se executar um conjunto de atividades que contribuam para atingir um objetivo comum. Estas subdividem-se num conjunto de fases que passam pela divisão, envio, difusão, execução e a consolidação (Veríssimo e Rodrigues, 2001). Que a seguir se detalham:

- **Divisão** (*splitting*): consiste em realizar a divisão de cada trabalho (P_a) em várias tarefas $P_a(1)$, $P_a(2)$, …, para serem executadas em alguns ou por todos os nós do sistema.
- **Expedição** (*dispatching*): consiste em alocar as tarefas aos nós em número adequado, de acordo com a capacidade de processamento de cada nó pela ordem requerida pelo sistema.
- **Difusão** (*diffusion*): consiste em difundir pelos nós relevantes as tarefas parciais.
- **Execução** (*execution*): é realizada nos nós relevantes resultando do processamento dos resultados $R_a(1)$, $R_a(2)$, …, que deverão ser consolidados na tarefa *Ra* resultante do processamento realizado no nó de consolidação.
- **Consolidação** (*consolidation*): é uma operação simples realizada no nó destinatário. O nó necessita conhecer a informação a consolidar, ou seja, $R_a(1)$, $R_a(2)$, …, para consolidar em R_a.

O esquema genérico do princípio de funcionamento do modelo de coordenação é apresentado na Figura 4-11.

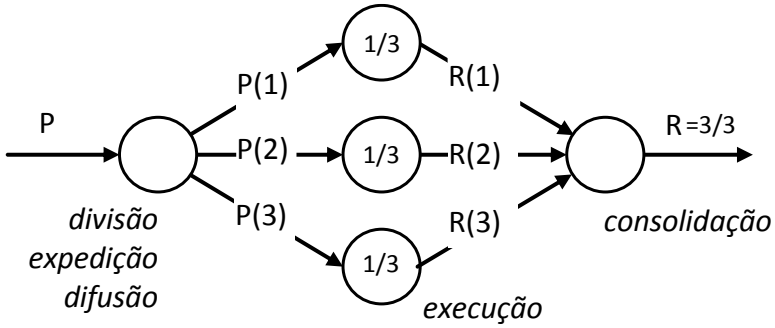


Figura 4-11. Diagrama de coordenação das atividades

A divisão e a expedição podem ser realizadas na fonte de emissão do pedido através de um algoritmo de paralelização que divide e envia em paralelo as tarefas para os diversos processos do sistema, disseminando as tarefas requisitadas conforme os casos. Em alternativa, estas operações poderão também ser realizadas no destinatário agindo de modo descentralizado. No entanto, e para a adoção desta alternativa, é necessário que os participantes possuam um algoritmo de concordância que, deterministicamente, divida e retenha as tarefas associadas a cada nó. Poderá também ser necessário aplicar critérios de ordenação à difusão. A aplicação deste critério depende das regras de divisão/expedição adotadas bem como do histórico de evolução do sistema. Em atividades descentralizadas de coordenação como seja o controlo distribuído é necessário que todos os nós possuam um conhecimento comum da evolução do sistema.

Quanto à consolidação esta poderá também ser realizada na fonte bem como no destinatário, no entanto, a adoção da primeira hipótese obriga a que todos os nós corram um algoritmo que permita reenviar os resultados para a fonte.

4.4.6.2 Partilha de recursos

A partilha refere-se ao conjunto de ações realizadas para garantir a correta execução das ações nos recursos partilhados, Figura 4-12. Esta atividade prende-se com a ordenação das tarefas antes da execução. Assim, se for possível determinar uma relação entre o emissor e os recetores, então uma ordenação casual dos pedidos deve ser assegurada para uma correta serialização. No caso dos vários elementos recetores não interagirem uns com os outros uma ordenação do tipo FIFO será suficiente para garantir uma correta serialização. Por outro lado, se a serialização é feita no destinatário o protocolo não necessita de ser ordenado.

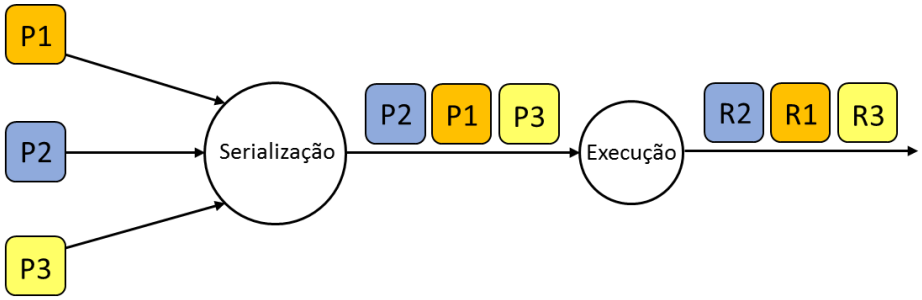


Figura 4-12. Atividades de partilha de recursos

Como se pode observar na Figura 4-12 várias tarefas P_1, P_2 e P_3 são enviadas diretamente para um recurso comum para serialização pelo que, após esta operação, as tarefas são enviadas para o recurso seguinte onde são executadas. Como resultado da execução produz resultados na mesma ordem em que as tarefas foram recebidas, ou seja, produzem os resultados R_2, R_1 e R_3, respetivamente.

4.4.6.3 Replicação

A replicação de processos prende-se com a necessidade de executar um conjunto de ações em vários nós esperando que produzam os mesmos resultados. Isto quer dizer que o mesmo processo P_a é executado em vários nós segundo um procedimento que envolve a difusão, a execução e a consolidação dos resultados, Figura 4-13.

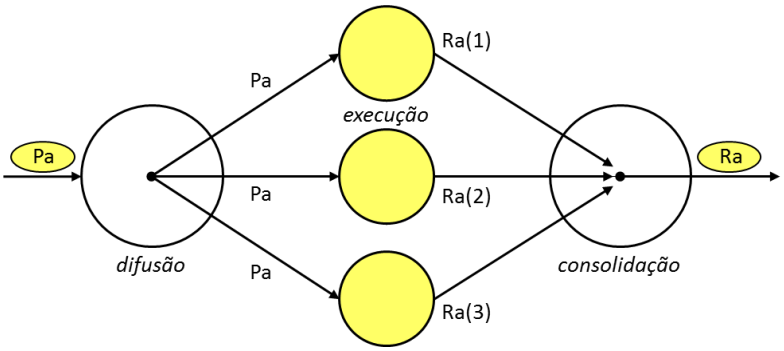


Figura 4-13. Diagrama de atividades de replicação

Dependendo do modelo utilizado na replicação o protocolo pode ou não necessitar que todas as mensagens sejam difundidas totalmente ordenadas garantindo-se assim o determinismo da replicação. Este processo é normalmente chamado de *active replication*, isto é, todos os participantes executam o mesmo conjunto de dados na mesma ordem (Guerraoui e Schiper, 1997). No caso de se utilizar a replicação para garantir a disponibilização de um valor, então será só necessário utilizar o primeiro valor produzido, $R_a = R_a(i)$. Por outro lado, se o modelo se destina a detetar a falha de valores, então a votação por maioria é necessária. Assim, os resultados da execução, $R_a(1)$, $R_a(2)$ e $R_a(3)$, são consolidados por maioria através da votação do resultado final garantindo deste modo que o resultado $R_a = vote(R_a(1), \ldots, R_a(n))$, enviado para o processo seguinte, estará correto. A consolidação dos resultados no destinatário obriga à execução de um algoritmo onde todos os resultados enviados e aceites pelo destinatário sejam consolidados.

Note-se, no entanto que, conceptualmente, um sistema distribuído é uma combinação de atividades de coordenação, de partilha e de replicação. Imaginemos uma base de dados replicada, para garantir que os dados estão sempre acessíveis, onde os registos são partilhados por vários utilizadores que competem entre si pelo acesso à base de dados. Por este facto todos os acessos

deverão encontrar-se sujeitos a ações de coordenação. Assim, se os utilizadores efetuarem um pedido P_a, P_b, …, estes poderão ser serializados por um protocolo *multicast*, assegurando que o sistema processa os pedidos na ordem pretendida (FIFO). A divisão, a expedição e a difusão, segundo um protocolo *multicast* com ordenação total, garante a disseminação dos pedidos na mesma ordem para todas as réplicas como se pode depreender da Figura 4-14.

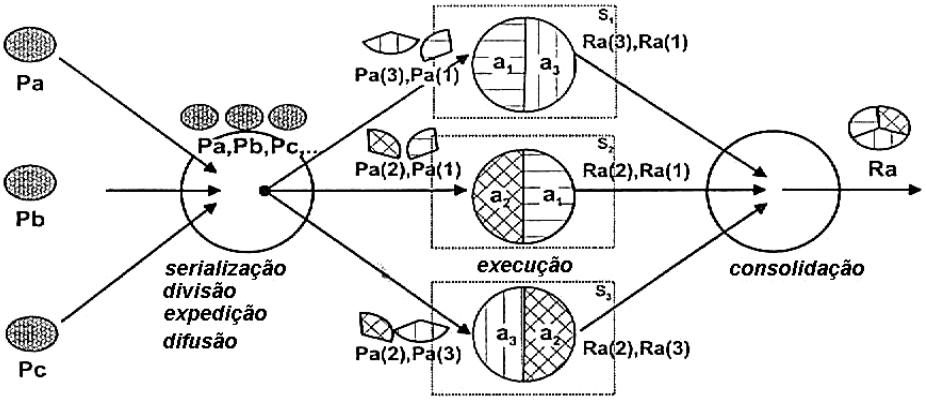


Figura 4-14. Ações combinadas (Veríssimo e Rodrigues, 2001)

Os processos $P_a(i)$ são enviados para execução para os vários nós (S_i) que, após execução produzem os resultados $R_a(i)$ de acordo com os processos recebidos e segundo a mesma ordem. Os resultados parciais são consolidados e enviados para o processo seguinte.

4.4.7 Arquitetura orientada a grupos

A programação com base nas arquiteturas orientada a grupos consiste basicamente na representação da informação e dos nós, dos objetivos e das atividades dos grupos participantes comunicando por difusão, ou por protocolos *multicast* bem definidos e sincronizados. Os processos comunicam entre si quer para dentro quer para fora do grupo recebendo e enviando informações quer do grupo onde pertence quer de grupos adjacentes. Dada a sua natureza de operação, em ciclo aberto, estes permitem apoiar facilmente a coordenação de atividades descentralizadas ou replicadas. Baseado em modelos hierárquicos estes sistemas interagem com os vários nós e os módulos interagem com os diversos elementos alocados nos respetivos locais. Assim, ao nível do nó, podem ser desenvolvidos vários blocos de entre os quais o bloco de deteção de falhas, poderá ser citado como exemplo. Este poderá ser utilizado para prestar serviços a todos os elementos do grupo nomeadamente no que se refere às informações referentes aos participantes no grupo de comunicações. Este módulo interagirá com o módulo de comunicações, responsável pela fiabilidade e pela garantia da ordenação das mensagens trocadas entre os nós.

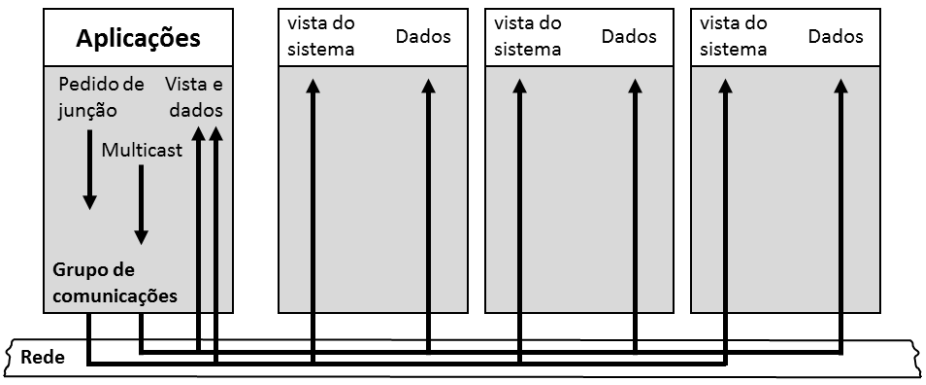


Figura 4-15. Comunicações segundo uma arquitetura orientada a grupos

Um terceiro módulo, integrado com estes dois últimos, servirá de suporte aos paradigmas dos sistemas distribuídos tal como, por exemplo, a gestão de réplicas. Assim sendo, a interação entre os elementos do grupo realiza-se pela formulação de um convite à participação no respetivo grupo e pela correspondente resposta ao pedido de junção. Quando aceites os vários elementos recebem uma vista global do grupo, informação sobre o grupo, bem como a atualização de todos os outros elementos, segundo o protocolo *multicast,* de acordo com o esquema apresentado na Figura 4-15.

As vantagens que se podem obter com a utilização de grupos de comunicação traduzem-se no facto de que os nós não necessitam de saber o número e a localização dos servidores. Estes limitam-se a enviar uma mensagem para o grupo esperando pela primeira resposta. De igual modo na disseminação o emissor produz a informação difundindo-a para os consumidores sem necessitar da informação do número ou da localização dos mesmos.

4.4.8 Memória compartilhada distribuída

A memória compartilhada distribuída (DSM – *Distributed Shared Memory*) é um paradigma intuitivo utilizado para emular o ambiente de execução de multiprocessador de memória compartilhada em sistemas distribuídos. O modelo é intuitivo no sentido de que os paradigmas são também válidos para programação concorrente em sistemas de memória compartilhada. A utilização de DSM garante-nos que a programação paralela desenvolvida para sistemas de partilha de memória com multiprocessadores pode ser facilmente transportada para um *cluster* de estações de trabalho. Assim, o objetivo final do DSM é proporcionar a ilusão da centralização dos sistemas de memória partilhada quer em termos de semântica quer em termos de performance onde todas a operações são atómicas. Isto quer dizer que a distribuição da memória deve apresentar-se de forma transparente para os projetistas. Neste sentido, há que ter em conta que todas as operações de escrita (W_i) são totalmente ordenadas e que as operações de leitura (R_j) retornam sempre o último valor escrito na memória. Assim sendo, e se a ordem temporal das operações não necessitar de ser respeitada, o modelo deve ser considerado como uma sequência consistente (Lamport, 1997). Os modelos antigos de consistência de memória são normalmente identificados como modelos de consistência forte.

A implementação desta metodologia está dependente da linguagem de programação usada bem como dos requisitos e da sua performance. O DSM pode ser implementado segundo uma aplicação de elevada performance trabalhando em paralelo. Ou seja, a sua implementação consiste na distribuição pelos vários nós disponíveis, de parte das tarefas a executar, tornando cada um dos nós responsáveis pela computação da porção do algoritmo que lhe foi alocado. Cada nó processa a sua parte do algoritmo até que o processo esteja concluído. Caso contrário será repetida uma nova ronda de processamentos. Esta metodologia poderá ainda ser utilizada na gestão da consistência dos objetos em memória, ou seja, através da utilização de réplicas consistentes dos objetos que pretendem ser acedidos, quer para leitura quer para escrita.

4.4.9 Barramento de Mensagens

O barramento de mensagens é uma abstração do processo. Este permite a troca de mensagens indiretamente, através do componente de interligação dos processos chamado de barramento, ou seja, de *bus*. Neste modelo alguns dos processos chamados de editores (*Publishers*) produzem mensagens para o barramento onde outros processos chamados de subscritores (*Subscribers*) consomem as mensagens que circulam no barramento. Este modelo de comunicação é ou pode também ser chamado de modelo de leitura de informação. Uma abordagem deste tipo permite resolver problemas simples das aplicações distribuídas pelo facto de que não é necessário que os emissores das mensagens e os recetores estejam ativos em simultâneo. Neste sentido as mensagens podem ser produzidas em qualquer momento e lidas

muito tempo depois, no entanto, e para que isto seja possível, é necessário que o barramento possua uma memória não volátil onde as mensagens são mantidas até que os recetores as leiam, isto é, as "puxem". Este modo de operação é chamado de modo assíncrono ou de interação não sincronizada.

As comunicações em barramento são normalmente utilizadas em sistemas em que os nós têm uma reduzida interação pelo que as mensagens produzidas poderão ser armazenadas em lotes de informação e consumida posteriormente. Associada a estas vantagens temos ainda de referir que a reconfiguração dos sistemas torna-se bastante fácil, quer para o editor quer para o subscritor, dado que se pode alterar o número, a identificação ou a localização dos subscritores sem alterar os editores e vice-versa. Nesta configuração o barramento pode apresentar-se como volátil, quando as mensagens que circulam no barramento são consumidas na hora ou desaparecem no *"éter"*, ou como um barramento persistente no caso de estas permanecerem no barramento até que sejam consumidas, Figura 4-16.

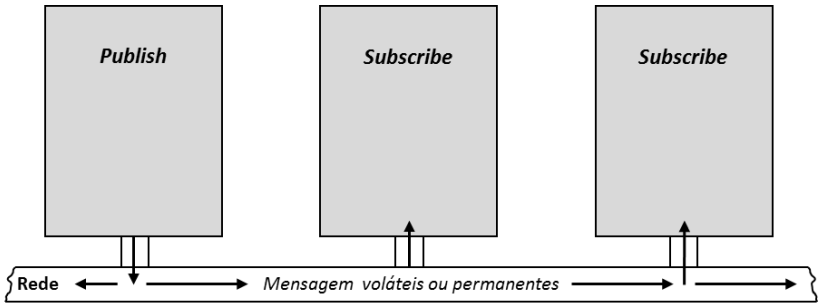


Figura 4-16. Implementação de um sistema Editor-Subscritor (Publish/Subscribe)

O endereçamento das mensagens pode ser conseguido usando uma caixa de correio, o método mais simples de endereçamento, onde as mensagens são depositadas ou em alternativa utilizado um outro qualquer esquema de endereçamento, por exemplo IP.

O editor especifica o nome das caixas de correio onde pretende depositar as mensagens e, para os subscritores, são também definidas caixas de correio onde estes vão recolher as mensagens. Á semelhanças das tradicionais caixas de correio, estas também deverão possuir uma capacidade máxima de receção que, quando cheias, deverá ser emitida uma informação de erro a enviar ao editor. Este poderá bloquear o processo, produzindo uma nova mensagem. Os subscritores removem as mensagens da caixa de correio ou em alternativa podem obter uma cópia das mesmas no barramento enquanto estas não são consumidas.

O endereçamento de mensagens apresenta-se como uma alternativa mais difícil de implementar dado que deve ser usado o endereço do nó. Neste caso as mensagens são publicadas no barramento com etiquetas de acordo com os endereços dos nós em questão utilizando uma ou mais palavras de identificação.

A implementação deste protocolo de comunicação é relativamente simples como se pode depreender do que foi exposto anteriormente. A abstração do barramento é materializada com a ajuda de um servidor pelo que, a atividade de publicação de informação torna-se bastante simplificada dado que a informação produzida é enviada para o servidor onde permanecerá armazenada numa memória não volátil. Os subscritores leem (*puxam*) as mensagens que pretendem receber diretamente do servidor. Os subscritores registam a intenção de receção de mensagens diretamente no servidor passando o servidor a ser responsável pela gestão e pela disseminação das mensagens em que os subscritores possuem interesse. Nesta situação, e se vários subscritores pretenderem receber as mesmas mensagens, deverá ser utilizada uma disseminação *multicast* para que todos os subscritores possam receber a mesma informação.

Uma outra abordagem ao problema de disseminação da informação via barramento refere-se à estratégia como as mensagens são lidas diretamente do servidor. Os subscritores contactam periodicamente o servidor recebendo assim as mensagens. Esta abordagem pode-se tornar menos eficiente, mas possui vantagens no caso de os sistemas possuírem subscritores que não se encontram permanentemente ligados pelo que os subscritores podem receber as mensagens de forma deferida, isto é, de forma não sincronizada, posteriormente.

4.5 Falhas associadas aos sistemas distribuídos

Os sistemas distribuídos são constituídos por vários nós independentes. Toda a interação é feita por troca de mensagens através dos canais de comunicação. Estes sistemas não possuem memória partilhada nem relógio global e, portanto, não possuem uma base de tempo comum para ordenação dos eventos. São geralmente constituídos por elementos não homogéneos e assíncronos.

Os sistemas distribuídos possuem redundância natural que pode ser utilizada, por si só, para a aplicação de técnicas de tolerância a falhas. Apesar de nos sistemas distribuídos existirem múltiplos processadores independentes e estes contribuírem para o aumento do risco de falhas, a distribuição é também importante para o incremento do desempenho do sistema e ao mesmo tempo para a redução do custo da implementação de tolerância a falhas. Garantir a confiança no funcionamento destes sistemas passa por solucionar problemas de consenso, de ordenação e atomicidade na troca de mensagens entre grupos de processos, sincronizar relógios quando necessário, implementar réplicas consistentes dos processos, garantir resiliência de dados e processos num ambiente sujeito a paragens de estações tanto clientes como servidoras, particionamento de redes, perda e atrasos de mensagens e, eventualmente, comportamento arbitrário dos componentes do sistema (Jalote, 1994).

Por outro lado, e de acordo com os desvios às especificações a que os processos estão sujeitos durante o seu funcionamento várias são as falhas associadas aos sistemas distribuídos e referidas na literatura (Hadzilacos e Toueg, 1993; Jalote, 1994; Sari, 2015). As falhas que normalmente ocorrem num sistema distribuído, segundo o modelo clássico são:

- **Colapso** *(Crash)*: a falha por colapso acontece quando um nó está a funcionar normalmente e para de repente. Depois desta ação não se consegue obter mais nenhuma informação sobre o mesmo. Esta é o tipo de falha mais simples e ocorre, por exemplo, quando o nó onde o processo está a ser executado é desligado.
- **Omissão** *(Omission):* a falha por omissão ocorre quando um nó não responde a um pedido. Neste caso, pode ter acontecido uma falha de receção do pedido ou de envio da resposta. O problema torna-se maior aquando da falha de envio, pois o nó que efetuou o pedido pode não estar ciente do problema. Se este requisitar novamente o mesmo serviço, o resultado poderá ser diferente do esperado.
- **Temporização** *(Timing)*: a falha de temporização ocorre quando a resposta de um nó chega fora do tempo esperado. Se a resposta chegar antes do tempo esperado, o *buffer* de receção pode não ter espaço suficiente para receber os dados. Por outro lado, se chegar muito tarde, o desempenho do sistema será afetado.
- **Resposta** *(Answer)*: a falha de resposta ocorre quando um nó envia uma resposta errada. Esta situação pode acontecer quando o nó responde a um pedido enviando um valor errado ou quando o nó reage de modo inesperado a um pedido que não consegue reconhecer.

- **Arbitrárias** *(Byzantine fault)*: a falha arbitrária é também conhecida como falha Bizantina. Esta é um dos problemas mais sérios, pois o nó pode estar a produzir saídas que não deveriam ser produzidas, mas que não são detetadas como incorretas.

Os modelos clássicos de falhas em sistemas distribuídos, mencionados anteriormente, foram definidos por Cristian (1991) e por Schneider (1990), Figura 4-17. Este último estendeu o modelo de Cristian definindo um novo modelo onde constam: *fail-stop*, *crash*, omissão de envio, omissão de resposta, temporização, resposta e arbitrariedade.

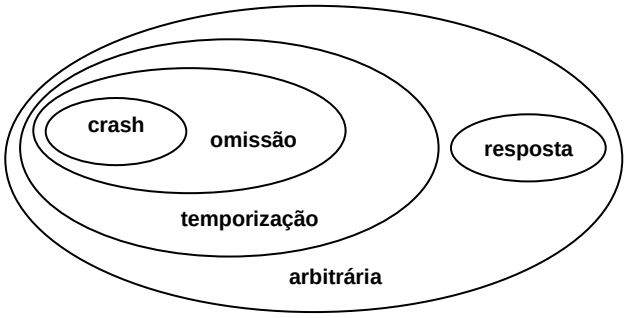


Figura 4-17. Modelo clássico de falhas em sistemas distribuídos (Jalote, 1994)

A proteção contra falhas em sistemas distribuídos passa por utilizar as características inerentes a estes sistemas tirando partido do seu processamento distribuído. A tolerância a falhas passa pela distribuição de réplicas idênticas garantindo que todas essas réplicas recebem a mesma mensagem e em simultâneo pelo que, a falha de uma réplica, não impedirá o correto funcionamento do sistema. Esta metodologia conduz a problemas da validação dos dados produzidos pelas réplicas. Nesta situação é necessário que todos os elementos replicados, que não falharem, consigam chegar a um consenso sobre o resultado final que, por vezes, pode ser uma tarefa bem complicada (Tanembaum e Steen, 2006). Lampor *et al.* (1982) afirmam, por sua vez, que o consenso só será possível se pelo menos dois terços dos processos replicados estão a funcionar corretamente, votação maioritária. Por outro lado, há que assegurar que as interligações entre componentes garantam que o sistema de comunicação execute a sua tarefa de forma confiável e segura (Veríssimo, 1996). A obtenção destas garantias conduz-nos à utilização do protocolo de transmissão designado de *atomic multicast*.

4.6 Conclusão

Neste capítulo foram abordadas as diversas formas estruturais do sistemas distribuídos. Abordou-se a sua construção e modo de funcionamento tendo em conta não só a suas interligações ao chão de fábrica como a comunicação com os elementos de decisão. Neste contexto são apresentadas a diversa topologias de rede quer estas sejam em anel, comunicação ponto-a-ponto, quer se trate de uma rede em árvore. Abordaram-se os diversos paradigmas das mesmas tendo em vista o envio de mensagens, quer sejam temporizadas quer não, usando pares de comunicação Cliente/Server ou Publish/Subscribe.

Os protocolos de comunicação multicast, o sincronismo dos nós (uso de relógios locais ou globais para a sincronização dos relógios dos diversos componentes) bem como a ordenação dos dados são apresentados como elementos preponderantes para a distribuição e replicação determinística dos sistemas de controlo.

Capítulo 5

Análise de alguns trabalhos relevantes

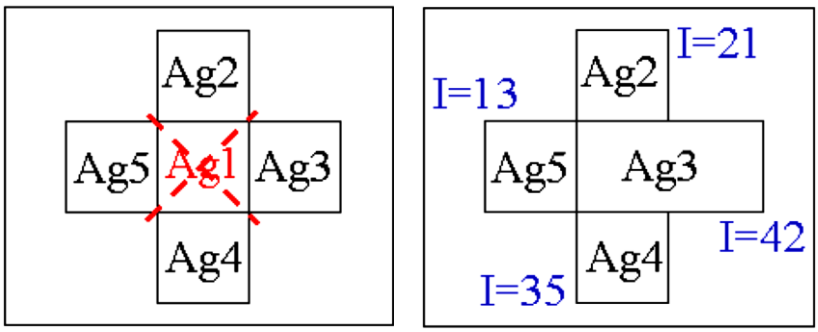
5.1 Introdução

Diversas abordagens à confiabilidade no funcionamento de sistemas distribuídos tolerantes a falhas foram já estudadas. Assim, referir-se-á alguns dos trabalhos que poderão ser considerados mais significantes tal como o DELTA-4 (Powell, 1991) associado a grandes sistemas de tempo real, o MARS (Kopetz *et al.*, 1989) um sistema distribuído com uma base de tempo global, o GUARDS (Powell, 2001) utilizando componentes de uso genérico, DEARCOTS (Pinho *et al.*, 2004) utilizando componentes de usos genérico e a linguagem de programação Ada95 ou ainda Bhat *et al.* (2018) usando requisitos de tempo de recuperação baseados na recuperação ativa e em *primary-backup*, por exemplo. Muitos outros trabalhos sobre a tolerância a falhas podem ser encontrados na literatura quer do ponto de vista da replicação suportada por *hardware*, por *software* ou mista, combinando *hardware* e *software*. Abordar-se-á ainda a tolerância a falhas em sistemas de tempo real e em aplicações de caráter industrial.

5.2 Tolerância a falhas baseada em hardware

A tolerância a falhas com base na replicação de *hardware* assenta na replicação dos componentes físicos dos sistemas. Embora seja uma abordagem mais cara é normalmente utilizada para o mascaramento de falhas, bem como para ações que visão prolongar o ciclo de vida dos sistemas. Algumas abordagens mais significativas à tolerância a falhas, com base no *hardware*, serão apresentadas seguidamente. Com estas pretende-se abordar, o melhor possível, os vários modos de redundância com *hardware*.

Proença (2003) aplica a tolerância a falhas na supervisão do controlo do tráfego ferroviário. Interligando cada agente de supervisão a cada um dos seus vizinhos, partilha a supervisão do troço da linha, trocando informação relativamente ao tráfego e à sua transferência entre zonas. Quando ocorre uma falha num agente, a ligação entre os supervisores é quebrada é despoletado um leilão, entre os agentes vizinhos, pela posse da área sem supervisão. A ideia subjacente ao processo de leilão consiste em encontrar o agente com menor volume de trabalho, agente que irá assegurar a supervisão da nova zona. Neste sentido os agentes trocam informações acerca da dimensão da zona por si supervisionada e do volume de tráfego associado sob a forma de um valor numérico conforme se apresenta na Figura 5-1a.



a) Falha de um agente supervisor *b) Nova situação de supervisão*

Figura 5-1. Controlo de tráfego ferroviário (Proença, 2003)

Cada supervisor contém informações acerca dos seus vizinhos bem como informações sobre a vizinhança de cada um dos seus vizinhos. Calculados os índices de disponibilidade, estes trocam essa informação de forma a permitir a avaliação da melhor proposta, pelo que após receção de todos os valores, o leilão é dado por encerrado selecionando-se o vencedor. O vencedor reconstitui a sua estrutura assumindo a nova área de supervisão, notificando os vizinhos da alteração entretanto efetuada, Figura 5-1b. O processo de eleição dos sistemas de controlo apresentado por Proença pode traduzir-se numa abordagem eficiente quando não são necessários requisitos de tempo real ou, em situações, em que os tempos de decisão são latos. O processo de eleição é um processo lento que irá sobrecarregar o sistema, em termos computacionais, uma vez que será necessário indagar a carga associada a cada sistema. Em termos de controlo de sistemas de tempo real os processos de eleição não são, normalmente, considerados dado que os tempos de decisão podem tornar-se elevados. É essencialmente por este facto que a ideia subjacente a esta metodologia não será implementada na nossa abordagem, no entanto a ideia da partilha de recursos revela-se bastante interessante.

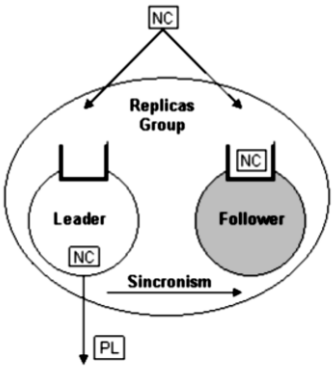


Figura 5-2. Replicação semi-ativa sincronizada (Rodrigues, 2008)

Em Rodrigues (2008) a tolerância a falhas é realizada com recurso à tolerância a falhas semi-ativa. O elemento de controlo é replicado num outro computador designado de "*seguidor*" comunicando com o "*líder*" por meio de mensagens sincronizadas, Figura 5-2. Nesta situação ambas as réplicas recebem as mesmas mensagens de controlo de um mesmo grupo e na mesma ordem, no entanto só a réplica líder executa as ações de controlo e processa os resultados enquanto o seguidor mantém-se atualizado sem executar qualquer ação de controlo nem produzir resultados. Esta sincronia permite à réplica seguidora continuar a executar o controlo da célula de produção do ponto correto após uma falha do líder (Rodrigues, 2008). A réplica líder informa a réplica seguidora do seu estado interno, enviando uma mensagem para a seguidora, solicitando a remoção de mensagens da sua fila de espera interna. Rodrigues (2008) utilizou ainda a redundância temporal e de informação, para além da redundância física da rede de comunicação,

implementadas no protocolo de comunicação da rede através do reenvio de mensagens. A duplicação do sistema de controlo noutro computador não é a abordagem pretendida para a nossa arquitetura. A abordagem de Rodrigues apresenta, no entanto, pontos de vista que serão transportados para a nossa implementação. Destes saliente-se o facto do sistema ser suportado por um conjunto de mensagens sincronizadas e ordenadas, base da nossa implementação. A replicação completa do sistema de controlo acarreta ainda um aumento dos tempos de processamento uma vez que as mensagens são enviadas em duplicado, uma para cada réplica. Adicionalmente o "líder" sobrecarrega o sistema com ordens de limpeza da fila de espera do "seguidor" aumentando o tempo de processamento.

Gulay *et al.* (2010) propõem uma abordagem mista à tolerância a falhas. Apresentam um estudo que tem por base o Hardware de Memória Transacional (HTM – *Hardware Transactional Memory*) para o desenvolvimento de um sistema tolerante a falhas. Com esta abordagem os autores pretendem mostrar que é possível obter tolerância a falhas com baixos custos utilizando a HTM modificada a que chamaram de *FaulTM*. Esta abordagem mista de *hardware-software* às técnicas de tolerância a falhas disponibiliza, ao programador, a flexibilidade necessária para decidir entre o desempenho e a confiabilidade. Neste sentido incidem o seu estudo sobre os problemas dos "erros suaves", nomeadamente os erros causados por partículas alfa, que podem dar origem a resultados incorretos ou em último caso a falhas do sistema. A abordagem consistirá numa série de passos que o programador terá de seguir que passam pela definição das secções vulneráveis do código, pela criação de uma *backup* de cada secção vulnerável, pela execução da sequência original e a de *backup* e pela comparação do conjunto de registos de transações na fase de confirmação. Assim sendo, numa arquitetura *multi-core*, apenas o processador principal é ocupado ficando os outros desocupados. O FaulTM tira partido desta inoperacionalidade para fins de confiabilidade, ou seja, para a deteção de erros suaves e permanentes.

Esta abordagem não se encontra diretamente relacionada com a implementação que se pretende desenvolver para a tolerância a falhas em ambiente 61499. No entanto alerta para o problema de ocorrência de erros ao nível do CPU dos sistemas computacionais que se traduzirão em falhas. A nossa implementação não visa colmatar este problema dos erros ao nível do CPU, mas sim, ter em atenção os desfasamentos entre sistemas. Esta nossa preocupação prende-se essencialmente com as diferentes velocidades de processamento ou erros que poderão resultar em tempos diferentes de processamento e consequente desordenação dos dados.

Jian *et al.* (2019) propõem uma abordagem à tolerância a falhas em sistemas distribuídos, tendo como base um protocolo BFT (*Byzantine Fault-Tolerant*) rápido e escalável, que chamaram de FastBFT. Neste sentido, associaram uma nova técnica de agregação de mensagens que combina ambientes de execução confiáveis, baseados em *hardware* (TEEs – *Trusted Execution Environments,* Intel SGX) com compartilhamento leve. Combinando essa técnica com várias outras otimizações (execução otimista, topologia em árvore e deteção de falhas), o FastBFT alcança baixa latência e alta taxa de transferência mesmo para redes de grande escala.

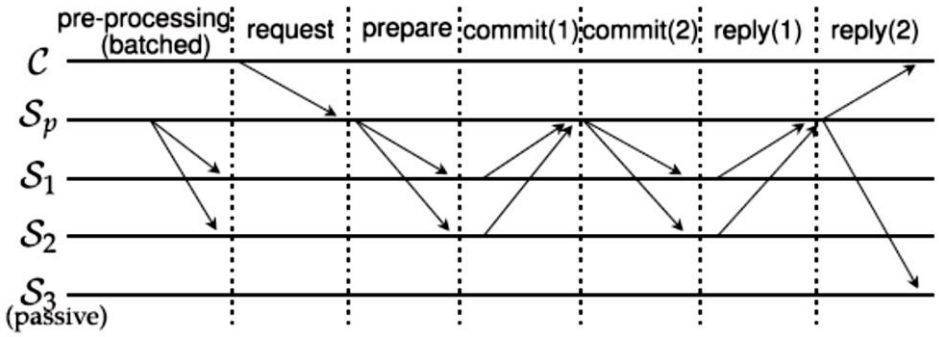


Figura 5-3. Padrão de mensagens do FastBFT (Jian et al., 2019)

A agregação reduz a complexidade da mensagem, uma vez que o FastBFT não requer operações com chaves públicas o que reduz, consideravelmente, a sobrecarga computação/comunicação. Esta arquitetura adota o paradigma BFT otimista, onde os nós encontram-se organizados segundo uma topologia em árvore, que requer apenas um subconjunto de nós para a execução do protocolo. Ao mesmo tempo juntam ainda um mecanismo de deteção de falhas simples que permite lidar com falhas não primárias de maneira eficiente, conforme esquema apresentado na Figura 5-3. O FastBFT garante a segurança em redes assíncronas assumindo-se que cada réplica mantém um TEE baseado em *hardware* com um contador monotónico e uma memória resistente à reversão. A escolha do paradigma otimista é justificada pelo compromisso entre eficiência e resiliência onde *f*+1 réplicas ativas concordam e executam os pedidos enquanto as outras *f* réplicas passivas apenas atualizam seus estados.

Os autores conseguiram otimizar e obter ganhos ao nível da latência e das taxas de transferência ao agregar a tolerância a falhas a um subconjunto de nós. Esta abordagem conduziu a um bom compromisso entre eficiência e resiliência. Esta é efetivamente uma das preocupações que temos no desenvolvimento do sistema replicado, uma vez que se pretende que este apresente baixa latência e um elevado compromisso entre eficiência e resiliência. Pretendemos efetuar a replicação de um conjunto restrito de elementos críticos e em número suficiente para a tolerância a falhas.

5.3 Tolerância a falhas baseada em software

Uma abordagem à tolerância a falhas baseada em *software* é uma questão fundamental para obter-se uma arquitetura confiável utilizando componentes de uso genéricos. Como não há *hardware* especializado, com propriedades de autocontrolo, é o *software* que deve gerir a replicação e a tolerância a falhas (Guerraoui e Schiper, 1997).

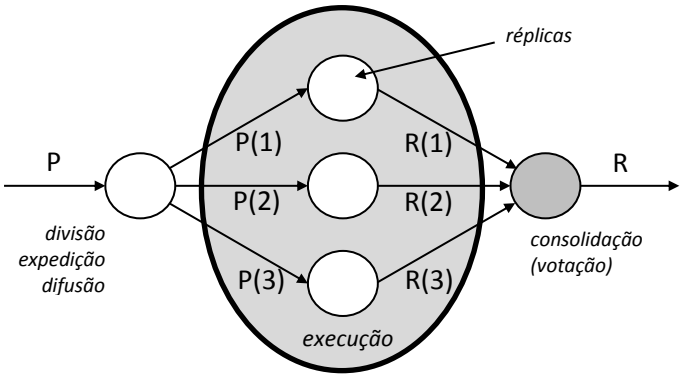


Figura 5-4. Replicação ativa de componentes de software

Assim, um sistema tolerante a falhas, baseado em *software*, é implementado pela coordenação de um grupo de componentes de *software* replicados em diferentes nós. A ideia é gerir o grupo de componentes de *software* com o fim de mascarar as falhas que possam ocorrer em algum dos seus membros. A coordenação destas réplicas dá a ilusão, aos outros componentes de *software*, que o grupo é constituído por um componente único de *software* livre de falhas (Powell, 1991) e, por isso, as falhas não se tornam visíveis. No entanto deve-se ter em conta que o uso de componentes genéricos replicados está, normalmente, associado a falhas não controladas dado que estes componentes não possuem, normalmente, mecanismos de autocontrolo, pelo que torna-se necessário utilizar técnicas de replicação ativa, Figura 5-4. Por outro lado, em aplicações baseadas em mecanismos dependentes do tempo, as diferentes velocidades de processamento dos nós replicados podem causar diferentes resultados. Como consequência destas diferenças, diferentes réplicas, mesmo que corretas, podem responder de

modo diferente às mesmas entradas, produzindo resultados inconsistentes se as mesmas não são comutativas. A inconsistência de resultados obtidos nas réplicas irá afetar o determinismo nos sistemas distribuídos (Poledna, 1994). Para obviar este problema poderá alcançar-se o determinismo se forem proibidos os pedidos de utilização dos mecanismos de sincronismo não determinísticos. Como consequência, o uso de multitarefas (execução de mais de um programa ao mesmo tempo) não seria possível, uma vez que a sincronização das tarefas e os mecanismos de comunicação estão inerentemente ligados à utilização de métodos não determinísticos. Este é o método utilizado por ambos os projetos MARS (Kopetz *et al.*, 1989) (sistema distribuído tolerante a falhas e de tempo real, arquitetura composta por um ou mais *clusters*) e Delta-4 (Powell, 1991) (sistema distribuído de tempo real, arquitetura composta por componentes de uso genérico). No primeiro, através de uma tabela de escalonamento estático, que garante a execução e o mesmo comportamento em todas as réplicas. No último, restringindo as réplicas para se comportarem como máquinas de estado (Schneider, 1990), quando é utilizada a replicação ativa.

É com base nas abordagens que estes autores preconizam que a nossa abordagem será implementada. A replicação será realizada ao nível do *software* a correr em componentes genéricos de baixo custo. A gestão das réplicas será também realizada pelo *software* garantindo-se a consistência e o determinismo das mesmas.

Pinho *et al.* (2016) abordam, também, o problema de tolerância a falhas em sistemas distribuídos utilizando a replicação de máquinas de estado (SMR – *State Machine Replication*) com algoritmos de ordenação total, consenso, como o Raft (Ongaro e Ousterhout, 2014). Neste sentido propõem uma abordagem PRaft (*Priority Raft*) de replicação de máquinas de estado considerando a prioridade das requisições. As mensagens recebidas são executadas prioritariamente, pelo que os processos não necessitam esperar pela confirmação da requisição para que estas sejam processadas. Estas são executadas no momento em que são rececionadas.

Os autores utilizam um protocolo de consenso com ordenação total das informações replicadas priorizando as mensagens, execução no momento em que são recebidas. Esta é uma abordagem contrária ao que pretendemos implementar. A nossa abordagem, para além de garantir o determinismo, compara as mensagens recebidas tratando-as de acordo com o instante em que se tornarão válidas. A nossa prioridade é o instante associado à mensagem pelo que a abordagem trabalhará com a mais antiga e não com a primeira a chegar.

Por sua vez, Kuvaiskii *et al.* (2016) utilizam a redundância ativa (TMR) na abordagem à tolerância a falhas transientes do CPU. Estes replicam para três cópias de instruções de verificação num programa de avaliação, periódica, para detetar e corrigir as falhas do CPU usando votação maioritária. Neste sentido, os autores propõem o ELZAR, uma estrutura de compilação que transforma aplicativos *multithread* não modificados, aproveitando instruções SIMD *(Single Instruction, Multiple Data)* disponíveis nos CPUs mais modernos, para suportar a redundância modular tripla usando extensões Intel AVX para vetorização.

Problema recorrente que aborda a tolerância a falhas em CPUs. Não apresenta uma relação direta com o problema que pretendemos resolver. No entanto não podemos descorar que a nossa abordagem é dependente do tempo e, como tal, dependente das diferentes velocidades ou de processamento ou mesmo de falhas de processamento dos CPUs.

Outros trabalhos podem ainda ser referidos não só no modo de abordagem da replicação e da redundância bem como ao nível da linguagem de programação utilizada. Domokos e Majzik (2005), usando um único equipamento, aplicam a redundância aquando do desenvolvimento da arquitetura tendo como base a modelação AOP (*Aspect Oriented Paradigm*). Estes desenvolveram, em UML (*Unified Modeling Language*), uma arquitetura inicial, não redundante, com determinados padrões relacionados com a redundância separando, deste modo, o desenvolvimento do modelo funcional do não funcional. Os modelos de tolerância a falhas e de

gestão de redundância e seus submodelos de análise específicos estavam disponíveis na forma de uma biblioteca padrão.

Outros autores usando também o UML (Bernardi *et al*. 2011) propõem a abordagem MARTE-DAM baseada num perfil para apoio ao desenvolvimento de modelos de análise quantitativa de confiabilidade, tendo como base de suporte o sistema distribuído, mesmo na presença de ataques maliciosos. Ao contrário das várias abordagens destinadas a compreender os modelos UML de confiabilidade, esta abordagem abrange diferentes aspetos de confiabilidade como seja a introdução de componentes tolerantes a falhas, que pode fornecer uma estrutura redundante suportada por decisores e por estratégias FT (*Fault Tolerance*). Para efeitos de desempenho e análise de confiabilidade e de avaliação, esta abordagem foi testada com modelos Determinísticos e Estocásticos Petri Nets (DSPN – *Deterministic and Stochastic Petri Nets*).

Estes dois últimos autores abordam o problema da redundância utilizando a linguagem UML. Não é esta linguagem que foi usada no desenvolvimento dos SIFBs, no entanto usaram uma abordagem que, em tudo, se pode considerar idêntica. Estas referências enquadram-se na multiplicidade de linguagem da IEC 61499. O encapsulamento da programação é igualmente um modo de programação desta nova norma.

Niz e Feiler (2009) propuseram uma abordagem para modelar e verificar formalmente os padrões de replicação na linguagem AADL (*Architecture Analysis and Design Language*) e, em seguida, analisaram os comportamentos potencialmente indesejados tendo como base dois modelos AADL, aplicados aos sistemas de voo, implementados em Simulink. O primeiro modelo define o comportamento desejado em sequências de chamadas síncronas e o segundo descreve a arquitetura de replicação. Utilizando a abordagem *primary-backup* os autores propõem duas réplicas: uma primária e outra de *backup*. As transições entre a primária e a de *backup* são realizadas por um sistema de eventos. Este último é então ligado a uma unidade controladora de transição que representa tanto o comportamento do operador humano como o módulo de deteção de falhas.

Lasnier *et al*. (2010) utilizam a mesma aproximação, *primary-backup*, acrescentando à linguagem AADL um anexo comportamental (BA – *Behavioral Annex*). O sistema foi modelado usando componentes AADL com ligações sincronizadas através da execução de eventos. Com a AADL-BA é possível identificar os estados em que a aplicação pode ficar bloqueada e descrever a sequência de chamadas realizadas das diferentes interligações. Esta abordagem apresenta dificuldades no desenvolvimento dos mecanismos de sincronização complexos em sistemas distribuídos e de tempo real com exclusão mútua. Para além disto, existem ainda dificuldades na aplicação desta abordagem tanto que o sistema central como o padrão de replicação devem ser especificado manualmente.

Gabsi e Zalila (2015) propõem uma nova metodologia de replicação usando as propriedades disponibilizadas pela AADL definindo e validando o modelo, as propriedades de replicação e o mecanismo de replicação desejado, segundo uma lista de definições de prioridades, tal como o modelo de replicação, o número de réplicas e uma lista de regras de transformação para chegar a um modelo AADL expandido, que contem as diferentes réplicas. Esta abordagem garante a redução da complexidade do sistema tolerante a falhas e a redução do tempo utilizado no desenvolvimento da aplicação devido, essencialmente, à geração automática do código do modelo AADL enriquecido por um conjunto de regras de transformação dos modelos, bem como a separação do projeto funcional e não funcional. Os requisitos de tolerância a falhas, dos componentes, são especificados separadamente pelo conjunto de propriedades.

As abordagens anteriores enquadram-se na abordagem aos problemas de tolerância a falhas segundo a abordagem *primary-backup*. Estes abordam a tolerância a falhas quer para sistema de controlo e de simulação de voo, em Simulink, bem como em sistemas distribuídos de tempo real. As aplicações são desenvolvidas numa linguagem específica que não será utilizada no nosso

modelo. Os autores definem, também, algumas regras que condicionam a construção do sistema. O nosso não se encontra restringido a nenhumas regras de restrições iniciais.

Yadav *et al.* (2016) propõem uma nova abordagem à tolerância a falhas em sistemas de tempo real baseada em NVP. Na solução proposta existem N versões do *software*, das quais *t* versões implementam apenas um subconjunto de todas as funcionalidades que são efetivamente críticas, enquanto (N-t) versões implementam todas as funcionalidades. Esta modificação da NVP permite maximizar a fiabilidade e reduzir os custos recorrendo à seguinte implementação: N é o número total de versões do *software*, *t* é o número de versões que contêm só funcionalidades críticas e (N-t) é o número de versões que contêm funcionalidades completas (críticas e não críticas). Todas as versões do *software* recebem as mesmas entradas, no entanto as versões *t* do *software* descartam as entradas que não são necessárias para a funcionalidade crítica (*Critical_flag=0*). Este valor de criticidade é também utilizado no algoritmo de decisão para determinar se a saída requerida é crítica (*Critical_flag=1*) ou não crítica.

Liew *et al.* (2017) apresentam um estudo sobre a programação de N versão para o desenvolvimento independente, baseado em *benchmarks*, de duas extensões para o KLEE (*symbolic virtual machine*) para suportar o raciocínio simbólico sobre a aritmética de ponto flutuante. Desenvolvido segundo um procedimento rigoroso de desenvolvimento independente de *benchmarks*, pretendem melhorar as ferramentas de desenvolvimento fornecendo uma base útil para futuros desenvolvimentos.

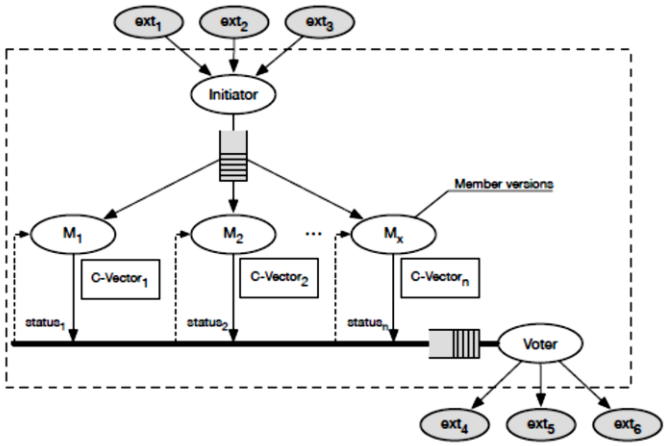


Figura 5-5. Framework de modelação NVP (Hu, 2017)

Por seu lado, Hu *et al.* (2017) propõem um padrão de modulação de tolerância a falhas baseado na programação de N versões que possa integrar-se, de forma transparente, nas aplicações existentes melhorando o seu funcionamento, mantendo as características de tempo. O modelo, desenvolvido em linguagem C, é constituído por um conjunto de componentes (*initiator*, *member versions* e o *voter*). Se por um lado, encapsularam diversos algoritmos alternativos para obtenção das mesmas saídas (*member version*), introduziram um novo componente (*initiator*) que visa auxiliar a modelação (Figura 5-5) agrupando as entradas nas quais o módulo originalmente trabalha, interligando-as ao *member version*.

Esta abordagem enquadra-se plenamente na nossa abordagem uma vez que a metodologia aplicada se cinge à replicação dos elementos de *software* que são críticos. Toda a restante estrutura permanecerá inalterada. Neste sentido podemos, igualmente, replicar *t* versões dos elementos de *software* efetivamente críticos para um ou mais componentes. Algumas das propostas apresentadas são efetivamente ideias que se integram na modelação da nossa abordagem. Estas referem-se essencialmente à linguagem de desenvolvimento, C, encapsulando

os algoritmos em FBs. A integração dos diversos membros, que poderemos dizer componentes distribuídos da nossa aplicação, é um fator de trabalho da aplicação em ambiente distribuído.

Sinca e Szász (2017) apresentam uma solução de redundância modular tripla (TMR) aplicada a sistemas de *hardware* digital. Estes abandonam, igualmente, a conhecida configuração de votação maioritária de 1 bit, ampliando-a e generalizando-a para o barramento de controle de um sistema de controlo digital. Com esta abordagem pretendem implementar um sistema de tolerância a falhas em sistemas digitais, em ambiente industrial, como seja o controlo confiável de servos bem como arquiteturas de *hardware* de computação paralela e distribuída de alta confiabilidade. Para isso desenvolveram, em Matlab/SimulinK um modelo que permitiu testar a capacidade de votação de um circuito de 4 bits digitais implementando-o fisicamente.

Por outro lado, Abdulhay *et al.* (2018), com vista a aumentar a confiabilidade dos sistemas, referem que os projetistas e os engenheiros devem criar sistemas inteligentes e altamente confiáveis adicionando redundância de *hardware*, de *software* e de tempo. Neste sentido os autores apresentam um estudo que visa melhorar a confiabilidade dos sistemas recorrendo à redundância de *hardware*. Para isso, estes passam da tradicional abordagem aos problemas de confiabilidade baseada na redundância modular tripla (TMR), técnica mais comum para tolerar um único erro SEU (*Single Event Upset*), para uma votação por maioria de redundância modular quintuplica (QMR). Esta alteração ao número de réplicas aumenta a capacidade de tolerância a erros passando de um para dois erros SEU tolerados. A consolidação dos resultados finais é também ela abordada segundo três novas propostas de votação maioritária (MV – *Majority Voter*) aplicado ao tratamento de imagens médicas.

A abordagem apresentada anteriormente visa melhorar as características destes modelos de confiabilidade aplicados a contextos industriais e médicos. Estes procuram a melhoria das características do modelo garantindo um resultado com um maior nível de confiança. A aplicação industrial dos modelos de tolerância a falhas e o aumento da confiabilidade dos sistemas é efetivamente um dos nossos objetivos. A abordagem industrial destes autores é uma abordagem a levar em conta no nosso modelo.

5.4 Conceito das Mensagens Temporizadas

As mensagens temporizadas são baseadas no conhecimento global do tempo de envio, no instante em que a tarefa se encontra pronta para a execução e na resposta dos piores tempos de execução das tarefas. Portanto, é possível, usando esse conhecimento global, ler a versão mais recente de um dado valor disponível para todas as réplicas.

Nesta abordagem, as mensagens são associadas com um tempo de validade. Este tempo de validade é definido como o instante em que o valor da mensagem se torna válido, ou seja, o instante em que o sistema é informado de que todas as réplicas já escreveram o valor. Tal prazo de validade é definido em (Poledna *et al.*, 2000). Por outro lado, e com o propósito de garantir que os elementos replicados leem o mesmo valor, é necessário armazenar várias versões da mesma mensagem. Os recetores devem ler a versão que tem o prazo de máxima validade do tempo de libertação da tarefa (Poledna *et al.*, 2000) ou seja a mensagem mais recente. Esta é a abordagem utilizada na arquitetura GUARDS a fim de garantir o comportamento determinista das replicadas (Wellings *et al.*, 1998). No projeto GUARDS este mecanismo é usado explicitamente na conceção, aplicação e execução, forçando os programadores do sistema a lidar simultaneamente com as exigências do sistema bem como com os problemas de replicação.

Uma vez que as mensagens temporizadas são baseadas no conhecimento global do tempo e no pior tempo de resposta da tarefa será possível usar este conhecimento global para ler a última versão do dado disponibilizado para todas as réplicas. Neste sentido haverá a necessidade de definir-se o tempo de validade de cada uma das mensagens no instante em que esta se torna

válida. As mensagens tornam-se válidas assim que todas as réplicas as receberem pelo que, o tempo de validade de uma mensagem $m_k(v)$ pode ser definido como (Poledna *et al.*, 2000):

$$m_k(v) = \begin{cases} W & -\ intraprocesso \\ W + \Delta + \varepsilon & -\ interprocesso \end{cases} \quad (5.1)$$

onde $m_k(v)$ é o tempo de validade da mensagem m_k, W é o pior tempo de resposta das mensagens registadas, Δ é o pior tempo de entrega da mensagem e ε é a máxima diferença entre os relógios do sistema. Por outro lado, e uma vez que no sistema replicado as mensagens são enviadas mais que uma vez para o recetor poderá haver a necessidade de armazenar várias versões da mesma mensagem. Neste contexto deverá ler-se a mensagem que possuir o tempo de validade mais antigo que o tempo de libertação da mesma. Segundo (Poledna *et al.*, 2000) este será definido como:

$$m_k.receção(tr_j) = max_{i=0}^{n}(m_k : m_k[i](v) \leq tr_j) \quad (5.2)$$

onde tr_j é o tempo associado à receção da mensagem e n é número de diferentes versões recebidas, até ao momento, para a mensagem m_k.

Deve-se considerar ainda que os sistemas distribuídos de tempo real necessitam de uma sincronização precisa entre os participantes e o relógio padrão apresentando, geralmente, desfasamentos na ordem dos milissegundos. Por outro lado, como ocorrem erros na configuração inicial dos relógios e escorregamentos (*Drift*), será necessário corrigi-los, periodicamente, para que se encontrem consistentes com o relógio padrão.

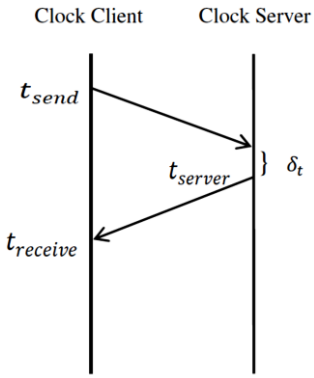


Figura 5-6. Sincronização entre o relógio do servidor e o relógio do cliente (Li et.al., 2015)

Assim sendo, poderão ser utilizados dois protocolos de sincronização de relógios para redes distribuídas que são o NTP (*Network Time Protocol*) baseado numa arquitetura *cliente/servidor* e PTP (*Precision Time Protocol*) baseado na arquitetura *escravo/mestre*. Li *et al.* (2015) propõem um mecanismo de sincronização de relógios em que o servidor de relógio, sincronizado por um padrão onde os clientes se encontram ligados por uma rede local usando o protocolo TCP/IP, vai corrigindo periodicamente o seu escorregamento conforme esquema apresentado na Figura 5-6. Estes estudaram o comportamento da rede LAN, atrasos de transmissão que normalmente são menores que a resolução mínima do relógio do sistema, em comparação com a WAN concluindo que as mudanças de carga a que as redes se encontram sujeitas durante as sucessivas

transmissões são pequenas. No entanto, ao utilizar apenas o *software* NTP, em LINUX, para sincronizar os relógios estes verificaram que, os erros de relógio medidos (*valor absoluto*) entre o cliente e o servidor são relativamente grandes, atingindo um máximo de 6 ms. Com a aplicação da metodologia proposta, utilização de um mecanismo de sincronização extra, reduziram o erro para valores na ordem dos 3 ms os quais poderão tornar-se significativos nos sistemas de tempo real mais exigentes.

É esta a metodologia que pretendemos implementar na nossa abordagem à replicação de sistemas distribuídos utilizando a norma IEC 61499. A utilização de mensagens temporizadas será o âmago do modelo a desenvolver. Por outro lado, a sincronização é também uma ferramenta essencial para a garantia do determinismo das réplicas, para a sincronização, pelo que adotaremos o protocolo de sincronização de tempo global NTP.

5.5 Sistemas tolerantes a falhas e de tempo real

O projeto MARS (Kopetz *et al.*, 1989) é um sistema distribuído tolerante a falhas desenvolvido para um sistema de tempo real, destinado a suportar aplicações de controlo de processo. A arquitetura é composta por um ou mais *clusters* distribuídos e interligados por uma rede em tempo real. Todos os componentes mantêm uma base de tempo global, permitindo-lhes sincronizar as suas ações e usar uma abordagem de acionamento temporal. Em MARS, os nós e o escalonamento da rede são determinados *offline* e armazenados numa tabela de escalonamento estático. Os componentes são concebidos para trabalharem como um sistema silencioso em caso de avaria (*silent fail*), através da utilização de *hardware* de autocontrolo, correndo em dupla redundância ativa, e com duas redes redundantes em tempo real, onde as mensagens são enviadas em duplicado.

O projeto DELTA-4 (Powell, 1991) teve como objetivo desenvolver uma arquitetura aberta e confiável para grandes sistemas distribuídos em tempo real. Em Delta-4 os nós são divididos em dois subsistemas diferentes: o hospedeiro, que é um componente de uso genérico (COTS – *Commercial Off-The-Self*), e a Rede NAC (*Network Attachment Controller*), componente silencioso em caso de avaria, utilizando um *hardware* de autocontrolo especializado. A necessidade de orientar os sistemas com requisitos mais rigorosos de tempo levou à utilização de uma arquitetura com especificações baseadas em arquiteturas de elevada *performance (XPA – eXtended Performance Architecture*) (Barrett *et al.*, 1990). XPA são sistemas constituídos por um conjunto de nós homogéneos distribuídos, ligados a uma rede LAN com propriedades de tempo real, onde o hospedeiro é também considerado como silencioso em caso de avaria. No entanto, ao contrário do NAC, o comportamento silencioso do hospedeiro é obtido através da utilização de técnicas de deteção de falhas silenciosas, onde o comportamento silencioso, em caso de avaria, é conseguido através de *software* de gestão das réplicas tendo por base uma redundância temporal.

O projeto GUARDS (Powell, 2001) visa desenvolver uma arquitetura genérica com base na utilização de componentes de uso genérico tentando minimizar o tempo de desenvolvimento e os custos associados a aplicações críticas de tempo real. A arquitetura é implementada com recurso a mecanismos de tolerância a falhas baseados em *software*, com a finalidade de lidar com a falta de fiabilidade dos componentes de uso genérico. A dificuldade em prever mecanismos de tolerância a falhas levou ao desenvolvimento de uma abordagem de replicação em dois níveis. No primeiro nível a arquitetura é constituída por um conjunto de canais, cada um contendo hospedeiros duplicados interligados por um sistema de memória partilhada. Estes canais são interligados a um segundo nível através de uma rede de comunicação ICN (*Interchannel Communication Network*), baseada em ligações unidirecionais em série, interligando os vários canais. Esta abordagem torna-se de difícil implementação tanto quanto maior for o número de canais a interligar. Por outro lado, o ICN tem de ser programado *offline* segundo uma tabela ordenada e estática, levando a uma sobrecarga de análise do sistema e ao consequente aumento

da dificuldade em realizar alteração no esquema da aplicação. Dado que esta arquitetura é orientada para a segurança ou para sistemas críticos e dado que a replicação dos canais é apenas motivada pela tolerância a falhas, não há necessidade dos sistemas possuírem mais de 3 ou 4 canais.

O projeto DEAR-COTS (Pinho, 2004) teve como objetivo desenvolver uma infraestrutura transparente e confiável para sistemas distribuídos em tempo real utilizando a linguagem de programação Ada95, em conformidade com o perfil *Ravenscar* (Burns, 1997). A arquitetura é genérica, desenvolvida com base na utilização de componentes de uso genérico, utilizando um mecanismo baseado em mensagens temporizadas para garantir o determinismo dos componentes replicados. Esta infraestrutura tem como objetivo permitir que o desenvolvimento das aplicações seja focalizado nos requisitos apresentados pelo sistema controlado, abstraindo-se dos mecanismos de baixo nível de suporte à replicação e à distribuição. Em DEAR-COTS a replicação é conseguida por duas formas distintas: primeiro, por uma camada de *software* subjacente, gestor de comunicação, que prevê os protocolos de comunicação adequados. Este oferece uma interface de comunicação de grupo, incluindo os mecanismos de comunicação necessários para a replicação e para a distribuição fornecendo os protocolos para a comunicação, *multicast*, e para consolidação dos dados replicados. Em segundo lugar, por uma abstração extra, por meio de um repositório de objetos de tarefas de interação Figura 5-7.

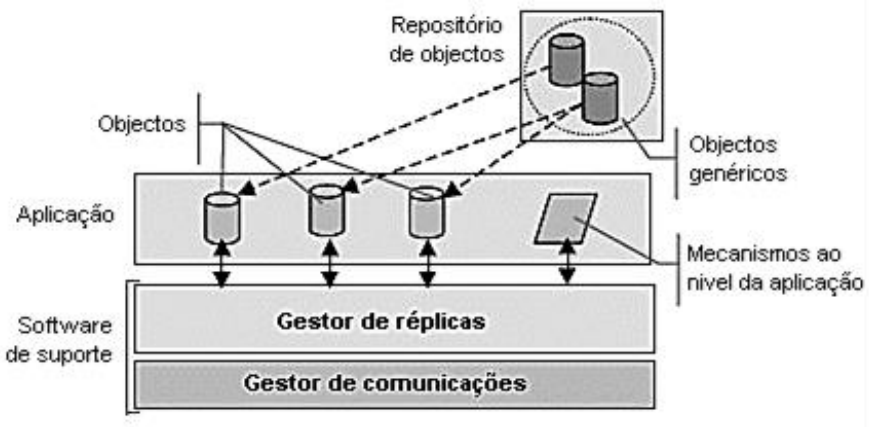


Figura 5-7. Infraestrutura de replicação (adaptada de Pinho, 2001)

Estes objetos têm por fim fornecer uma interface transparente, no qual os processos desconhecem os problemas de replicação e de distribuição. O seu tempo de execução é suportado pelo gestor de réplicas, que é também ele uma camada de *software* subjacente da aplicação. Esta camada é também responsável pela interface com os mecanismos de comunicação fornecidos pelo gestor de Comunicação. Juntas, essas duas camadas de *software*, constituem o *software* de suporte HRTS (*Hard Real-Time Subsystem*) usado na replicação ativa. O HRTS e uma camada intermédia localizada entre a aplicação e os componentes do sistema (Pinho, 2001).

As abordagens apresentadas por estes três autores são a base de desenvolvimento do nosso problema. A redundância implementada na nossa abordagem é uma redundância simples, replicação de um ou mais componentes, interligados por uma única rede de comunicação. Os nós do sistema distribuído são constituídos por elementos de baixo custo e de uso genérico e pelo protocolo de sincronizado de relógios NTP. A base de tempo associada a cada componente permite a sincronização das suas ações usando uma abordagem temporal. Os tempos de latência e de transmissão das mensagens são definidos *offline* e mantem-se estáticos. Pretendemos implementar uma infraestrutura transparente e fiável que suporte a tolerância a falhas suportadas por *software*. Os diversos objetos desenvolvidos e alocados ao repositório de objetos são responsáveis pelas comunicações, em *multicast*, e pela gestão das réplicas.

Wei *et al.* (2007) propõem uma nova abordagem à tolerância a falhas integrada com um algoritmo de escalonamento, baseado em prioridade fixa, para explorar a redundância e a escalabilidade de sistemas distribuídos tolerantes a falhas e de tempo real. Estes adotam, como base de suporte à tolerância a falhas, a *backup* primária com vista à tolerância a falhas permanentes de *hardware*. Neste sentido, consideram um típico sistema distribuído constituído por um conjunto de processadores, considerados idênticos, que executam uma série de tarefas periódicas e em tempo real. A este é associado um conjunto de cópias primárias de tarefas em tempo real periódicas, independentes e preensivas bem como, um conjunto de cópias de segurança das tarefas a que os autores chamaram de Tercos. Estes consideram ainda que a parte redundante da técnica (RP – *Redundant part*) deverá priorizar as tarefas dando prioridade às mais altas onde a parte de cópia (BP – *Backup part*) deve ser executada considerando que a falha é encontrada. Este procedimento aumentará a ocupação e a carga do processador associado enquanto outros poderão falhar. Tercos atribui a cada cópia de tarefa e processador dois cenários, *backup* ativa ou passiva. No primeiro cenário, não ocorre nenhuma falha no processador, enquanto no segundo caso, um processador falha durante a fase de execução das tarefas. Estes referem que o Tercos apresenta-se como uma estratégia eficiente de escalonamento (segundo a análise exata do pior tempo de resposta para uma determinada tarefa periódica em um único processador, WCRT – *Worst Case Response Time*) de um sistemas distribuídos de tolerância a falhas e de tempo real.

Peng e Yang (2015) propõem um algoritmo de escalonamento de tolerância as falhas denominado de RRFTGS (*Resource-Recovery Global Tolerant a Fault*) usando, igualmente, cópias passivas e ativas (*backups*). O modelo proposto reduz o custo da tolerância a falhas e o cálculo *online* da sua escalabilidade torna-se baixo pelo que, é adequado para sistemas de tempo real sólidos (*hard real-time systems*) com multiprocessadores.

Estes autores propõem abordagens com *backups*, modelo que não será implementado na nossa abordagem, visando a deteção de falhas em *hardware*. Estes modelos apresentam um escalonamento de tarefas eficiente jogando com os piores tempos de execução (WCET). A priorização é aplicada sacrificando a ocupação e a carga associada aos processadores. No entanto, esperam uma redução dos custos de implementação tornando-o adequado a sistemas de tempo real robustos.

Tabbaa *et al.* (2011) apresentam o problema da tolerância a falhas usando um algoritmo com base na replicação ativa programando ε+1 réplicas de cada tarefa para obter a tolerância a falhas requerida. Estes pretendem escalonar as tarefas de tolerância a falhas para os diversos nós de processamento heterogéneos em sistemas de computação em *cluster*. Usando o algoritmo DAG (*Directed Acyclic Graph*) tratam as informações respeitantes ao tempo de execução das tarefas nos processadores dos sistemas de destino, tempos de comunicação e o número de falhas do processador que devem ser toleradas pelo algoritmo de escalonamento. Assim, ao mapearem o algoritmo para os diversos recursos procuram minimizar o tamanho do escalonamento tolerando um determinado número de falhas silenciosas do processador (*fail-stop*). Na sua abordagem à tolerância a falhas os autores consideram que os nós são dinâmicos e a sua ordem é compatível com as restrições de precedência do gráfico, o que poderá ser conseguido se os nós livres forem escalonados. A ordem dos nós livres é definida por um valor de prioridade. Assim sendo, o escalonamento das tarefas de tolerância a falhas para um mapeamento DAG é realizado em *clusters* com processadores heterogêneos com comunicações replicadas uma vez que cada uma das réplicas ε+1, de cada uma das tarefas, receberá a mesma mensagem das réplicas ε+1 de cada uma das réplicas precedentes.

A abordagem apresentada pelos autores manifesta-se como uma abordagem pouco eficiente uma vez que se verifica a duplicação de mensagens. Esta abordagem é realizada com recursos a escalonamentos que podem englobar tanto os tempos de comunicação como o número de falhas a tolerar por cada *cluster*.

Bhat *et al.* (2018) apresentam uma abordagem centrada nos requisitos de recuperação temporal (RTR – *Recovery Time Requirement*) considerando a tolerância a falhas de *software* em sistemas críticos de segurança e de tempo real. Neste sentido propõem várias técnicas que permitam determinar a redundância das tarefas satisfazendo os requisitos do RTR. Estes desenvolvem as suas metodologias usando heurísticas computacionais eficientes que permitam encontrar uma alocação viável das tarefas e das suas cópias redundantes. Para isso, os autores utilizam um sistema distribuído constituído por vários nós computacionais que comunicam entre si por mensagens assumindo-se que as tarefas com alta prioridade são imediatamente processadas em detrimento das tarefas com menor prioridade. Cada uma destas tarefas possui um WCET *i* com um período de tempo *i* e um prazo de validade implícito *i* que se igualam. Assim, propõem a heurística TPCDC+R (*Tiered Placement Constrained Decreasing* con *Cold standby* e *Recovery-time*), com recuperação temporal, para que as restrições RTR possam ser contempladas. Complementam esta proposta com mais duas heurísticas adicionais denominadas RTT (*Recovery Time-Tiered*) e TRTI (*Tiered Recovery-Time Constraint Increasing*) que priorizam as restrições de RTR na sequência de alocação de tarefas.

Esta é uma abordagem centrada nos requisitos temporais com base na redundância de *software*. Estes associam um WCET e prazo de validade a cada uma das mensagens transmitidas garantindo a sua execução de acordo com a sua prioridade. Combinam a sua abordagem com duas heurísticas com vista a priorização em detrimento do tempo em que se tornaram válidas. Esta priorização pode levar à retenção e, consequente expiração de uma mensagem, com baixa prioridade. Este modelo não se coaduna com a nossa abordagem.

5.6 IEC 61499

Sousa *et al.* (2015a) abordaram o problema da tolerância a falhas no contexto IEC 61499. Estes, utilizando o ambiente de desenvolvimento *Eclipse 4diac*, tendo como suporte de implementação a plataforma de execução FORTE IEC 614499, criaram alguns novos FBs destinados a suprir os vários cenários de comunicação entre réplicas, já referidos por Sousa e Santos (2007) e por Santos e Sousa (2010). Neste trabalho os autores desenvolvem novos pares de comunicação *publish/subscribe* capazes de garantirem uma comunicação *multicast* entre réplicas bem como o seu determinismo, Figura 5-8. Estes novos SIFBs, trabalhando com mensagens temporizadas, garantiam que todas as réplicas recebiam os mesmos dados, na mesma ordem e segundo o instante em que foram disponibilizados. Deste modo, as diversas camadas da arquitetura de comunicação (SIFBs adicionadas pelo designer da aplicação) foram implementadas com recurso do campo de identificação ID do respetivo SIFB de comunicação (sintaxe/semântica especificada no FORTE) pelo que, na comunicação *publish/subscribe* a cadeia de identificação "225.0.0.1:61499" assumirá automaticamente o formato "fbdk []. Ip [225.0.0.1:61499]". A camada respeitante à comunicação entre réplicas deverá ser convertida para "fbdk [].replication[time].ip[225.0.0.1:61499]" de modo a garantir o envio do instante em que os dados ou eventos são disponibilizados.

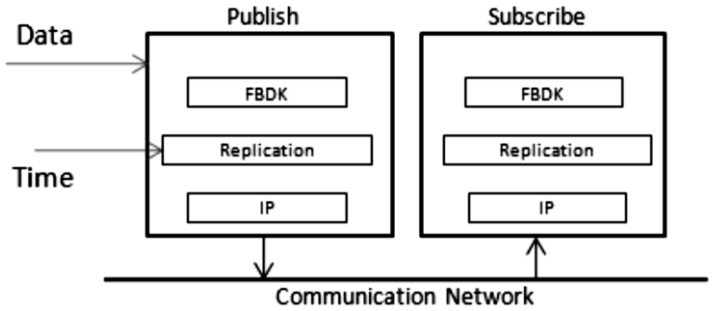


Figura 5-8. Publish/Subscribe layered architecture (Sousa et al., 2015b)

Pela integração destas características a eficiência do sistema de controlo dependerá da sua capacidade de tolerar falhas ou da sua capacidade de resposta a situações imprevistas. Esta maior ou menor eficiência do sistema traduz-se na forma como este lida com a ocorrência de situações excecionais minimizando as consequências da falha. Nesta situação a descentralização do controlo poderá apresentar-se como uma solução vantajosa se cada elemento de *software* (elemento de controlo IEC 61499) possuir capacidades, de forma autónoma, para lidar com a falha ou situação imprevista. Cada elemento de *software* controla uma dada secção do sistema de manipulação pelo que torna-se importante que os restantes elementos possam colmatar a falha de um dado elemento sem que o desempenho do sistema seja posto em causa. Neste sentido deve ser considerada uma abordagem transparente para a replicação e para a distribuição permitindo uma programação mais simples, de modo que os programadores se abstraiam dos detalhes associados à distribuição e aos problemas de replicação.

A abordagem apresentada neste artigo encontra-se, de algum modo, em consonância com o objetivo do modelo que se pretende desenvolver. No entanto, uma abordagem que tem por finalidade criar novos blocos função que integre todo o modelo de replicação poderá tornar-se uma tarefa complexa. Isto obriga também à criação de pelo menos dois FBs que contemplem a replicação simples, mas também TMR. A modelação que se propõe é uma modelação mais simples uma vez que tira partido de todos os objetos disponibilizados pelo repositório do aplicativo. Isto é, utiliza-se em primeiro, lugar todos os FBs standards e só depois se desenvolvem novos FBs estritamente necessários ao desenvolvimento da aplicação. Desenvolvimento da interface de replicação usando, principalmente, os SIFBs disponibilizados pela norma.

Lednicki *et al.* (2013) apresentam um modelo para determinar o pior tempo de execução de modelos de *software* IEC 61499, analisando, hierarquicamente cada nível de composição. Este modelo pode ser utilizado em conjunto com o aplicativo de *software* ou reutilizado noutro sistema. Estes autores definem a informação segundo dois conjuntos de dados agrupados em WCET de eventos e WCET de período, Figura 5-9. Assim, o conjunto de dados do evento é considerado como as entradas com informações associadas tendo em conta a execução iniciada pela chegada dos eventos de entradas aos blocos de função. O valor de WCET representa o valor máximo do tempo que um FB necessita para executar a sua funcionalidade (tarefa), desde a entrada de um evento até que a sua atividade interna seja concluída (saída de evento), independentemente do caminho ou dos caminhos internos de execução. Os autores definem uma série de premissas necessárias à análise dos diversos princípios de funcionamento dos FBs (BFB, CFB, SIFB, etc.).

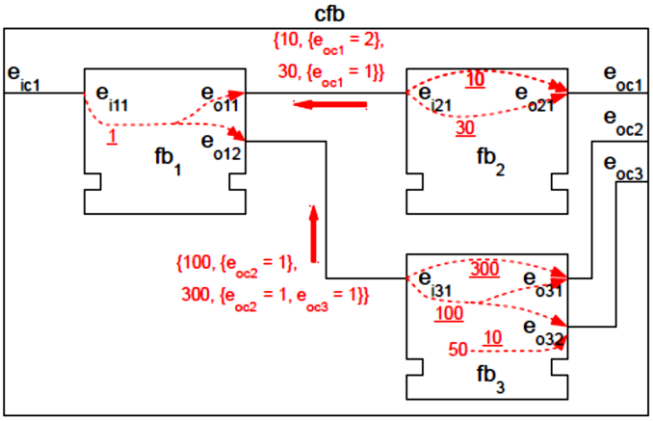


Figura 5-9. Exemplo da análise de um Composit Function Block (Lednicki et al., 2013)

A obtenção de sistemas distribuídos de tempo real passa pelo desenvolvimento de aplicações onde o domínio do tempo assuma um papel preponderante. O conhecimento dos parâmetros

WCET e WCRT, por exemplo, são fundamentais para a sincronização das réplicas. Os autores apresentam vários procedimentos para a obtenção/medição dos WCET dos diversos FBs tendo em conta os eventos de entrada, as tarefas a executar, de acordo com os caminhos internos, e da sua finalização (evento de saída). Estes tempos são fundamentais para determinar os tempos de libertação dos dados processados por cada uma das réplicas. Metodologia que será utilizada no modelo a desenvolver.

5.7 Tolerância a falhas em contexto industrial

Schütz *et al.* (2013) apresentam um conceito de implementação de *software* para automação industrial capaz de melhorar a confiabilidade do sistema recorrendo a *soft* sensores. Este conceito é apresentado tendo como suporte da aplicação dispositivos de automação standards tais como os controladores lógicos programáveis (PLCs) e requisitos de tempo real. O sistema é constituído por até cinco cilindros hidráulicos, controlados individualmente, acoplados a sensores de pressão e de distância. Os sensores são fisicamente ligados por um sistema de barramento de campo a vários PLCs, que executam o software de controlo. Neste modelo de redundância, os autores consideram que cada nó identificado representa um ponto de medição o qual é equipado com um sensor convencional e um *soft* sensor predefinido como elemento de referência. Um valor de qualidade em cada um dos nós identificados representa a precisão (desvio padrão) do valor medido ou calculado no nó. Durante a execução o modelo de tolerância é calculado com base nos sensores que apresentam melhor precisão verificando se não são violadas as restrições de funcionamento da máquina e de produção. O modelo definido pode ser comparado com uma estrutura em árvore de falhas. Esta descreve as dependências entre a precisão das funcionalidades da máquina (componentes) e os valores de referência usados nos sensores (sensores convencionais e *soft*) numa estrutura modular hierárquica (Figura 5-10).

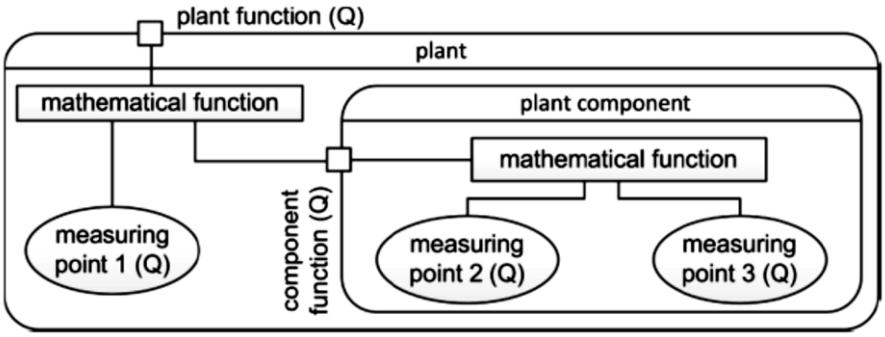


Figura 5-10. Exemplo do modelo de tolerância

A deteção de falhas com base no modelo de redundância implementada usa diferentes fontes de informação para identificar um sensor com defeito. No caso de um ponto de medição, são vários os sensores disponíveis (redundância de *hardware*) ou medições que podem ser usadas para calcular um valor num determinado ponto de medição usando dependências analíticas (*soft* sensores). Por outro lado, se pelo menos três canais de informação se encontrarem disponíveis, poderá ser identificado um canal defeituoso por votação maioritária. Neste sentido, a definição de intervalos de valores de controlo precisos impõem limites de ação que podem conduzir a deteção de erro sem falha (limite muito pequeno) ou ao risco de não deteção de erro (limite muito grande). A estrutura de controlo do sistema foi implementada sobre uma hierarquia de Blocos Função IEC 61131-3, encapsulando cada componente de controlo, agente de *software*, num FB.

Os autores apresentam uma abordagem à tolerância a falhas suportadas pelos PLCs de uma máquina hidráulica. Utilizam a IEC 61131 (programada por encapsulamentos do *software* em FBs) dentro de todas as restrições que a norma e os PLCs podem apresentar. A comparação de valores

esperados e os respetivos desvios funcionam como elementos de controlo dos parâmetros que poderão conduzir a erros que advirão dos limites de precisão impostos, leituras erróneas. Com o tratamento destes valores segundo a IEC 61499 traduzir-se-ia numa mais-valia uma vez que se poderia tirar partido das características associados aos sistemas distribuídos que a respetiva norma preconiza. A recolha destes sinais poderia ser obtida num ambiente distribuído e dados por um qualquer componente de baixo custo com capacidade para os executar.

5.8 Conclusão

Abordamos neste capítulo uma série de trabalho desenvolvidos com o objetivo da tolerância a falhas em sistemas distribuídos e, também, de tempo real. A tolerância a falhas é abordada tendo em conta o escalonamento de tarefas que podem ser priorizadas e, como tal, tratadas segundo o seu grau de prioridade, deixando para processamento posterior as de menor prioridade. Outras aproximações procuram minimizar as falhas de funcionamento dos CPUs garantindo que os desfasamentos entre os processadores e, possíveis maus funcionamentos, não conduzem a erros de execução utilizando, não só mensagens temporizadas, mas também a redundância das mesmas.

A abordagem centrada nas mensagens temporizadas é uma abordagem que nos permite obter o determinismos das réplicas quer tendo por base o *software*, o *hardware* ou ambos. Poledna utilizou as mensagens temporizadas, associando a cada um dos dados, disponibilizados, o instante em que estes se tornam válidos. Constata-se ainda que o conhecimento global do tempo e o pior tempo de resposta de cada uma réplica é fundamental para se garantir o determinismo do sistema. Este conhecimento do tempo de validade das mensagem terá de ser obtido pela sincronização de relógios por meio de um dos protocolos de tempo NTP, tendo por base uma arquitetura *cliente/servidor*, ou PTP, arquitetura *escravo/mestre*. Alguns dos autores referidos utilizaram ainda, como base de desenvolvimento dos seus trabalhos, componentes de uso genérico que será, também, a base dos trabalhos que se pretendem desenvolver.

Capítulo 6

Norma IEC 61499

6.1 Introdução

Os sistemas de controlo suportados por sistemas computorizados estão presentes num largo espectro de utilizações, que vão desde a automação industrial e do controlo de processos, da utilização doméstica e dos serviços até à robótica e às aplicações críticas em sistemas fixos e móveis. Em todas estas áreas de aplicação, são utilizados computadores para controlar o meio envolvente, sendo esperado que estes reajam a estímulos, externos e internos, de acordo com os requisitos impostos pelo ambiente a controlar.

No âmbito do controlo de processos industriais é dominante a utilização de Controladores Lógicos Programáveis (PLCs – *Programmable Logic Controllers*). Assim sendo, e dada a crescente utilização de PLCs para a implementação de sistemas de controlo industrial, torna-se necessário que estes sejam capazes de responder a requisitos de tempo para que a execução de programas com restrições de tempo real e elevada fiabilidade seja possível. Neste sentido, a Comissão Eletrotécnica Internacional (IEC) publicou várias normas que visam regulamentar a programação dos PLCs utilizados em automação industrial das quais se destaca a norma IEC 61131-3 (IEC 61131, 2013). Por outro lado, com a utilização de redes de comunicação industriais, usadas para interligar os PLCs, desenvolveram-se verdadeiros sistemas distribuídos e em tempo real para os quais, as linguagens de programação preconizadas na norma IEC 61131-3, controlo centralizado e execução cíclica, não eram capazes de dar resposta. Outros fatores como as diferentes solicitações dos mercados (prazos, qualidade, disponibilidade, etc.), a crescente competitividade dos mercados globais e as novas exigências, em termos das aplicações de controlo flexíveis e distribuído, levou a que a IEC pensasse em alternativas à norma IEC 61131.

Foi devido, principalmente a estas razões, que uma nova norma, a IEC 61499 (IEC 61499, 2012), foi criada. Esta vem definir uma nova estrutura para o desenvolvimento de sistemas de controlo distribuído permitindo também que novas potencialidades sejam exploradas, como seja a confiança no funcionamento, pela tolerância a falhas, tendo como base a replicação de componentes críticos.

A nova norma internacional revela-se então como uma tentativa para fornecer os conceitos arquiteturais em falta na IEC 61131. Na IEC 61499 define-se os meios para a portabilidade de *software* (Hanisch e Vyatkin, 2004) entre diferentes fabricantes, incorpora-se o encapsulamento de funções, distribuição dos sistemas e a execução por eventos. Neste sentido foram introduzidas uma série de características que permitem desenvolver sistemas de controlo distribuído (IPMCS – *Industrial Process Measurement and Control Systems*), nomeadamente:

- ***Operabilidade*** – capacidade dos dispositivos de diferentes fabricantes operarem em conjunto garantindo ao programador do sistema abstração do *hardware*.
- ***Portabilidade*** – capacidade das ferramentas de *software* aceitarem e interpretarem bibliotecas produzidas por outras ferramentas.
- ***Modularidade*** – capacidade do sistema ser dividido em partes distintas e independentes por meio de programação estruturada. Permite a reutilização dos módulos de *software*.
- ***Event-driven*** – a execução dos algoritmos é orientada a eventos apoiada por um fluxo de dados.

A título de resumo pode dizer-se que a arquitetura IEC 61499 foi concebida na perspetiva do aumento da procura de sistemas de controlo que utilizam automação distribuída. Incorporando várias soluções que pretendem fazer frente aos desafios da automação distribuída a IEC 61499 propõe uma série de linguagens de desenvolvimento quer ao nível dos sistemas distribuídos quer dos sistemas de controlo. Esta vem colmatar a lacuna existente entre as populares linguagens de programação de PLCs, preconizadas na IEC 61131, e os sistemas distribuídos (Vyatkin, 2009).

6.2 Conceitos básicos

A norma IEC 61499 define uma série de funcionalidades básicas, funcionais e estruturais, que podem ser utilizadas na definição de especificações e no desenvolvimento e implementação de sistemas de controlo distribuído. Estas funcionalidades podem apresentar-se como combinações puras de *software* ou como uma abstração lógica do *software* e do *hardware*.

Assim, o desenvolvimento de um qualquer sistema de controlo, suportado pela IEC 61499, possui como pedra basilar a funcionalidade designada de "bloco função" (FB – *Function Block)*. O FB é considerado a entidade básica de um qualquer componente de *software* portável (IEC 61499, 2012). É com base neste princípio, que a norma, disponibiliza várias funcionalidades de *software* entra as quais se podem destacar os blocos função básicos (BFB – *Basic Function Block*), os FBs compostos (CFB – *Composit Function Block*, constituídos por vários FBs que por sua vez podem conter outros blocos função compostos) até aos blocos função de interface utilizados nas comunicações entre módulos (SIFB – *Service Interface Function Block*).

Embora o termo bloco função seja já conhecido da norma IEC 61131, um bloco função, nesta nova norma, apresenta-se com uma estrutura e características diferentes dos blocos função preconizados na norma IEC 61131. Neste sentido a IEC 61499 predefine uma grande variedade de blocos de função standards e específicos para a manipulação de eventos e dados. Destes destacam-se a divisão ou a concatenação (*split event* ou *merge event*), a geração de eventos com atraso (*event delay*), etc.

Na Figura 6-1 apresenta-se o aspeto geral da interface do bloco função preconizado pela IEC 61499 (note-se que o aspeto gráfico exterior do bloco função não é normativo). Como se depreende da figura os FBs são constituídos por um bloco superior normalmente designado de cabeça do bloco função, local onde se realiza a entrada e a saída de eventos, e por um bloco inferior chamado de corpo do bloco função, onde se realiza a entrada e a saída de dados.

Neste sentido o bloco função, considerado como um bloco de *software* onde são executadas as funções básicas de controlo, assume um papel preponderante no desenvolvimento e no controlo de sistemas, quer sejam centralizados quer sejam distribuídos. Cada sistema é composto por uma diversidade de FBs interligados ou, em alternativa, por um ou mais algoritmos e por uma máquina de estados denominada de ECC (*Execution Control Chart*). Os ECC definem os estados internos e as transições entre estados que devem ser ativados, após a chegada de um evento ou devido à alteração do valor de uma variável interna. Os ECC definem ainda os algoritmos que

devem ser executados e qual os eventos que os algoritmos devem gerar uma vez terminada a sua execução.

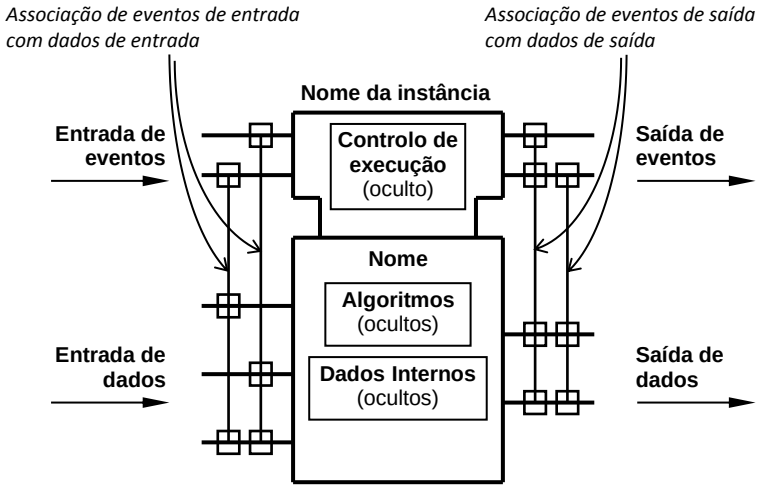


Figura 6-1. Interface de um bloco função básico (adaptada de IEC 61499, 2012)

Cada bloco função encontra-se interligado por eventos e/ou dados, constituindo uma rede de FBs. Os eventos são utilizados para a ativação dos FBs, enquanto os canais de dados são utilizados para transferir a informação entre FBs (dados do tipo inteiro, booleano, *string*, etc.). Nos FBs compostos os eventos que chegam são simplesmente transmitidos à rede interna do FB enquanto nos FBs básicos a receção de eventos implica na ativação do ECC e na mudança de estado do mesmo. Por cada ativação, um fragmento do algoritmo é executado e um novo evento pode ser gerado. Em consequência desta ativação, os algoritmos recebem dados dos canais de entrada produzindo novos dados que colocam nos canais de saída. Neste sentido, toda e qualquer especificação associada a um FB têm como função controlar a execução do bloco função, especificar qual o algoritmo que deve ser invocado, após a entrada de um evento, e qual o estado de execução da função de controlo que deve ser especificado através do Gráfico de Controlo de Execução. Os algoritmos contidos nos FBs podem ser escritos em qualquer linguagem de programação propostas pela norma desde que seja suportada pelo recurso que a executa. Estes podem ser escritos em todas as linguagens definidas na norma IEC 61131-3 (*Ladder* – LD, *Structured Function Chart* – SFC, *Instruction List* – IL, *Structured Text* – ST, *Function Block Diagram* – FBD) bem como com linguagens como o Java, o C, o C++, etc.

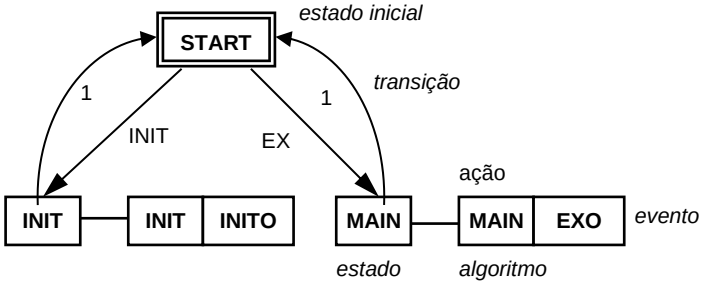


Figura 6-2. Exemplo de ECC – Gráfico de Controlo de Execução (adaptada de IEC 61499, 2012)

Na Figura 6-2 apresenta-se um exemplo de um gráfico de controlo de execução constituído por uma máquina de estados finitos e respetivo estado inicial, ECC. Este gráfico é composto por estados associados a ações (identificadas pelas caixas retangulares) e transições que estabelecem ligações seletivas entre os estados (representadas por setas orientadas). As ações (obedecendo a relações "n" para "n", ou seja, um estado pode possuir mais que uma ação e uma ação pode ser

utilizada por mais do que um estado) são compostas por algoritmos que, após invocação do evento *INIT (evento de inicialização)*, produzem eventos que estão na sequência da execução do algoritmo ativado (algoritmo "*INIT*" produz o evento de saída *INITO* (evento de confirmação da inicialização). A cada transição encontra-se associada uma condição booleana (expressão lógica) que pode utilizar um ou mais eventos – variáveis de entrada, variáveis de saída ou variáveis internas ao bloco função. Os eventos de entrada são representados através de condições booleanas, definidas como verdadeiras (*True*), sempre que se verificar a ocorrência de um evento, passando novamente a falsas (*False*), imediatamente após, a transição de todos os eventuais estados da transição se encontrarem transpostos. Por outro lado, só pode existir um único estado ativo a cada momento, pelo que é necessário que todas as transições que saem de um dado estado sejam mutuamente exclusivas.

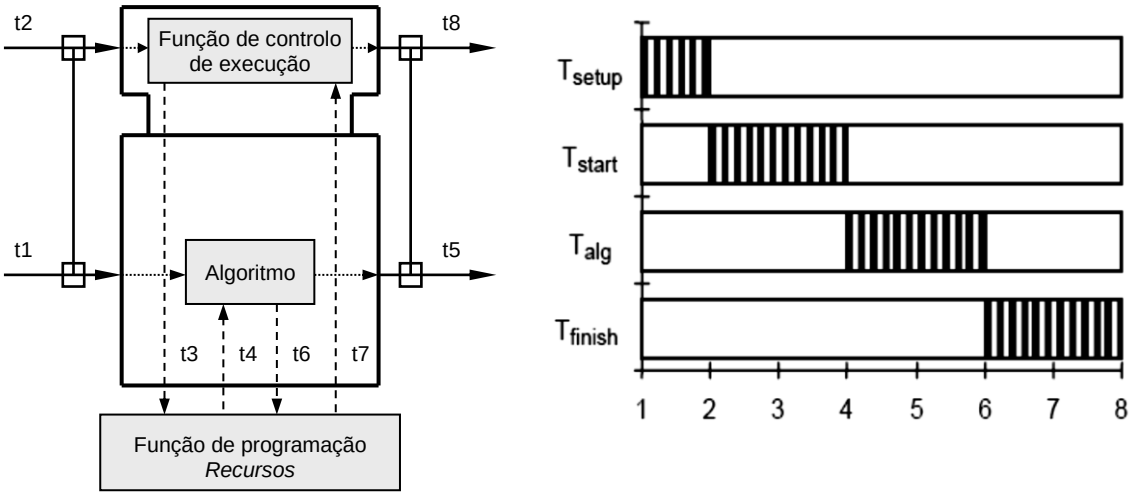


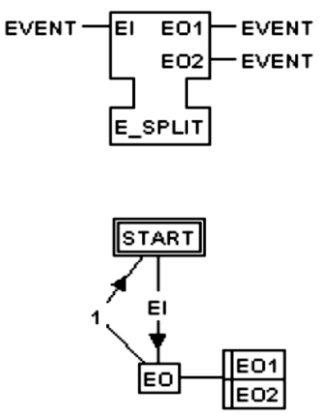
Figura 6-3. Modelo e tempo de execução de um Bloco Função (adaptada de IEC 61499, 2012)

A chegada de um evento ativa o ECC (Figura 6-3) que por sua vez ativa os algoritmos e os recursos associados à sua execução de acordo com a seguinte sequência:

1. A ocorrência de um evento de entrada, associado a um dado de entrada, despoleta a execução do bloco (tempo de ativação T_{setup} = t2 – t1).
2. A função de controlo de execução notifica a função escalonamento de recursos avaliando os estados da função ECC. Se nenhuma transição de saída do estado atual é ativada o processo segue para o passo 8 (t8). Caso contrário, se uma ou mais variáveis de transição de estado se encontram ativas, uma única transição de estado ocorre. O estado atual é substituído pelo estado seguinte preparando-se para a execução do novo estado. A função de controlo de execução notifica a função de controlo de recursos preparando o algoritmo para uma nova execução (T_{start} = t4 – t2, *tempo de entrada de eventos necessário para dar início ao ECC*).
3. O algoritmo de execução é iniciado (T_{alg} = t6 – t4, *tempo de execução do algoritmo*). Este estabelece a ligação entre os valores das variáveis de saída associando-as aos eventos de saída através do qualificador *with* (t5). Por fim a função de controlo de recursos é notificada do fim de execução do algoritmo de execução (t6).
4. A função de escalonamento de recursos invoca a função de controlo de execução (t7) assinalando os correspondentes eventos de saída, limpando os valores booleanos das variáveis correspondentes ao disparo dos eventos de entrada e de saída (T_{finish} = t8 – t6, *tempo de execução de um evento até à sua saída*).

Neste sentido pode dizer-se que o bloco função básico (BFB) é uma abstração de um componente de *software* adaptado às necessidades de controlo do sistema. Isto significa que as

especificações do bloco função podem ser feitas sem nenhum conhecimento especial do *hardware* em que será posteriormente executado. A título de exemplo, e para melhor ilustrar o que se acabou de expor, apresenta-se na Figura 6-4 a interface, o ECC e o código da duplicação de um evento de entrada de um bloco função básico, previamente definido pela norma IEC 61499 (Bloco função *E-Split*).




```
FUNCTION_BLOCK E_SPLIT (* Split an event *)
EVENT_INPUT
 EI; (* Input event *)
END_EVENT
EVENT_OUTPUT
 EO1; (* First output event *)
 EO2; (* Second output event, etc. *)
END_EVENT
EC_STATES
 START; (* Initial State *)
 EO: (* Extensible *)
 ->EO1, (* Output first event *)
 ->EO2; (* Output second event, etc. *)
END_STATES
EC_TRANSITIONS
 START TO EO := EI ;
 EO TO START := 1;
END_TRANSITIONS
END_FUNCTION_BLOCK
```

Figura 6-4. Interface, ECC e algoritmos do bloco função básico E_Split (IEC 61499, 2012)

6.3 Blocos função compostos

Os blocos função compostos (CFB), ao contrário dos blocos função básicos, são compostos por uma rede de blocos função interligados entre si (agregação de instâncias de FBs). Nestes, os eventos/dados de entrada e de saída, associados aos blocos função internos, interligam-se com os eventos/dados de entrada e de saída do bloco função composto estabelecendo a sequência de invocação. Na Figura 6-5 apresenta-se um esquema das interligações de um bloco função composto e as respetivas ligações.

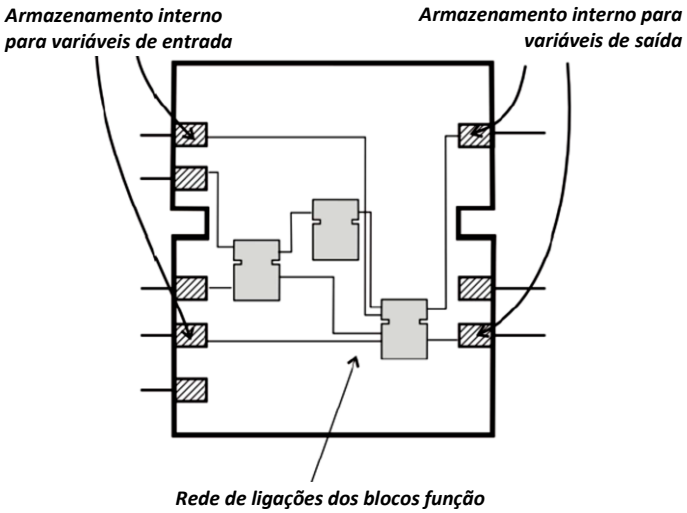


Figura 6-5. Bloco função composto (adaptada de Hanisch and Vyatkin, 2004)

Os CFB podem ainda ser constituídos por blocos função básicos, interligados entre si, ou por outros blocos função compostos organizados hierarquicamente. Na Figura 6-6 apresenta-se um esquema de uma possível interligação de blocos função compostos no seio de um bloco função composto.

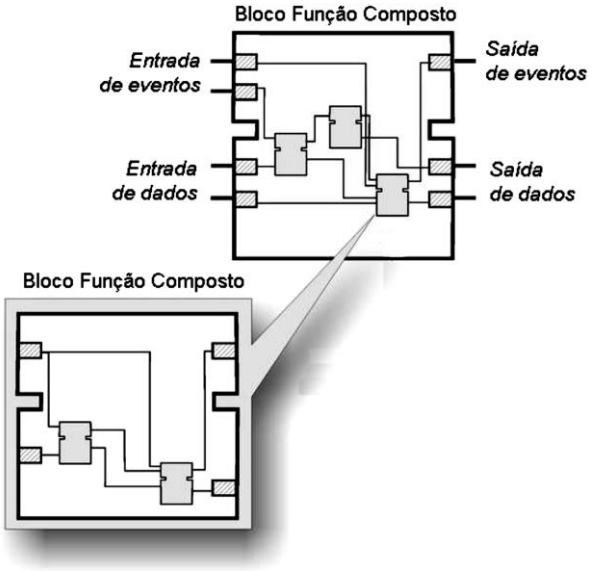


Figura 6-6. Composição hierárquica de um bloco função composto (adaptada de Vyatkin, 2015)

Por outro lado, os CFB ao não possuírem variáveis internas, exceto as utilizadas para guardar os valores de entrada e de saída dos eventos e dos dados, tornam-se dependentes do comportamento dos blocos função e das suas interligações. Assim, cada evento de entrada do bloco função composto deve encontrar-se ligado a um e só um evento de entrada do bloco função subsequente. Por sua vez os eventos de entrada dos blocos função devem estar ligados a não mais do que um evento de saída do bloco função anterior e vice-versa. No que respeita ao funcionamento dos dados de entrada (*data*) dos blocos função compostos podem não se encontrar ligados a nenhum bloco função, como podem estar ligados a mais do que um. No entanto, cada um dos dados de entrada dos blocos função só pode ser ligado a uma única saída de dados do bloco anterior. Do mesmo modo que os eventos, os dados de saída dos blocos função compostos devem encontrarem-se ligados a uma e só uma saída de dados dos blocos função que constituem o CFB. Nestes blocos deve-se utilizar o qualificador *with* para associar os eventos e os dados quer sejam de entrada quer de saída. Na Figura 6-7 apresenta-se um exemplo de utilização de blocos função compostos e a associação de eventos e dados.

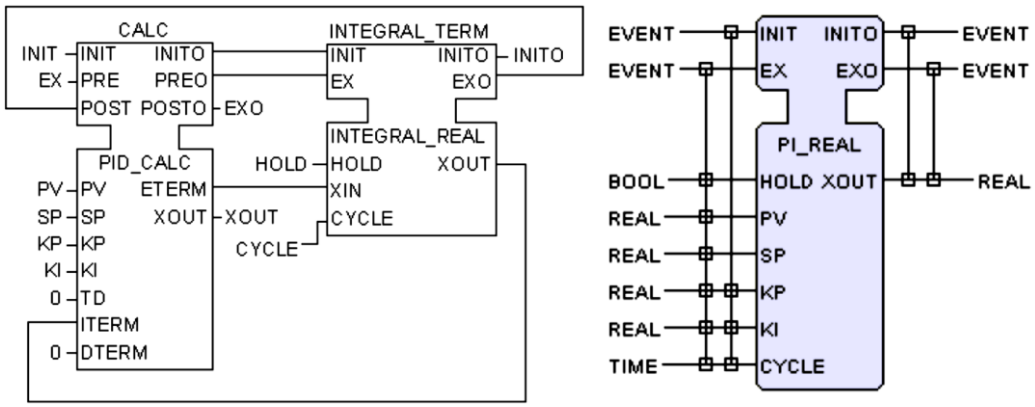


Figura 6-7. Bloco função composto, estrutura interna e interface (IEC 61499, 2012)

Note-se ainda que os blocos função compostos não possuem um ECC, no entanto, e segundo Vyatkin (2007), um bloco função básico dentro do bloco função composto poderá desempenhar esse papel, controlando a evolução do mesmo, ver Figura 6-8.

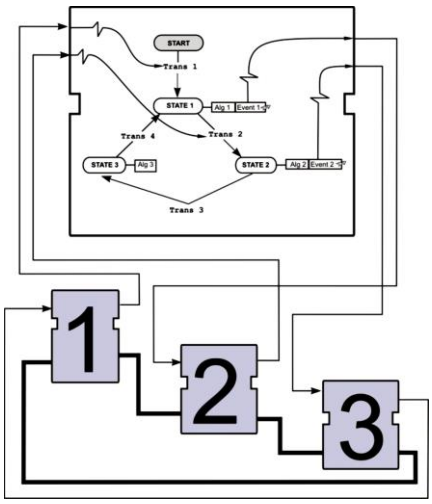


Figura 6-8. Bloco função utilizado como ECC num bloco função composto (Vyatkin, 2015)

Deve-se referir ainda que dentro da rede de FB é impossível que múltiplos FBs sejam ativados simultaneamente (a execução de um CFB é linear) pelo que um evento de saída não pode ser direcionado para mais que um evento de entrada.

6.4 Subaplicações

As subaplicações, à semelhança dos blocos função compostos, são também aplicações compostas por uma rede de blocos função interligados entre si. Na Figura 6-9 apresenta-se um esquema das interligações das entradas e saídas dos elementos que podem constituir uma subaplicação.

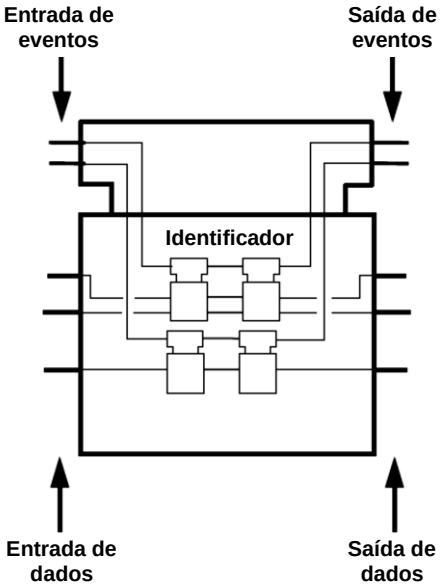


Figura 6-9. Esquema de um FB do tipo subaplicação (adaptada de IEC 61499, 2012)

Os eventos/dados de entrada e de saída associados aos blocos função internos interligam-se com os eventos/dados de entrada e de saída da subaplicação, estabelecendo a sequência de invocação dos blocos função. Note-se ainda que os elementos que integram estas interligações

podem ser constituídos por blocos função básicos ou por outras subaplicações organizadas hierarquicamente. Nestas o qualificador *with* (interligação) não é utilizado mantendo-se os eventos e os dados quer de entrada, quer de saída desassociados. Na Figura 6-10 apresenta-se um exemplo de utilização de subaplicações.

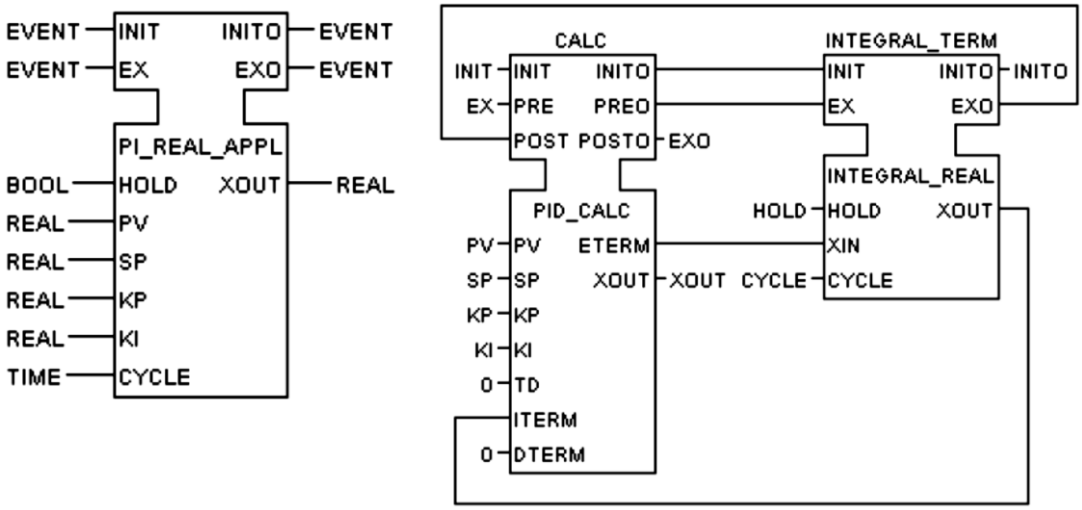


Figura 6-10. Interface e estrutura de uma subaplicação (IEC 61499, 2012)

6.5 Bloco Função para Interface de Serviços

Os FBs usados como interfaces de serviços são blocos especiais que, ao contrário dos blocos básicos e dos blocos função compostos, não se destinam a ser desenvolvidos pelos programadores. Os blocos de interface de serviço (SIFB – *Service Interface Function Blocks*) são diferentes dos restantes pelo facto de possuírem um único algoritmo no seu interior, são fornecidos pelos fabricantes do *software* com o intuito de proporcionar às aplicações uma interface com os serviços de comunicação e de gestão do processo (p. ex. controladores, blocos de entradas/saídas remotas, etc.). As interfaces SIFB possuem a mesma aparência de um bloco função básico utilizando semânticas e comportamentos específicos que permitem a interação entre as aplicações e os recursos do sistema. Na Figura 6-11 apresenta-se um exemplo de duas interfaces de serviços utilizados na gestão.

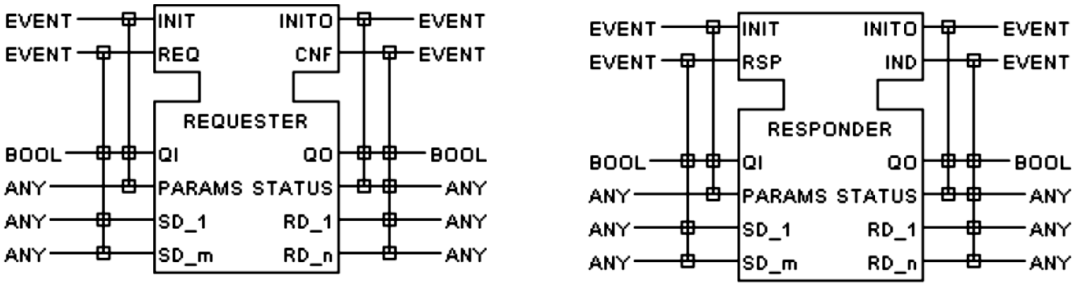


Figura 6-11. Exemplo de SIFB REQUESTER e RESPONDER (IEC 61499, 2012)

Os SIFBs são por isso abstrações de componentes de *software* destinados a aplicações de serviços. Estes fornecem serviços à aplicação em que se integram gerando eventos de saída (causados, por exemplo, pela leitura de valores de sensores, encerramento de um recurso, envio de uma mensagem, acesso remoto de dados, etc.) dependentes unicamente e fundamentalmente do algoritmo que os gere. Neste sentido, e de modo a facilitar a execução dos algoritmos, a norma

pré-define alguns parâmetros, de entrada e de saída, fundamentais para a execução do serviço. Entre estes podemos encontrar o parâmetro *INIT*, o *INITO*, o *QI* para entrada do qualificador, etc. (ver Figura 6-11). A juntar aos grupos de serviços mencionados anteriormente (comunicação e processo) a norma previu ainda outros tipos de serviços dos quais se destaca as interfaces gráficas.

Os SIFBs podem também ser especificados por diagramas de sequência temporal, onde os SIFBs são representados por duas linhas verticais definindo dois campos de interação, aplicação e recurso. Os serviços prestados por estes blocos, esquematizados na Figura 6-12, referem-se ao "estabelecimento normal" do serviço, "serviço normal" e ao "fim da aplicação de inicialização".

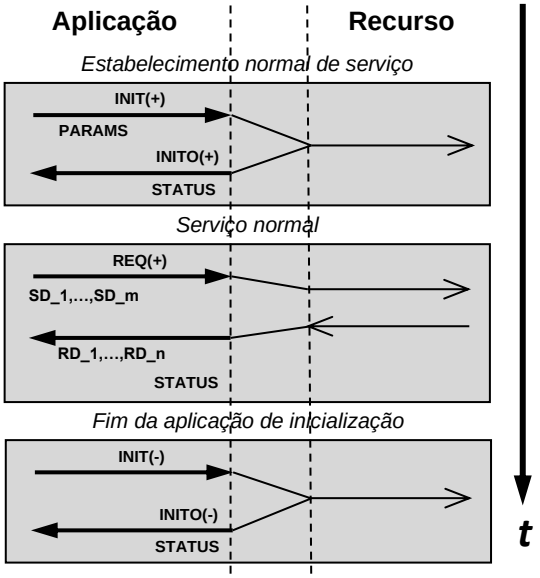


Figura 6-12. Diagrama de sequência temporal da aplicação (adaptada de Hanisch e Vyatkin, 2004)

O evento de entrada *INIT* destina-se à inicialização/finalização do serviço de acordo com o valor de entrada, verdadeiro ou falso, da variável booleana *QI*. Assim, a notação *INIT+* indica a ocorrência de um evento de entrada *INIT*, associado ao qualificador *QI* com valor verdadeiro, enquanto *INIT-* corresponde a um valor de *QI* falso. Os parâmetros de entrada, representados por *PARAMS* (ver Figura 6-11), representam os parâmetros de serviço associados à inicialização do serviço pelo que, no final da inicialização/finalização, o SIFB responde através do evento *INITO* indicando o sucesso da inicialização/finalização com a variável booleana de saída *QO* igual a verdadeira, ou indicando o insucesso, com *QO* igual a falso.

Os dados de entrada necessários para a execução do serviço são indicados pelas entradas *SD_1…SD_m* associadas, através do qualificador *with*, com o evento de entrada *REQ* (*Request-event*). Os dados de saída *RD_1…RD_n* recebem os dados processados, resultado do serviço prestado, associados ao evento de saída *CNF* (*Confirm-event*) que representa a confirmação da execução do serviço. A saída *STATUS* fornece informações sobre o estado do serviço através da ocorrência de um evento de saída (Figura 6-12).

Um caso particular de utilização destas interfaces funcionais prende-se com os sistemas de comunicação, *Communication Interface Function Blocks*, CIFB. Para isso, a norma definiu dois padrões de comunicação designados por *PUBLISH*/*SUBSCRIBE* para a comunicação unidirecional e *CLIENT*/*SERVER* para comunicações bidirecionais. Estes blocos, disponibilizados pelo fornecedor do *software/hardware*, podem ser ajustados para uma rede particular ou para mecanismos de comunicação de uma aplicação específica. Na Figura 6-13 apresentam-se os blocos *PUBLISH* e *SUBSCRIBE* utilizados na transferência de dados unidirecionais através de uma rede de comunicação.

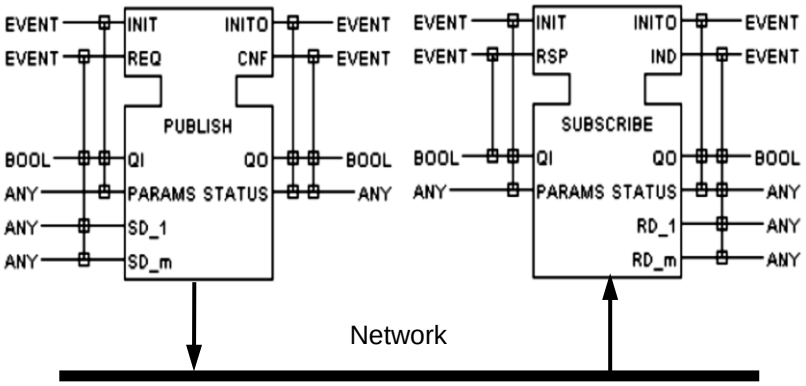


Figura 6-13. Interface de comunicação unidirecional Publish/Subscribe (adaptada de IEC 61499, 2012)

O *Publish* (Editor) disponibiliza os dados a enviar para a rede através de *SD_1, …, SD_m*, provenientes de um ou mais blocos função da aplicação, inicializando ou finalizando o serviço de acordo com o descrito anteriormente. A pedido do evento *REQ*, os dados que precisam ser publicadas são enviadas pelo editor através da rede que, após publicação, notifica o editor do seu envio através do evento de saída *CNF*. Por outro lado, o *Subscribe* (Subscritor) inicializado pela aplicação lê os dados *RD_1, …, RD_m* dando início à transferência dos mesmos. O *Publish* envia os dados acionando o evento IND, na saída do *Subscribe*, para notificar os aplicativos de leitura de que os novos valores dos dados estão disponíveis nas saídas *RD_1 ... RD_m* do *SUBSCRIBE*. O aplicativo de leitura notifica o *SUBSCRIBE* através do evento *RSP* de que os dados foram lidos. Na Figura 6-14 apresenta-se o diagrama de sequência temporal deste processo.

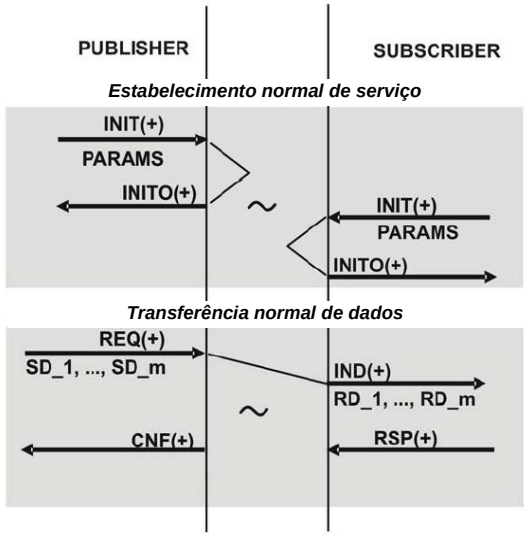


Figura 6-14. Estabelecimento da comunicação e transferência normal de dados (Hanisch e Vyatkin, 2004)

Note-se ainda que no FBDK esses dois padrões de comunicação são implementados usando-se os serviços do protocolo de Internet. Através da utilização destes pacotes de comunicação, o programador pode desenvolver aplicações distribuídas comunicando através de uma rede local (Ethernet) e até mesmo por Internet. Assim, e uma vez que se utilizará o protocolo preconizado pela IEC 61499, apenas será necessário indicar no ID de cada um dos CIFBs o endereço IP e a porta de comunicação de acordo com o formato *address:port*, ou seja, por exemplo 192.168.0.2:61499, onde 192.168.0.2 é o IP e 61499 o número da porta de comunicação que, será automaticamente convertido para fbdk[].ip[192.168.0.2:61499]. Então, e para que se possa comunicar através do par *PUBLISH/SUBSCRIBE* é necessário usar o serviço *multicast* do protocolo UDP (*User Datagram*

Protocol) e, como já foi referido, fornecer aos FBs um endereço IP *multicast* e uma porta. Os FBs *Publish* enviarão dados para a rede pelo endereço/porta IP *multicast* e os FBs *Subscribe* recebem-nos no mesmo endereço IP/porta *multicast*. Assume-se ainda que poderão existir vários *Subscribers* para um único *Publish*. Os endereços IP *multicast* podem ser escolhidos do intervalo 224.0.0.0 a 239.255.255.255, com o intervalo de endereços entre 224.0.0.0 e 224.0.0.255 reservado para o sistema, enquanto a porta será arbitrária. Um exemplo do parâmetro ID do *Publish/Subscribe* será, por exemplo, 225.0.0.1:61499 em que, 225.0.0.1 é o endereço IP do grupo *multicast* e 61499 é a porta de comunicação selecionada.

Para o par *CLIENT/SERVER*, comunicação ponto-a-ponto, i.e., apenas um e só um *Client* pode enviar e receber dados de e para um único *Server*. A ligação TCP/IP deverá ser configurada tendo em conta o *Server* alocado a um dado dispositivo enquanto o *Client*, alocado a outro dispositivo, será ligado ao server através do mesmo IP, porta de escuta do server. O FB *Client* enviará uma solicitação ao FB *Server* usando o endereço IP e a porta do mesmo pelo que, o *server* responderá após o processamento dos dados recebidos. Um exemplo dos parâmetros do ID usado para o *Server* seria “localhost:61501” e o ID do *Client* seria 192.168.0.2:61501. O atributo "*localhost*" é apenas para fins informativos e não será usado pelo FORTE, ou seja, o endereço IP do servidor será sempre o mesmo que o endereço IP do dispositivo. Apenas o atributo de porta é usado. Note-se, no entanto que, ao contrário dos FBs *Publish/Subscribe*, os FBs *Client/Server*, trabalhando em pares, implica que apenas um único *Client* possa ser ligado a um *Server* de cada vez, enquanto um *Client* só poderá ser ligado apenas a um *Server*.

A juntar a estes grupos de serviços mencionados anteriormente, existem ainda outros tipos de serviços que podem ser fornecidos através dos SIFBs como sejam:

- *Graphical User Interface* (GUI) – serviço de Interface Homem-Máquina (HMI) permitindo a supervisão e o controlo do sistema (versão 8.4, por exemplo).
- *Gestão de comunicações* – que asseguram as comunicações entre os diferentes componentes, fazendo uso das capacidades das redes de comunicação. Permitem fornecer comunicações de baixo e alto nível.
- *Gestão de processo* – serviços utilizados na implementação de *drivers* de recursos eletrónicos utilizados nas aplicações (sinais I/O, ficheiros, aplicações externas, entre outros) (Christensen, 2018).

Deve-se notar que por princípio os SIFBs não são implementados pelos utilizadores. Eles são fornecidos com as ferramentas de desenvolvimento de modo a poderem ser utilizados, diretamente, numa qualquer aplicação.

6.6 Especificações do sistema

6.6.1 Recursos e Dispositivos

Um sistema industrial de medição e controlo, vulgarmente designado de IPMCS, é modelado como um conjunto de equipamentos interligados que comunicam uns com os outros por meio de uma rede de comunicação. Este processo pode ser modelado como uma aplicação residente num único dispositivo considerado, segundo a IEC 61499, como um elemento indivisível e atómico (Strasser *et al.*, 2006) ou dividido por vários dispositivos. Este poderá ainda conter uma interface de processo e uma interface de comunicação com zero ou mais recursos como se pode ver na Figura 6-15.

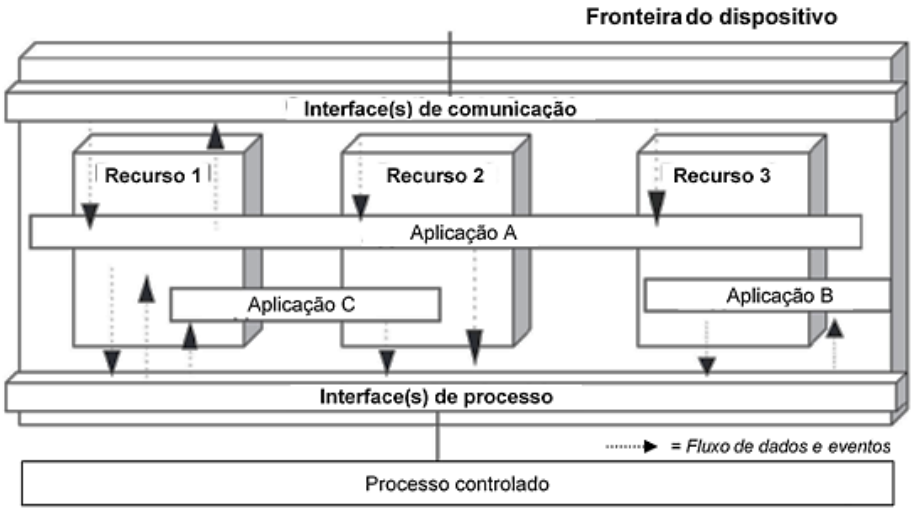


Figura 6-15. Modelo do dispositivo (adaptada de Hanisch e Vyatkin, 2004)

A *interface de processo* fornece um mapeamento entre o processo físico (medições analógicas, I/O, etc.) e os recursos. As informações trocadas entre o processo físico e os recursos são apresentadas como dados, como eventos ou ambos. Por sua vez, a *interface de comunicação* fornece um mapeamento entre os recursos e as informações trocadas através da rede de comunicação. Os serviços prestados por estas interfaces vão desde a comunicação de dados ou eventos ao suporte de serviços de programação, configuração, diagnóstico, etc.

Um recurso pode ser visto como uma unidade funcional disponibilizada por um dispositivo com controlo de funcionamento independente. Apresentando-se como um elemento autónomo pode ser desenvolvido, configurado com diversos parâmetros, iniciar-se, ser apagado, sem que isso afete os demais recursos existentes no dispositivo. O recurso fornece as ferramentas necessárias para a execução das aplicações, Figura 6-16.

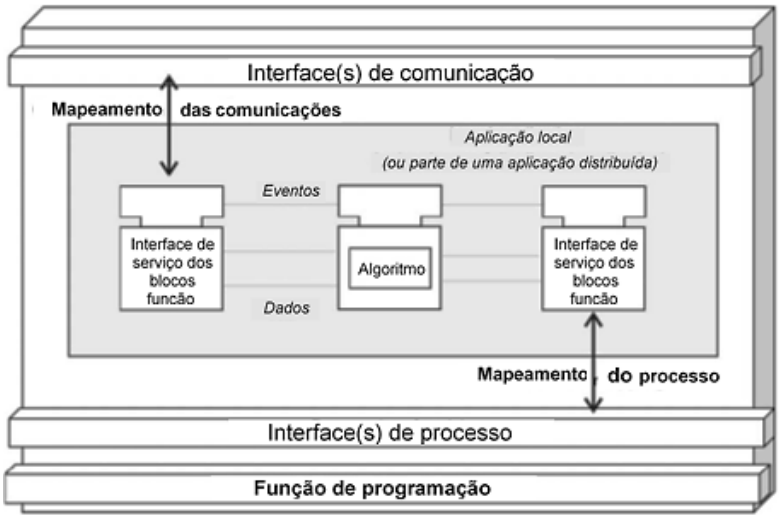


Figura 6-16. Modelo do recurso (adaptada de Hanisch e Vyatkin, 2004)

Os recursos recebem os dados e/ou eventos do processo e/ou das interfaces de comunicação, processando-os e reenviando-os, novamente, para o processo e/ou para as interfaces de comunicação, de acordo com as especificações das aplicações. Para além disso, estes fornecem também os meios físicos para a execução dos algoritmos (armazenamento de dados,

algoritmos de controlo de execução, eventos, etc.) bem como os recursos de *software* para gerir e controlar o comportamento dos blocos função e suas funções de controlo.

6.6.2 Configuração do sistema

Um sistema é considerado um conjunto de um ou mais dispositivos interligados, entre si, comunicando com o processo a controlar através, de sensores e sinais de atuação. O sistema modelado pode ser desenvolvido como uma aplicação residente num único dispositivo, aplicação C (Figura 6-17), ou pode ser distribuída entre os vários dispositivos do sistema, aplicações A e B. A aplicação subdividida pelos vários dispositivos do sistema, poderá executar, nos diferentes componentes, diferentes tarefas (Khalgui *et al.* (2005) considera uma tarefa *T* a consequência da execução de um FB ativado pela ocorrência de um evento de entrada, *EI*). Estas poderão ser: a leitura de dados de entrada num determinado dispositivo, o controlo de processo noutro e a saída de dados alocados a um outro qualquer dispositivo, etc.

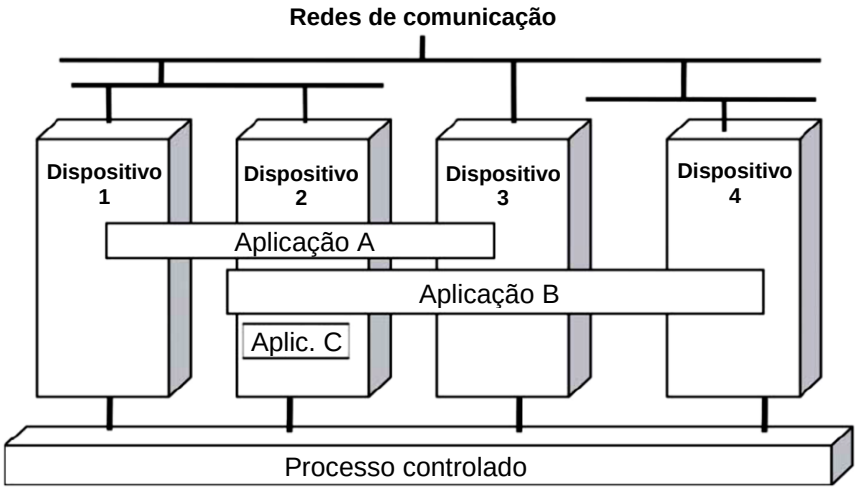


Figura 6-17. Configuração do sistema (adaptada de Hanisch e Vyatkin, 2004)

6.6.3 Aplicação

Segundo a IEC 61499 uma aplicação é uma rede de instâncias cujo os dados e eventos de entrada e de saída encontram-se interligados. Nestas os FBs e as subaplicações são considerados os nós da rede das instâncias e os dados e os eventos, de entrada e de saída, os elementos de interligação, ver Figura 6-18.

Por outro lado, uma aplicação distribuída entre vários recursos do mesmo dispositivo ou de diferentes dispositivos pode ser considerada como um passo intermediário no desenvolvimento de sistemas distribuídos. Neste sentido, e embora a norma defina completamente a funcionalidade do sistema, não especifica a estrutura dos mesmos em termos de distribuição pelos dispositivos computacionais, nem o modo como executar esta distribuição. Assim há que definir um conjunto específico de elementos que possam ser separados e atribuídos por um qualquer dispositivo de acordo, por exemplo, com o esquema que é apresentado na Figura 6-18. O modo de e como fazer a separação do sistema será sempre uma decisão do programador bem como a alocação de quantos e quais os módulos a distribuir pelos diversos dispositivos.

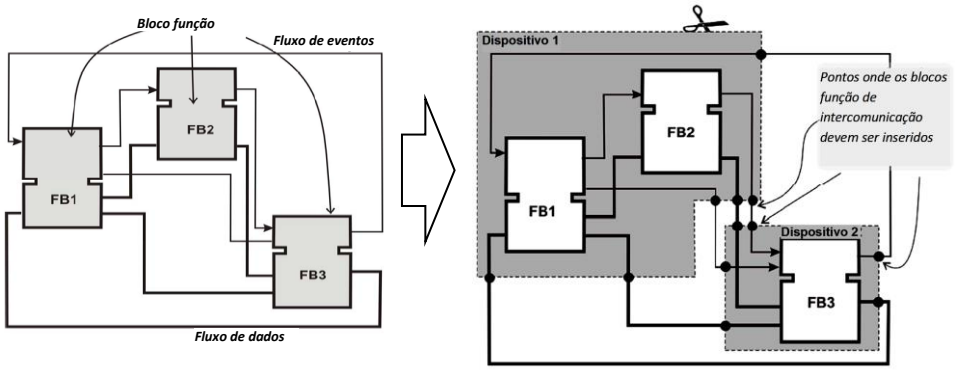


Figura 6-18. Modelo de uma aplicação/Aplicação distribuída (adaptada de Hanisch e Vyatkin, 2004)

No entanto, não basta só definir a estrutura computacional dos dispositivos, distribuindo os elementos da aplicação há que definir explicitamente o processo de comunicação entre estes pelo que deverá adicionar-se, nos locais onde a separação ocorreu (ver Figura 6-18), blocos funcionais de comunicação (*PUBLISH/SUBSCRIBE* para comunicações unidirecionais, ou *CLIENT/SERVER*, para comunicações bidirecionais). A configuração final do sistema, com os respetivos módulos de comunicação inseridos (*Publish* e *Subscribe*), pode ser visualizada na Figura 6-19.

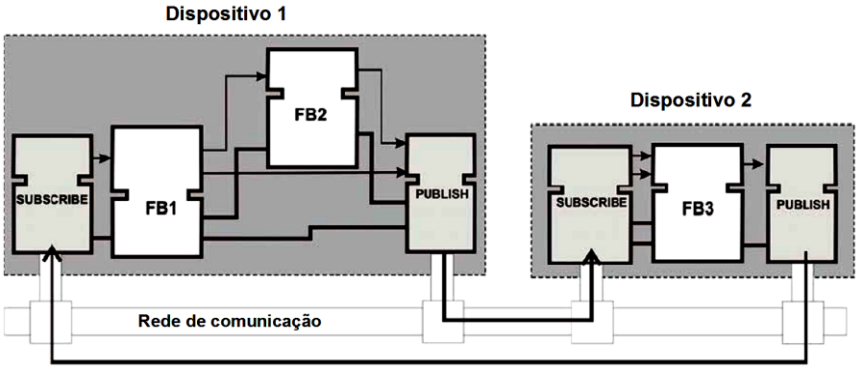


Figura 6-19. Aplicação distribuída ligada por blocos função de comunicação (adaptada de Hanisch, 2004)

De notar que uma aplicação só poderá ser considerada um produto final quando todos os FBs instanciados estiverem mapeados para os dispositivos e recursos utilizados, tomando em atenção que há que utilizar FBs específicos para estabelecer as comunicações entre os diferentes dispositivos ou recursos. Para isso, e como já foi referido há que utilizar os SIFBs de acordo com as especificações pretendidas para o sistema. Na Figura 6-20 apresenta-se um exemplo concreto, não distribuído, de um sistema de controlo de um cilindro de duplo efeito (Santos e Sousa, 2010).

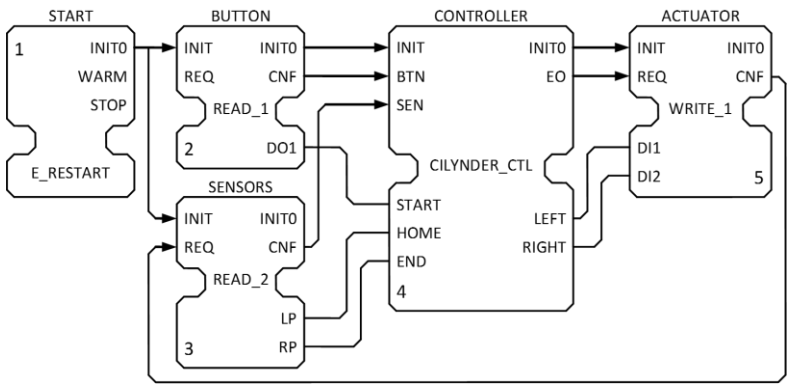


Figura 6-20. Sistema de controlo de um cilindro de duplo efeito (Santos e Sousa, 2010)

Capítulo 7

Infraestrutura de Replicação IEC 61499

7.1 Introdução

Os sistemas de controlo por computador são usados numa vasta e diversificada gama de aplicações. Estes podem ser encontrados em áreas como a automação industrial, o controlo de processos, na robótica, nos sistemas móveis, na aviação e em aplicações especiais. Em todas estas aplicações, os computadores são utilizados para controlar o ambiente circundante, reagindo a estímulos externos de acordo com as exigências do ambiente a controlar. Assim, é esperado que estes respondam corretamente às solicitações, tanto em valor como no domínio temporal, de tal modo que as correções do sistema dependam não apenas do resultado lógico da computação, mas também do momento em que os resultados são produzidos.

Por outro lado, os sistemas computorizados tornaram-se cada vez mais fiáveis dado que são desenvolvidos para executarem corretamente as suas tarefas mesmo na presença de componentes que apresentem falhas. Estes sistemas possuem mecanismos de tolerância a falhas que lhes permite prestar o serviço, de acordo com as especificações, apesar da existência de falhas (Laprie *et al.*, 1995). Assim, para que estes requisitos possam ser cumpridos, há a necessidade de garantir a redundância da aplicação, redundância de *software* ou de *hardware*, num contexto distribuído. Consequentemente que a redundância de *hardware* e/ou de *software* exige mecanismos de gestão das réplicas, apoiados por protocolos de comunicação adequados, que garantam a consistência da informação, bem como a consolidação dos valores produzidos pelos componentes, pelos recursos ou pelo *software* replicado. No entanto, a utilização destes mecanismos acarreta, geralmente, no aumento da complexidade de desenvolvimento da aplicação.

Nos sistemas distribuídos tolerantes a falhas utiliza-se a replicação de dados e de serviços para manter a disponibilidade e, sob certas condições, o seu desempenho. Quando um nó que possui um componente crítico do sistema falha, uma réplica desse mesmo componente, localizado num outro nó, assume de forma transparente a execução dos requisitos do elemento em falha. Por outro lado, com a implementação de réplicas, o problema da manutenção dos componentes replicados obriga à utilização de metodologias de controlo que garantam a integridade de todos os elementos replicados. Estas metodologias consistem em protocolos que têm por base a tolerância ativa (Schneider, 1990), a tolerância passiva ou *primary-backup* (Budhiraja, 1993) e a semi-ativa (Powell, 1991).

Associado aos problemas inerentes ao controlo das réplicas surgem os problemas de comunicação entre os elementos replicados e alocados aos diferentes nós do sistema. De facto, o

mecanismo de comunicação a implementar deverá garantir que todos os elementos que constituem o objeto replicado, grupo de elementos que constituem cada uma das réplicas (subaplicação), processem o mesmo conjunto de dados. Sendo assim, um qualquer processo poderá enviar informação para todos os elementos replicados, de forma transparente, sem ter de saber quantas são as réplicas e onde estão localizadas.

7.2 O problema

As aplicações de controlo industrial são, normalmente, baseadas em Controladores Lógicos Programáveis (PLCs) utilizando linguagens de programação padronizadas e definidas segundo a norma IEC 61131-3 (IEC 61131, 2013). A ligação de vários PLCs através de redes de comunicação conduziu a que estas aplicações se transformassem em sistemas de controlo distribuído onde, na maioria delas, é necessário complementá-las com sincronização em tempo real e atributos de tolerância a falhas, de modo a proporcionar a necessária confiança no funcionamento. No entanto, a semântica preconizada na norma IEC 61131, para as suas diversas linguagens de programação, não se encontra adequada para os sistemas distribuídos e aplicações de controlo de automação flexível. Em consequência a nova arquitetura IEC 61499 (IEC 61499, 2015) foi concebida integrando várias soluções que poderão dar respostas aos problemas de automação distribuída. Assim sendo, a norma IEC 61499 propõe uma linguagem de programação ao nível dos sistemas de controlo distribuído que faz a ponte entre as linguagens de programação dos PLCs e a dos sistemas distribuídos (Vyatkin, 2009).

Associado à natureza distribuída das aplicações de controlo surgem novos desafios, entre eles, a multiplicidade de linguagens de programação utilizadas na programação de *hardware* distribuído (geralmente PLCs interligados através de redes com sensores inteligentes, atuadores, inversores de frequência e dispositivos de interface homem-máquina – HMI) incluindo a lógica associada a cada um dos nós de controlo e suas interações que são, ou poderão ser, também um problema. Por outro lado, de modo a minimizar estes desafios, há ainda os novos recursos do *hardware*, como a fiabilidade, que permitem a introdução de técnicas de tolerância a falhas na arquitetura da aplicação através da replicação dos dispositivos (os dispositivos devem ser entendidos como entidades de *hardware* e *software*). Esta abordagem garante que se uma qualquer réplica falhar, e se as restantes réplicas continuarem em funcionamento, estas serão capazes de mascarar a existência da réplica em falha perante a restante aplicação. A replicação a nível de *hardware* é alcançada através da replicação do dispositivo juntamente com o *software* a ser executado, com custo elevado, enquanto a replicação ao nível do *software*, menos dispendiosa, permite uma menor granularidade da replicação. Com esta abordagem é possível replicar somente os elementos de *software* críticos, executando-se uma única cópia dos elementos não críticos.

Várias foram as arquiteturas propostas para a tolerância a falhas todas elas explorando um dos três métodos para a introdução da diversidade, ou seja, conceber/implementar diversidade, diversidade de dados e diversidade temporal (Pullum, 2001). Para além destas, ainda poderão ser tomadas em conta duas das principais abordagens associadas à recuperação ou à tolerância de falhas identificadas – recuperação para a frente e recuperação para trás. A recuperação para a frente (reexecução do elemento de *software* que falhou, com a intenção de recuperar a falha intermitente) e recuperação para trás (uso de pontos de verificação de bons estados conhecidos a partir dos quais a recuperação é tentada). Ambas as diversidades temporais apresentam desvantagens quando utilizadas em contexto de aplicações de tempo real devido, essencialmente, aos atrasos adicionais introduzidos quando se recupera de uma falha. No entanto, o *software* tolerante a falhas poderá utilizar qualquer uma destas metodologias, por exemplo a utilização de blocos de recuperação (Horning *et al*., 1974), em aplicações sensíveis ao tempo e no caso em que o tempo necessário para a recuperação das falhas assumirá um papel preponderante.

Não obstante ao que foi referido anteriormente, as estruturas baseadas na recuperação para a frente, como seja a programação de N-Versões (Elmendorf, 1972; Avizienis e Chen, 1977) com suas variações (Laprie *et al.,* 1990), programação N-Copias (Ammann e Knight, 1988), bem como as variações sobre a abordagem de blocos de recuperação (kim e Welch, 1989; Nguyen e Liu, 1998), são as preferidas para as aplicações de tempo real. Assim, quando se utiliza a recuperação para a frente, todas as réplicas devem ser mantidas em sincronismo, para que estas possam processar o mesmo conjunto de dados e de eventos de entrada e na mesma ordem (Geurraoui e Schiper, 1997) ou seja, de forma determinística, consolidando-se, de alguma forma, todas as saídas das réplicas. Por outro lado, a juntar a todas estas considerações, deve-se ter em conta que o standard IEC 61499 não explicita o modo como tratar a replicação de componentes, muito menos outras questões relacionadas com a sincronização de réplicas e o determinismo, deteção e contenção da falha e as restrições de tempo real que podem ser impostas à aplicação.

Assim sendo, a resolução destes problemas passará pela utilização de algoritmos determinísticos que garantam que a informação disponibilizada em cada nó segue uma ordenação bem definida e que garanta a mesma perceção de evolução do sistema a todos os dispositivos ou componentes. Por outro lado, dependendo do modelo de replicação adotado e do protocolo de difusão, poderá ou não ser necessário que as mensagens difundidas se encontrem ordenadas a fim de garantir o determinismo das réplicas. O determinismo do sistema poderá então ser obtido pelo uso de mensagens temporizadas, por ordens prioritárias, por *token* numerado, etc., e como tal garantir que a informação recebida e processada, por todo o código lógico, responderá exatamente da mesma forma para o mesmo conjunto de entradas.

Neste sentido, algumas das questões que poderão ser colocadas prendem-se com o modo como a IEC 61499 lida, ou poderá lidar, com os sistemas distribuídos tolerantes a falhas, ou seja: Como poderá interagir perante os possíveis cenários de transferência de eventos e dados? Como se poderá garantir a sua integridade? Serão os FBs standards, nomeadamente, os SIFB de comunicação, capazes de assegurarem, por si só, o determinismo das réplicas? Qual será o procedimento a adotar para o garantir?

Assim sendo, a título de uma primeira resposta a estas conjeturas, podemos assumir que as mensagens temporizadas, sincronizadas e devidamente ordenadas assumirão um papel relevante, na garantia do determinismo. Naturalmente que as outras questões relacionadas com o modo como proceder quanto à interligação dos dispositivos, dos recursos ou dos equipamentos (note que se considerará equipamento como o conjunto de componentes eletromecânicos que constituem o transportador, que passaremos a designar de *conveyor*), do uso de FBs standards ou dos procedimentos a adotar por exemplo, não poderão, ainda, ser respondidas, no entanto pode-se apontar caminhos que possam levar-nos a um entendimento das mesmas. Neste sentido, consideremos um cenário de transferência de objetos constituído por um conjunto de *conveyors*, como o representado na Figura 7-1, que se encontrará sujeito às mais variadas configurações (*layouts*) e interações. Naturalmente que nesta implementação e no que se refere ao processo de controlo, poderemos definir outros modos de funcionamento como sejam a prioridade de receção (em função dos *conveyors* ativos) em detrimento do tempo de ativação ou do tempo de receção, por exemplo. No entanto, num sistema replicado de tempo real, o determinismo das réplicas será o fator preponderante na garantia de processamento dos mesmos valores e na mesma ordem e, como tal, condicionante da ação a realizar pelos componentes críticos, sujeitos a replicação.

Na Figura 7-1 apresenta-se, um possível troço, de um sistema de transporte, inspeção e de verificação de um sistema de manuseamento de malas de um aeroporto (BHS), constituído por cinco equipamentos de transporte lineares e um rotativo. O *conveyor* C3 recebe malas de C1 e de C2 pelo que a sua disponibilidade deverá ser a mais elevada possível, tal e qual o equipamento R1 que receberá malas de C3 e de C4. Então, do ponto de vista da disponibilidade, é-lhes atribuído um papel fundamental no processo e como tal, torna-os num dos elementos críticos do sistema.

Nesta situação, os *conveyors* C3 e R1, ao nível do controlo, deverão ser replicados para um ou mais dispositivos garantindo-se assim a redundância do sistema.

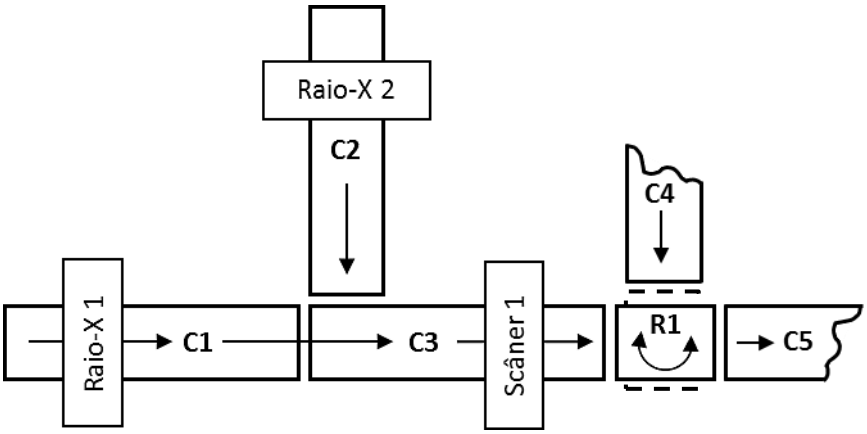


Figura 7-1. Convergência de conveyors, eleição linear e rotativa

É perante estes cenários que o nosso problema se coloca. As questões que advirão incidem no modo como a IEC 61499 lida com os presentes cenários e como serão realizadas as transferências entre *conveyors* que poderão estar sujeitos a replicação.

7.3 Modelo de replicação 61499

Como já foi referido anteriormente, a maioria das técnicas de tolerância a falhas é realizada com a replicação de um ou vários componentes críticos, pelo que as réplicas devem ser tão independentes quanto possível de forma a minimizar uma falha comum, que possa causar a falha de todas as réplicas. Isto pode ser conseguido pela execução das réplicas em *hardware* distinto, pela diversidade de criação ou por diferentes implementações do mesmo algoritmo, ou seja, pela replicação de *software*. Assim sendo, e independentemente da abordagem, o objetivo principal é garantir que, se uma das réplicas falhar, as réplicas restantes continuarão a funcionar e, portanto, mascararão a existência da réplica em falha perante a remanescente aplicação, tornando-a o mais transparente possível.

Embora a transparência seja um dos objetivos principais da *framework* a desenvolver, deve considerar-se que uma utilização completamente transparente dos mecanismos de replicação/distribuição introduzirão aumentos do tempo de desenvolvimento e custos desnecessários. Por outro lado, deve-se ter em conta que durante a fase de desenvolvimento da aplicação, o uso da replicação e da distribuição não é considerado uma abordagem transparente uma vez que, posteriormente, o programador, na fase de configuração do sistema realizará a replicação e a alocação dos componentes no sistema distribuído.

Uma aplicação IEC 61499 é vista como sendo uma composição de vários dispositivos, subaplicações ou elementos unitários de processamento interligados, que a cada invocação dos eventos, processam os dados, gerando novos eventos e/ou dados. Assim sendo, a replicação será feita ao nível da subaplicação, do FB ou do dispositivo (ao nível dos componentes) pelo que este será considerado atômico e indivisível para fins de replicação e, portanto, as aplicações não precisam de ser completamente replicadas. Assim, dependendo do grau de fiabilidade exigida, somente os componentes mais críticos serão escolhidos para a replicação e como tal deve-se considerar o componente como uma entidade completamente ortogonal e independente das unidades da aplicação já existentes (BFB, CFB, Subaplicações, etc.) e definidas na IEC 61499. Por outro lado, a partilha de dados e eventos entre diferentes réplicas, residentes nos diferentes nós,

deverão ser sincronizados, para que se possa garantir que todas as réplicas recebem o mesmo conjunto de dados e na mesma ordem.

Neste sentido, cada tarefa (ação resultante da invocação de um FB, enquanto elemento de *software*) será realizada pela invocação de um evento podendo, no entanto, ser realizada inúmeras vezes. Assim, cada uma destas tarefas poderá ser realizada periodicamente pelo *runtime* (invocação temporal), esporadicamente, em resultado da invocação de uma outra tarefa, ou pelo ambiente. Como resultado desta invocação cada tarefa será propagada para uma outra, partilhando os seus dados e eventos pelo que, deve ser assegurado que, quando replicadas, devem ter a mesma perceção dos dados. Esta perceção será obtida usando-se uma difusão de dados do tipo *multicast* que garanta que todas as réplicas recebem o mesmo conjunto de dados na mesma ordem.

7.3.1 Unidade de replicação

No que diz respeito à tolerância a falhas, usando a replicação de componentes, deve-se definir o que se entende por unidade de replicação. Assim, define-se componente replicado (unidade de replicação), como um elemento de *software*, considerando como elemento mais simples de replicação o FB e o mais complexo a subaplicação, que faz parte integrante da aplicação e que será replicado de acordo com a sua criticidade. Neste sentido, a maior ou menor complexidade do sistema replicado depende da replicação parcial ou total do mesmo, pelo que se verificará um aumento das interações entre os diferentes componentes e, consequentemente, das ações de consolidação das diferentes saídas. Como tal, a aplicação será dividida por vários componentes, cada um com um conjunto de tarefas, que interagem como um todo. Cada um destes componentes poderá ser replicado para mais do que um nó de processamento, ou replicado num só nó. Em cada nó poderão coexistir vários componentes e, como tal, várias tarefas e subaplicações. Cada um destes componentes será considerado como uma abstração do sistema distribuído pelo que estes serão usados para estruturar a replicação e o sistema.

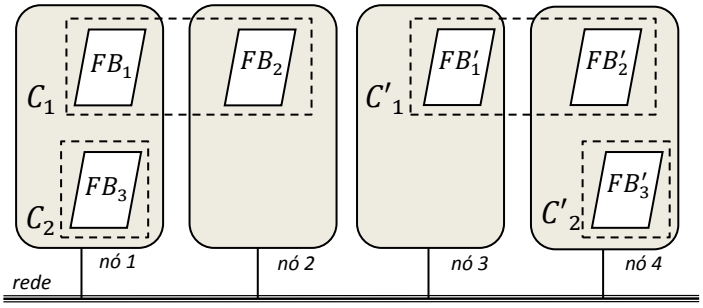


Figura 7-2. Replicação ativa

Na Figura 7-2 apresenta-se, a título de exemplo, uma aplicação com três FBs (FB_1, FB_2 e FB_3) divididos por dois componentes (C_1 e C_2) que por sua vez são replicados (C'_1 e C'_2) em outros nós. O componente C_1 engloba o FB_1, alocada ao nó 1, e o FB_2, alocada ao nó 2. As suas réplicas, blocos função FB'_1 e FB'_2 encontram-se alocadas aos nós 3 e 4, respetivamente. O componente C_2, que engloba um único FB (FB_3), encontra-se alocado ao nó 1, enquanto a sua réplica foi alocada, somente e na totalidade, ao nó 4. Note-se ainda que, o grau de replicação dos componentes presentes no exemplo da Figura 7-2 é de 2, e que para se tolerar o comportamento erróneo das réplicas, seria necessário usar $2 * f + 1$ réplicas para se tolerar f falhas. Assim, para que se possa tolerar uma única falha num processo P, seriam necessárias 3 réplicas do processo, ou seja, processo inicial P, e as suas réplicas P_1 e P_2.

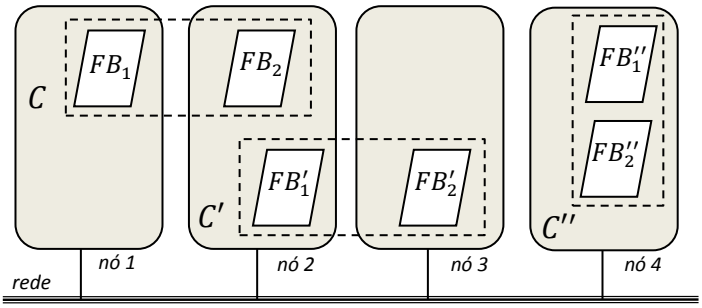


Figura 7-3. Replicação de uma subaplicação

Por seu lado as subaplicações, para além de poderem ser distribuídas por vários nós, podem também ser replicadas num ou em mais nós. Na Figura 7-3 apresenta-se a replicação de uma subaplicação C (componente constituído por dois FBs, duas tarefas) que se encontram distribuídas pelos nós 2 e 3 ou, em alternativa, replicada na totalidade num único nó (nó 4).

Neste sentido deve considerar-se que o componente não imporá restrições, quanto ao modo como a aplicação deve ser estruturada. No entanto devemos considerar que os FBs alocados ao mesmo componente possuem algum modo de relacionamento pelo que, caberá ao programador do sistema decidir qual a estrutura a adotar para a mesma. A decisão poderá ser tomada tendo em conta não só os padrões de interações entre os FBs, mas também o relacionamento entre os mesmos. Em suma, cabe ao programador a decisão de alocação dos componentes.

7.3.2 Cenários de interação entre componentes

Antes de se definir quais as interações possíveis entre componentes, deve-se relembrar que, o componente é considerado a unidade de replicação e como tal o componente será também considerado como um elemento portador de falhas, pelo que, a existência de falhas ao nível dos FBs conduzirá a falhas nos componentes. No entanto, se um componente falhar, produção de valores incorretos ou não produção de qualquer valor, o conjunto de componentes replicados não falhará, uma vez que a consolidação das saídas mascarará o componente em falha. Isto é, o componente não apresentará falhas, uma vez que a interação entre os FBs internos de cada uma das réplicas não necessita de ser consolidada, a não ser que os seus resultados sejam disponibilizados para um outro qualquer componente ou para o sistema.

É importante realçar que os mecanismos de replicação são mecanismos essenciais para o correto funcionamento do sistema e, por consequência, deverão evitar/prevenir todo e qualquer erro de *software*. Uma vez que a ocorrência deste poderá provocar uma falha que, a ocorrer num único nó ou componente, poderá ser, facilmente, mascarada pela sua réplica. Como tal, e se for necessário um elevado grau de fiabilidade do sistema, esta poderá ser conseguida recorrendo as metodologias de diversidade (Pullum, 2001).

Assim sendo, dado que o objetivo principal desta tese é a realização da replicação em sistemas distribuídos, tendo como base de replicação a IEC 61499, apenas dois mecanismos de interação, entre FBs, podem ser utilizados, isto é, a passagem de eventos bem como a passagem de dados. Assim, de acordo com a replicação ativa, todo os FBs do sistema, ou pelo menos parte dos FBs, deverão ser replicados. No entanto, deve-se ter em conta que o aumento do número de réplicas irá diminuir a eficiência do sistema, devido ao aumento de FBs replicados e das suas interações, aumento do número de mensagens trocadas. Esta relação, grau de replicação e eficiência, encontra-se intimamente relacionada dado que, o aumento do grau de replicação implica na diminuição da eficiência e vice-versa. Sendo assim, aquando do desenvolvimento de um sistema tolerante a falhas e de tempo real, a eficiência não deve ser considerada como a meta final, contudo, esta pode ser melhorada se diminuirmos o grau de redundância.

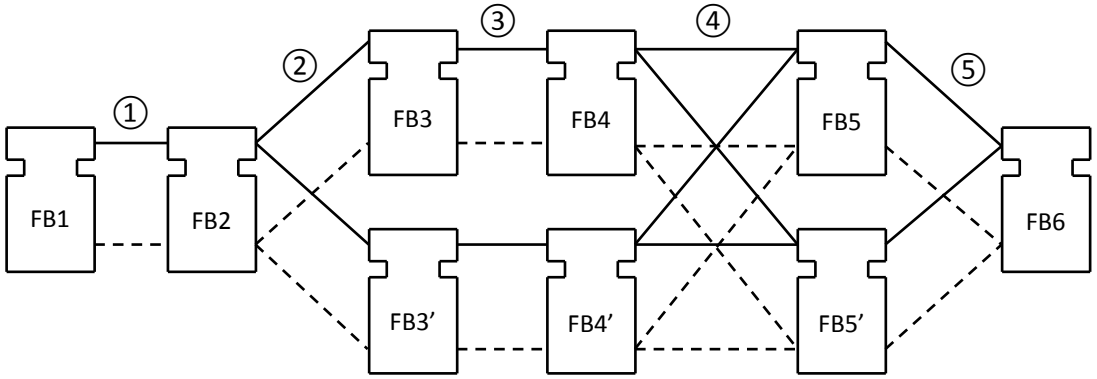


Figura 7-4. Cenários de interação entre FBs replicados e não replicados (Santos e Sousa, 2008)

Na Figura 7-4 são apresentados diversos modos de interação evidenciando os dois únicos mecanismos de interação. Assim, na passagem de eventos e dados entre elementos replicados e não replicados, poderão ser consideradas diversas possibilidades de configuração da aplicação distribuída. No entanto, para a *framework* IEC 61499, as interações entre os diversos elementos poderão, de forma genérica, apresentar-se segundo os vários cenários que a seguir se enunciam (Sousa e Santos, 2007):

1. Passagem de eventos e dados entre FBs não replicados, standard preconizado pela norma (passagem entre FB1 e FB2).
2. Passagem de eventos e dados entre um FB não replicado e vários FBs replicados (passagem de FB2 para FB3 e FB3').
3. Passagem de eventos e dados entre FBs replicados contidos no mesmo componente. Note-se que nesta situação o componente replicado seria considerado como um outro FB contendo os dois FBs internos, interagindo, através de uma rede interna (passagem entre FB3 e FB4 e entre FB3' e FB4', igual à passagem preconizada na passagem 1, interação standard).
4. Passagem de eventos e dados entre FBs replicados, dentro de componentes replicados, para outros FBs replicados em outros componentes replicados (passagem de FB4 para FB5 e FB5' bem como de FB4' para FB5 e FB5').
5. Passagem de eventos e dados de um componente replicado para outro não replicado (passagem entre FB5 e FB5' para FB6, cenário inverso ao da passagem 2).

A implementação de todos estes possíveis cenários de interação, entre os diversos elementos replicados e não replicados, conduz à realização de uma análise mais aprofundada destas interações. Neste sentido não será necessário tecer considerações sobre o **cenário 1** uma vez que a interação entre os FBs 1 e 2 é o processo normal de interação proposto na IEC 61499.

No **cenário 2** (*difusão multicast*), um único valor de dados fica disponível num FB não replicado para várias cópias de um segundo FB replicado. Para isso será necessário utilizar um protocolo de difusão *multicast* que garanta que todas as réplicas recebem os dados na mesma ordem bem como no tempo em que foram disponibilizados.

No **cenário 3** (*determinismo*), todas as réplicas devem permanecer consistentes e sincronizadas com o objetivo de se poder comparar as saídas. No entanto, uma vez que na replicação de *software* pode ser atribuído a cada nó um conjunto distinto de componentes, a aplicação de um qualquer algoritmo de escalonamento em cada um dos nós poderá resultar numa sequência de execução diferente que, por sua vez, poderá levar a que duas réplicas processem as mesmas entradas em diferentes ordens e, por conseguinte, produzir saídas distintas se estas

entradas não forem comutativas (Poledna *et al.*, 2000). Assim, considere-se, a título de exemplo, um FB a replicar (FB1) que ao receber um evento *e1*, executa um algoritmo que produz uma saída de dados *d1* usada por FB2. Duas réplicas de FB1 e FB2 são executadas em dois nós de processamento distintos (FB1a e FB2a sobre *pa* e FB1b FB2b sobre *pb*). Cada réplica de FB1 enviará o valor *d1* para a respetiva réplica FB2 no mesmo nó de processamento (FB1a em *pa* enviará para FB2a em *pa*, enquanto FB1b em *pb* enviará *d1* para FB2b em *pb*), Figura 7-5 (Sousa e Santos, 2007).

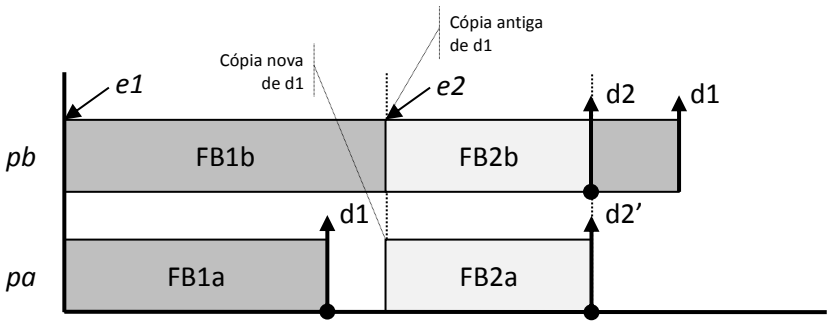


Figura 7-5. Réplicas inconsistente, adaptado para IEC 61499 de Poledna et al. (2000)

Considere-se agora que, devido a um desfasamento das velocidades de processamento dos recursos em relação aos dois nós de processamento, *pa* irá terminar a execução da réplica FB1a um pouco mais cedo do que a réplica FB1b em *pb*. Durante este lapso de tempo FB1a, em *pa*, terminou a sua execução (e, portanto, disponibiliza *d1* para FB2a) enquanto, em *pb*, a réplica FB1b ainda se encontra em execução, o evento *e2* é despoletado o que provocará o início da execução de FB2a. Se FB2 tem prioridade de execução superior a FB1, FB2b vai antecipar FB1b em *pb*. Em *pa*, também irá executar-se FB2a, assim que FB1a terminar a execução, não ocorre preempção do processo. Em *pa*, FB2a terá um novo valor para *d1*, enquanto em *pb*, FB2b será executado com uma cópia antiga da mesma variável. Nesta situação, as duas réplicas executar-se-ão com diferentes valores de entrada tornando-se inconsistentes, isto é, as variáveis internas vão ficar atualizadas com valores distintos em cada umas das réplicas disponibilizando-se valores distintos para *d2*.

Esta dificuldade em manter o determinismo em sistemas distribuídos pode ser superada em primeiro lugar através do uso de mecanismos temporais que garantam o determinismo (Powell *et al*., 1991; Schneider, 1990). Em segundo lugar pela garantia de que todas as réplicas processem as mesmas decisões de acordo com uma ronda, pré-programada, de consenso (sobrecarga excessiva do sistema até 80% (Schneider, 1990)) e, em terceiro lugar, pela utilização de *hardware* dedicado. Poderá ainda ser conseguido o determinismo pela utilização de mensagens temporizadas (Poledna *et al.*, 2000) ordenadas segundo uma execução virtual sem qualquer comunicação entre processadores e com o sincronismo de todos os relógios dos nós do processamento distribuído.

Neste cenário, os dados que passam entre os FBs dentro de cada réplica ou componente replicado, poderão ser decompostos em três alternativas: i) cada FB dentro do componente replicado é completamente idem-potente, isto é, produz os mesmos resultados, independentemente da ordem pela qual recebe os pedidos; ii) cada FB dentro do componente replicado só recebe eventos de uma única fonte (e pode, portanto, ser considerada como uma única tarefa) ou, em alternativa, enquadrar-se na condição i); e iii) o FB não se enquadra em nenhuma das opções atrás referidas. Assim, na alternativa i), as réplicas estarão sempre sincronizadas, não é necessário implementar protocolos de controlo especiais, enquanto na alternativa ii), a abordagem às mensagens temporizadas será uma alternativa válida e eficiente. Na alternativa iii), será necessário realizar a consolidação dos dados entre réplicas a cada transferência de dados, mesmo entre FB dentro da mesma réplica.

No **cenário 4** (ver Figura 7-4), tal como acontece no cenário 2, deverá utilizar-se uma difusão do tipo *multicast* na distribuição dos dados para cada uma das réplicas recetoras. Por outo lado, ao contrário do cenário 2, a difusão *multicast* terá que ser executada em cada uma das réplicas pelo FBs responsáveis pelo envio. Além disso, todos os dados enviados pelas réplicas originais têm que ser consolidados num único valor. Isto pode ser conseguido por cada FB que recebe os dados, desde que todos eles sigam o mesmo algoritmo de consolidação (algoritmo de votação).

O **cenário 5** é semelhante ao cenário anterior (cenário 4) onde todas as saídas de valores têm de ser consolidadas no final do envio. Neste cenário apesar de existir apenas uma única cópia do FB recetor, poderá ser utilizada, de igual modo, uma difusão do tipo *multicast*.

Tecer-se-á agora algumas considerações sobre os mecanismos de transferência de eventos entre FBs. Considere-se então o **cenário 2**, dado que o cenário 1 é considerado o cenário normal de interação da IEC 61499 (passagem de eventos entre FBs não replicados), em que um FB não replicado envia eventos para um FB replicado. Nesta situação, e à semelhança da passagem de dados no mesmo cenário (2), deve-se utilizar um protocolo de difusão *multicast* que garanta que todas as réplicas recebem os eventos e concordam com o momento em que o evento foi enviado. Esta anuência é fundamental para satisfazer os pressupostos em que se baseiam as mensagens temporizadas.

No caso do **cenário 3** não se verifica a necessidade de utilização de protocolos especiais devido à relação causal entre o envio e receção de eventos dos FBs. Por outro lado, não há a necessidade de estabelecer acordos entre as réplicas no momento em que o evento foi enviado, uma vez que as mensagens temporizadas mascararão as diferenças temporais. O **cenário 4** é semelhante ao cenário 2, exceto na consolidação dos eventos disponibilizados, pois todas as réplicas que recebem eventos devem consolidá-los em função do tempo em que o evento foi disponibilizado de modo a cumprir-se os pressupostos, mais uma vez, do protocolo de mensagens temporizadas. A utilização da difusão *multicast* na transmissão de eventos por parte de cada uma das réplicas emissoras obriga a que todas as réplicas recetoras recebam, exatamente, os mesmos dados e, por conseguinte, votar com o mesmo algoritmo e produzir os mesmos resultados para o tempo de consolidação. O algoritmo de votação não poderá ter qualquer elemento de aleatoriedade e deverá ser considerado de elevada confiabilidade.

O **cenário 5** é similar ao cenário 4, exceto no facto de um único FB recetor decidir sobre a consolidação do valor de saída. Neste caso, não é necessário decidir sobre o tempo em que o dado foi disponibilizado para o FB recetor, uma vez que não será necessário usar mensagens temporizadas, mas sim sobre a validação dos valores recebidos.

Nestes cenários de interação o uso de mensagens temporizadas (Poledna *et al.*, 2000) tornar-se-ão indispensáveis quando se partilham dados com componentes replicados. Nestas situações o tempo de validade das mensagens é primordial quando uma tarefa lê um valor uma vez que, ela deve ler o valor mais recente que possua o tempo de validade mais antigo, que o tempo de execução da mesma. Assim, cada vez que uma tarefa é executada adicionar-se-lhe-á, na origem, o tempo de disponibilidade da mesma. Neste sentido, quando tarefas pertencentes a diferentes componentes interagem, haverá a necessidade de consolidar os dados ou os eventos a disseminar entre componentes replicados. Neste sentido, o conjunto de componentes replicados deverá ser definido como um grupo pelo que, estas interações deverão ser designadas de comunicação intergrupo, em vez de intercomponente. As interações entre estes grupos serão designadas de *um-para-muitos* (1:N), *muitos-para-muitos* (N:N) e *muitos-para-um* (N:1) conforme o esquema apresentado na Figura 7-6. Estas mesmas interações poderão ser consideradas nas relações entre componentes, não replicados, uma vez que a interação *um-para-um* (1:1) será o standard da IEC 61499. Nesta situação *Gn* e *Gm* assumir-se-ão como FBn e FBm (Santos e Sousa, 2010).

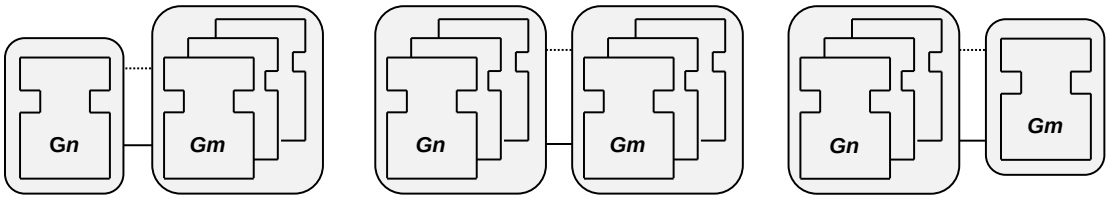


Figura 7-6. Intergrupos ou intercomponentes/tarefas: 1:N, N:N e N:1 (Santos e Sousa, 2010)

7.3.3 Estrutura da framework

A estrutura de replicação a adotar será implementada tendo como base distintas camadas de suporte essenciais à replicação, ou seja, a *Gestão de comunicações* e a *Gestão de réplicas*. A primeira camada, *Suporte do software*, será responsável pela implementação dos protocolos de comunicação com recurso a blocos função de comunicação (SIFB) e sua interligação com os sistemas de replicação e de armazenamento dos dados. Este componente de gestão fornecerá as interfaces de comunicação, SIFBs *standards* já disponibilizados pela norma, necessárias para assegurar a replicação e a distribuição do sistema, ou seja, os mecanismos para a implementação do protocolo de comunicação *multicast*, os algoritmos de replicação e de consolidação, Figura 7-7. A segunda camada, *Gestão de réplicas*, disponibiliza os objetos necessários para a ação de desenvolvimento do sistema distribuído e replicado (elementos disponibilizados pelo *Repositório de objetos*, nomeadamente, os SIFB de comunicação) e os meios para a compilação e a execução do *runtime* da aplicação (exportação e compilação do FORTE).

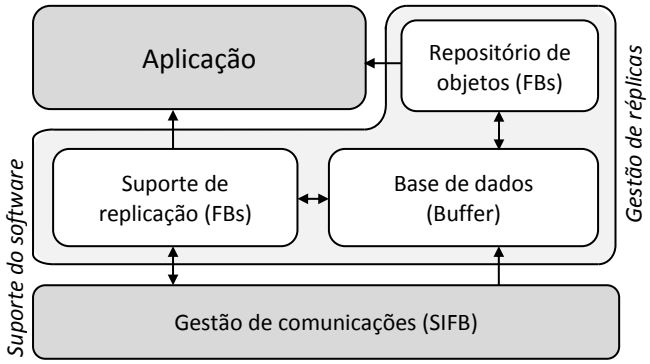


Figura 7-7. Estrutura da framework (Santos e Sousa, 2010)

O *Repositório de objetos* será usado durante o desenvolvimento da aplicação e na fase de configuração, disponibilizando um conjunto genérico de objetos com diversas capacidades (inerentes à *framework 4diac*) bem como os objetos desenvolvidos pelo programador. Estes objetos (BFB, CFB, SIFB, etc.), instanciados com dados e eventos apropriados, são incorporados na aplicação permitindo o desenvolvimento da mesma, mantendo-se o foco nos requisitos de controlo do sistema, deixando para mais tarde a implementação dos mecanismos de distribuição, de replicação e de tempo real, quando necessários.

Este repositório permite a edição, reutilização e adição de novos objetos, desenvolvidos pelo programador, bem como a sua disponibilidade para a incorporação em novas aplicações fornecendo a transparência e os FBs necessários ao desenvolvimento das aplicações. O *Suporte de replicação* será também responsável pela consistência dos componentes replicados bem como, pela interação entre os objetos e a aplicação (tempos de validade das tarefas, alteração dos estados da aplicação, etc.).

7.4 Sincronização de réplicas com recurso a SIFB *Client/Server*

A IEC 61499 define uma série de funcionalidades básicas, funcionais e estruturais que podem ser utilizadas na definição de especificações, no desenvolvimento e implementação de sistemas de controlo distribuído. É com base neste princípio, que a norma, disponibiliza vários objetos de *software* entre os quais se podem destacar os blocos função de serviço utilizados para implementar as comunicações entre componentes (SIFB – *Service Interface Function Block*). Este repositório de objetos disponibiliza uma grande variedade de blocos de função standards e específicos, que operam, essencialmente, sobre a manipulação de eventos e de dados. É com base nestes pressupostos que se definiram as estratégias de resolução dos problemas associados à replicação, isto é, de sincronismo e de determinismo das mesmas.

Assim, a solução de sincronismo proposta e implementada visa dar respostas aos cenários de interação entre componentes, apresentados no item 7.3.2. Neste sentido foi desenvolvida uma abordagem de sincronização tendo como base de comunicação, os SIFBs de comunicação standards, do tipo *Client/Server*, ver Figura 7-8. Para isso desenvolveu-se um modelo constituído por três réplicas e um elemento de votação alocadas a um único dispositivo, designado de PC_1, e por um dispositivo de monitorização e de interação com a aplicação designado de HMI. A cada uma das réplicas é atribuída a responsabilidade de execução dos FBs principais "F_ADD e ITI_SUM" sendo que a última instância, alocada a um quarto recurso, é utilizada para realizar a consolidação dos valores enviados pelas réplicas, FB votador “VOTER”.

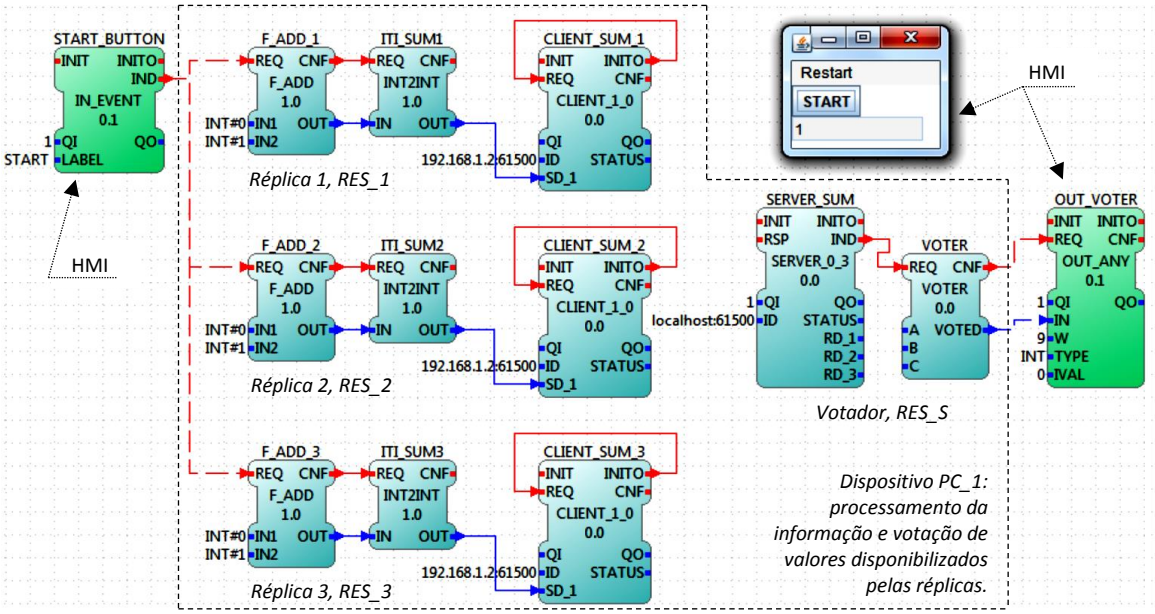


Figura 7-8. Estrutura do sistema distribuído e replicado usando SIFB Client/Server (Santos e Sousa, 2018)

O bloco função START_BUTTON, alocado ao dispositivo HMI, é utilizado como elemento de inicialização das réplicas sincronizando o seu estado de ativação, réplicas idem-potentes. Estas, adicionam os valores inteiros de 0 e 1, em resposta ao evento de inicialização, que, após notificação sequencial e temporal (predefinida *offline*) dos *clients,* disponibilizarão, através dos canais de comunicação, os dados a disseminar. A instância VOTER esperará pelos dados a consolidar. O valor elegido é enviado para o bloco função OUT_VOTER que o apresentará na janela de interação (janela do HMI).

Todas as instâncias da aplicação encontram-se a correr num único computador simulando um ambiente distribuído virtual, constituído por dois dispositivos (PC_1 e HMI), comunicando com o auxilio dos protocolos de comunicação TCP/IP (comunicação ponto-a-ponto) por meio de uma

rede Ethernet, alocada ao PC e fundamental para a execução da aplicação, Figura 7-9. Note-se, no entanto, que não faz sentido testar a aplicação num único PC (sem necessidade de sincronização de relógios) uma vez que o sistema em teste não é efetivamente um sistema distribuído, mas sim uma simulação do mesmo. No entanto é uma ferramenta importante para a simulação da abordagem de sincronização dos dados recebidos das réplicas.

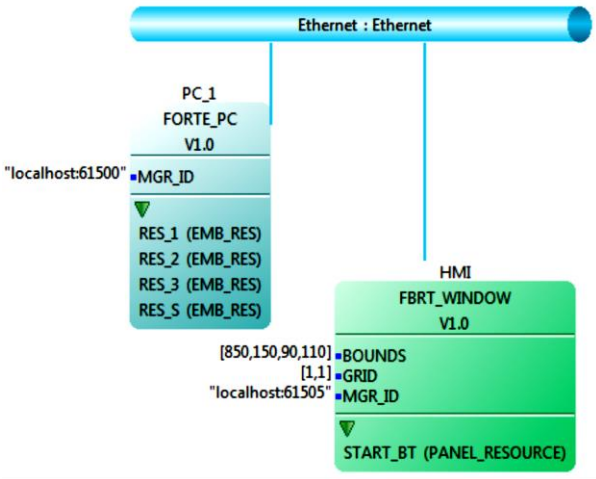


Figura 7-9. Visão da distribuição do sistema *e da rede de comunicações*

7.4.1 Interação entre réplicas em modo Client/Server

De entre os cenários apresentados na Figura 7-4 esta abordagem poderá ser aplicada aos cenários 2 e 5 sincronizando-se as réplicas recetoras bem como o elemento de votação. Deste modo, o par de comunicação *Client/Server*, usando os protocolos de difusão da Internet (IP), assegurará as diversas implementações, mapeando-as para os diferentes protocolos de comunicação subjacentes, pelo que as aplicações distribuídas poderão comunicar via rede local (por exemplo, *Ethernet*) e até mesmo por uma rede global, via Internet.

A comunicação entre os diferentes dispositivos e as diferentes réplicas será realizada com recurso a SIFBs de comunicação implementados segundo uma arquitetura em camada (Hofmann, 2011). Neste sentido, a comunicação entre instâncias será realizada utilizando-se diferentes versões standard dos SIFBs *cliente/server* assentes no protocolo de interligação IP, tendo como base de transmissão o protocolo TCP, ou seja, ligação TCP/IP. A execução, em ambiente FORTE, é realizada através de protocolos de difusão (ponto-a-ponto) suportada por endereços IP e pela definição das portas de comunicação (*endereço:porta*). Neste sentido o programador precisará apenas de configurar as camadas de comunicação de cada um dos blocos função de comunicação utilizados, o que será realizado através da indicação do parâmetro de identificação ID usando-se a seguinte sintaxe: layer [layer_parameter]. As comunicações entre o *client* e o *server* são implementadas entre a camada superior, normalmente FBDK utilizando o protocolo TCP, e a camada inferior, geralmente a camada IP. Isto significa usar, por exemplo, a sintaxe fbdk[].Ip[192.168.1.2:61500].

No caso da interação do tipo *client/server* a comunicação é realizada ponto-a-ponto, ou seja, apenas um *CLIENT* pode enviar e receber dados de e para um servidor. O *SERVER*, alocado ao dispositivo 1 (PC_1, recurso RES_S), possui um ID associado ao endereço do seu hospedeiro (*localhost:61500)* e o número da porta de comunicação, 61500. Todos os clientes possuem um ID composto pelo endereço IP do dispositivo de alocação do servidor e pelo mesmo número de porta usado no ID do *server.* Deste modo, no modelo de sincronização apresentado, os IDs dos *CLIENTs*

(localizados no dispositivo PC_1) assumirão o endereço IP do hospedeiro (*localhost*) e o número da porta de comunicação que, por defeito e salvo alteração, é o número da norma, Figura 7-10.

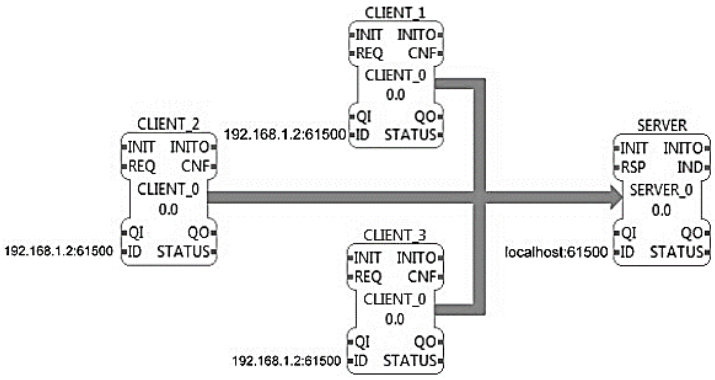


Figura 7-10. Difusão ponto-a-ponto (Client/Server), cenário 5 (muitos-para-um)

Note-se, no entanto, que todos os dados enviados pelas réplicas deverão ser consolidados num único valor. Neste caso o FB recetor irá decidir sobre a consolidação do valor de saída, por meio do algoritmo de votação ou de sincronização, pelo que este terá que ser determinístico e não poderá ter qualquer elemento de aleatoriedade. Assim sendo, na abordagem ao problema de sincronização do cenário 5, utilizando uma abordagem assente única e simplesmente nos pares de comunicação *client/server*, os tempos em que os dados são disseminados para o FB recetor não terão que ser, necessariamente, considerados. Neste cenário o *server* esperará até que todos os *clients* enviem os dados ou até que seja obtido o número de eventos/dados necessários à sua execução, sincronização das réplicas emissoras, onde o votador, enquanto elemento de finalização, apenas terá que votar os valores recebidos independentemente da ordem de receção. No entanto, a aplicação desta abordagem ao cenário 2 carece da garantia de sincronização e do determinismo das réplicas dado que estas produzirão valores diferentes caso não seja possível obter-se a sincronização das mesmas. Por outro lado, os eventos/dados recebidos deverão ser consolidados num único evento/dado antes de serem enviados para os FBs subsequentes (Dubinin, 2018). Neste sentido é possível criar uma estrutura de suporte à replicação de aplicações IEC 61499 utilizando as diversas implementações dos SIFBs standards de comunicação, considerando-se as diferentes existências dos diversos componentes replicados. Assim, várias versões dos SIFBs de comunicação de suporte à replicação terão que ser utilizadas, dependendo se o remetente, o destinatário ou ambos são replicados (Santos e Sousa, 2008; Sousa, 2014). Assim sendo, usando-se as implementações corretas de interação para cada um dos cenários apresentados na Figura 7-4, poderemos restringi-las a apenas dois modos de interação que serão, para além da típica implementação do protocolo de comunicação *um-para-um* preconizada pela norma (cenários 1 e 3, *1:1*), a comunicação *um-para-muitos* (cenário 2 e 4, *1:N*) e a comunicação *muitos-para-um* (cenário 4 e 5, *N:1*).

É de salientar como já foi referido, que numa abordagem à sincronização suportada por SIFB de comunicação *client/server* só um único cliente poderá encontrar-se ativo no momento de envio dos dados. Isto quer dizer que em cada uma das réplicas o *client* só poderá encontrar-se ativo durante o período de tempo necessário para o envio e a transmissão da informação. Esta imposição leva a que o programador tenha de definir *offline*, para cada réplica, o tempo de ativação e de duração de cada ligação ao servidor assegurando a sua sequencialidade. Neste sentido, o *Server* recebe os dados dos vários SIFB de comunicação *Client*, alocados às réplicas, de acordo com um protocolo de *token* temporizado, assumindo um fluxo de informação em tempo real por réplica e por ciclo de funcionamento.

O *token* temporizado, utilizado nesta abordagem é considerado como um elemento disparado em intervalos de tempo predefinidos e que transita ao longo de uma rede do tipo *token*

ring constituída por 3 recursos e com uma identificação única no intervalo [1, 2 e 3], numeração associada às réplicas. O *token* circula em torno de um pseudo-anel (Figura 7-11) do recurso *i* para os recursos *i+1*, *i+2*, … até ao recurso *i+(n-1)*, retornando para *i*. A este processo de transição há que associar aos tempos de funcionamento a latência do sistema, conhecida como a latência do anel (*token walk-time* (W)), que corresponderá à soma de todas as latências de todos os W*i* dos recursos (tempos de espera e de propagação do *token*).

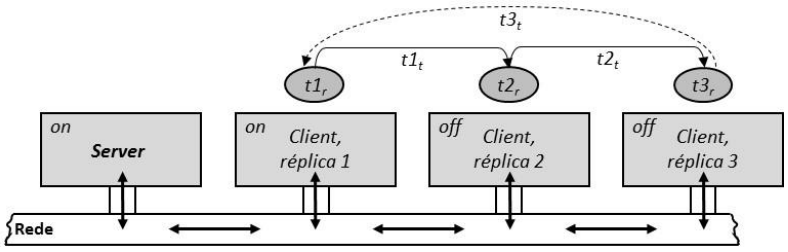


Figura 7-11. Esquema de passagem do token temporal, pseudo-anel

Neste modelo de implementação o tempo necessário para completar o ciclo será dado pelo tempo limite de operacionalidade do último recurso (réplica 3) mais o tempo mínimo necessário para se efetuar a receção dos dados pelo servidor. O tempo de latência total W_{Token} de transmissão do sistema corresponderá a:

$$W_{Token} = \sum_{i=1}^{n}(Ti_{retenção} + Ti_{transmissão}) \quad (7.1)$$

Assim sendo, o *token* viaja entre os recursos de forma pseudo-circular onde cada cliente poderá transmitir os dados, somente, quando possuir o *token*. Os clientes enviam os dados e/ou eventos um após o outro, no intervalo de tempo de ativação predefinido, *offline*, enquanto o servidor aguarda até que todos os dados sejam recebidos. Nesta abordagem definiu-se o processo de posse e de passagem de *token,* para cada uma das réplica, de acordo com a implementação apresentada na Figura 7-12. Neste esquema apresentam-se os tempos associados à ativação de cada um dos SIFBs *client*, para cada uma das réplicas, e por consequência os tempos de posse e de transição entre as três réplicas do sistema.

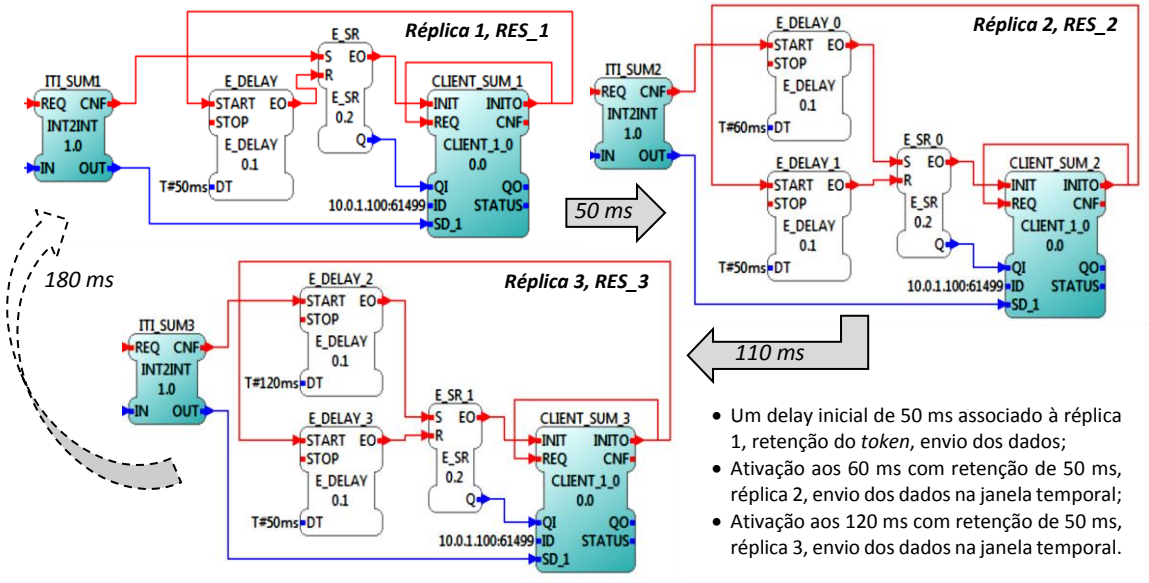


Figura 7-12. Arquitetura de passagem do token temporal entre réplicas

Neste sentido, de acordo com a arquitetura do sistema, a réplica 1 recebe o *token* assim que a operação de adição for concluída. O evento de conclusão define o início de ativação do *client* associado "CLINT_SUM_1" retendo, durante 50 ms, a ligação ao server. Do mesmo modo que o *client* da réplica 1 ficou ativo após a execução do FB "ITI_SUM"(ver Figura 7-8) todos os FB *delay*, de inicialização do *client*, entram, imediatamente, em contagem decrescente para a ativação do *client* associado. Assim, a réplica 2, encontra-se a enviar dados para o server passados 60 ms do início de ativação da aplicação, retendo a ligação durante 50 ms. No fim deste tempo a ligação ficará desfeita e o sistema fica em espera até ativação do *client sum 3*. 50 ms depois o processo encontra-se concluído pelo que o *token* pode retornar à origem. Tome-se em atenção ainda que o *token* não retornará efetivamente à origem uma vez que não será feita uma passagem física do mesmo para a réplica 1, isto é, o *token* "morre" na réplica 3. No entanto, este processo não ficará bloqueado na réplica 3 uma vez que retomará o percurso normal assim que o botão de START for novamente pressionado, ativação do *token* da réplica 1. Assim sendo, o tempo total de circulação do *token* será dado pela equação (7.2) pelo que o *server* deverá esperar um tempo mínimo para a receção completa dos dados de:

$$W_{Token} = \sum_{i=1}^{n}(50 + 10) + (50 + 10) + (50 + 10) = 180\,ms \tag{7.2}$$

Esta espera e o respetivo sincronismo das réplicas emissoras só é possível ser obtido dado que se optou por utilizar SIFBs com interfaces e estruturas diferentes. Os *CLIENTs* apresentam interfaces com uma só entrada (SD_1, CLIENT_1_0) enquanto o *SERVER* apresenta uma interface com três saídas. Estas diferenças condicionam o par de comunicação o que leva, necessariamente, a que o *server* tenha que receber três dados (RD_1, RD_2 e RD_3, SERVER_0_3). Este condicionalismo é imposto pela estrutura de funcionamento interna do SIFB *server* uma vez que a indicação de finalização da receção, evento IND, só será emitido quando associado aos três dados, ou seja, quando se verificar a receção de um número de dados igual a três (totalidade dos dados esperados), ver Figura 7-13.

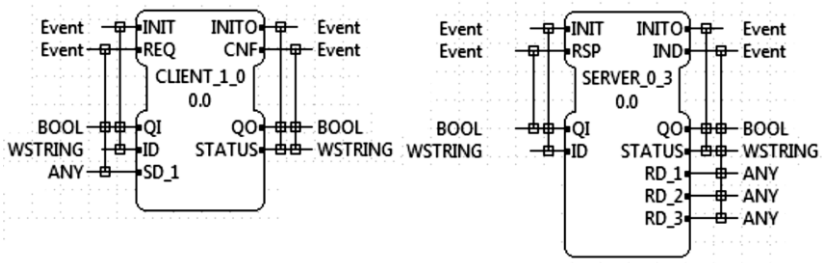


Figura 7-13. Client/Server interface usada nos cenários 4 e 5 (Santos et al., 2018)

7.4.2 Consolidação e votação do resultado, par Client/Server

Os votadores podem ser desenvolvidos de acordo com as mais variadas técnicas de tolerância a falhas, no entanto estes possuem como fim último a comparação dos resultados de duas ou mais variáveis e decidirem qual o resultado correto, se existir. Existem de facto muitos tipos de votadores (Pullum, 2001) e a decisão sobre qual o algoritmo de votação a usar, dependerá da semântica requerida pela aplicação, no entanto, para que a votação seja viável, todas as réplicas devem enviar os dados no tempo esperado.

Assim, na abordagem apresentada, espera-se que os eventos e os dados sejam enviados de e para as réplicas (cenários 2 e 5), localizadas nos diferentes recursos, sem que nenhum tempo seja incluído, ou seja, sem indicação do instante em que o evento se tornou disponível. Neste

contexto, no que se refere ao cenário 5, diferentes réplicas a comunicarem com um FB não replicado, o processo de sincronização é realizado com recurso a pares *Client/Server*. As réplicas, possuindo cada uma um *Client* (com interface diferente da do S*erver*, ver Figura 7-13), enviam mensagens, sem qualquer inclusão de tempo, para um mesmo *Server* de acordo com os tempos predefinidos para a passagem do *token*. Este escalonamento das respostas das réplicas é conseguido pela definição dos tempos de disparo de cada cliente, tempo de passagem do *token*, de acordo com uma ordenação predefinida para a qual, a ocorrência de um qualquer evento ou dado, se encontra forçada à priori. O servidor aguardará até que o conjunto de eventos e dados, provenientes dos clientes, sejam recebidos, Figura 7-14. É somente neste momento que os dados são consolidados, isto é, se realiza a eleição do dado. Note-se que, de acordo com o princípio de funcionamento dos pares *Client/Server*, o SIFB *Server* deverá encontrar-se inicializado antes que qualquer um dos *Clients* sejam inicializados. Este pressuposto encontra-se materializado no esquema apresentado na Figura 7-14 pelo ponto *Start server*.

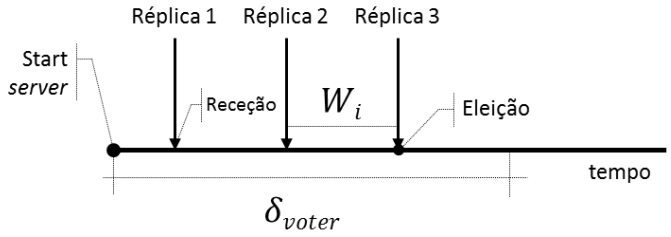


Figura 7-14. Consolidação de dados com receção completa, sem falhas

Nesta situação, o votador receberá todo o conjunto de dados enviados por todos os recursos, realizando a votação imediatamente após a receção do último dado. Os dados são difundidos para o *server* esperando que este elemento de votação os receba assim que estes sejam disseminados na rede. Por outro lado, no caso de uma ou mais mensagens falharem (Figura 7-15) o votador aguardará até δ_{voter}, executando, somente, nesta fase a eleição do dado. Note-se que este procedimento deverá ser implementado num contexto *muitos-para-um* e que o tempo de decisão será dependente do pior tempo de resposta do último dado recebido.

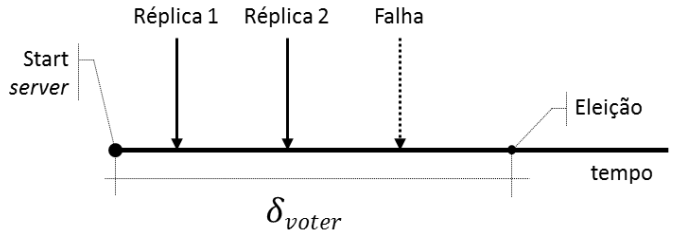


Figura 7-15. Consolidação de dados com receção incompleta, com falhas

O FB VOTER (Service Interface FB (SIFB), escrito em C++) foi desenvolvido para realizar a consolidação dos dados através de uma votação maioritária. Este processa os dados disponibilizados pelo *server*, independente da ordem de receção, após ativação do evento IND, indicador de finalização de receção de dados no server, Figura 7-16. O votador, alocado ao recurso S (RES_S), recebe os dados provenientes das diferentes réplicas consolidando-os num único valor a transferir para o FB subsequente, visualização na janela de interação do dispositivo HMI, isto é, no OUT_VOTER.

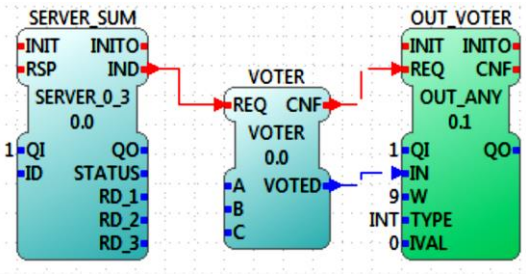


Figura 7-16. Server, votador e output do valor elegido

Nesta abordagem, uma vez que a replicação é tripla e modular (TMR), foi utilizada uma votação maioritária que, no caso das três réplicas, é feita por comparação simples dos valores recebidos em A e C. O mecanismo de votação determina qual o valor a eleger de acordo com o *Algoritmo 1*. Parte do possível código de implementação da votação maioritária é apresentado em seguida:

Algoritmo 1 – Votador

```
1:   void FORTE_VOTER::setInitialValues(){
2:      A() = 0;
3:      B() = 0;
4:      C() = 0;
5:      VOTED() = 0;
6:   }

7:   void FORTE_VOTER::executeEvent(int pa_nEIID){
8:      switch(pa_nEIID){
9:      case scm_nEventREQID:
10:        if (A() == C())
11:            VOTED() = A();
12:        else
13:            VOTED() = B();
14:        sendOutputEvent(scm_nEventCNFID);
15:        break; }
16:  }
```

A interação deste bloco função com o restante sistema é realizado através das variáveis de entrada e de saída definidas nas linhas 2 a 5 (*Algoritmo 1*, *setInitialValues*). A execução deste bloco encontrar-se-á em espera até que o evento REQ (*scm_nEventeREQID*) seja ativado (linha 9) desencadeando a execução da ação de comparação e escolha do inteiro a votar (linhas 9 a 13). O valor de saída VOTED será disponibilizado pelo evento CNF, saídas interligadas (ver interface do votador, Figura 7-17b), que confirma a conclusão da execução do FB (*scm_nEventCNFID*). A eleição do dado a visualizar é realizada tendo em conta o conceito binário dos valores recebidos e, como tal, as combinações possíveis dos valores recebidos enquadrar-se-ão numa potência de 2^3 [0,0,0; 0,0,1; … ;1,1,0; 1,1,1].

Há que considerar ainda que o modelo disponibiliza os dados ao votador de acordo com a ordem em que este os recebeu. Assim sendo, os dados rececionados no *server* são independentes do tempo em que estes são disponibilizados pelo que os dados serão sempre ordenados de acordo com a ordem de chegada. O mecanismo de decisão irá validar os dados consolidando-os num único valor que estará compreendido entre 0 e 1 ou, em caso de não receção de valores, o valor 5. Se 0 e 1 são os valores esperados das operações realizadas pelas réplicas o valor 5 será o valor injetado no votador pelo CFB "WCET_SER", FB WCET para a abordagem *Client/Server*. Uma possível simulação do envio de dados e dos valores rececionados pelo votador pode ser visualizada na Figura 7-17a.

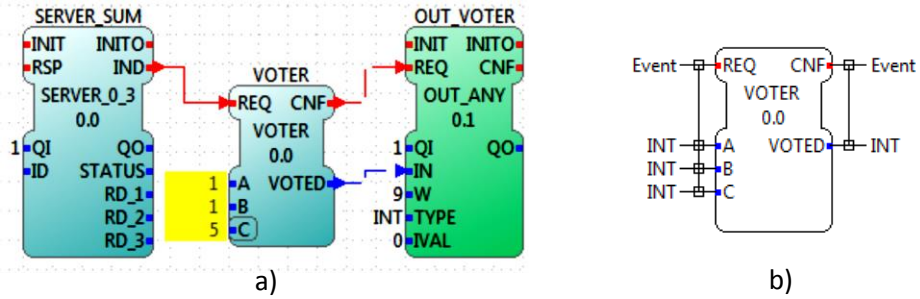


Figura 7-17. Componente de consolidação dos dados, e interface do votador

O valor de saída, valor a enviar para o FB "OUT_VOTER", é resultado da comparação de A e C onde, em caso de igualdade, a escolha recairá sempre sobre A. Por outro lado, no caso de não ser verificada esta igualdade, o valor de saída assumirá o valor de B que, no exemplo apresentado na Figura 7-17a, será igual a 1. Note-se que nesta primeira abordagem ao problema da replicação de sistemas optou-se por usar o valor 5 como indicador de uma falha de comunicação ou como indicador de expiração do tempo de receção dos dados. Na Figura 7-17b apresenta-se a interface do VOTER e respetivas interligações.

7.4.3 Worst-Case Execution Time, par Client/Server

Existem de facto muitos tipos de votadores (Pullum, 2001) cuja escolha depende da semântica da aplicação, mas para que a votação seja possível, todas as réplicas terão que enviar os dados no tempo esperado. Assim, para garantir-se o consenso, todas as réplicas precisam enviar os dados para o *server* num intervalo de tempo preestabelecido que contemple não só o tempo de processamento lógico, mas também a latência do sistema, WCET (*Worst-Case Execution Time*) (Lednicki, 2013). No modelo WCET desenvolvido (Figura 7-18) o votador solicita os dados ao servidor após a ativação do evento REQ lendo-os de uma só vez. Neste sentido, quando a execução de uma ou mais réplicas for interrompida, o *Server* aguardará indefinidamente pela receção dos dados e, como tal, o votador não poderá produzir resultados. Neste caso é preciso forçar o *server* a disponibilizar os dados esperados e emitir o evento IND (*Service Indication*) que indicará a receção completa dos mesmos.

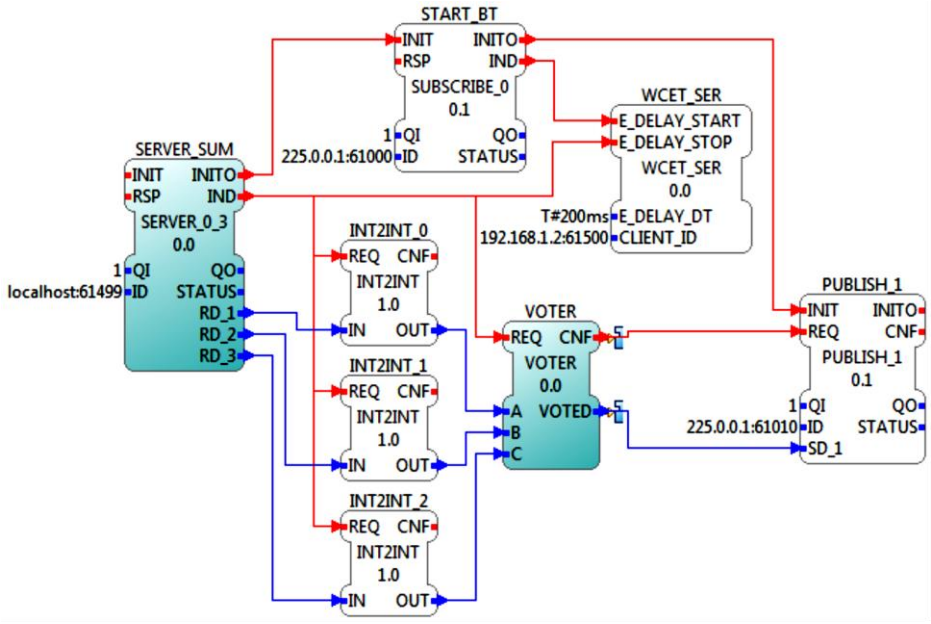


Figura 7-18. Implementação do CFB Worst-Case Execution Time para Client/Server (CFB WCET_SER)

O facto é que ao garantir-se a sincronização do sistema através do uso do SIFB *Server* iremos incorrer num impasse (*deadlock*) sempre que se verifique a não receção de qualquer um dos dados enviados. Este impasse só poderá ser quebrado pelo envio de dados adicionais que caracterizam a maior latência do sistema ou uma falha de comunicação entre SIFBs. O CFB "WCET_SER" foi desenvolvido com o objetivo de terminar com o impasse a que o *server* poderá estar sujeito sempre que um dado não seja recebido, ver Figura 7-18.

Neste sentido, para obviar os bloqueios ocasionais do componente de votação utilizando uma abordagem *client/server*, foi desenvolvido e adicionado aos objetos de repositório, o CFB "WCET_SER" que será inicializado pelo evento de sincronização das réplicas START_BT. Este CFB encontrando-se parametrizado para um tempo de ativação de 200 ms (ligeiramente superior ao tempo necessário para que o *token* percorra todas as réplicas, ver (7.2)), e possuirá o mesmo ID do *server* (192.168.1.2:61500). Note-se que este CFB, alocado ao recurso S (RES_S, Figura 7-9, PC_1), só poderá ser utilizado em conjunto com um *server*, uma vez que utiliza um SIFB *Client* para gerar os dados em falta. A estrutura do CFB e as interligações da sua rede interna são apresentadas na Figura 7-19.

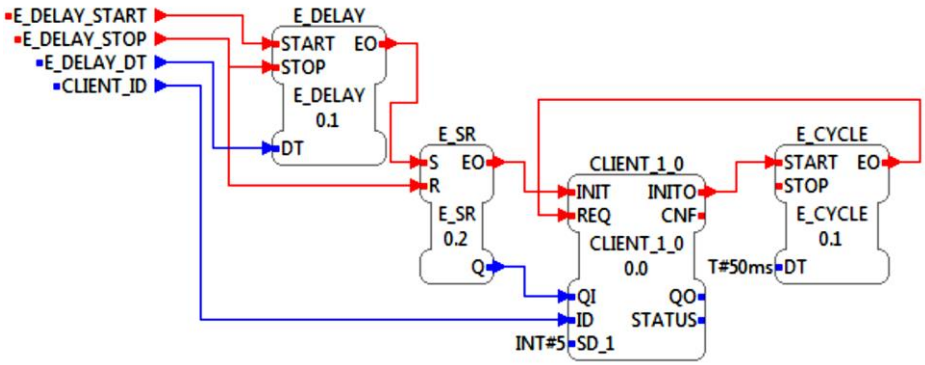


Figura 7-19. Rede interna do Composite Function Block WCET_SER

Este CFB é constituído por um bloco função E_DELAY, ativado pelo evento E_DELAY_START, e é parado, imediatamente, assim que o *Server* confirme a receção de todos os dados. O evento IND do *Server*, ligado ao evento E_DELAY_STOP faz a paragem da ativação do CFB. O SIFB CLIENT_1_0 inicializa-se após a passagem do tempo de espera predefinido para o FB E_DELAY_DT (200 ms) ativando imediatamente o FB E_CYCLE que, ciclicamente a cada 50 ms, ativa o evento REQ do SIFB CLIENT_1_0. O SIFB *Client* envia para o *Server* o inteiro 5 tantas vezes quantas os dados em falha. WCET_SER torna-se inativo após a receção de todos os dados, evento IND do SIFB *Client*, ver Figura 7-18. Nesta implementação o SIFB *Client* do WCET só será inicializado quando todos os outros se encontrarem desativados. O SIFB permanecerá ativo pela ação do bloco E_SR que o ativa e desativa em função dos tempos de receção dos dados recebidos ou perdidos durante a difusão dos mesmos.

Para finalizar esta abordagem podemos dizer, em modo de conclusão, que o modelo será capaz de realizar a sincronização das réplicas emissoras e, como tal, dos dados enviados, se os eventos ocorrem todos no intervalo de tempo predefinido. A ocorrência de um qualquer evento intermédio conduz ao descontrolo das réplicas uma vez que pode despoletar a ativação de *tokens* em simultâneo. A ativação de mais do que um *token* leva à ativação de mais que um *client* levando o sistema para um estado de erro que conduz à sua paragem. Por outro lado, de um ponto de vista dirigido estritamente para o votador, considerando que este se mantinha em funcionamento, seriam consolidados valores que, provavelmente, apontariam para uma falha de comunicação o que se traduziria, nesta abordagem, na votação do inteiro 5. Assim sendo, não se pode considerar este modelo como sendo uma boa abordagem uma vez que se encontra condicionado pela não ocorrência de eventos intermédios embora se possa assegurar a ordenação

dos dados desde que estes se enquadrem, obrigatoriamente, nos intervalos de tempo preestabelecidos.

7.5 Sincronização de réplicas com recurso a *Publish/Subscribe*

Com esta abordagem ao desenvolvimento da *framework* de replicação pretende-se dar resposta aos problemas evidenciados com a implementação *Client/Server*. Neste sentido, para se manter a identidade do modelo o mais semelhante possível mantiveram-se as mesmas instâncias, replicadas e não replicadas, mantendo-se uma estrutura idêntica à desenvolvida anteriormente. Esta nova abordagem será implementada utilizando-se os pares de comunicação SIFB *Publish/Subscribe*, de acordo com a arquitetura apresentada na Figura 7-20. A abordagem foi, implantada, com todas as suas instâncias, num único computador.

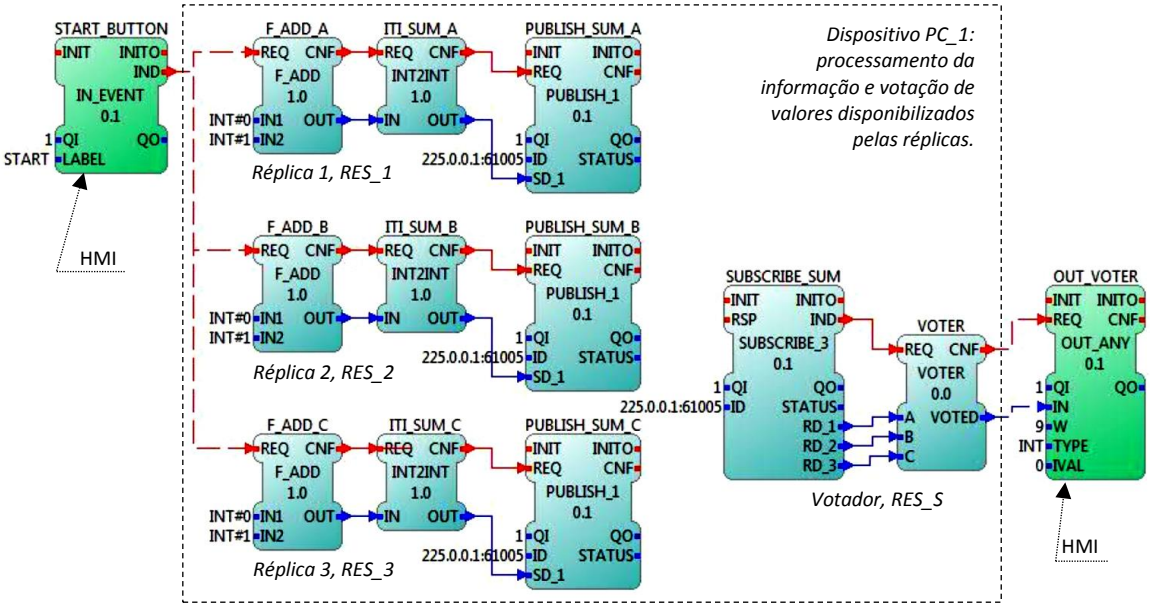


Figura 7-20. Estrutura de um sistema distribuído e replicado usando SIFB Publish/Subscribe

Assim, num cenário de sincronização *Publish/Subscribe*, os FBs devem possuir IDs que sigam as regras dos endereços de difusão *multicast* enquadrados no intervalo 224.0.0.0 até 239.255.255.255. Nesta implementação os *PUBLISH* encontram-se alocados quer ao dispositivo 1 (visualização, HMI) quer ao dispositivo 2 (execução das réplicas). Por seu lado, os *SUBSCRIBE*, alocados aos dois dispositivos, que se encontrem no mesmo segmento de rede que o *publish* receberão os dados e os eventos enviados por este, se possuírem o mesmo ID que o *publish* (Vyatkin, 2015). Assim sendo, os dados e eventos publicados serão recebidos pelo *subscribe* localizado no dispositivo 1 (eventos de sincronização do votador, comunicação unidirecional) e no dispositivo 2 (evento de sincronização das réplicas) no instante, aproximadamente, em que foram disponibilizados.

7.5.1 Interação entre réplicas em modo Publish/Subscribe

A comunicação entre as diversas instâncias será realizada utilizando-se as versões standard de SIFBs *Publish/Subscribe* assentes em protocolos de comunicação suportados por endereços IP, protocolo de comunicação UDP/IP. A execução da aplicação é realizada por difusão *multicast* suportada por IPs e pelas portas de comunicação, seguindo as regras de endereçamento da difusão *multicast* (Figura 7-21), que garantam a disseminação dos dados enviados pelas diversas

réplicas, isto é, de *muitos-para-um*, cenário 5 ou a ativação das réplicas, cenário 2, *um-para-muitos*.

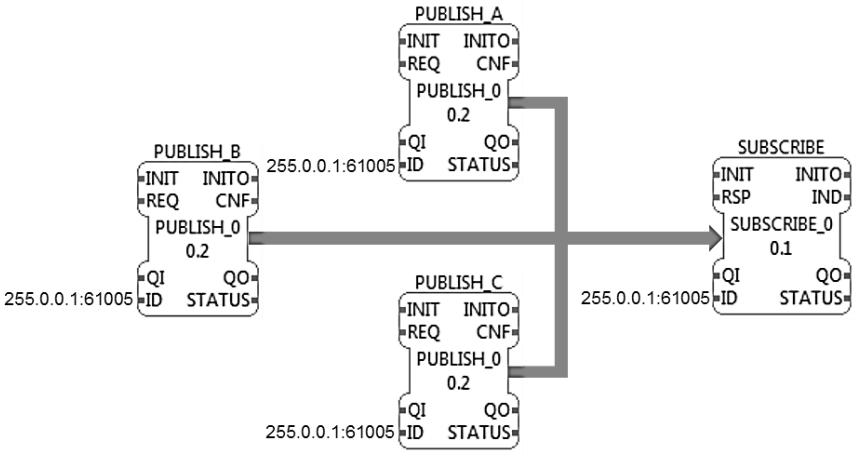


Figura 7-21. Difusão multicast unidirecional (Publish/Subscribe), cenário 5 (muitos-para-um)

Por outro lado, na sincronização das réplicas, cenário 2, também presente nesta implementação, será, igualmente, necessário um protocolo de difusão *multicast* que garanta a disseminação dos dados ou eventos produzidos, isto é, a passagem de eventos de *um-para-muitos*. Esta abordagem não garante à partida o determinismo uma vez que poderão ser disseminados vários eventos, ou dados, de cada uma das réplicas. Assim, numa situação de disparos múltiplos o determinismo permitirá distinguir a possível desordenação dos eventos numa situação em que, por exemplo, a ordem de desativação do sistema chegue antes da ordem de ativação. Assim sendo, mesmo sem possuirmos a garantia de que estes SIFB asseguram o determinismo das réplicas, poderemos modelar toda a aplicação com os pares *PUBLISH/SUBSCRIBE,* Figura 7-22.

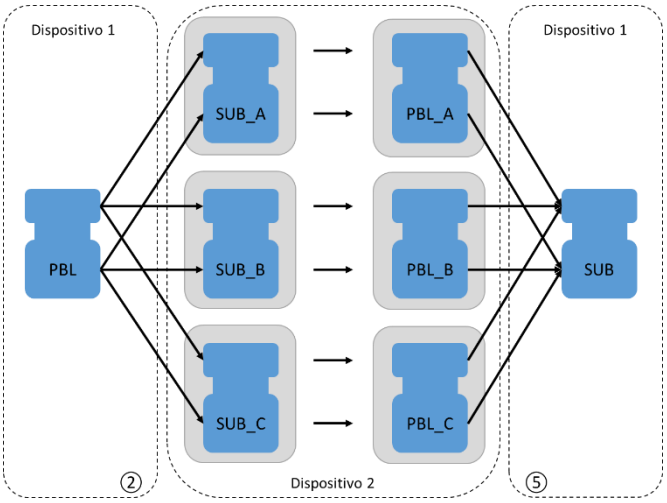


Figura 7-22. Passagem de eventos/dados, um-para-muitos (PBL/SUB) e muitos-para-um (PBL/SUB)

Assim sendo, o SIFB de comunicação *Publish*, à esquerda (PBL), é responsável pela disseminação de eventos do dispositivo 1 (evento de sincronização das réplicas, botão de inicialização da aplicação, *Start*) para o controlo lógico, no dispositivo 2. Os SIFBs de comunicação *Subscribe* alocados ao dispositivo 2 e replicados pelos vários recursos (SUB_X), recebem os eventos enquanto os *publish*, à direita, interagem, por sua vez, com o dispositivo 1, enviando os dados para consolidação no votador.

Note-se, no entanto, que na passagem de eventos no cenário 2 os SIFB de comunicação possuem a mesma interface gráfica, ou seja, o mesmo número de entradas e de saídas, contrariamente ao que se passa no cenário 5. É, efetivamente, por esta diferenciação que também nesta implementação, o *Subscribe* aguardará até que todos os dados sejam recebidos. Esta espera só será possível dado que utilizou-se SIFBs com interfaces diferentes, *Publish* com uma só entrada de dados (SD_1, PUBLISH_1) e um *Subscribe* com três saídas, que receberá os três dados replicados (RD_1 a RD_3, SUBSCRIBE_3), Figura 7-23.

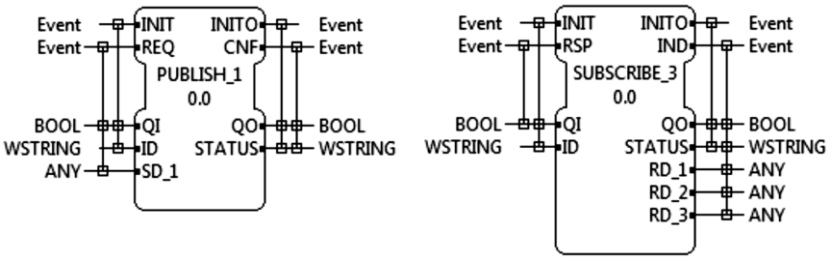


Figura 7-23. Publish/Subscribe interface usada no cenário 5

7.5.2 Consolidação e votação do resultado, par Publish/Subscribe

Como já foi exposto na abordagem anterior os eventos e os dados são recebidos das réplicas, localizadas nos diferentes dispositivos, sem que nenhum tempo de validade lhes seja incluído, ou seja, sem registo do instante em que os eventos se tornam válidos. Assim sendo, no cenário 5, as diferentes réplicas, possuindo cada uma um *Publish* (com interface diferente da do *Subscribe*, ver Figura 7-23), enviam os dados a consolidar para o *subscribe* alocado ao componente não replicado. O *subscribe* aguardará até que todos os eventos e dados sejam recebidos. É somente neste momento, que os dados são enviados para consolidação, isto é, para o votador. Esta espera é, igualmente, conseguida pela utilização de diferentes interfaces do par *Publish/Subscribe* que condicionam a evolução do sistema garantindo-se, deste modo, o sincronismo do mesmo. O votador, alocado ao recurso S, recebe os dados das diferentes réplicas consolidando-os num único valor a transferir para o FB subsequente, ver Figura 7-20. A consolidação dos dados será realizada segundo os pressupostos enunciados para os pares *Client/Server* excetuando-se o tempo de inicialização do *Subscribe*, Figura 7-24a.

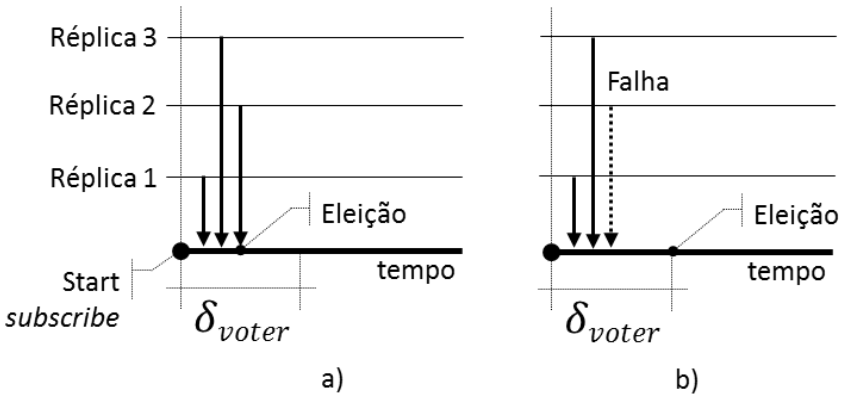


Figura 7-24. Consolidação de dados, pares Publish/Subscribe

No caso de uma ou mais mensagens falharem (Figura 7-24b) o votador aguardará até δ_{voter}, executando, somente, nesta fase a eleição do dado. Note-se que este procedimento deverá ser implementado num contexto *muitos-para-um* e que o tempo de decisão será dependente do pior tempo de resposta do último dado recebido.

O votador compara os valores A e C que, em caso de igualdade, elegera A como valor consolidado caso não seja verificada a igualdade entre A e C, o valor de saída assumirá o valor de B. Note-se que nesta implementação foi mantido, igualmente, o valor 5 como indicador de uma falha de comunicação ou como indicador de expiração do tempo de receção dos dados.

7.5.3 Worst-Case Execution Time, par Publish/Subscribe

A consolidação dos valores para votação só será viável se todas as réplicas enviarem os dados no tempo esperado. Sendo assim, todas as réplicas precisam enviar os dados/eventos para o *subscribe* num intervalo de tempo esperado que contemple não só o tempo de processamento lógico, mas também a latência do sistema, WCET (Lednicki, 2013). No modelo desenvolvido (ver Figura 7-20) o votador solicita os dados ao *Subscribe* após a ativação do evento REQ. Neste sentido, quando a execução de um ou mais dispositivos ou recursos for interrompida, o servidor aguardará indefinidamente pelos dados e o votador não poderá produzir resultados. É, pois, nesta situação que o *Subscribe* necessita ser forçado a produzir os dados esperados e emitir o evento IND que indica a receção completa dos mesmos.

O facto é que ao garantir-se a sincronização do sistema através do uso do *subscribe* iremos incorrer, novamente, num impasse (*deadlock*) sempre que se verifique a não receção de qualquer um dos dados. Este impasse só será quebrado pelo envio de dados adicionais que caracterizem a maior latência do sistema ou falha de comunicação entre os SIFBs. O CFB "WCET_SUB" foi desenvolvido com o objetivo de terminar com o impasse a que o *subscribe* poderá estar sujeito sempre que um dado não seja recebido, Figura 7-25.

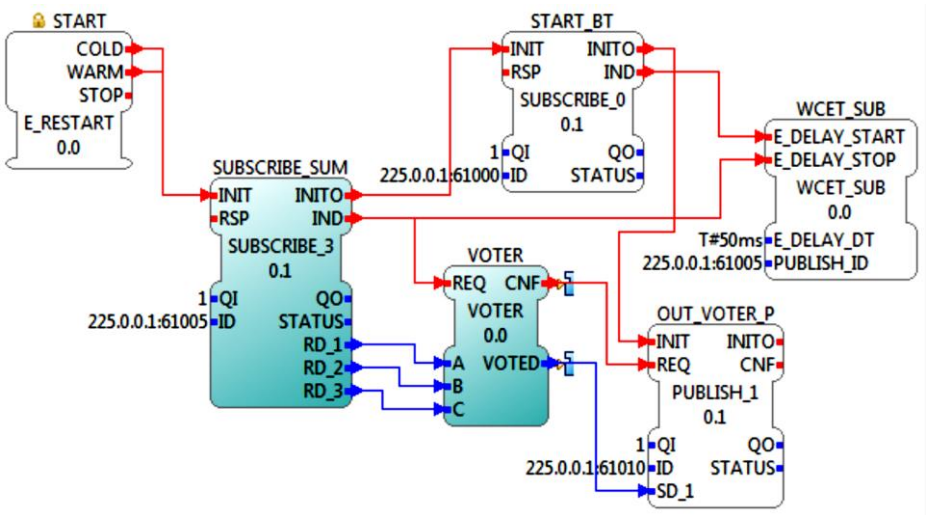


Figura 7-25. Implementação do CFB Worst-Case Execution Time (WCET_SUB)

Neste novo modelo de replicação utiliza-se CFB "WCET_SUB" que será também inicializado pelo *Subscribe* START_BT. Este encontra-se parametrizado para um tempo de ativação de 50 ms, possuindo o mesmo ID do *subscribe* (225.0.0.1:61005) associado ao votador. Note-se que este CFB, alocado ao recurso S (REC_S, Figura 7-9, PC_1), só poderá ser utilizado em conjunto com um *Subscribe,* uma vez que utiliza um SIFB *Publish* para gerar os dados em falta. A estrutura do CFB e as interligações da sua rede interna são apresentadas na Figura 7-26.

Este CFB apresenta uma construção idêntica ao CFB desenvolvido para o modelo *Client/Server*. O FB E_DELAY é ativado pelo START_BUTTON, evento E_DELAY_START interligação ao evento IND do *subscribe* START_BT, e é parado, imediatamente, assim que o *subscribe* confirme a receção de todos os dados, evento IND interligado a E_DELAY_STOP. O PUBLISH_1 inicializa-se após o tempo de espera (50 ms) ativando imediatamente E_CYCLE que, ciclicamente, a cada

50 ms, ativa o evento REQ de PUBLISH_1. O SIFB *Publish* envia para o S*ubscribe* o inteiro 5 tantas vezes quantas os dados em falha, ou seja, tantas vezes quantos os atrasos ou colapso da ligação entre os componentes replicados. WCET_SUB torna-se inativo após a receção de todos os dados, envio do evento IND do SIFB *Subscribe*, ver Figura 7-25.

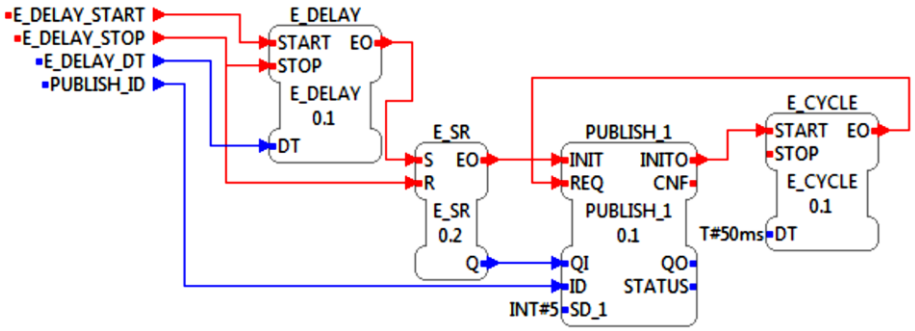


Figura 7-26. Composite Function Block WCET_SUB

É de salientar que a abordagem *PUBLISH/SUBSCRIBE* é mais eficiente, em termos temporais, do que a replicação realizada com pares *CLIENT/SERVER*. Nesta abordagem o envio dos dados é realizado assim que um qualquer dos *publishers* receba um evento na entrada REQ, enquanto na utilização de pares *Client/Server* o envio dos dados processa-se sequencialmente e de acordo com os tempos predefinidos para a ativação de cada *Client*. Os dados encontrar-se-ão disponíveis de acordo com a ordem de disparo de cada *Client* pelo que será necessário esperar pela ativação do FB "CLIENT_SUM_3" (último *client* a enviar dados o que acontecerá 170 ms após o início do envio dos dados pelo CLIENT_SUM_1). Na replicação suportada por pares *PUBLISH/SUBSCRIBE* não há a necessidade de esperar pela sua vez para enviar os dados, pois o envio dos mesmos acontece sempre que estes estejam disponíveis e o evento REQ seja ativado.

7.6 Sincronização de réplicas com recurso a *Timed Messages*

Como já foi referido ao longo deste texto, as mensagens que se tornaram disponíveis para o sistema ou para um outro componente do mesmo deverão ser consolidadas. Para isso, é necessário garantir o determinismo dos componentes replicados, replicação ativa. Assim sendo, o determinismo será conseguido pela aplicação do conceito de mensagens temporizadas (*timed messages*, Poledna *et al.*, 2000). Desta forma, o determinismo será obtido se o sistema for capaz de garantir que todos os componentes replicados usam os mesmos valores para determinar a ordem de execução. Neste sentido, a sincronização entre processadores deverá ser realizada através de um protocolo de sincronização de relógios onde o instante de disponibilidade das tarefas é obtido em função do pior tempo de execução da tarefa disponibilizada. Assim, de acordo com o mapeamento dos cenários de replicação apresentados na Figura 7-4 (cenários 2 e 5), deveremos sincronizar os relógios da aplicação de modo que, aos dados a disseminar possam ser associados os instantes de disponibilidade, definidos pelos tempos de execução dos FBs aos quais os eventos/dados se encontram interligados, isto é, imediatamente após a sua execução.

Neste sentido, a fim de implementar o protocolo de mensagens temporizadas, o FB responsável pelo envio, deve juntar aos dados o instante de validação associado aos mesmos, uma vez que o FB recetor não terá qualquer maneira de determinar o instante de validação destes. Este tempo de validação, será na realidade, o pior tempo de execução do FB que disponibiliza os dados (uma vez que na *framework* IEC 61499, os dados de saída só são disponibilizados quando o algoritmo terminar a sua execução) acrescido dos tempos de envio, que poderá ser determinado *offline* (Khalgui *et al.* 2005). Assim, a abordagem às mensagens temporizadas só fará sentido se a passagem de mensagens for o único método usado para transferir dados entre tarefas. Sendo

assim, uma tarefa *T* será considerada como uma entidade de processamento que recebe os dados, executa um algoritmo e produz novos dados a serem enviados para outra tarefa (Khalgui *et al.*, 2005). No caso das mensagens temporizadas utilizadas no contexto da IEC 61499, a passagem de dados será de facto limitada à passagem de mensagens. No entanto, se um FB receber eventos de entrada (se for ativada a sua execução) a partir de mais do que uma fonte, o FB não pode ser modelado como uma única tarefa, exigindo uma tarefa para cada uma das fontes de eventos de entrada. Neste caso as variáveis internas, assim como o ECC (máquina de estado interno de um BFB) ou outros quaisquer algoritmos têm de ser considerados como sendo dados passados entre tarefas, fora do controlo do protocolo de mensagens temporizadas. A caraterização de uma tarefa T_k pode ser, num contexto de passagem de mensagens, definida como:

$$T_k = \{dados\ processados, instante\ de\ validade\} \tag{7.3}$$

Assim, na interação entre elementos replicados com elementos não replicados ou com outros replicados (cenário 4, passagem N:N), a difusão dos dados deverá ser realizada com recurso à difusão *multicast* (ligação do tipo UDP/IP) executada por cada uma das réplicas responsáveis pelo envio dos mesmos. No entanto, para se assegurar que todas as réplicas trabalham com o mesmo conjunto de dados e que produzem os mesmos valores de saída, os dados recebidos terão que ser ordenados e disponibilizados de acordo com os tempos em que estes se tornaram válidos. Esta sincronização e consequente determinismo terão que ser obtidos pelo desenvolvimento de um FB que garanta a ordenação, a espera e a disponibilização dos dados, no tempo desejado. Este será o ponto basilar para a obtenção do determinismo uma vez que a IEC 61499 nada especifica quanto à sua implementação. Na Figura 7-27 apresenta-se a arquitetura que será utilizada como exemplo para os testes de determinismo que irão ser realizados ao longo desta tese.

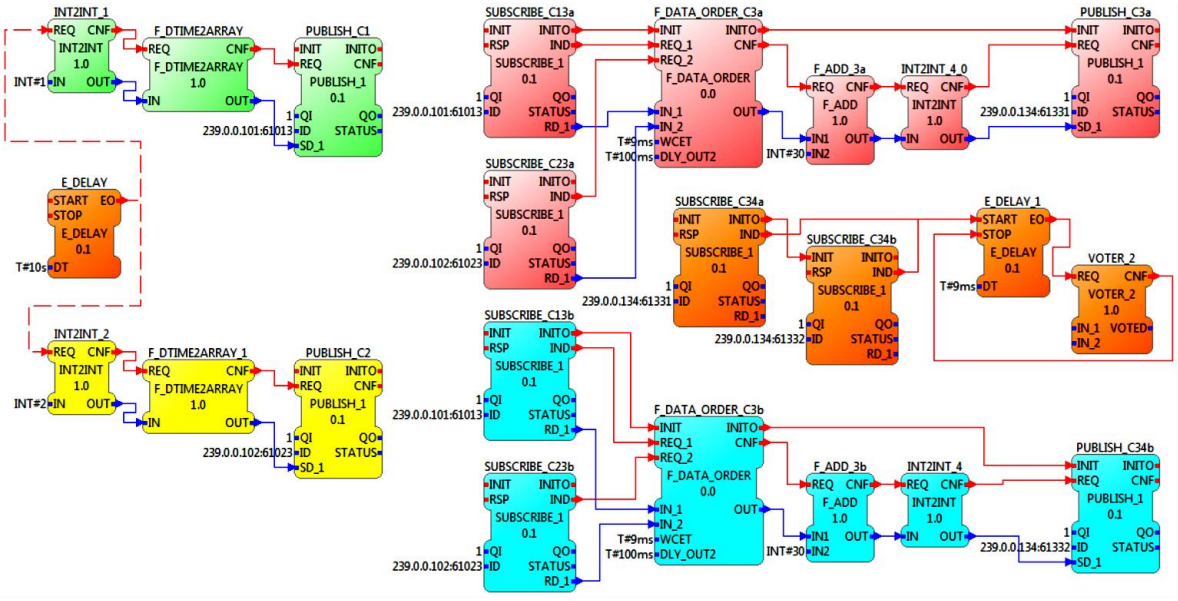


Figura 7-27. Arquitetura do exemplo determinístico implementado

Neste sentido, a introdução dos instantes em que os dados se tornam válidos, leva-nos à criação de mecanismos de transmissão de mensagens temporizadas, fundamentais para a obtenção do determinismo. Para isso foram desenvolvidos novos SIFBs que possam ser garantes do sincronismo das réplicas. Assim, de acordo com a proposta apresentada por Santos e Sousa (2010), interposição de um FB de gestão de réplicas (*RManag*), os novos blocos função foram intercalados quer agregados ao *Publish* (F_DTIME2ARRAY, criação de mensagens temporizadas,

gestão do tempo), quer agregados ao *Subscribe* (F_DATA_ORDER, sincronização e ordenação das mensagens, gestão de mensagens). Por outro lado, uma vez que a aplicação se encontra distribuída e replicada por vários nós, torna-se necessário que, para além da utilização de um mecanismo de distribuição que garanta a atomicidade do sistema (SIFB *Publish*, que permite comunicação do tipo *multicast*), se utilizem mensagens temporizadas (mensagens provenientes dos e para os elementos replicados). Para isso, será necessário intercalar em cada um dos componentes que comuniquem com componentes replicados o bloco função (SIFB) F_DTIME2ARRAY de agregação do instante de validade do dado, Figura 7-28.

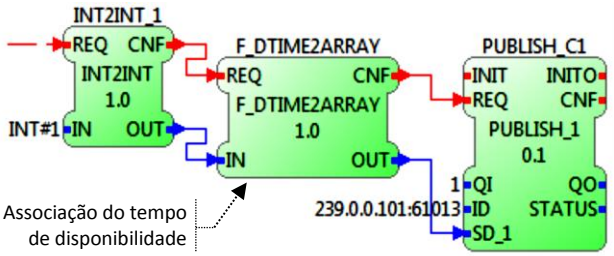


Figura 7-28. Intercalação do SIFB F_DTIME2ARRAY

Na Figura 7-29 apresenta-se a interface do Service Interface FB F_DTIME2ARRAY desenvolvido para a criação de mensagens temporizadas formatadas como {*dado, tsec, tnsec*}.

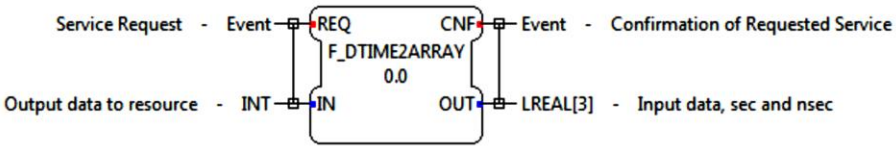


Figura 7-29. Interface do FB de criação de mensagens temporizadas

Da análise da estrutura interna do SIFB, *Algoritmo 2*, podemos apresentar parte do possível código de implementação de mensagens temporizadas desenvolvido para o FORTE POSIX. Este objeto de repositório, *Service Interface Function Block* desenvolvido pelo programador, é definido unicamente como a interface de validação do instante de disponibilidade das mensagens, onde o instante de validade da tarefa é determinado no emissor (após execução do FB e de acordo com as informações disponíveis no recurso ou no dispositivo, ver Figura 7-28).

Algoritmo 2 – Criação de mensagens temporizadas

```
1:  void FORTE_F_DTIME2ARRAY::setInitialValues(){
2:     IN() = 0;
3:     OUT_Array().fromString("0"); }

4:  struct timespec tsa;

5:  void FORTE_F_DTIME2ARRAY::executeEvent(int pa_nEIID){
6:     switch(pa_nEIID){
7:     case scm_nEventREQID:
8:        clock_gettime(CLOCK_REALTIME, &tsa);
9:        OUT()[0] = IN();
10:       OUT()[1] = tsa.tv_sec;
11:       OUT()[2] = tsa.tv_nsec;
12:       sendOutputEvent(scm_nEventCNFID);
13:    break; }
14: }
```

A mensagem a disseminar para as réplicas é constituída pelos dados, a data/hora em segundos e a sua respetiva fração em nanossegundos. O evento CNF (*Confirmation of Requested Service*) confirmará a disponibilidade da mensagem, notificando, neste caso, o *publisher*, associado ao recurso ou dispositivo.

A interação deste módulo com o restante do sistema é realizada através das variáveis de entrada e saída definidas nas linhas 2 e 3 (*Algoritmo 2*, *setInitialValues*). A execução do SIFB encontrar-se-á em espera até que o evento REQ (*scm_nEventeREQID*) seja ativado (linha 7). A ativação deste evento desencadeia a ação de associação do instante de disponibilidade aos dados recebidos e a criação de um array, do tipo real (LREAL), constituído pelos dados e o respetivo instante de validade (linha 9 a 11). As mensagens de saída OUT (dados de saída com o instante associado) serão disponibilizadas segundo o formato [*dado*, *tempo em s*, *tempo em ns*]. O evento CNF confirma a conclusão da execução do SIFB (*scm_nEventCNFID*). A ativação do evento CNF, associado a OUT, disponibilizará as mensagens em OUT.

Vários outros problemas terão ainda de ser resolvidos para que se possa tirar partido do determinismo das réplicas, principalmente, a garantia de sincronismo dos relógios dos diferentes dispositivos. Assim, para que todos os relógios trabalhem no mesmo espaço temporal, quer seja universal ou associado somente ao servidor de tempo da rede local, instalou-se um servidor NTP (*Network Time Protocol*), em ambiente Linux, que deverá sincronizar todos os relógios, para o valor de referência do servidor, dos equipamento que possam encontrar-se interligados à rede local (minicomputadores do tipo RPis), ver processo de instalação do servidor e dos clientes no Anexo A. Esta sincronização é realizada em função do tempo real obtido por sincronização externa realizada em função do UTC (*Universal Time Coordinated*) ou em função do TAI (*Temps Atómic International*) recebidos por difusão (RPi com ligação à Internet).

Uma vez sincronizados os relógios dos equipamentos, será possível implementar o mecanismo de obtenção do determinismo das réplicas. Neste sentido, para que este seja possível, agregou-se aos *Subscribe* o bloco função composto F_DATA_ORDER, agregação realizada nos nós replicados, que garantirá o determinismo dos mesmos uma vez que, no desenvolvimento do mesmo, se adicionaram funcionalidades específicas de gestão de réplicas. A interface, desenvolvida e adicionada ao repositório de objetos do *4diac*, receciona as mensagens temporizadas enviadas dos elementos não replicados (ou replicados) ordenando, os valores em *buffer*, tendo em conta o valor mais recente, correspondente ao dado com tempo de validade mais antigo, ou seja, o dado que foi validado há mais tempo. Na Figura 7-30 apresenta-se a interface do CFB suas interligações e dependências desenvolvidas para garantir o determinismo das réplicas. Note-se que a indicação, manual (*offline*), de um valor de WCET é fundamental para que a sincronização do sistema não incorra num impasse (*deadlock*) sempre que se verifique a não receção de qualquer um dos dados.

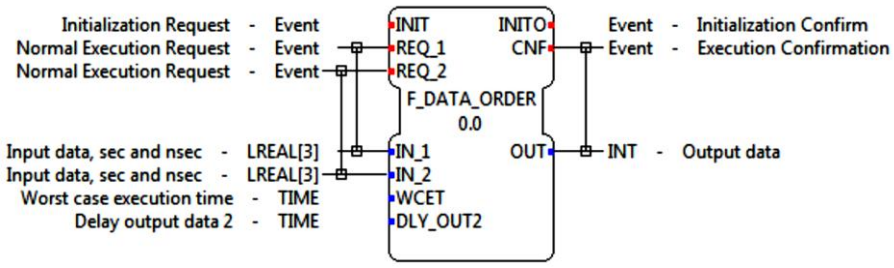


Figura 7-30. Bloco função composto desenvolvido para a obtenção do sincronismo

Este tempo deverá ser calculado tendo em conta não só o tempo máximo de transmissão, mas também os possíveis desfasamentos de sincronização dos relógios locais. A estrutura interna do CFB, suas interligações e rede interna são apresentadas na Figura 7-31.

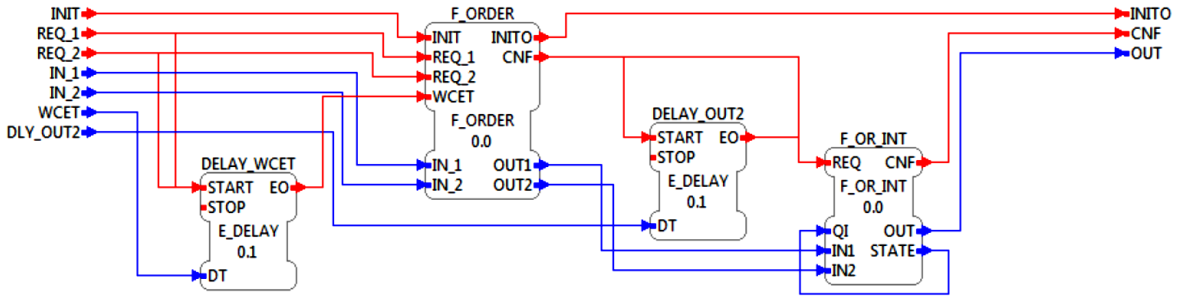


Figura 7-31. Arquitetura do CFB F_DATA_ORDER usado para sincronizar e consolidar os dados

Refira-se ainda que o valor de WCET, introduzido manualmente (em *offline*), encontra-se associado ao evento de entrada DLY_OUT2 que ativará o FB, com atraso na ativação, "DELAY_WCET" que irá gerar um valor inteiro (variável *delay*, *Algoritmo 3*), de indicação da passagem do tempo de espera, fundamental para a execução do SIFB de ordenação. O FB DELAY_WCET é ativado ou reativado pelos eventos independentes REQ, reiniciando o tempo de espera, enquanto o conjunto formado pelo SIFB F_OR_INT (um "OU" de inteiros) e o FB DELAY_OUT2 é responsável pela gestão de dados e, como tal, pelo controlo dos tempos de disponibilidade dos mesmos.

O SIFB "F_ORDER" é a base da sincronização do sistema e, como tal, a interface de escolha do valor com o instante de validade mais antigo. A estrutura da interface do SIFB de ordenação é apresentada na Figura 7-32 e o possível código para a ordenação de mensagens, usando o FORTE, é apresentado no *Algoritmo 3*. Este objeto de reposição é definido como a interface de armazenamento (*buffer*) das mensagens, recebidas nas réplicas. A interface esperará até que a confirmação do serviço seja recebida, evento CNF, ou que tenha passado o tempo definido como WCET.

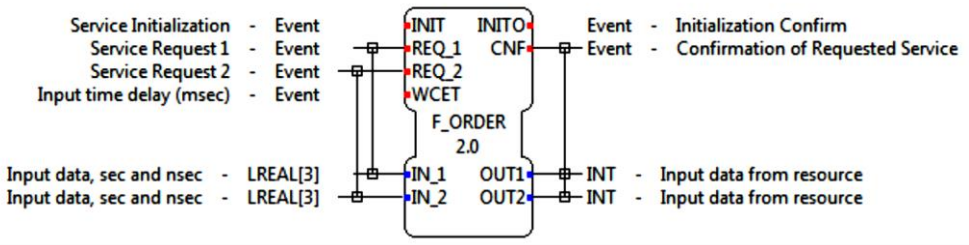


Figura 7-32. Interface do bloco função F_ORDER

O possível código de implementação deste service interface FB F_ORDER foi desenvolvido tendo como objetivo principal a obtenção do determinismo das réplicas é apresentado no *Algoritmo 3*. Este objeto de repositório, SIFB desenvolvido pelo programador, é definido como o elemento de sincronização das réplicas.

A sua estrutura interna é constituída por um *buffer*, dos dados disseminados para a rede, e pelo seu processamento em função do instante em que estes se tornaram disponíveis. A interação do SIFB com os restantes elementos da réplica é realizada através das variáveis de entrada e de saída, variáveis de inicialização do FB. Estas, bem como os seus valores de inicialização, são definidas nas linhas 1 a 5 (*Algoritmo 3*) enquanto as variáveis auxiliares de processamento são definidas nas linhas 7 a 10. A execução deste módulo encontrar-se-á em espera até que qualquer um dos dados propagados ao longo da rede do sistema distribuído seja recebido. Assim, e após ativação de um qualquer evento REQ (*scm_nEventREQ_xID*) o sistema transfere para as variáveis auxiliares os valores associados ao array propagado e recebido no formato [dado, tempo *s*, tempo *ns*]. Neste mesmo instante dá-se a inicialização do WCET através da ativação do FB DELAY_WCET

(ver Figura 7-31) enquanto o evento de entrada WCET, associado a F_ORDER, permanecerá inativo.

Algoritmo 3 – Ordenação de dados

```
1:  void FORTE_F_ORDER::setInitialValues(){
2:          IN_1_Array().fromString("0");
3:          IN_2_Array().fromString("0");
4:          OUT1() = 0;
5:          OUT2() = 0;
6:  }
7:  /*Variável públicas para os dados e para os tempos (tempo)*/
8:  int delay = 0;
9:  double d1, sec1, nsec1;
10: double d2, sec2, nsec2;
11: void FORTE_F_ORDER::executeEvent(int pa_nEIID){
12:         switch (pa_nEIID){
13:            case scm_nEventINITID: //evento de saída de inicialização
14:               sendOutputEvent(scm_nEventINITOID);
15:               break;
16:            case scm_nEventREQ_1ID:
17:               d1 = IN_1()[0], sec1 = IN_1()[1], nsec1 = IN_1()[2];
18:               break;
19:            case scm_nEventREQ_2ID:
20:               d2 = IN_2()[0], sec2 = IN_2()[1], nsec2 = IN_2()[2];
21:               break;
22:            case scm_nEventWCETID:
23:               delay = 1;
24:               break;
25:         } /****** fim da receção dos eventos, fim do SWITCH/CASE ******/
26:         /*Tratamento dados recebidos, ordenação tempo de disponibilidade*/
27:         if (d1 != 0 && d2 != 0 && delay != 0){
28:            if (sec1 < sec2){
29:               OUT1() = d1, OUT2() = d2;
30:               sendOutputEvent(scm_nEventCNFID);
31:            }
32:            if (sec1 > sec2){
33:               OUT1() = d2, OUT2() = d1;
34:               sendOutputEvent(scm_nEventCNFID);
35:            }
36:            if (sec1 = sec2){
37:               if (nsec1 <= nsec2){
38:                  OUT1() = d1, OUT2() = d2;
39:                  sendOutputEvent(scm_nEventCNFID);
40:                  d1 = 0, sec1 = 0, nsec1 = 0, d2 = 0, sec2 = 0, nsec2 = 0;
41:               }
42:               else{
43:                   OUT1() = d2, OUT2() = d1;
44:                  sendOutputEvent(scm_nEventCNFID);
45:                  d1 = 0, sec1 = 0, nsec1 = 0, d2 = 0, sec2 = 0, nsec2 = 0;
46:               }
47:            }
48:            delay = 0;
49:         }
50:         if (d1 != 0 && delay != 0 && d2 == 0) {
51:            OUT1() = d1, OUT2() = d2;
52:            sendOutputEvent(scm_nEventCNFID);
53:            d1 = 0, sec1 = 0, nsec1 = 0, d2 = 0, sec2 = 0, nsec2 = 0;
54:            delay = 0;
55:         }
56:         if (d2 != 0 && delay != 0 && d1 == 0) {
57:            OUT1() = d2, OUT2() = d1;
58:            sendOutputEvent(scm_nEventCNFID);
59:            d1 = 0, sec1 = 0, nsec1 = 0, d2 = 0, sec2 = 0, nsec2 = 0;
60:            delay = 0;
61:         }
62:   } /****** fim do void FORTE_F_SINCHRONIZE ******/
```

O evento WCET será ativado após a passagem do tempo associado a DELAY_WCET transferindo para a variável *delay*, variável associada ao tempo de sincronização, o valor 1 (linha 23), validação da expiração do tempo associado a WCET. A escolha do dado com o instante de maior existência é feita ao longo das linhas 26 a 49, assumindo-se que ambos foram rececionados no intervalo de tempo WCET. Ao longo das linhas 50 a 61 faz-se a eleição de um único dado (d1 ou d2) rececionado, dado que será propagado ao FB subsequente, após expiração do tempo WCET.

Neste sentido, quando o gestor de dados, elemento de *software* que garante a obtenção do determinismo, ler os valores recebidos, trabalha com os valores mais recentes que possuam os instantes de validação mais antigos associados à validação da tarefa. Por outro lado, deve ainda assumir-se o pressuposto de que as mensagens são enviadas apenas localmente dentro de um único processador (Poledna *et al.*, 2000). Na Figura 7-33 apresenta-se o esquema de como o determinismo poderá ser obtido nos componentes replicados em função do tratamento das mensagens temporizadas recebidas e tratadas segundo o conceito do SIFB desenvolvido.

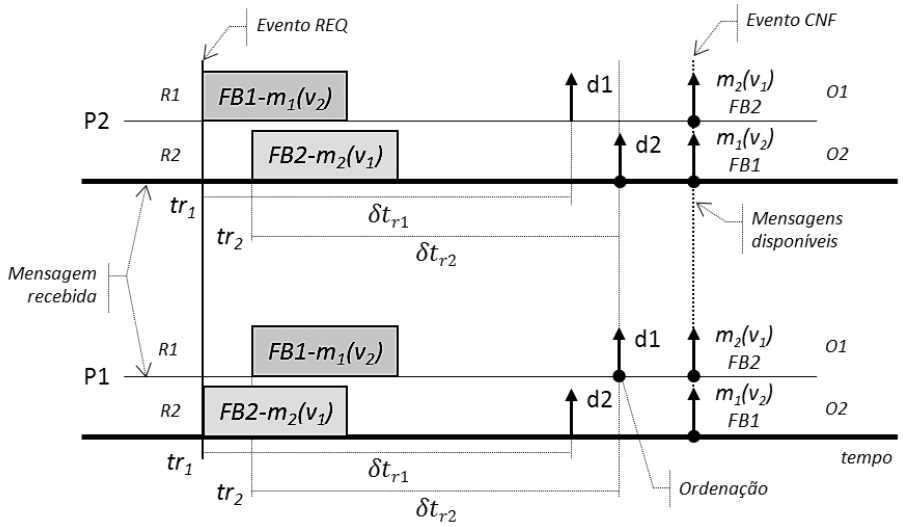


Figura 7-33. Consistência de réplicas, adaptada à IEC 61499 (Timed messages)

A cada um dos FBs é associada uma tarefa pelo que estes enviarão uma mensagem de ordem m_k onde, nesta implementação, o índice *k* se encontra associado ao número do respetivo FB ou tarefa. Por outro lado, teremos de assumir que as mensagens, com o instante de execução associado, chegarão ao seu destino antes do tempo limite predefinido para a execução da mesma (δt), tempo que é ativado após receção da primeira mensagem. E, como tal, as mensagens a disseminar serão constituídas por um conjunto composto pelos dados e pelos instantes de validação identificando-as como $m_k(v_i)$, ou seja, uma mensagem temporizada de ordem *k* disponibilizada no instante de validade v_i. Neste sentido, teremos ainda de assumir que, num sistema de tempo real distribuído e tolerante a falhas, o conjunto de processadores utilizados não serão necessariamente iguais pelo que, cada um deles terá de possuir um relógio local, que se encontrarão, aproximadamente, sincronizados com todos os outros processadores e que não diferem mais do que um tempo ε.

As mensagens enviadas pelos FBs são recebidas e lidas pela ativação dos eventos REQ (*Service Request*, *R1* e *R2*) onde tr_i representa os instantes de recessão nos processos *P1* e *P2*, ou seja, nas réplicas. δtr_i correspondente ao tempo de espera, associado ao evento *i* recebido, para o qual o mesmo se encontrará disponível. Com a chegada de um novo evento reinicia-se o tempo de espera δtr_i, associado ao novo evento, ao fim do qual se procederá à ordenação dos eventos de acordo com o instante de validação v_i. A mensagem que apresente o valor mais recente e que possua o

instante de validação mais antigo associado à sua validação será alocada a *O1* (*OUT1*) enquanto a segunda será alocada à saída *O2* (*OUT2*). A disponibilidade destes dados será realizada pela ativação do evento CNF (*Service Confirmation*) que os tornará válidos, em simultâneo, para os FBs subsequentes. Note-se que os dados são disponibilizados em simultâneo pelo que caberá ao FB seguinte, integrado na gestão de réplicas, fazer a gestão de leitura dos mesmos, leitura do dado *d1* (OUT1) que após um compasso de espera lerá o dado *d2* (OUT2).

Por outro lado, há ainda que considerar que a cada FB associa-se uma tarefa e como tal, esta será o resultado do processamento do algoritmo associado, uma solicitação, uma tarefa. No entanto, de acordo com o esquema apresentado na Figura 7-33 este princípio não poderá ser cumprido uma vez que o FB associado terá dupla solicitação e, obviamente, teria que possuir duas tarefas. Naturalmente que o FB tratará, internamente, estas duas solicitações como sendo duas tarefas distintas disponibilizando, em conjunto com o evento de confirmação do serviço, uma única tarefa que, para todos os efeitos, é composta por dois dados. Isto é, a execução do FB de gestão de réplicas não permite que seja disponibilizado o dado *d1* e passado *δtr* o dado *d2* o que contraria o conceito das mensagens temporizadas. No entanto, estes poderão ser lidos com um intervalo de tempo $\delta_{libertação}$ gerido pelo suporte de replicação pelo que a libertação dos dados será realizada de acordo com o esquema apresentado na Figura 7-34.

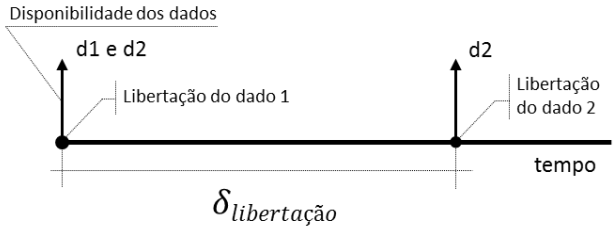


Figura 7-34. Libertação de dados, gestão de réplicas

O tempo $\delta_{libertação}$ será definido pelo pior tempo de execução do CFB F_DATA_ORDER associado ao tempo de execução do F_ADD (ver Figura 7-27). O conjunto destes tempos serão utilizados para definir o tempo de ativação do SIFB F_OR_INT responsável pela libertação do segundo dado (d2). Na Figura 7-35 apresenta-se a estrutura de interface do SIFB de gestão de libertação do dados, após ordenação.

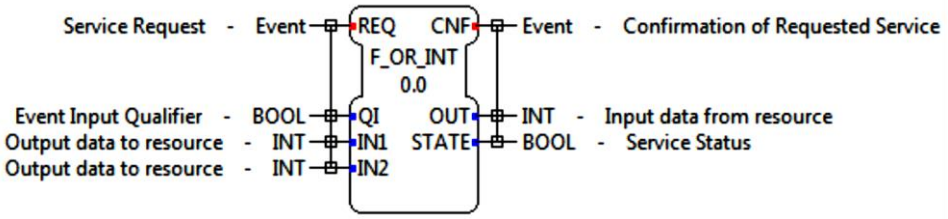


Figura 7-35. Service interface function block F_OR_INT

O código desenvolvido para a gestão de libertação dos dados é apresentado no *Algoritmo 4*. A interação deste módulo com o restante é realizado através das variáveis de entrada definidas nas linhas 2 e 3 (variáveis booleanas). A realização da ação é efetuada por ativação do evento REQ em combinação com o qualificador QI. A essência desta abordagem traduz-se numa ação de comutação dos valores de entrada resultantes das combinações dos estados das variáveis booleanas QI e STATE. Esta comutação é conseguida pela interligação da saída STATE que a cada ativação altera o valor lógico do qualificador QI (passagem de *false* a *true* e vice-versa). Esta alternância de valores leva a que o algoritmo leia, alternadamente, os valores de *d1* e *d2* presentes, em simultâneo, nas entradas IN1 e IN2, respetivamente, ver Figura 7-31.

Algoritmo 4 – Libertação dos dados, alternância de valores

```
1:      void FORTE_F_OR_INT::setInitialValues(){
2:         QI() = false;
3:         STATE() = false;
4:      }
5:      void FORTE_F_OR_INT::executeEvent(int pa_nEIID){
6:         switch(pa_nEIID){
7:         case scm_nEventREQID:
8:            if (QI() == false){
9:               OUT() = IN1();
10:              STATE() = !STATE();
11:              sendOutputEvent(scm_nEventCNFID);
12:           }
13:           if (QI() == true){
14:              OUT() = IN2();
15:              STATE() = !STATE();
16:              sendOutputEvent(scm_nEventCNFID);
17:           }
18:        break;
19:        }
20:     }
```

7.7 Conclusão

Neste capítulo foram apresentadas diversas abordagens à distribuição e à replicação de *software* em sistemas de tempo real tendo como foco, sempre que possível, a utilização dos elementos standards constantes do repositório de objetos do *4diac*. Com efeito analisaram-se três abordagens aos problemas de sincronização de réplicas suportadas pelos SIFB de comunicação *Client/Server* e *Publish/Subscribe* e *Timed Messages*.

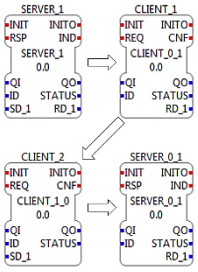
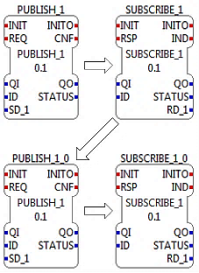
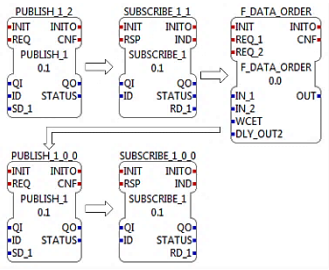
A utilização destes pares de SIFB de comunicação, construídos de acordo com as propostas apresentadas, permitem garantir o sincronismo das réplicas votadoras, a espera de elementos disponibilizados por cada componente replicado, mas poderão não garantir, por si só, o determinismo das réplicas em aplicações IEC 61499. Por este motivo desenvolveu-se um SIFB de sincronização que, associado aos pares *Publish/Subscribe*, asseguram o determinismo das réplicas. Esta associação, combinada com um SIFB de associação temporal aos dados, são a combinação final para a obtenção do sincronismo das réplicas.

Assim, quer para a replicação com recurso aos pares *Client/Server* quer para os *Publish/Subscribe* é possível assegurar-se a sincronização de dados sem se garantir o determinismo. *Client/Server* pode ser utilizado em qualquer uma das abordagens, quer em ações de divergência quer de convergência. No entanto, uma abordagem com esta base conduz ao desenvolvimento de sistemas replicados e distribuídos com tempos de execução mais elevados, passagem do *token* entre *Clients*, onde a aplicação de tempo real apresenta menor eficiência temporal. Por outro lado, uma abordagem *Publish/Subscribe*, a mais utilizada em sistemas distribuídos, apresenta os mesmos comportamentos da implementação anterior, mas com significativos ganhos de eficiência temporal. Esta abordagem não necessita de tempos de espera para envio de dados uma vez que os dados podem ser disseminados simultaneamente segundo um protocolo *multicast* UDP/IP.

A terceira abordagem (*Timed messages*), suportada por pares *Publish/Subscribe* associados a SIFB de sincronização das réplicas garantem o determinismo das mesmas podendo ser utilizados, igualmente em todos os cenários apresentados na Figura 7-4.

Na Tabela 7.1 apresenta-se uma comparação dos modelos propostos, suas interações e propriedades, vantagens e desvantagens associadas à sua implementação.

Tabela 7.1. Comparação dos modelos de replicação

Arquitetura	Interface
	- 4 interfaces SIFB distintas, ligação do *server* a *client* (réplica) e ligação *client* a *server* (votação); - Desenvolvimento mais demorado, com custos mais elevados, com base na complexidade dos SIFB de comunicação; - Tempos de processamento mais elevados, passagem do *token* entre *clients* e tempo de disseminação dos dados; - Sistema com menor eficiência temporal.
	- 2 interfaces SIFB distintas, ligação *publish* a *subscribe* e vice-versa (ligação a réplicas e ao votador); - Desenvolvimento mais rápido, menores custos, com base na simplicidade dos SIFB de comunicações; - Tempos de processamento reduzidos, tempos de ativação do SIFB e de disseminação dos dados; - Sistema com maior eficiência temporal.
	- Características idênticas ao desenvolvimento de sistemas distribuídos com *publish/subscribe*; - Ligeiro aumento dos tempos de processamento resultantes da obtenção do determinismo.

Capítulo 8

Validação da framework de replicação

8.1 Introdução

A utilização de componentes de baixo custo no desenvolvimento de aplicações distribuídas permite-nos desenvolver sistemas de tempo real tolerantes a falhas com elevada confiabilidade. Estes dispositivos com capacidade para suportar a replicação ativa de *software* são, também eles, um elemento de replicação devido, essencialmente, à sua facilidade de integração e ao seu custo muito reduzido. Por outro lado, com o surgimento da IEC 61499 e de aplicativos que permitiram a sua implementação, é possível desenvolver aplicações, de forma transparente, mantendo-se o foco nos requerimentos de controlo, abstraindo-nos das tarefas de distribuição e de replicação. Neste sentido, a utilização de ferramentas de desenvolvimento como a *framework Eclipse 4diac™*, assumiram um papel preponderante no desenvolvimento e na reconfiguração de aplicações distribuídas. No entanto, a replicação, também ela uma forma de distribuição, deverá ser tratada com mais cuidado, uma vez que a norma bem como as ferramentas de desenvolvimento nada referem ou possuem para a garantir. Assim, caberá ao programador desenvolver os meios que garantam a replicação e o determinismo das mesmas. Este deverá alocar a cada um dos dispositivos os meios necessários para a execução da aplicação garantindo a interação e a integridade da mesma. Os modelos que iremos validar ao longo deste capítulo são apresentados como soluções genéricas de sincronização e de obtenção do determinismo que poderão ser adotadas no desenvolvimento de uma qualquer aplicação IEC 1499. A *framework* tem como base os elementos de *software*, que podem ser replicados, de um repositório de objetos standards e desenvolvidos pelo programador, que permitirão esconder os detalhes da replicação e da distribuição.

8.2 Validação da sincronização com os SIFB *Client/Server*

Já alguns cenários de passagem de mensagens foram abordados quer ao nível das interações quer da sincronização. Por outro lado, do ponto de vista da interação entre componentes (ver item 7.3.2) dos cinco cenários apresentados poderemos dizer que somente dois são efetivamente necessários, *um-para-muitos* (1:N, cenário 2 e 4) e *muitos-para-um* (N:1, cenário 4 e 5). Se num deles é necessário garantir o determinismo e, consequentemente, o sincronismo nos elementos replicados (FB não replicado que interliga-se com FBs replicados) no outro, será necessário, unicamente, garantir a sincronização e consequente consolidação dos dados recebidos (FBs replicados que interligam-se com um FB não replicado).

8.2.1 Cenário de interação, muitos-para-um

Num cenário de interação, entre vários elementos replicados e um não replicado (N:1, cenário 5), a difusão dos dados deverá ser realizada com recurso à difusão *ponto-a-ponto*, a executar por cada uma das réplicas responsáveis pelo envio, Figura 8-1. Os dados recebidos terão que ser consolidados num único dado ou evento pelo que será necessário assegurar o sincronismo das réplicas emissoras. A única cópia do FB recetor (FB6) esperará até que todas as réplicas enviem os dados a consolidar ou, em alternativa, até que se verifique a passagem do tempo de espera predefinido. A consolidação dos dados, ou seja, a escolha ou mesmo a eleição do dado poderá ser realizada maioritariamente. Neste cenário não será necessário a utilização de mensagens temporizadas na difusão dos dados, uma vez que a consolidação será realizada em função dos dados recebidos e não do instante em que estes se tornaram válidos, será realizada em função das tarefas executadas e recebidas.

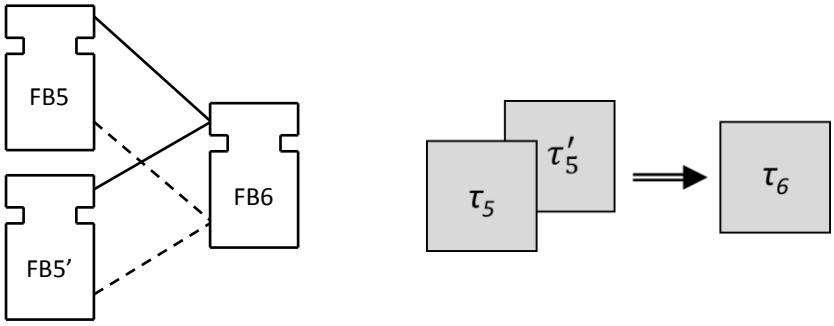


Figura 8-1. Exemplo de interação muitos-para-um (N:1, cenário 5)

Assim, para se obter a sincronização, a norma IEC 61499 disponibiliza-nos uma série de standards de comunicação que poderão, em conjunto com o repositório da *framework* de desenvolvimento, ser utilizadas para esse fim. Neste sentido poderão ser utilizadas várias combinações dos pares de comunicação *Client/Server* onde a sincronização, será obtida com recursos a diferentes interfaces dos SIFBs *Client* e *Server*, isto é, diferente número de entradas (*Data to be Send*) e de saídas (*Received Data*), ver Figura 7-13.

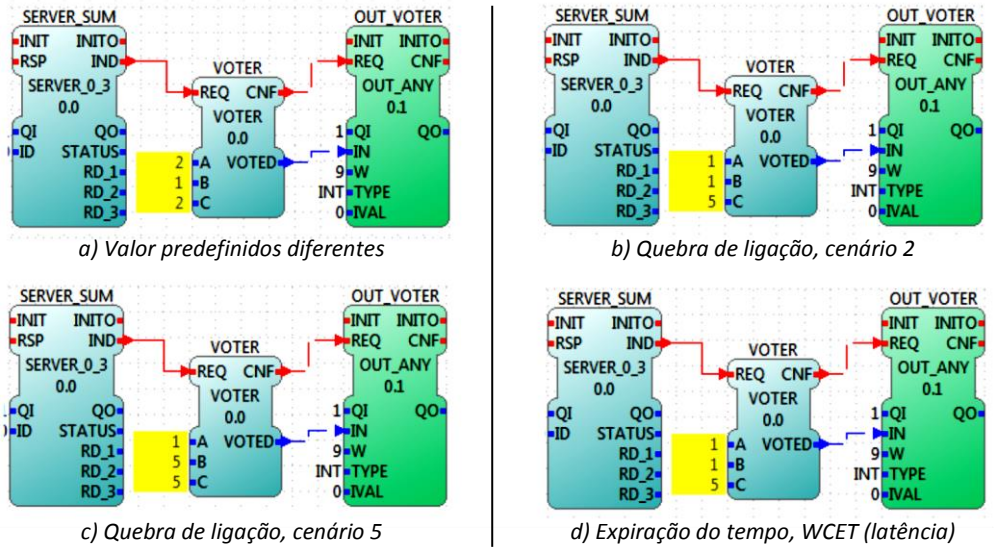


Figura 8-2. Cenários possíveis de falha do sistema replicado, testes de erro

Neste sentido a validação desta abordagem foi realizada com recurso a testes simples, tendo como base o exemplo apresentado na Figura 7-8. Nesta validação utilizaram-se os standards de comunicação SIFB *Client/Server* para a consolidação dos valores disponibilizados pelas réplicas (cenário *muitos-para-um*). O sistema replicado, com todas as instâncias da aplicação, foi executado num único computador. A inicialização das réplicas e respetiva sincronização foi realizada por difusão *multicast* por ação do botão *START*. Esta implementação foi testada e validada com êxito. Para isso, foram realizados diversos cenários de simulação de falhas prováveis, parando-se e iniciando-se cada um dos recursos, alterando-se os valores predefinidos ou por colapso do sistema de comunicação entre recursos e dispositivos (não envio de dados) de acordo com o procedimento seguinte (ver Anexo B):

1. ***Alteração dos valores predefinidos*** *(Resposta errada – Answer).* Alteraram-se os valores de entrada (IN1, ver Figura 7-8) dos FBs F_ADD_1 e F_ADD_3 para INT#1, ou seja, dois inteiros com valor 1 nas entradas IN1 e IN2 de cada um dos FBs. O valor esperado da soma seria 2 o que foi confirmado pelo votador, valor em maioria (Figura 8-2a). Ver o esquema apresentado na Figura 11-1 do Anexo B. Note-se que nesta abordagem a ordenação dos dados é meramente casual.
2. ***Colapso da ligação entre o HMI e o recurso 1*** *(Omission).* Foi quebrada a ligação entre o FB "START_BUTTON" (alocado no dispositivo HMI) e o FB "F_ADD_1" (alocado no dispositivo PC_1). Ao não se ativar o recurso 1 não se verifica o envio dos dados para o votador e, como tal, o FB "WCET" enviará para o votador o valor 5 correspondente à expiração do tempo de processamento. Nesta situação o votador confirmará o valor 1 uma vez que este se encontra em maioria. O valor 5 encontra-se associado à entrada C, embora a falha corresponda ao recurso 1, uma vez que foi o último valor a ser recebido (valores ordenados por ordem de receção). Note-se que o *Server* se limita única e simplesmente a receber os dados independentemente da ordem em que estes são enviados (Figura 8-2b). Ver esquema apresentado na Figura 11-2 do Anexo B.
3. ***Colapso de ligação entre os recursos e o votador (cenário 5)*** *(Crash).* Foi quebrada a ligação entre os recursos 1 e 2 e o FB "VOTER". Neste caso, quando não se ativam os FBs *client* não se verifica o envio de dados para o *server* (votador) e como tal, o FB "WCET" enviará para o votador o valor 5, correspondente à expiração do tempo de processamento, tantas vezes quantas as necessárias. Os valores 5 encontram-se associados às entradas B e C, embora as falhas de comunicação correspondam aos recursos 1 e 2. Note-se que os valores 5 só são enviados para o *server* após a expiração do tempo de processamento das réplicas (Figura 8-2c). Ver o esquema apresentado na Figura 11-3 do Anexo B.
4. ***Expiração do tempo de processamento (WCET)*** *(Timing).* Ocorre devido ao maior tempo de latência da rede ou pelo facto do *server* não rececionar os dados que poderão ter sido enviados pelas réplicas (Figura 8-2d).

A sincronização dos dados recebidos no votador foi testada com sucesso obtendo-se os resultados esperados para os ensaios, cumprindo-se os requisitos preestabelecidos para este modelo de sincronização, o votador elegeu os dados maioritários. Da simulação de colapso de linha de comunicação ou de temporização de envio foi também possível verificar que os dados recebidos encontravam-se em consonância com os valores esperados. Os testes foram realizados respeitando o tempo máximo de receção dos dados e com o disparo de um único evento.

8.3 Validação da sincronização com os SIFB *Publish/Subscribe*

O sistema foi novamente testado, de acordo com esta nova abordagem e o esquema apresentado na Figura 7-20. Assim, sabendo que se utilizaram SIFBs de comunicação

Publish/Subscribe em todas as interações (*um-para-muitos*, cenário 2, e *muitos-para-um*, cenário 5) do sistema replicado, com todas as suas instâncias executadas num único computador, foi também ele testado com êxito. Nesta nova abordagem seguiu-se, igualmente, o procedimento de validação adotado no item 8.2.1 que poderá ser extrapolado para esta nova implementação. Os resultados obtidos encontram-se em concordância com os testes anteriores sendo, também eles, representados pela Figura 8-2.

A sincronização dos dados recebidos foi realizada com sucesso obtendo-se os resultados esperados para os ensaios. Da simulação de colapso de linha de comunicação ou de temporização de envio, foi também possível verificar que os dados recebidos encontravam-se em consonância com os valores esperados. Os testes foram realizados respeitando o tempo máximo de receção dos dados e com o disparo de um único evento.

Esta abordagem apresenta melhorias consideráveis dado que os tempos de desenvolvimento tornam-se menores devido, essencialmente, à uniformização do processo, replicação utilizando unicamente SIFBs de comunicação *Publish/Subscribe*. Por outro lado, verificou-se que, para além dos ganhos de eficiência no tempo e na diminuição do custo de desenvolvimento, os ganhos significativos do sistema são obtidos em relação ao tempo real de funcionamento uma vez que o sistema se tornou temporalmente mais eficiente. Os tempos de controlo do sistema poderão ser mais reduzidos operando, ao nível dos ms, com dois ou mesmo um dígito.

8.4 Configuração do exemplo de aplicação

As telas transportadoras, vulgarmente designados por transportadores (*conveyors*), usados para o transporte de um qualquer produto ou material, são sistemas de transporte onde a replicação se pode tornar uma ferramenta extremamente útil. Um exemplo perfeito para a aplicação da replicação são os sistemas de transporte de bagagens em aeroportos (BHS – *Baggage Handling Systems*). Os BHS, sistemas automatizados de transporte e de manipulação de bagagens desde o local de "*check-in*" até ao local de embarque, são um exemplo clássico de um sistema de automação distribuída complexa e que exige alto desempenho, confiabilidade e flexibilidade.

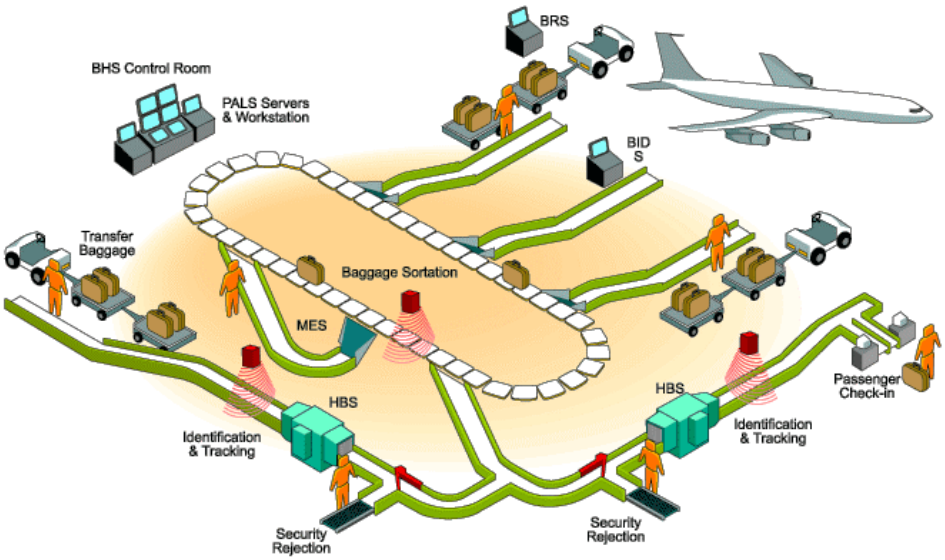


Figura 8-3. Diagrama simplificado de um BHS de um aeroporto (Pteris, 2019)

Do ponto de vista da sua utilização os BHS tornaram-se uma componente importante no processo de transporte, até mesmo desejáveis (Vickers, 1998), dada a sua flexibilidade de reconfiguração e da utilização de módulos de mecatrónica inteligente. Estas características

assumem um papel preponderante no crescimento dos BHS, e por isso, tornam-se vitais para o processamento eficaz da bagagem visto que os aeroportos, de um modo geral, encontram-se constantemente em expansão e adaptação aos espaços existentes. Na Figura 8-3 é apresentado um esquema simplificado de um sistema BHS de um aeroporto.

Os BHS são sistemas modulares controlados, normalmente, ao nível mais baixo dos sistemas, por uma hierarquia centralizada de controladores lógicos programáveis (PLCs) que interagem com sensores, atuadores e dispositivos de controlo embebidos, integrando-se e trabalhando perfeitamente. São constituídos por uma combinação modular de transportadores, sistemas de desvio, raios-X e de diversos *scanners* dispostos de acordo com as necessidades do aeroporto que serão dinamicamente adaptados de acordo com as novas rotas, alterações do equipamento ou devido a avarias. É, pois, devido a esta natureza modular, à sua dispersão e ao elevado número de elementos a controlar que uma abordagem de controlo distribuído fará todo o sentido. A aplicabilidade do modelo de referência IEC 61499 na implementação desta arquitetura de controlo foi já estudada por Black e Vyatkin (2010), que demonstrou a capacidade de descentralização do sistema de controlo, sua reconfiguração e escalabilidade, e por Yan e Vyatkin (2011) que, usando o ISaGRAF como *framework* de implementação, criaram uma rede de teste com mais de 50 nós de controlo, simulando um aeroporto de pequeno porte, provando que a IEC 61499 pode cumprir os níveis necessários de flexibilidade e de distribuição.

Naturalmente que não será viável, neste momento, desenvolver e testar a aplicação num sistema real de manipulação e de transporte de bagagens de um aeroporto (BHS). Neste sentido, a aplicação de controlo distribuído será desenvolvida, numa primeira fase, num ambiente de simulação emulando esse mesmo sistema. Este ambiente de simulação é realizado sobre um conjunto de cinco minicomputadores de baixo custo (Raspberry Pi) que, por sua vez, simularão o envio e a receção de objetos, quer com sistemas lineares quer rotativos. Se por um lado, os transportadores lineares recebem os objetos, movem-nos e finalmente transferem-nos para o próximo *conveyor*, movimentação numa única direção, os rotativos, para além de os transportarem, terão de girar em torno do seu eixo vertical, transportando-os e direcionando-os para dois destinos distintos, divergência. Estes, sistemas poderão ainda ser usados na receção de objetos de distintos procedimentos encaminhando-os para um único destino, convergência.

É então neste *modus operandis* que o problema de receção dos objetos se torna crítico. O sistema de controlo terá de decidir que objeto recebe em primeiro lugar não esquecendo que tem mais um em espera. Assim, se o considerarmos como o elemento crítico do sistema e se replicarmos o seu controlo, mais problemas advirão desta sua replicação. No entanto, correremos o risco de processar informação distinta em cada réplica e, como tal, o sistema não operar ou entrar num processo de vai-e-vem, rotação à esquerda e rotação à direita, com a respetiva perda e/ou deterioração dos objetos. É neste sentido que a introdução do determinismo e da sincronização será a garantia de que as réplicas irão operar com os mesmos dados e na ordem em que estes ficaram disponíveis. Na Figura 8-4 apresenta-se a configuração do exemplo de aplicação que iremos usar como base para a distribuição e para a replicação.

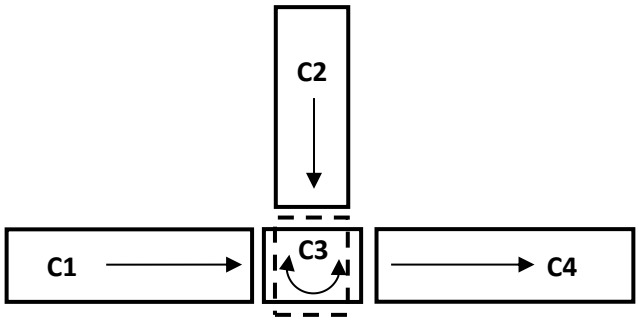


Figura 8-4. Configuração modular da aplicação, quatro conveyors

Analisando a configuração apresentada na figura anterior será fácil perceber que o sistema será constituído por quatro tarefas alocadas a quatro componentes. O *conveyor* 1 (componente C1) e o *conveyor* 2 (componente C2) que alimentam, esporadicamente, o *conveyor* 3 (componente C3). Nesta situação C3 interagirá com C1 e C2, em função dos eventos e dos dados rececionados, e decidirá que objeto receber em primeiro lugar. O *conveyor* 3 passará, posteriormente, o objeto recolhido ao *conveyor* 4 (componente C4). Assim, seremos confrontados com dois possíveis cenários que passaremos a enunciar:

1. ***Transferência de C1 ou C2 para C3***: a transferência de um objeto dos *conveyors* C1 ou C2 para C3. Estes apresentam-se isoladamente pelo que só ocorrerá a receção de um único evento durante o período de tempo esperado. O determinismo cingir-se-á à validação do evento rececionado após ocorrência de δt, ver Figura 7-33.
2. ***Transferência de C1 e C2 para C3***: a transferência de um qualquer objeto presente, no mesmo instante, nos transportadores C1 e C2 para C3 carece de um mecanismo de decisão que defina qual dos *conveyors* enviará, em primeiro lugar, o objeto transportado. Neste cenário, C3 receberá a transferência do componente escolhido permanecendo em *standby* o segundo componente. C3 permanecerá ativo até que os dois eventos sejam processados.

Assim, se depurarmos ainda mais o sistema, poderemos identificar outras tantas tarefas que, encapsuladas em cada um dos componentes, nos permitirão controlar cada um dos *conveyors*. Estas serão constituídas pelos sensores de entrada e de saída, tarefas periódicas de leitura dos valores dos sensores, o controlo do sistema e/ou os atuadores de cada um dos sistemas de transporte.

8.4.1 Arquitetura do sistema transportador

Os sistemas de transporte, bem como os BHS, sendo considerados como unidades físicas modulares centradas em torno de um componente mecatrónico transportador, poderão ser vistos como a representação de uma secção de um sistema mais complexo de transporte. E, como tal, a arquitetura de modelação de cada um destes elementos pode ser desenvolvida considerando-a como uma abordagem orientada a objetos.

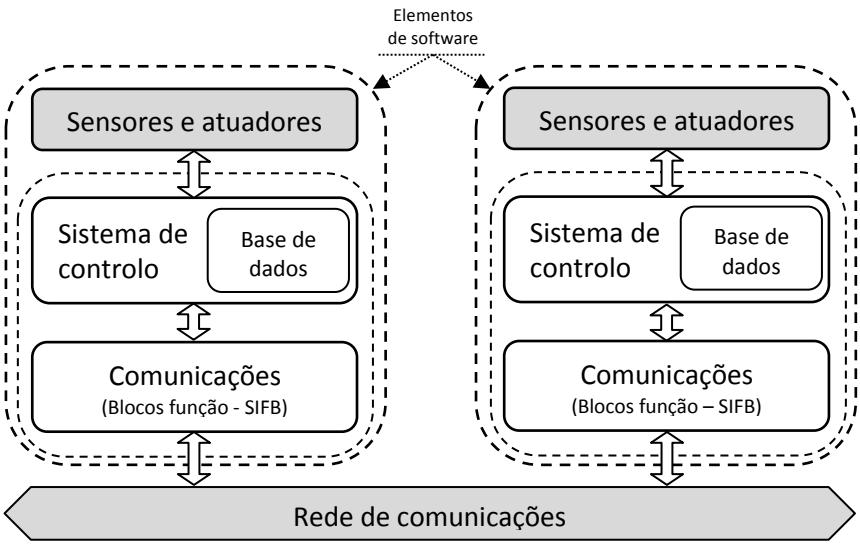


Figura 8-5. Arquitetura interna dos conveyors enquanto elementos de software

Considerando-se ainda que os *conveyors* são constituídos por vários sensores (responsáveis pela deteção das malas ou peças e pelo controlo de velocidade do transportador), por atuadores (responsáveis pela alteração do percurso das malas ou das peças, desvios de direção), por motores (responsáveis pela movimentação) e por um sistema de controlo (responsável pela tomada de decisões com base nos sinais dos sensores e da informação recebida de outros elementos), a modelação do sistema, como um todo, pode ser realizada tendo como base o modelo genérico apresentado na Figura 8-5. Cada um destes módulos interligar-se-ão com o módulo seguinte, topo-a-topo, por divergência ou por convergência de módulos de acordo a construção física do mesmo pelo que, o componente, enquanto elemento de *software*, pode ser reutilizado inúmeras vezes, reconfiguração de um novo *layout*, e, quando necessário, replicado.

Assim sendo, os transportadores adjacentes recebem a informação contida na base de dados, número de malas em trânsito, por exemplo, bem como informação considerada relevante para o correto funcionamento do transportador seguinte. O sistema de controlo interage com os sensores e atuadores do "*conveyor*" em função da informação recebida dos "*conveyors*" adjacentes, enquanto os blocos função de comunicação (SIFBs), estabelecem os requisitos necessários à implementação de um sistema de controlo distribuído.

8.4.2 Exemplo simplificado da aplicação

Neste item será exemplificado, recorrendo à configuração apresentada na Figura 8-4, como o sistema será desenvolvido e distribuído usando-se o Repositório de objetos disponibilizado pela *framework* de desenvolvimento *Eclipse 4diac™* (ver Anexo C). As tarefas que se encontram identificadas na Figura 8-5 não serão consideradas neste desenvolvimento uma vez que se consideram, nesta fase, encapsuladas em cada um dos FBs usados na modelação do sistema. Neste sentido, serão apresentados cada um dos sistemas de transporte (*conveyors*) como um único FB de encapsulamento com uma única tarefa associada. Na Figura 8-6 esquematiza-se a configuração da aplicação, a associação aos componentes com as respetivas tarefas bem como a sua distribuição pelos recurso ou dispositivos.

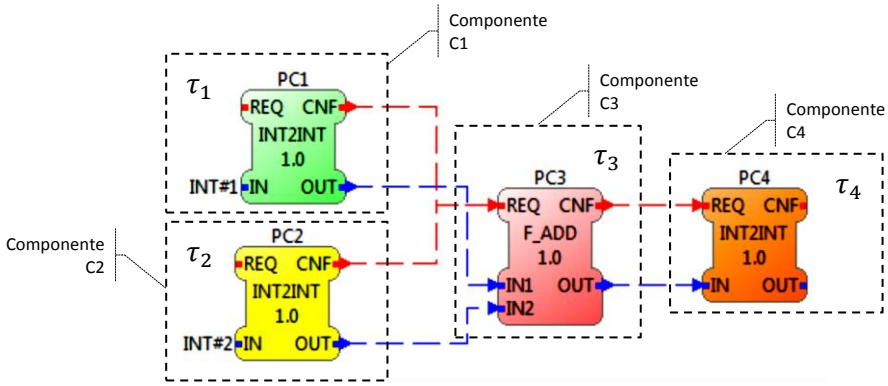


Figura 8-6. Esquematização do sistema distribuído

Note-se que cada um destes componentes representam um único "*conveyor*" e uma única tarefa (FB) que, nesta implementação ao nível do *software*, se cinge à informação de que possui um objeto disponível para entrega. Os componentes C1 e C2 representam os alimentadores do componente C3, componente rotativo, enquanto C4 representa o componente de continuidade.

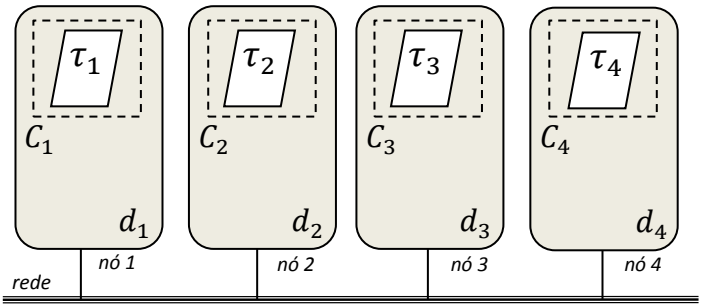


Figura 8-7. Exemplo de distribuição da aplicação e suas interações

Na Figura 8-7 apresenta-se a alocação da aplicação e respetivas tarefas aos diversos nós. Assim, o nó 1 (dispositivo 1) encontra-se configurado com o componente C1, enquanto o nó 2 encontra-se configurado com o componente C2 (dispositivo 2). O nó 3 (dispositivo 3) recebe o componente C3 (a replicar posteriormente) e, finalmente, o dispositivo 4 (nó 4) é configurado com o "*conveyor*" de continuidade, componente C4.

8.5 Implementar a *framework* de replicação com a IEC 61499

No item 8.4.2 foi apresentado um exemplo simplificado de uma aplicação, composta por quatro "*conveyors*", e o modo como os quatro componentes foram distribuídos pelos diversos dispositivos. Naturalmente que esta implementação será novamente utilizada para explicar como o sistema será replicado e redistribuído e como será modificado ao longo do seu desenvolvimento. Assim, consideremos que cada um dos FBs representados na Figura 8-8 são o encapsulamento de toda a arquitetura interna do componente e, como tal, serão replicados e distribuídos, individualmente, para cada um dos dispositivos. Note-se ainda que, os dispositivos usados como suporte do sistema de replicação são considerados de uso corrente e de baixo custo (Raspberry Pi, Anexo D) e executarão os diferentes *runtime* das instâncias de lógica do FORTE. Na Figura 8-8 apresenta-se a configuração final da distribuição/replicação dos quatro componentes utilizados na implementação da *framework* de replicação com a IEC 61499.

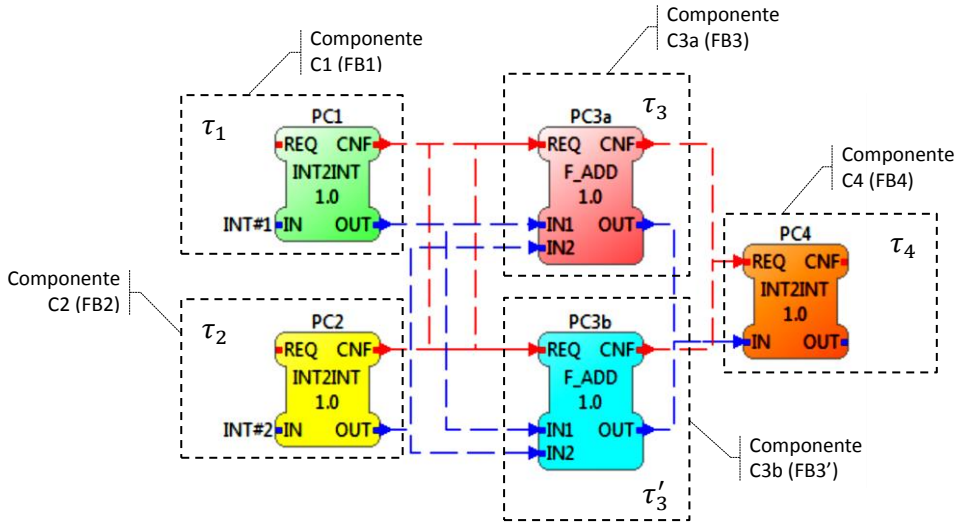
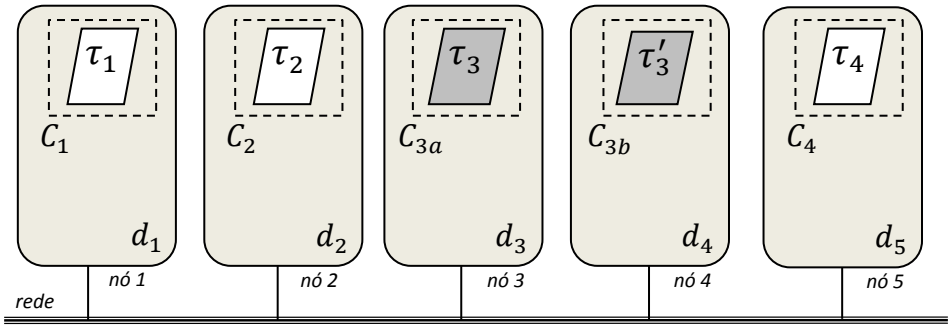


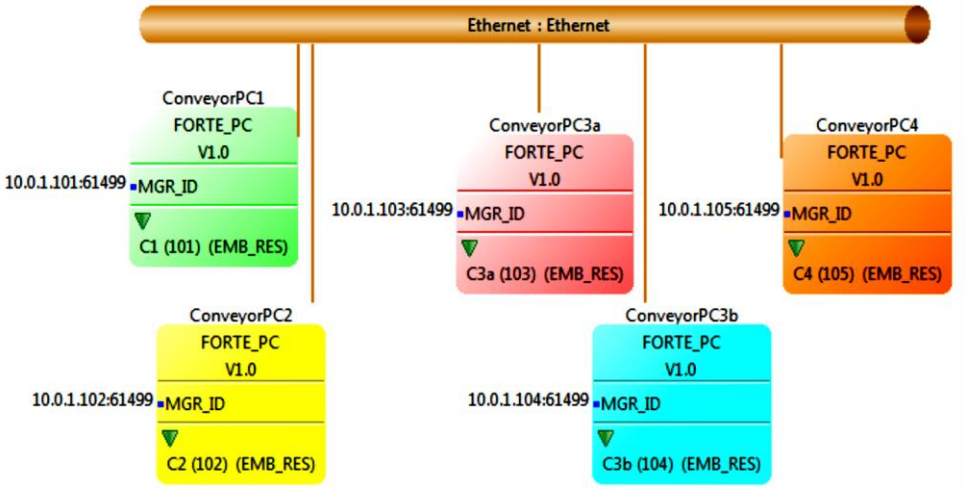
Figura 8-8. Esquematização do sistema replicado, comunicação 1-para-N e N-para-1

A alocação física do sistema distribuído e replicado pelos diversos minicomputadores pode ser traduzida no esquema apresentado na Figura 8-9a. Assim, o nó 1 (dispositivo 1) encontra-se configurado com o componente C1 (FB1), enquanto o nó 2 (dispositivo 2) encontra-se configurado

com o componente C2 (FB2). Os dispositivos 3 e 4 (nós 3 e 4) recebem uma réplica do componente C3, designadas de C3a (FB3) e C3b (FB3'), e, como tal, as réplicas da tarefa τ_3 (τ_3 e τ_3'). Finalmente o nó 5 (dispositivo 5) encontra-se configurado com o componente de continuidade, componente C4 (FB4).



a) Exemplo da replicação dos componentes e suas interações



b) Visualização da implementação do sistema em FORTE

Figura 8-9. Visão da distribuição dos componentes do sistema e da rede de comunicações

O FORTE, enquanto *runtime* de execução da aplicação, terá como função suportar, após compilado em C++, a execução do sistema desenvolvido e por isso, terá que incluir todos os dispositivos, recursos, FBs desenvolvidos e os do repositório, disponibilizados pela iniciativa *Eclipse 4diac* (ver Anexo C).

Neste sentido, no processo de compilação do FORTE será necessário definir os *targets* associados às plataformas de destino que irão receber o executável quer este seja um típico PC, com Windows, quer seja um sistema embebido, com Linux. Deste modo, todo e qualquer elemento de *software* desenvolvido ou alterado pelo utilizador (FBs, SIFBs ou outros) deverá ser implementado em C++ para que este possa ser incorporado numa nova compilação do FORTE, ou seja, uma alteração ou um novo FB implica numa nova compilação. Assim sendo, a aplicação foi distribuída e replicada pelos diversos dispositivos alocando-se a cada um, um único recurso (FB). A cada um destes dispositivos alocaram-se as instâncias do FORTE (FBs que representam os componentes) identificando-os com os respetivos pares *IP:porta* (por exemplo, 10.0.1.101:61499, ConveyorPC1) enquanto, a visualização da evolução do sistema foi realizada com recurso ao ambiente de desenvolvimento 4DIAC-IDE (versão 1.8.4). A aplicação foi distribuída, replicada e implantada fisicamente, tendo em conta a distribuição das diversas instâncias pelos cinco dispositivos, Figura 8-9b.

A cada um destes elementos corresponde uma distribuição física do FORTE pelo que cada um deles correrá diferentes implementações da aplicação. O resultado final da distribuição das instâncias pelos diversos dispositivos é apresentado na Figura 8-10. Os passos necessários à implementação da aplicação são apresentados no Anexo E.

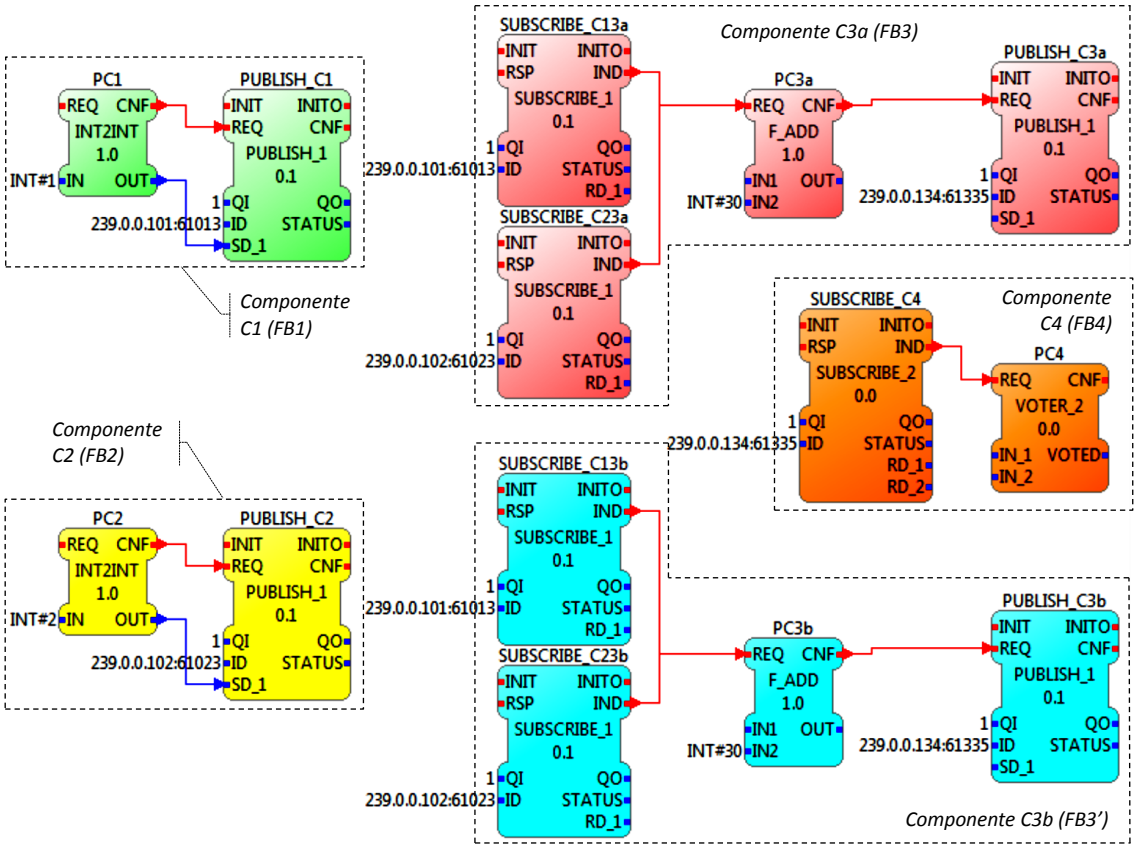


Figura 8-10. Esquematização das comunicações Publish/Subscribe da aplicação replicada

8.5.1 *Infraestrutura física de replicação*

A implementação física das instâncias replicadas e não replicadas foi realizada com recursos a componentes de baixo custo com capacidade de execução do FORTE e do FBDK. Neste sentido, o exemplo de aplicação desenvolvido, foi distribuído pelos cinco dispositivos, designados de ConveyorPC1 (RPi1) a ConveyorPC4 (RPi5).

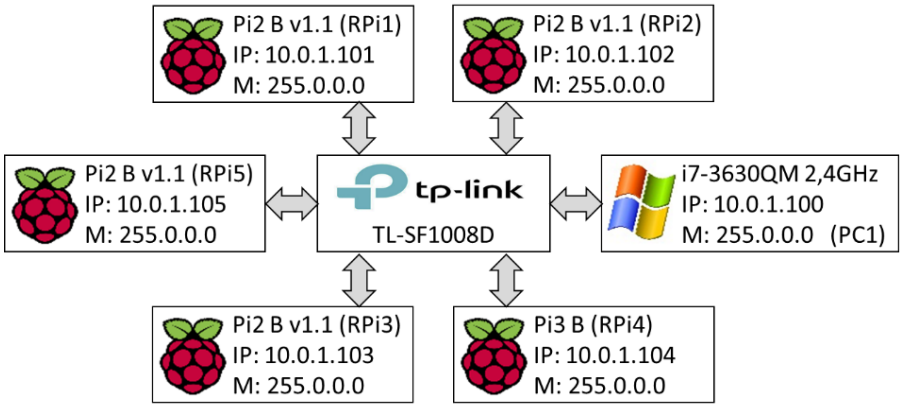


Figura 8-11. Interligação dos equipamentos utilizados na rede da aplicação desenvolvida

O PC1 funciona como suporte ao desenvolvimento e monitorização do sistema, enquanto as restantes instâncias encontram-se distribuídas pelos restantes dispositivos, minicomputadores do tipo Raspberry Pi, interligados por um *switch* de 8 portas, de acordo com o esquema apresentado na Figura 8-9. Na Figura 8-11 apresenta-se o diagrama da rede local utilizada na implementação da aplicação bem como os IPs e *netmask* dos referidos dispositivos.

Naturalmente que este processo de distribuição não se limitará unicamente à interligação dos diferentes dispositivos a uma rede local ou de Internet. Assim, para além de ter que se selecionar o *hardware* que se pretende associar à rede, há que ter em conta que estas plataformas possam executar o FORTE ou que, de alguma maneira, se possa aceder aos endereços de entrada e saída dos mesmos. Desta lista de possíveis dispositivos que correm o FORTE pode-se salientar o Raspberry Pi (utilizado nesta tese), o BeagleBone Black, ou um outro qualquer dispositivo com Linux ou mesmo um qualquer computador alocado à rede. Note-se ainda que a rede que constitui a aplicação que agora se desenvolve é constituída por sistemas operativos distintos que respeitam o standard POSIX e o standard WIN32 (ver Anexo F) e, como tal, o FORTE a executar em cada um destes dispositivos terá que respeitar estes standards.

Assim sendo, partiu-se de uma arquitetura desenvolvida com base na utilização de componentes de uso genérico e de baixo custo, desenvolveu-se um mecanismo de passagem de mensagens temporizadas que possa garantir o determinismo dos componentes replicados bem como a execução, por parte de todos os participantes, do mesmo conjunto de dados e na mesma ordem. Para isso foram desenvolvidos um par de FBs capazes de garantir, com base nos SIFBs de comunicação definidos pela norma e disponibilizados pelo *Eclipse 4diac™*, a comunicação *multicast* bem como assegurar o determinismo das réplicas. Estes novos SIFBs, trabalhando com mensagens temporizadas, visam garantir que todas as réplicas processam os mesmos dados, na mesma ordem e segundo o instante em que foram disponibilizados.

Usando como ambiente de desenvolvimento o 4DIAC-IDE da plataforma *Eclipse 4diac™* implementaram-se e distribuíram-se os FBs pelos dispositivos remotos, conjuntamente com uma cópia da compilação do FORTE, ver o esquema de distribuição apresentado na Figura 8-12. Como interface com o utilizador foi utilizado, o ambiente Windows (correndo uma compilação WIN32 do FORTE e monitorização HMI realizada com recurso ao repositório de objetos FBRT) enquanto o FORTE utilizado para executar os FBs que contêm a lógica principal da aplicação, INT2INT, F_AAD e a propagação dos valores, foram alocados aos elementos remotos (RPis).

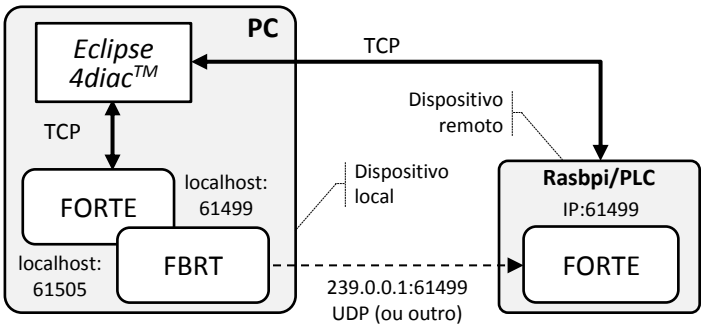


Figura 8-12. Arquitetura da implementação de aplicações remotas (adaptado de: 4diac, 2019c)

Na Figura 8-13 apresenta-se a estrutura base do exemplo de aplicação, utilizado nesta primeira abordagem. Cada uma das tarefas encontra-se distribuída, como um elemento de *software* único (FB), por cada um dos diversos equipamentos (RPis). As tarefas τ_1 e τ_2 foram alocadas aos RPi1 e RPi2 (componentes C1 e C2 localizados nos nós 1 e 2 representando os dois "*conveyors"* de alimentação), enquanto a tarefa τ_3 alocou-se ao RPi3 (componente C3, nó 3). Por outro lado, e uma vez que a tarefa τ_3 foi considerada associada ao elemento crítico do sistema,

procedeu-se a replicação completa, não só do elemento de *software,* mas também do *hardware* (réplica completa do respetivo dispositivo). Assim, as réplicas, cópia integral do *software*, serão executadas em diferente *hardware* produzindo as tarefas τ_3 e τ'_3. O componente C3, de acordo com a sua replicação, passará a ser designado de C3a e C3b, respetivamente (alocação aos RPis 3 e 4). Esta estrutura implementar-se-á de acordo com o esquema apresentado na Figura 8-13.

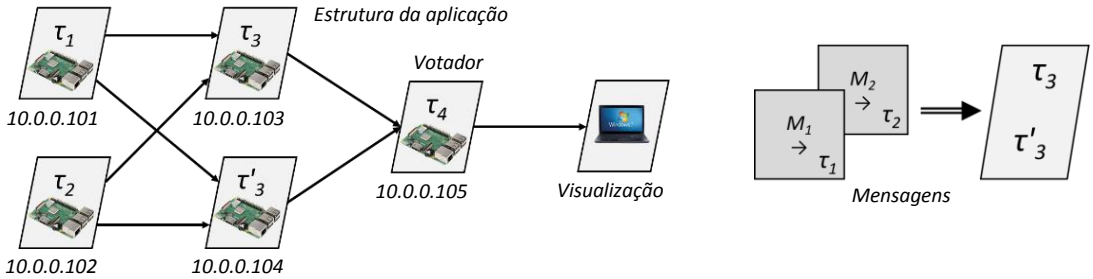


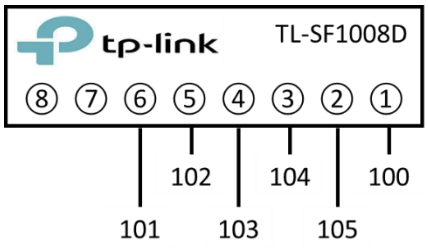
Figura 8-13. Representação da estrutura física de replicação

8.5.2 Detalhe do equipamento utilizado

Naturalmente que antes de se mostrarem e analisarem resultados há que especificar as características dos equipamentos utilizados na aplicação. Na Tabela 8.1 mostra-se a informação pormenorizada e todas as características dos elementos de controlo (PCs) bem como do(s) elemento(s) de interligação. No esquema apresentado na Figura 8-14 mostra-se o posicionamento e as respetivas interligações do *switch* com os diversos dispositivos de computação.

Tabela 8.1. Características do equipamento utilizado na aplicação

Equipamento	Características	
RPi1 (10.0.1.101) **RPi2 (10.0.1.102)** **RPi3 (10.0.1.103)**	Software: Kernel: CPU: Model:	*Raspbian GNU/Linux 9.4 (stretch)* *4.14.50-v7+ #1122 (2018)* *ARMv7 rev 5 (v7l), 900 MHz, Cortex-A7, 32 bits* *RPi 2 B Rev 1.1*
RPi4 (10.0.1.104)	Software: Kernel: CPU: Model:	*Raspbian GNU/Linux 8.0 (jessie)* *4.4.34-v7+ #930 (2016)* *ARMv7 rev 4 (v7l), 1.2GHz, Cortex-A53, 64 bits* *RPi 3 B Rev 1.2*
RPi5 (10.0.1.105)	Software: Kernel: CPU: Model:	*Raspbian GNU/Linux 9.9 (stretch)* *4.19.42-v7+ #1219 (2019)* *ARMv7 rev 5 (v7l), 900MHz, Cortex-A7, 32 bits* *RPi 2 B Rev 1.1*
PC1 (10.0.1.100)	Software: CPU: Model:	*Windows 7 Professional* *Service Pack 1* *Intel Core i7-3630QM CPU 2.4GHz, 64 bits* *ASUS K55V Series*
Switch **8 portas**	Protocols: Data Rates: Buffer size:	*IEEE 802.3, IEEE 802.3u, IEEE 802.3x, CSMA/CD* *10/100 Mbps em half duplex* *20/200 Mbps em full duplex* *2Mb*



Portas:

1 – PC1 ip 10.0.1.100
2 – RPi2 ip 10.0.1.105
3 – RPi3 ip 10.0.1.104
4 – RPi2 ip 10.0.1.103
5 – RPi2 ip 10.0.1.102
6 – RPi2 ip 10.0.1.101

Figura 8-14. Esquema de ligação do sistema ao switch (experiência 1)

8.6 Validação da sincronização com *Timed Messages*

As mensagens temporizadas (*Timed messages*) são utilizadas para garantir o determinismo dos sistemas tolerantes a falhas e de tempo real. Assim, a associação do instante de disponibilidade permite-nos ordenar os dados recebidos em cada uma das réplicas do sistema pela ordem em que estes se tornaram válidos. Neste contexto, as mensagens difundidas de e para as réplicas deverão ser enviadas usando um protocolo de disseminação do tipo *multicast*. Esta condição leva-nos para a utilização de pares de comunicação *Publish/Subscribe*. Assim sendo, pela análise do exemplo de aplicação proposto (ver Figura 8-8), será fácil enquadrar, o envio de mensagens, no cenário de interação 2 (1:N). As réplicas recebem dados de FBs não replicados que terão de ser sincronizados. Esta sincronização e consequente determinismo serão obtidos em função dos SIFBs desenvolvidos pelo programador que garantem a ordenação, a espera e a disponibilidade dos dados no tempo desejado. Este será o ponto basilar do determinismo uma vez que a IEC 61499 nada especifica quanto à sua implementação.

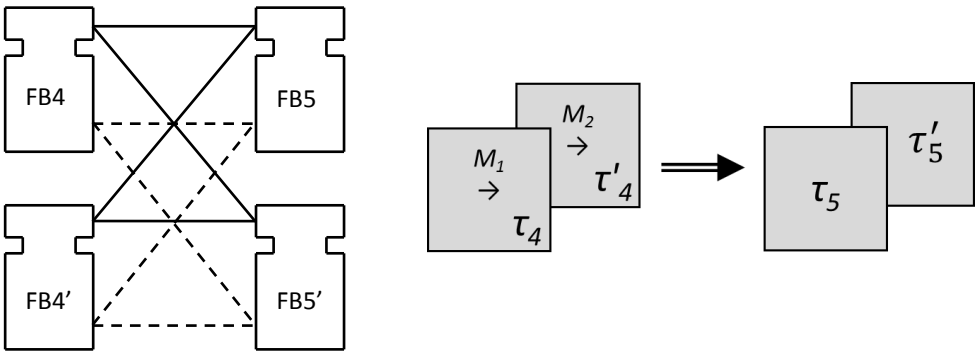


Figura 8-15. Exemplo de interação muitos-para-muitos (1:N)

Por outro lado, será fácil compreender que a associação de componentes representados na Figura 8-13 conduz-nos necessariamente à utilização de comunicações suportadas pelo protocolo *multicast,* uma vez que a comunicação de dados assenta, essencialmente, numa base de transmissão de *um -para-muitos* (tarefas τ_1 e τ_2, FB1 e FB2, a comunicarem com as tarefas τ_3 e τ_3', FB3 e FB3', e estas com τ_4, FB4). Na obstante desta necessidade, deveremos ainda ter em conta que esta estratégia permitirá, também, o uso mais eficiente da largura de banda disponível na infraestrutura uma vez que, os pacotes serão replicados, somente, para os *hosts* associados. Neste sentido, será preciso informar o *kernel*, de cada um dos *host*, a que grupo *multicast* se encontram associados (IP_ADD_MEMBERSHIP). Para isso, foi necessário criar um *socket* que associe todos os IPs das placas de interface de rede (NIC – *Network Interface Card*) pertencentes a esta rede privada. Este *socket* permitirá realizar a associação da rede privada 10.0.0.0 (rede em que todos os dispositivos se encontram interligados) a um *router multicast* com ip 224.0.0.0, ou seja, garantir a associação de cada um dos respetivos IPs ao grupo, tornando-os membros desse mesmo grupo. O roteamento dos IPs foi realizado usando a informação disponibilizada no Anexo G, alínea a). Para isso desenvolveu-se um pequeno *Script* (em *Bash*) que será executado durante

a fase de inicialização de cada um dos Raspberry Pi chamado de *4diac_st.sh*. O *script* de inicialização apresentado na Figura 11-34 (Anexo G, alínea b) associa a interface *eth0* ao *route* especificado garantindo-se a comunicação em *multicast* entre os diversos dispositivos. Os ips dos Raspberry ficam associados ao *router multicast* definido.

Por outro lado, e dado que o sistema tem de encontrar-se sincronizado foi ainda necessário implementar um servidor de tempo (NTP) utilizado como protocolo de sincronização de relógios dos computadores. Após a instalação deste, verificou-se que a sincronização do nosso servidor com o servidor de referência apresentava, há altura, um *reach* de 377 (indicação de que não houveram falhas nas 8 últimas consultas) e um *offset* (deslocamento) de -2.398 milissegundos (diferença de tempo entre o nosso servidor e o de referência), ver exemplo da instalação no Anexo A. Ao *Script 4diac_st.sh*, referido anteriormente, adicionaram-se as linhas necessárias à inicialização do *Servidor* e dos *Clientes* NTP, Figura 11-35 (Anexo G, alínea c). Ressalve-se que a diferença entre servidor e cliente NTP reside, essencialmente, na configuração dos mesmos e, como tal, na configuração do ficheiro */etc/ntp.conf* (ver Anexo A).

A validação física do sistema distribuído passaria então pelo teste da estrutura de modo a garantir não só a distribuição dos componentes replicados C3a e C3b (alocados aos dispositivos com o ip 10.0.1.103 e 10.0.1.104, respetivamente) mas também, dos restantes componentes, ver Figura 8-9. Para isso procedeu-se à interligação física dos diversos equipamentos com recurso ao *switch* de 8 portas que assegura as ligações dos cinco RPis, sistema operativo *Debian* (Linux), e do PC operando com o Windows 7 Profissional (monitorização do processo).

8.6.1 Validar o determinismo dos SIFBs de comunicação Publish/Subscribe

Como já foi referido em itens anteriores o *4diac*, de acordo com a norma IEC 61499, disponibiliza uma série de ferramentas de comunicação fundamentais para a interligação com os módulos distribuídos. Estes SIFB de comunicação asseguram não só a comunicação entre os diferentes nós, mas também, a comunicação em *multicast* ou *broadcast* entre os diversos elementos do sistema distribuído. Estas caraterísticas inerentes aos pares de comunicação *Publish/Subscribe* e *Client/Server*, são sem dúvida um grande passo para a realização de sistemas distribuídos, libertando-nos da complicada tarefa de desenvolver códigos, mais ou menos extensos, para a obtenção do *multicast* ou do *broadcast*. Por outro lado, no que se refere ao determinismo dos sistemas distribuídos, quer o *4diac* quer a IEC, nada indica que, aquando do desenvolvimento de sistemas replicados e distribuídos, haja garantia de este ser obtido. Assim, e com o intuito de validar o possível determinismo da *framework* desenvolvida, optamos por construir um modelo de teste de replicação que nos possa elucidar sobre a hipótese de ser obtido o determinismo com os FBs de comunicação standards disponibilizados pela aplicação de desenvolvimento, pares *Publish/Subscribe*.

Neste sentido foi desenvolvida a arquitetura de replicação apresentada na Figura 8-16 que visa ilustrar parte do sistema físico distribuído. A interligação entre os diferentes dispositivos é realizada tendo por base uma série de IPs *multicast* e de portas distintas. Os endereços dos IPs usados foram definidos dentro da gama 239.0.0.XXX traduzindo, tanto quanto possível, as diversas interligações entre dispositivos. Assim, e a título de exemplo, explana-se o ID 239.0.0.101:61013 onde 239 se associa ao intervalo *multicast* (224.0.0.0 a 239.255.255.255), 101 ao ip do RPi (10.0.1.**101**) e a porta 61013 à correspondente ligação entre o componente 1 (C1 – FB1) e as réplicas do componente 3 (recurso a verde (componente C1) aos recursos rosa e ciam, componentes C3a (FB3) e C3b (FB3') replicados).

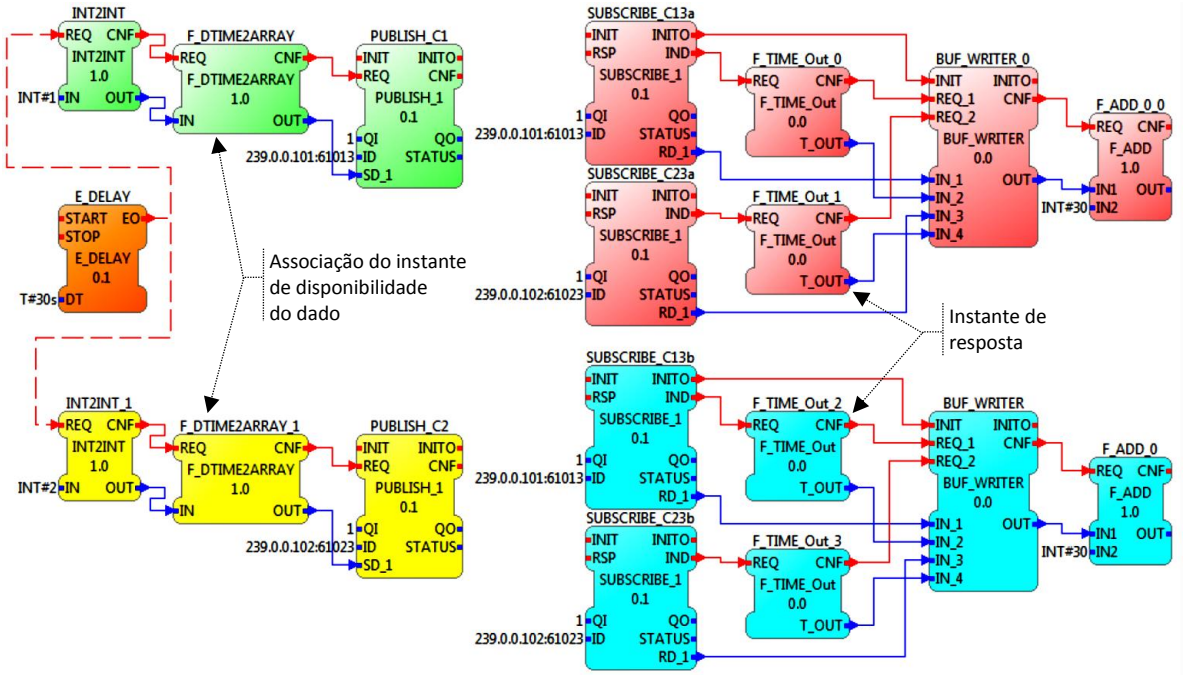


Figura 8-16. Validação do determinismo dos SIFB de comunicação Publish/Subscribe

Analisando a figura anterior será fácil perceber que, para além dos SIFBs de comunicação (*Publish/Subscribe*), dois novos FBs (F_TIME_Out e BUF_WRITTER) foram adicionados à arquitetura base, apresentada na Figura 8-10. Neste sentido e para se entender o processo de criação de mensagens temporizadas tornou-se visível o posicionamento do FB F_DTIME2ARRAY responsável pela adição do instante de validade dos dados provenientes de FBs INT2INT, introdução dos dados de valor 1 ou 2 simulando a ativação dos componentes 1 e 2, respetivamente.

O objeto, F_TIME_Out, é definido como a interface de agregação dos tempos de receção/disponibilidade dos dados às réplicas, pelo que o instante de disponibilidade do evento é determinado após a receção do mesmo (imediatamente após a disponibilidade do dado pelo *subscribe*). Note-se ainda que no nó recetor, o evento só será temporizado aquando da validação do serviço solicitado (evento IND) e após a ativação do evento REQ, ou seja, na resposta da réplica. Na Figura 8-17 apresenta-se a interface do FB F_TIME_Out desenvolvido para a definição dos instantes de execução de um evento {*tsec, tnsec*} definidos por dois reais.

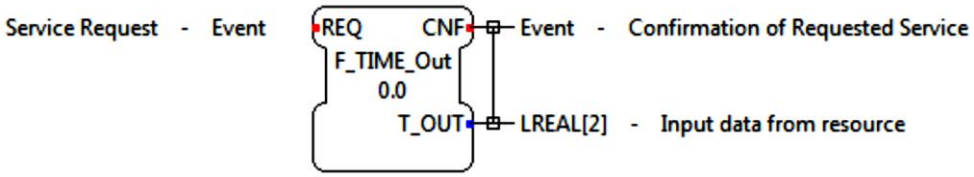


Figura 8-17. Interface do FB F_TIME_Out [sec, nsec]

Parte do possível código de implementação dos tempos de receção/disponibilidade a usar no FORTE é apresentado no *Algoritmo 5*. Este elemento de *software*, Service Interface Function Block, bem como todos os definidos na *framework* de desenvolvimento *4diac*, poderão ser utilizados em novas aplicações e, como tal, na reconfiguração do sistema ou de novos sistemas. Ressalva-se que os módulos desenvolvidos bem como a compilação do FORTE foram realizados tendo em conta um ambiente de trabalho suportado pelo sistema operativo *Debian* Linux, ou seja, OS *Raspbian*.

Algoritmo 5 – Tempo de disponibilidade

```
1:   void FORTE_F_TIME_Out::setInitialValues(){
2:      T_OUT_Array().fromString("0"); }

3:   struct timespec tsr;

4:   void FORTE_F_TIME_Out::executeEvent(int pa_nEIID){
5:      switch(pa_nEIID){
6:      case scm_nEventREQID:
7:         clock_gettime(CLOCK_REALTIME, &tsr);
8:         T_OUT()[0] = tsr.tv_sec;
9:         T_OUT()[1] = tsr.tv_nsec;
10:        sendOutputEvent(scm_nEventCNFID);
11:     break;
12:     }
13:  }
```

O SIFB BUF_WRITTER foi desenvolvido única e simplesmente para criar um *buffer* de receção de dados provenientes dos nós C1 e C2 (valor 1 ou 2) e dos respetivos instantes de disponibilidade dos dados na origem, e de receção nos elementos replicados (componente 3, C3a e C3b). Este, não sendo considerado um elemento fundamental para o determinismo da aplicação, não será tratado exaustivamente, no entanto, é essencial à validação do mesmo. Note-se que a ferramenta *4diac* disponibiliza FBs de registo, com um máximo de 10 variáveis (*type Library/utils, "CVS_WRITTER_10"*), no entanto, a nossa escolha não recaiu na utilização de um qualquer destes módulos predefinidos no *4diac*, mas sim no desenvolvimento de um específico, capaz de efetuar e registar, diferidamente, o *buffer*. Procedimento que não se encontra aplicado aos blocos função WRITTER disponibilizado pela aplicação – registo de dados feito a cada ativação do evento CNF.

Assim sendo, como base de análise do comportamento determinístico do sistema, foi utilizada, numa primeira abordagem, a estrutura apresentada no esquema seguinte, Figura 8-18. É com base nesta estrutura, constituída pelos cinco RPis (dispositivos ,C1 a C4, com cinco tarefas associadas τ_1 a τ_4), que se definiram os tempos de ciclo de disponibilidade dos eventos, se caracterizaram as mensagens temporizadas a disseminar pela rede, se definiu a replicação de *hardware* e de *software*, se concebeu a sincronização inicial do sistema, bem como a sua distribuição e interligação à rede Ethernet.

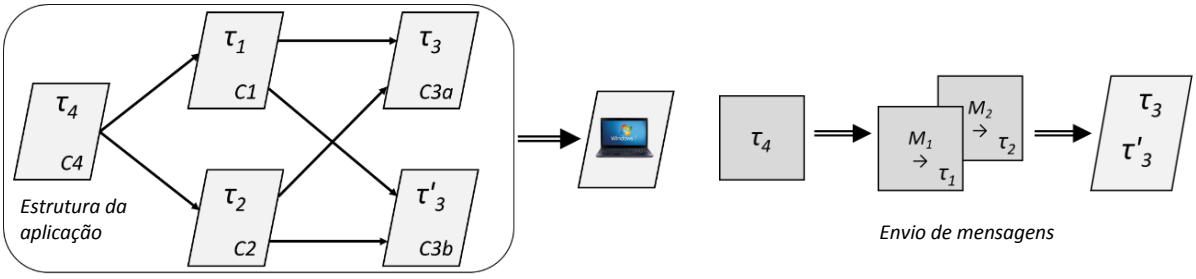


Figura 8-18. Estrutura base do teste

Poderemos então deduzir que os testes de determinismo foram realizados sobre réplicas que se encontram a correr em dois nós distintos (nós 3 e 4, C3a e C3b, portas 3 e 4) e que a disseminação das mensagens é realizada nos nós 1 e 2 (portas 5 e 6) e dirigidas para os nós 3 e 4. O nó 5 (C4, porta 2) funciona como elemento de sincronização, inicial, da aplicação. A monitorização do sistema distribuído é realizada no PC ligado na porta 1 (ver Figura 8-14).

8.7 Validação numérica da comunicação *Publish/Subscribe*

Para clarificar-se o possível determinismo da aplicação, recorrendo unicamente a elementos standard da *framework 4diac*, pares de comunicação *Publish/Subscribe*, foram lançados na rede 10.000 eventos com intervalos de frequência de disponibilidade de 1Hz e de 0,5Hz, isto é, com intervalos de 1000 ms e 500 ms. Para isso a aplicação foi dividida em dois cenários distintos de transmissão de eventos e de dados. Assim sendo, a aplicação é composta pelos componentes C1 e C2 (não replicados, com as tarefas τ_1 e τ_2, FB1 e FB2) que comunicam, individualmente, com os componentes C3a e C3b (replicados, compostos pelas tarefas τ_3 e τ_3', FB3 e FB3'), comunicação *1-para-N*. Pelo componente C4 (não replicado, composto pela tarefa τ_4, FB0), utilizado na sincronização inicial dos componentes C1 e C2, comunicação *1-para-N*, e pelo elemento "Windows", considerado como o sexto componente da aplicação, usado na interação com o utilizador (HMI), ou seja, na monitorização do sistema, ver esquema apresentado na Figura 8-18.

8.7.1 Testes ao fluxo de mensagens sem tráfego adicional

Na arquitetura apresentada Figura 8-16 mostra-se a distribuição das tarefas pelos diferentes nós da aplicação. No entanto, esta esquematização não é suficiente para que o processo de geração de eventos possa ser compreendido. Neste sentido, deveríamos tornar visíveis todos os elementos de *software* que se encontram encapsulados em cada um dos recursos, C1 (101, FB1) e C2 (102, FB2), associados aos dispositivos "*ConveyorPC1*" e "*ConveyorPC2*", por exemplo, ver Figura 8-9b). Por outro lado, deveríamos, também, analisar todos os recursos alocados a cada um dos dispositivos, no entanto, explanar-se-á, unicamente, um dos dispositivos sincronizados pelo recurso C4 (105, FB0) uma vez que, de algum modo, os restantes elementos encontram-se visíveis na arquitetura apresentada na Figura 8-16.

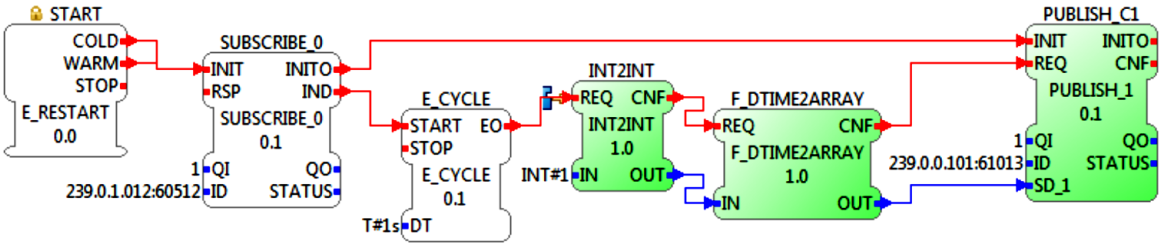


Figura 8-19. Estrutura do recurso C1 (101), dispositivo "ConveyorPC1"

Na estrutura do recurso C1 (101), apresentado na Figura 8-19, são visíveis todos os FBs que constituem este elemento de *software*. Assim, para além do FB inerente a cada recurso, FB de ativação START, responsável pela auto inicialização do recurso, adicionaram-se o FB de comunicação SUBSCRIBE_0, responsável pela comunicação *multicast* (ID: 239.0.0.012:60512) entre o recurso C4 (105) e os recursos C1 (101) e C2 (102), e o bloco função E_CYCLE, responsável pela frequência de lançamento dos eventos, ou seja, os períodos de disponibilidade do evento.

O FB E_CYCLE, sincronizado com o recurso C2 (102), dispositivo *"ConveyorPC2"*, envia para o INT2INT, FB subsequente, eventos periódicos com um intervalo de 1s e/ou 0,5s (1Hz e/ou 0,5Hz). Estes disparos, do ponto de vista do teste, visam testar o determinismo da aplicação e, como tal, a ordem em que os dados são recebidos nas réplicas (um inteiro de valor 1 é lançado no nó C1 (FB1), supostamente, em simultâneo com um inteiro de valor 2 no nó C2 (FB2), a cada intervalo de tempo). O instante de disponibilidade do dado é-lhe adicionado imediatamente após confirmação do seu processamento, evento CNF, por ativação do evento de entrada REQ de F_DTIME2ARRAY. A latência da rede é definida pela diferença dos instantes de disponibilidade e

de receção do respetivo evento, FBs F_TIME_Out associados a cada uma das réplicas, ver Figura 8-16.

O sistema foi reinicializado e deixado a estabilizar durante algumas horas. A rede encontrava-se livre de tráfego adicional de modo que os pacotes intercruzados pudessem ser recebidos e registados, por um *buffer* de 20.000 linhas (10.000 eventos disparados em cada um dos componentes C1 e C2), alocado às respetivas réplicas. Dos eventos registados, em cada uma das réplicas C3a (103) e C3b (104), constam os dados recebidos (inteiro correspondente ao envio de C1, valor 1, e de C2, valor 2), os instantes, em segundos, de disponibilidade dos dados nos respetivos nós (variável *tsa – timestamp to availability*, *Algoritmo 2*), associando-se ainda o instante em nanossegundos e os instantes de receção em segundos e nanossegundos (variável *tsr – timestamp to receiving*, *Algoritmo 5)*. Esta estrutura é disseminada sob a forma de um vetor constituído por três variáveis do tipo *LREAL* [*dado, sec, nsec*].

Na Tabela 8.2 são apresentadas as características de cada uma das tarefas e sua alocação aos componentes, enquanto na Tabela 8.3 apresentam-se os dados referentes à propagação e respetiva distribuição pelos elementos replicados e não replicados. Os tempos são apresentados em milissegundos.

Tabela 8.2. Características das tarefas executadas

Tarefas	Tipo	Componente	Nó
$\boldsymbol{\tau_1}$	*Periódica*	*C1*	*1*
$\boldsymbol{\tau_2}$	*Periódica*	*C2*	*2*
$\boldsymbol{\tau_3}$	*Periódica*	*C3a*	*3*
$\boldsymbol{\tau_3'}$	*Periódica*	*C3b*	*4*
$\boldsymbol{\tau_4}$	*Esporádica*	*C4*	*5*

Tabela 8.3. Características das mensagens, passagem de eventos sem tráfego adicional

Fluxo	Bytes	Período (*ms*)	De	Para	Protocolo
S1	*1*	-	τ_4	τ_1, τ_2	*Multicast*
S2	*24*	*1000/500*	τ_1	τ_3, τ_3'	*Multicast*
S3	*24*	*1000/500*	τ_2	τ_3, τ_3'	*Multicast*

Note-se que as tarefas, bem como as mensagens, são externas aos nós uma vez que a comunicação é realizada intercomponentes. A mensagem referente à tarefa τ_4 (fluxo S1, FB4) é efetuada segundo um cenário de comunicação *um-para-muitos*. As mensagens, referentes às tarefas τ_1 e τ_2 (fluxos S2 e S3, FB1 e FB2), são efetuadas, também elas, segundo um cenário de comunicação *um-para-muitos*. Estes cenários encontram-se preconizados na IEC 61499 pelo que, nestes casos, é suficiente utilizar os SIFBs de comunicação disponibilizados pelo *software*, ou seja, os pares *Publish/Subscribe*.

Na Tabela 8.4 apresentam-se os tempos de resposta para cada fluxo de mensagens (rede sem tráfego adicional) usando o protocolo de difusão *multicast*. O WCRT (*Worst-Case Response Time*) representa o pior tempo de resposta para a transmissão de mensagens e o Período representa a frequência com que as mensagens são disponibilizadas. A Ordenação indica a percentagem de mensagens recebidas na mesma ordem nas duas réplicas. Note-se que o tempo de resposta é

definido como o intervalo entre o instante de disponibilidade da mensagem (*tsa*), no emissor, e o instante de receção da mensagem completa no recetor (*tsr*), ou seja, *tsr-tsa*.

Tabela 8.4. Tempos associados ao fluxo de informação sem tráfego adicional

Fluxo	Período	WCRT (*ms*)		Ordenação da receção
		C3a	C3b	
S1	-	-	-	
S2	*1000*	*6,720*	*3,979*	*93,35%*
S3	*1000*	*6,765*	*3,806*	
S1	-	-	-	
S2	*500*	*6,760*	*6,236*	*97,94%*
S3	*500*	*7,207*	*8,045*	

Da observação da tabela anterior será fácil verificar que o tempo mais elevado de transmissão de mensagens, para um envio periódico de 1000 ms, ocorre com S3 (envio de mensagens temporizadas do componente C2, não replicado, para a réplica C3a, tempo de resposta de 6,765 ms).

Por sua vez, para um envio periódico de 500 ms verifica-se que o tempo mais elevado de transmissão de mensagens ocorre, também ele, com S3 (envio de mensagens temporizadas do componente C2, não replicado, para a réplica C3b, tempo de resposta de 8,045 ms). É ainda de salientar que, embora a ordenação apresente valores de validação consideravelmente elevados, não se verificou o determinismo das réplicas. Note-se, no entanto, que ao aumentarmos a frequência de disponibilidade das mensagens a disseminar, a percentagem de concordância dos dados, nas réplicas, também aumentou. Na Figura 8-20 apresentam-se os diagramas correspondentes aos tempos de resposta nas réplicas C3a e C3b, piores tempos de resposta.

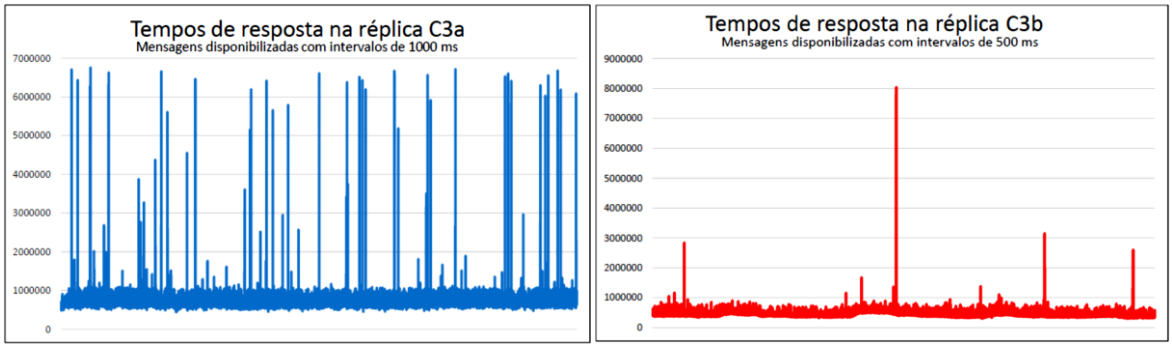


Figura 8-20. Tempos de resposta das réplicas sem tráfego adicional (nsec)

8.7.2 Teste ao fluxo de mensagens com tráfego adicional

Neste segundo teste optamos por introduzir tráfego na rede. Pretendia-se, igualmente, verificar o comportamento determinístico das réplicas e quais as alterações que poderiam advir do lançamento de pacotes extra na rede. A estrutura física não sofreu alterações (manteve-se a posição de interligação dos equipamentos) mas tornou-se ativo o sexto componente, PC1. O tráfego de pacotes extra lançados no sistema foi realizado entre o ip 10.0.1.100 (PC1), utilizando o comando *ping –t* (envio de pacotes de 32 bytes por segundo), e o ip 10.0.1.105 (RPi5), comando *$ sudo ping –f* (envio de 100 ou mais pacotes por segundo, taxa de envio de 56(84) *bytes of data*).

A estrutura de fluxo de tarefas e mensagens ficou agora definida de acordo com o esquema apresentado na Figura 8-21. Esta nova estrutura é constituída pelos mesmos cinco RPis (dispositivos C1 a C4) e seis tarefas, τ_1 a τ_5.

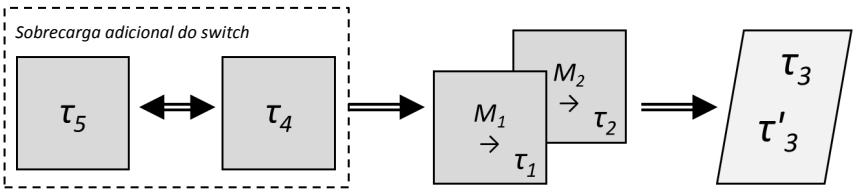


Figura 8-21. Estrutura do teste utilizando tráfego adicional

Na Tabela 8.5 são apresentadas as características de cada uma das tarefas e sua alocação aos componentes, enquanto na Tabela 8.6 apresentam-se os dados referentes à propagação e respetiva distribuição pelos elementos replicados e não replicados. Os tempos são apresentados em milissegundos.

Tabela 8.5. Características das tarefas executadas com tráfego adicional

Tarefas	Tipo	Componente	Nó
$\boldsymbol{\tau_1}$	*Periódica*	*C1*	*1*
$\boldsymbol{\tau_2}$	*Periódica*	*C2*	*2*
$\boldsymbol{\tau_3}$	*Periódica*	*C3a*	*3*
$\boldsymbol{\tau_3'}$	*Periódica*	*C3b*	*4*
$\boldsymbol{\tau_4}$	*Espor./Period.*	*C4*	*5*
$\boldsymbol{\tau_5}$	*Periódica*	*C5*	*6*

Note-se que as tarefas τ_4 e τ_5 (fluxos S1 e S6 e vice-versa) são difundidas segundo um cenário de comunicação *um-para-um*. Este é o cenário base de comunicação da IEC 61499 pelo que é suficiente utilizar os SIFBs de comunicação disponibilizados pelo *software 4diac*, ou seja, os pares *Publish/Subscribe* ou *Client/Server*, quando necessário. Neste teste as tarefas e as mensagens continuam a ser externas aos nós de alocação dos componentes e por isso a comunicação volta a ser realizada intercomponentes. As tarefas τ_4 e τ_5 são usadas como elementos de sobrecarga do *switch* aumentando os tempos de processamento dos *chips* ASIC (*Application Specific Integrated Circuit*) de processamento das *frames*.

Tabela 8.6. Características das mensagens, passagem de eventos com tráfego adicional

Fluxo	Bytes	Período (*ms*)	De	Para	Protocolo
S1	*1*	*-*	τ_4	τ_1, τ_2	*Multicast*
	56(84)	*ping -f*	τ_4	τ_5	*Unicast*
S2	*24*	*1000/500*	τ_1	τ_3, τ_3'	*Multicast*
S3	*24*	*1000/500*	τ_2	τ_3, τ_3'	*Multicast*
S6	*32*	*ping -t*	τ_5	τ_4	*Unicast*

Na Tabela 8.7 apresentam-se os tempos de resposta para cada fluxo de mensagens (rede com tráfego adicional) usando-se os protocolos de difusão *multicast* e *unicast*. São, igualmente, apresentados o WCRT, o Período e a Ordenação, indicando-se a percentagem de mensagens recebidas, na mesma ordem, nas réplicas C3a e C3b. A análise de mensagens trocadas entre os componentes C4 e C5, fluxos S1 e S6, não serão abordadas.

Tabela 8.7. Tempos associados ao fluxo de informação com tráfego adicional

Fluxo	Período	WCRT (*ms*)		Ordenação da receção
		C3a	C3b	
S1	-	-	-	
S2	*1000*	*6,527*	*2,602*	*94,33%*
S3	*1000*	*6,464*	*2,942*	
S1	-	-	-	
S2	*500*	*5,843*	*2,457*	*93,43%*
S3	*500*	*6,248*	*2,569*	

Da observação da tabela anterior será fácil verificar que o tempo mais elevado de transmissão de mensagens, para um envio periódico de 1000 ms, ocorre com S2 (envio de mensagens temporizadas do componente C1, não replicado, para a réplica C3a, tempo de resposta de 6,527 ms).

Por sua vez, para o envio periódico de 500 ms, verifica-se que o tempo mais elevado de transmissão de mensagens ocorre com S3 (envio de mensagens temporizadas do componente C2, não replicado, para a réplica C3a, tempo de resposta de 6,248 ms). A ordenação apresenta, também, valores de validação elevados, no entanto, não se verificou o determinismo das réplicas. Note-se, ainda que ao aumentarmos a frequência de disponibilidade das mensagens a disseminar, a percentagem de concordância dos dados nas réplicas diminui ligeiramente. Na Figura 8-22 apresentam-se os diagramas correspondentes aos tempos de resposta na réplica C3a, piores tempos de resposta.

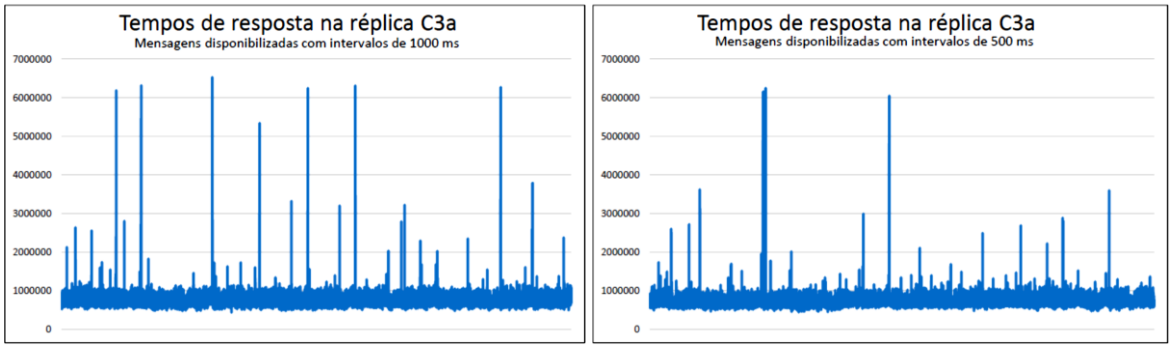


Figura 8-22. Tempos de resposta das réplicas com tráfego adicional de C4 para C5 e vice-versa (nsec)

8.7.3 Testes ao fluxo de mensagens com tráfego adicional nas réplicas

Diversas configurações de teste poderiam ser implementadas e verificadas quanto ao seu comportamento determinístico. Assim, dado que não poderemos testar exaustivamente todas elas, implementou-se mais um cenário de comunicações entre elementos que possam, de algum modo, afetar o envio e a receção das mensagens. Neste sentido, não foram efetuadas alterações

físicas à rede (mantiveram-se as interligações e posições dos equipamentos) desativando-se, somente, o sexto componente, PC1. O tráfego de pacotes extra lançados no sistema foi deslocado para o ip 10.0.1.**101** (RPi1, FB1), utilizando-se o comando *$ sudo ping – f* (envio de 100 ou mais pacotes por segundo com uma taxa de envio de 56(84) *bytes of data*), dirigindo-o para a réplica C3a (ip 10.0.1.**103**, FB3). Com este novo teste de validação pretendia-se aferir quais seriam as alterações que poderiam advir da geração de pacotes adicionais no nó C1 e como poderia afetar a sua receção na réplica C3a. A estrutura de fluxo de mensagens foi definida de acordo com o esquema apresentado na Figura 8-23. Esta nova estrutura é constituída pelos mesmos cinco RPis (dispositivos C1 a C4) e por uma tarefa extra, envio de pacotes de C1 para C3a, que se designará de τ_x.

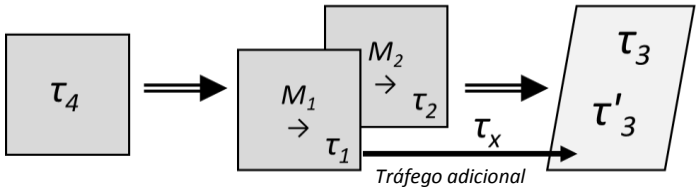


Figura 8-23. Estrutura de testes utilizando tráfego adicional no sentido C1 para C3a

Na Tabela 8.8 são apresentadas as características de cada uma das tarefas e sua alocação aos componentes, enquanto na Tabela 8.9 apresentam-se os dados referentes à propagação e respetiva distribuição pelos elementos replicados e não replicados. Os tempos são apresentados em milissegundos.

Tabela 8.8. Características das tarefas executadas com tráfego adicional entre C1 e C3a

Tarefas	Tipo	Componente	Nó
$\boldsymbol{\tau_1}$	*Periódica*	*C1*	*1*
$\boldsymbol{\tau_2}$	*Periódica*	*C2*	*2*
$\boldsymbol{\tau_3}$	*Periódica*	*C3a*	*3*
$\boldsymbol{\tau_3'}$	*Periódica*	*C3b*	*4*
$\boldsymbol{\tau_4}$	*Esporádica*	*C5*	*5*
$\boldsymbol{\tau_x}$	*Periódica*	*C1*	*1*

Tabela 8.9. Características das mensagens, passagem de eventos com tráfego entre C1 e C3a

Fluxo	Bytes	Período *(ms)*	De	Para	Protocolo
S1	*1*	*-*	τ_4	τ_1, τ_2	*Multicast*
S2	*24*	*1000/500*	τ_1	τ_3, τ_3'	*Multicast*
S3	*24*	*1000/500*	τ_2	τ_3, τ_3'	*Multicast*
Sx	*56(84)*	*ping -f*	τ_x	τ_3	*Unicast*

Note-se que a tarefa $\boldsymbol{\tau_x}$ (fluxo Sx) é difundida segundo um cenário de comunicação *um-para-um* recorrendo a um par SIFB de comunicação do tipo *Publish/Subscribe*, cenário IEC 61499 de difusão dos eventos e dados.

Na Tabela 8.10 apresentam-se os tempos de resposta para cada fluxo de mensagens (rede com tráfego adicional nas réplicas, C3a) usando os protocolos de difusão *multicast* e *unicast*. São, igualmente, apresentados o WCRT, o Período e a Ordenação, indicando-se a percentagem de mensagens recebidas, na mesma ordem, nas réplicas C3a e C3b. A análise da informação trocada entre o componente C1 e C3a, fluxo Sx, não será abordada.

Tabela 8.10. Tempos associados ao fluxo de mensagens com tráfego adicional entre C1 e C3a

Fluxo	Período	WCRT (*ms*)		Ordenação da receção
		C3a	C3b	
S1	-	-	-	
S2	*1000*	*6,208*	*1,556*	*97,09%*
S3	*1000*	*6,448*	*1,941*	
S1	-	-	-	
S2	*500*	*7,196*	*3,178*	*94,34%*
S3	*500*	*6,575*	*3,235*	

Da observação da tabela anterior será fácil verificar que o tempo mais elevado de receção de mensagens, para um envio periódico de 1000 ms, ocorre com S3 (envio de mensagens temporizadas do componente C2, não replicado, para a réplica C3a, tempo de resposta de 6,448 ms). Por sua vez, para o envio periódico de 500 ms verifica-se que o tempo mais elevado de transmissão de mensagens ocorre com S2 (envio de mensagens temporizadas do componente C1, não replicado, para a réplica C3a, tempo de resposta de 7,196 ms). Os resultados obtidos encontram-se em concordância com os obtidos nos testes anteriores não se verificando, de igual modo, o determinismo das réplicas. Note-se, no entanto que ao aumentarmos a frequência de disponibilidade das mensagens a disseminar, a percentagem de concordância dos dados nas réplicas, diminui consideravelmente. Na Figura 8-24 apresentam-se os diagramas correspondentes aos tempos de resposta na réplica C3a, piores tempos de resposta.

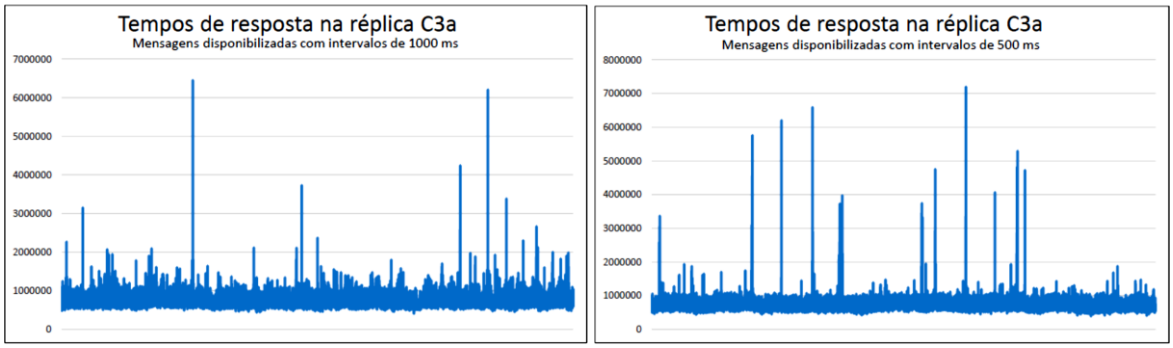


Figura 8-24. Tempos de resposta das réplicas com tráfego adicional de C1 para C3a (nsec)

A título de resumo dos testes realizados podemos verificar que a ordenação dos dados, nas duas réplicas usando unicamente os pares de comunicação *Publish/Subscribe*, ronda um valor médio próximo dos 95 %. Deste modo torna-se notório que os SIFB de comunicação, por si só, não são capazes de garantir o determinismo das réplicas pelo que, será necessário utilizar um elemento de ordenação, a desenvolver e a alocar pelo programador. De acordo com os testes efetuados os tempos de resposta das mensagens são máximos para S3 (7,207 ms em C3a e 8,045 ms em C3b) quando as mensagens são enviadas com intervalos de 500 ms, sistema sem

tráfego adicional. Na Figura 8-25 mostram-se as relações entre os dados ordenados recebidos nas réplicas e os maiores tempos de resposta.

Pode-se verificar ainda que de modo geral a percentagem de concordância, com tráfego adicional, entre as réplicas cresceu, ligeiramente, em relação ao teste sem tráfego (1000 ms). É, no entanto, de salientar os testes de C1 → C3a, tráfego adicional de C1 para a réplica C3a (período de 1000 ms), onde a concordância deste ensaio situou-se em cerca de 97%. O aumento de tráfego no *switch* parece conduzir ao aumento da concordância dos dados para o ciclo de disparos de 1000 ms.

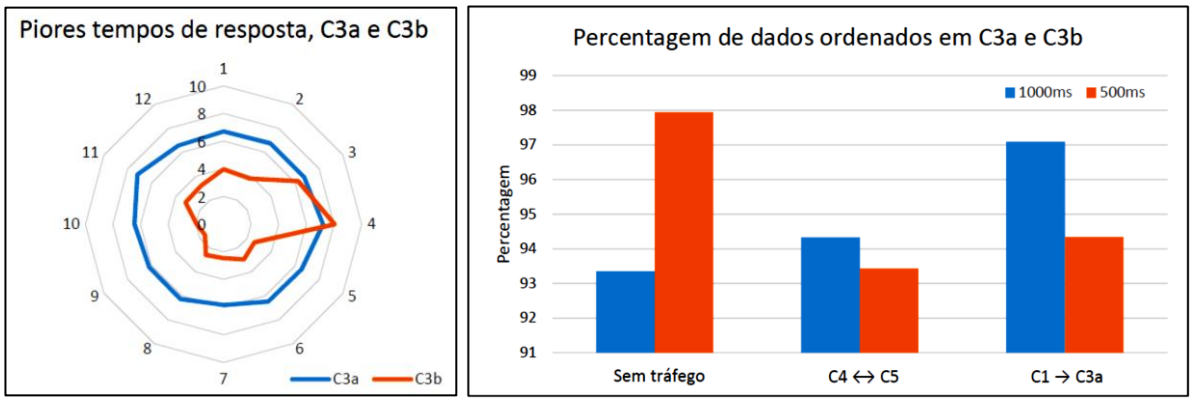


Figura 8-25. Relação dos tempos de resposta e percentagem de dados ordenados nas réplicas

8.8 Testes numéricos aplicados ao sistema replicado

A clarificação do modelo desenvolvido passa pela implementação de um exemplo abrangente que retrate a situação em estudo, ver Figura 8-8. Este modelo será constituído por seis tarefas ($\tau_1 \ldots \tau_6$) e cinco componentes ($C_1 \ldots C_5$) (FB1 a FB4) que serão distribuídas e replicadas pelos diversos nós de acordo com a estrutura apresentada na Figura 8-26. Cada um dos componentes foi distribuído e alocado, como um todo, a um único nó.

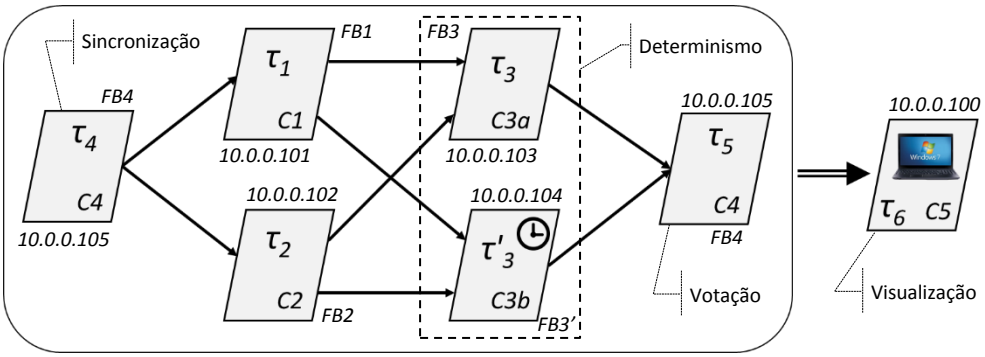


Figura 8-26. Estrutura do exemplo de aplicação usado nos testes numéricos

Para isso foi realizada uma replicação simples do exemplo de acordo com a estrutura apresentada na Figura 8-9 mantendo-se suas interligações e posições na rede Ethernet com taxas de comunicação de 10/100 Mbps. A aplicação foi dividida pelos cinco dispositivos (RPis) alocando-se a cada um deles uma única tarefa. Ao componente C5 alocou-se a tarefa τ_6 que se caracteriza, essencialmente, pela visualização da evolução do sistema. Na Figura 8-27 são apresentados os pormenores de implementação da aplicação desenvolvida, usada nos testes de validação do modelo da *framework* de replicação. A alocação de cada um dos componentes aos nós é representada por diversas cores correspondendo a cor verde ao componente C1 (não replicado,

envio de mensagens temporizadas, FB1) e a cor amarela ao componente C2 (não replicado, envio de mensagens temporizadas, FB2). As réplicas do sistema são definidas pelas cores rosa e *cian* (componentes C3a (FB3) e C3b (FB3'), réplicas recetoras de mensagens temporizadas) e a cor laranja corresponde ao elemento de votação componente C4 (não replicado, FB4).

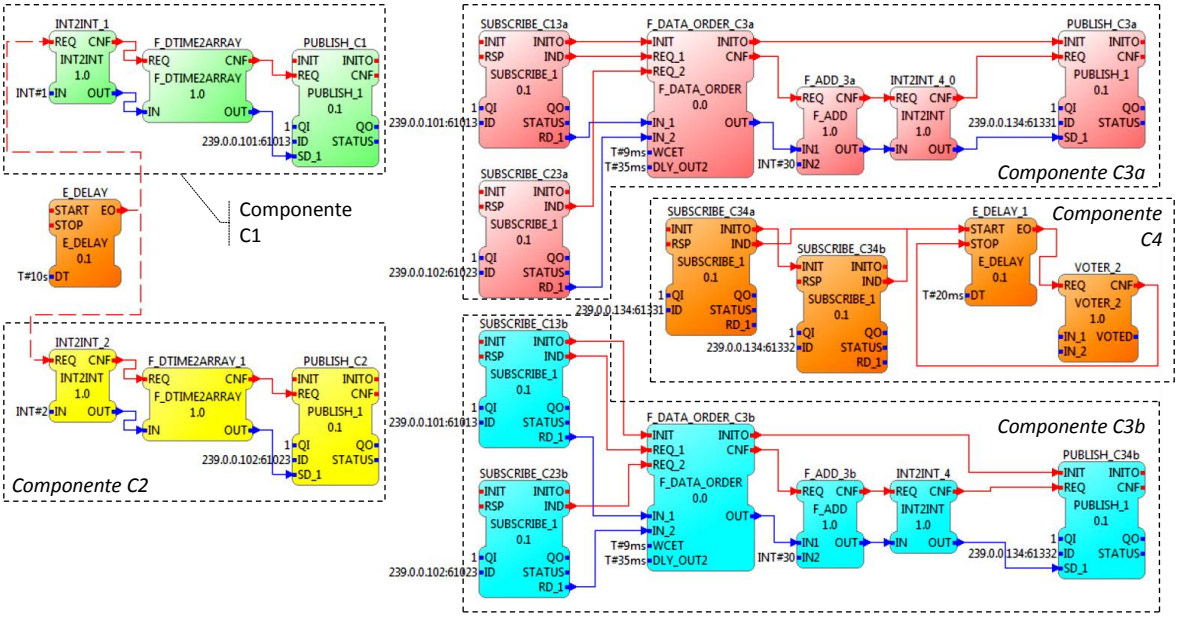


Figura 8-27. Arquitetura do exemplo de aplicação usado nos testes numéricos

A estrutura do fluxo de informação e de mensagens foi definida de acordo com o esquema apresentado na Figura 8-28. As mensagens temporizadas têm origem nos componentes C1 e C2 e são consolidadas nas réplicas do componente C3 (tarefas τ_3 e τ_3').

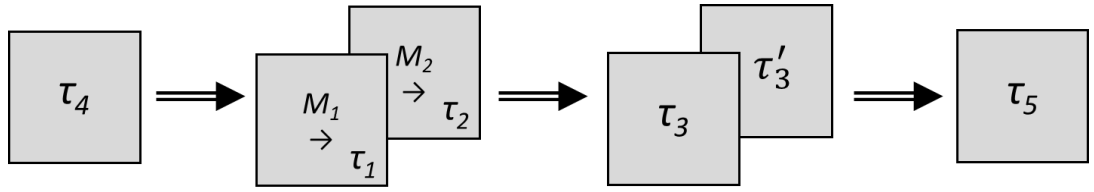


Figura 8-28. Esquema de passagem de informação e de mensagem do exemplo de aplicação

A consolidação dos dados dependerá não só das mensagens temporizadas, mas também do valor associado ao WCET. A definição deste valor encontra-se dependente não só dos tempos máximos de transmissão das mensagens (Tabela 8.11) mas também do máximo desvio associado às diferenças de tempos (*offset*) entre os relógios locais e o relógio do servidor local de tempo, servidor NTP alocado ao componente C3b (ver valores de desvio na Tabela 8.12). Embora os valores associados ao *offset* sejam obtidos em *ms* na Tabela 8.12 as variações dos relógios são apresentados em *µs*. BCRT (*Best Case Response Time*) é definido como os tempos mínimos de resposta das mensagens temporizadas obtidos nas réplicas.

Tabela 8.11. Caracterização dos tempos das mensagens, máximos e mínimos

Fluxo	WCRT (*ms*)	De	Para	BCRT (*ms*)	De	Para
S2	*7,169*	τ_1	τ_3	*0,170*	τ_1	τ_3
S3	*8,045*	τ_2	τ_3'	*0,131*	τ_1	τ_3'

Tabela 8.12. Caracterização do offset de sincronização dos relógios NTP local (µs)

IP		delay (µs)	offset (µs)	jitter (µs)	poll (s)
10.0.1.101	*max.*	*497*	*150*	*45*	*16*
	min.	*449*	*-26*	*4*	
10.0.1.102	*max.*	*511*	*164*	*42*	*16*
	min.	*436*	*-56*	*4*	
10.0.1.103	*max.*	*566*	*118*	*44*	*16*
	min.	*471*	*-37*	*7*	
10.0.1.105	*max.*	*563*	*104*	*58*	*16*
	min.	*498*	*-21*	*7*	

8.8.1 Análise dos tempos de resposta de consolidação

A análise dos tempos de resposta das mensagens nas réplicas, usando o protocolo de comunicação *multicast*, visa determinar o tempo de espera, na fase de decisão ($\delta_{decisão}$), necessário para que o SIFB consolide os dados recebidos. Este atraso será dependente do pior tempo de resposta das réplicas bem como do melhor tempo de resposta das mesmas (processamento das mensagens), tendo como referência o tempo inicial comum a todos os nós de envio. Por outro lado, deveremos considerar que sendo o tempo de envio das mensagens comum a todos os nós (sincronização dos relógios locais) existiram pequenas variações ou erro nas leituras de relógio (*jitter*) que, tal qual o *offset*, tornaram os relógios só aproximadamente sincronizados. Assim, sabendo-se o pior e o melhor tempo de resposta às mensagens (ver Tabela 8.11) poderá definir-se o tempo necessário para a decisão ($\delta_{decisão}$). Este pode ser definido como:

$$\delta_{decisão} = \max_{\forall m \in res(m)} \{W_m\} - \min_{\forall m \in res(m)} \{W_m\} + \varepsilon \tag{8.1}$$

onde $max\{W_m\}$ é o pior tempo de resposta da mensagens (máximo valor de WCRT) e $min\{W_m\}$ é o melhor tempo de resposta para a mensagens (mínimo valor de BCRT), considerando o mesmo tempo de referência. $res(m)$ é o conjunto de mensagens replicadas recebidas e ε o máximo desvio dos relógios locais. Na Figura 8-29 apresentam-se as relações de tempos referidas.

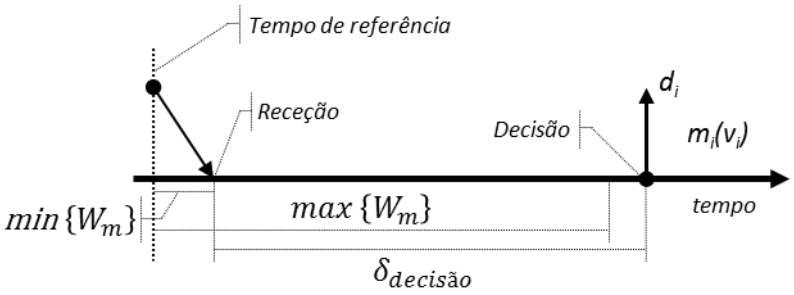


Figura 8-29. Esquema da estrutura de consolidação de dados

Assim sendo o tempo de *decisão* ($\delta_{decisão}$) para a consolidação dos dados será definido em função do pior tempo de resposta dos dados correspondente a um tempo de 8,045 ms (ver Tabela 8.11) e o melhor tempo de resposta dos dados correspondente a um tempo de 0,131 ms. O máximo desvio entre relógios (ε) será de 164 µs pelo que o tempo de decisão será de:

$$\delta_{decisão} = 8{,}045 - 0{,}131 + 0{,}164 = 8{,}078 \cong 9\ ms \tag{8.2}$$

Neste cenário o pior tempo de decisão (Figura 8-29) para a consolidação das mensagens, pode ser definido considerando-se todas as mensagens recebidas no conjunto $res(m)$ pelo que o pior tempo de decisão será dado por:

$$W^{decisão} = \max_{\forall_m \in res(m)} \{W_m\} + \delta_{decisão} \tag{8.3}$$

Nesta abordagem o pior de tempo decisão será então de:

$$W^{decisão} = 8{,}045 + 8{,}078 = 16{,}123\ ms \tag{8.4}$$

8.8.1.1 Análise do tempo de disponibilidade dos dados

A análise dos tempos de disponibilidade dos dados recebidos nas réplicas visa determinar o intervalo de tempo de processamento dos dados (δ_{wcet}). Este será definido como o tempo necessário à execução das operações de sincronismos das réplicas, o tempo de propagação ao FB subsequente e respetiva execução. As medições são realizadas no interior de cada uma das réplicas pelo que estas estarão dependentes, unicamente, dos relógios locais não sendo necessário estabelecer condicionantes extras, tal como os erros de leituras dos relógios (*jitter*) ou o *offset*. Este tempo pode ser calculado de acordo com Lednicki *et al.* (2013) e será definido como:

$$W(cfb) + W(sifb) \tag{8.5}$$

$$cfb = sifb = \{\langle E_i, E_o\rangle, \langle e_s, e_d\rangle\} \tag{8.6}$$

onde E_i é o conjunto de eventos de entrada e E_o o conjunto de eventos de saída, e_s é a porta da fonte de ligação do evento e e_d a porta do elemento de ligação.

Na Figura 8-30 ilustra-se a análise do bloco função composto de ordenação (ver Figura 7-31) e sua associado ao elemento de adição. O elemento bloco função composto (*cfb*) é constituído por quatro FBs que recebem dois eventos de entrada disseminando um único evento de saída.

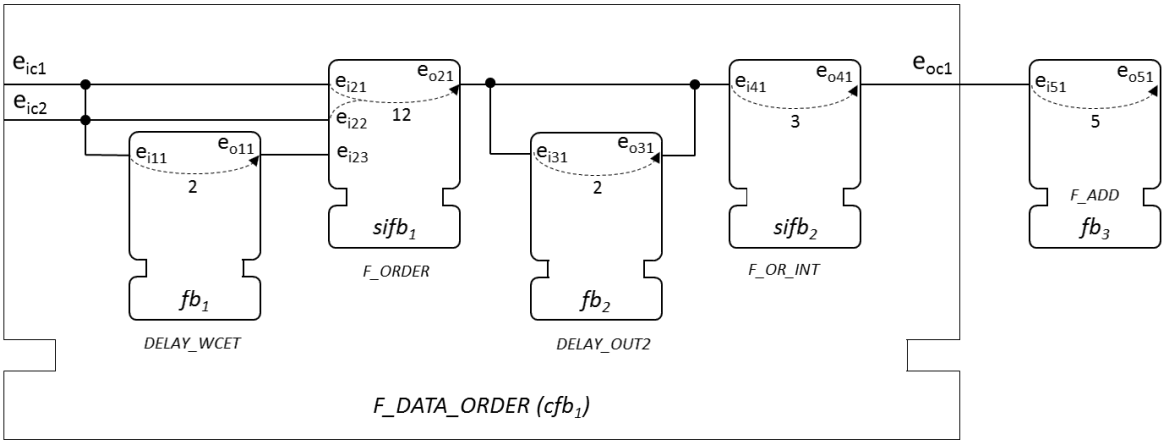


Figura 8-30. Análise WCET do CFB F_DATA_ORDER e de F_ADD_3 (eventos associado à Figura 7-31)

Na figura anterior é apresentado graficamente, por linhas tracejadas, os WCET partindo-se dos eventos de entrada (e_i) dirigidos para os eventos de saída (e_o) onde, a geração de múltiplos eventos de entrada, é representada por múltiplas setas ou percursos. A análise do WCET do CFB *F_DATA_ORDER* será realizada começando nas portas de entrada dos eventos e_{ic} até à propagação às portas dos eventos de saída e_{oc}. A análise termina com a obtenção dos tempos do fb_3 que em conjunto com o cfb_1 definirão os tempos necessários à operacionalidade do elemento sincronizador. Com base em (8.2) assumiremos os seguintes WCET para os FBs em análise:

$$W(fb_1) = \langle 9, \{\langle e_{i11}, \{\langle W, \{e_{o11} = 1\}\rangle\}\}\}\rangle \tag{8.7}$$

$$\begin{aligned} W(sifb_1) = \langle 0, \{\langle e_{i21}, \{\langle W, \{e_{o21} = 1\}\rangle\}\}\}\rangle, \\ \langle 0, \{\langle e_{i22}, \{\langle W, \{e_{o21} = 1\}\rangle\}\}\}\rangle, \\ \langle 9, \{\langle e_{i23}, \{\langle W, \{e_{o21} = 1\}\rangle\}\}\}\rangle \end{aligned} \tag{8.8}$$

$$W(fb_2) = \langle 25, \{\langle e_{i31}, \{\langle W, \{e_{o31} = 1\}\rangle\}\}\}\rangle \tag{8.9}$$

$$W(sifb_2) = \langle 0, \{\langle e_{i41}, \{\langle W, \{e_{o41} = 2\}\rangle\}\}\}\rangle \tag{8.10}$$

$$W(fb_3) = \langle 0, \{\langle e_{i51}, \{\langle W, \{e_{o51} = 1\}\rangle\}\}\}\rangle \tag{8.11}$$

Neste sentido, para se obter o WCET(cfb_1) deveremos utilizar unicamente as entradas que apresentam os valores máximos. Estes valores serão definidos em função das portas de entrada do CFB e das portas dos eventos de saída. Os valores definidos corresponderão, usando o método da normalização do máximo elemento, neste caso, mantendo-se as duas entradas, aos máximos obtidos para o cfb_1 analisado (*F_DATA_ORDER)*. Os valores de WCET serão dados por:

$$\begin{aligned} WCET(cfb_1) = \langle \{\langle e_{ic1}, \{\langle W_1, \{e_{oc1} = 2\}\rangle, \\ \langle W_2, \{e_{oc1} = 2\}\rangle\}\}\rangle, \\ \langle \{\langle e_{ic2}, \{\langle W_1, \{e_{oc1} = 2\}\rangle, \\ \langle W_2, \{e_{oc1} = 2\}\rangle\}\}\rangle \end{aligned} \tag{8.12}$$

Onde a normalização suprema dos resultados traduz-se num valor duplo e igual dos tempos que será definido como:

$$WCET(cfb_1) = \langle \{\langle e_{ic1}, e_{ic2}\ \{\langle W, \{e_{oc1} = 2\}\rangle\}\}\rangle\} \tag{8.13}$$

Para a determinação dos WCET associados ao bloco função composto e ao bloco F_ADD_3 foram realizados vários testes registando-se 3000 valores associados a cada um dos FB dos elementos em estudo. Da análise dos dados, em *ms*, determinaram-se os seguintes WCET para cada um dos FBs alocados às réplicas:

$$W(fb_1) = \langle 9, \{\langle e_{i11}, \{\langle 2, \{e_{o11} = 1\}\rangle\}\}\}\rangle \tag{8.14}$$

$$\begin{aligned} W(sifb_1) = \langle 0, \{\langle e_{i21}, \{\langle 12, \{e_{o21} = 1\}\rangle\}\}\}\rangle, \\ \langle 0, \{\langle e_{i22}, \{\langle 12, \{e_{o21} = 1\}\rangle\}\}\}\rangle, \\ \langle 9, \{\langle e_{i23}, \{\langle 12, \{e_{o21} = 1\}\rangle\}\}\}\rangle \end{aligned} \tag{8.15}$$

$$W(fb_2) = \langle 25, \{\langle e_{i31}, \{\langle 2, \{e_{o31} = 1\}\rangle\}\rangle\}\rangle \quad (8.16)$$

$$W(sifb_2) = \langle 0, \{\langle e_{i41}, \{\langle 3, \{e_{o41} = 2\}\rangle\}\rangle\}\rangle \quad (8.17)$$

$$W(fb_3) = \langle 0, \{\langle e_{i51}, \{\langle 5, \{e_{o51} = 1\}\rangle\}\rangle\}\rangle \quad (8.18)$$

Neste sentido, usando o método anteriormente indicado obteremos os seguintes valores para a análise em questão:

$$\begin{aligned} WCET(cfb_1) = \langle \{\langle e_{ic1}, \{\langle 26, \{e_{oc1} = 2\}\rangle, \\ \langle 28, \{e_{oc1} = 2\}\rangle\}\rangle\}\rangle, \\ \langle \{\langle e_{ic2}, \{\langle 26, \{e_{oc1} = 2\}\rangle, \\ \langle 28, \{e_{oc1} = 2\}\rangle\}\rangle\}\rangle \end{aligned} \quad (8.19)$$

Onde a normalização suprema dos resultados seria:

$$\begin{aligned} WCET(cfb_1) = \langle \{\langle e_{ic1}, \{\langle 28, \{e_{oc1} = 2\}\rangle\}\rangle\}\rangle \\ \langle \{\langle e_{ic2}, \{\langle 28, \{e_{oc1} = 2\}\rangle\}\rangle\}\rangle \end{aligned} \quad (8.20)$$

Assim sendo o WCET de operacionalidade de cada uma das réplicas será definido como:

$$\begin{aligned} \delta_{wcet\ réplica} &= \langle \{\langle e_{ic1}, \{\langle 28, \{e_{oc1} = 2\}\rangle\}\rangle\}\rangle + \langle 0, \{\langle e_{i51}, \{\langle 5, \{e_{o51} = 1\}\rangle\}\rangle\}\rangle \\ &= \langle \{\langle e_{ic1}, e_{ic2}\ \{\langle 33, \{e_{oc1} = 2\}\rangle\}\rangle\}\rangle \end{aligned} \quad (8.21)$$

A equação anterior definirá o tempo mínimo necessário para a libertação do segundo dado (tempo de simulação), pelo que o valor a utilizar deverá ser maior ou igual ao obtido. Neste caso optamos por utilizar o valor de 35 ms para a disponibilização do dado dois, adição do tempo de execução do FD INT2INT (2ms). Na Figura 8-31 apresentam-se os tempos de funcionamento de cada um dos FBs (associados às réplicas), de standby e respetivo ciclo de funcionamento das réplicas.

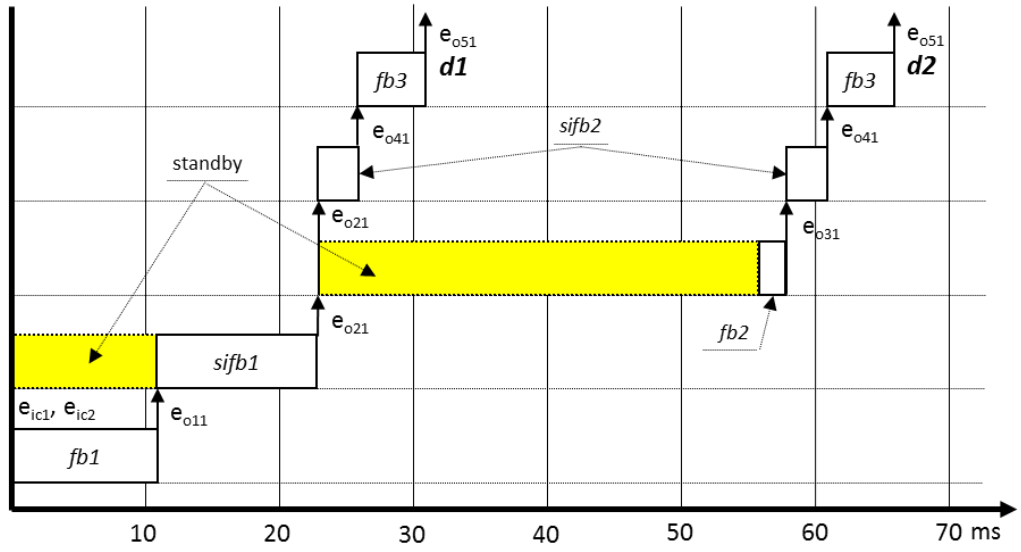


Figura 8-31. Diagrama dos tempos de operação das réplicas

8.8.2 Validação comportamental do sistema (um switch)

No teste completo do sistema manteve-se a estrutura física da implementação inicial das interligações e posições do equipamento em relação ao *switch* (experiência 1, ver Figura 8-14). Com este teste pretendia-se aferir o modelo desenvolvido e como tal, a capacidade de se garantir o determinismo das réplicas (componentes C3a e C3b) bem como da eleição do valor rececionado no dispositivo 4. Para isso, caracterizaram-se, na Tabela 8.13, cada uma das tarefas e sua alocação aos componentes, enquanto na Tabela 8.14 apresentam-se os dados referentes à passagem de eventos e de mensagens e respetiva distribuição pelos elementos.

Tabela 8.13. Características das tarefas executadas pelo sistema replicado

Tarefas	Tipo	Componente	Nó
$\boldsymbol{\tau_1}$	*Periódica*	*C1*	*1*
$\boldsymbol{\tau_2}$	*Periódica*	*C2*	*2*
$\boldsymbol{\tau_3}$	*Periódica*	*C3a*	*3*
$\boldsymbol{\tau_3'}$	*Periódica*	*C3b*	*4*
$\boldsymbol{\tau_4}$	*Esporádica*	*C4*	*5*
$\boldsymbol{\tau_5}$	*Periódica*	*C4*	*5*

Note-se que os SIFB de comunicação são todos da família *Publish/Subscribe* usando um protocolo de comunicação do tipo *multicast*. A tarefa τ_4 é, mais uma vez, utilizada na sincronização dos componentes C1 e C2 e encontra-se alocada ao componente C4 também responsável pela escolha do valor recebido das réplicas.

Tabela 8.14. Características das mensagens e da passagem de eventos, sistema replicado

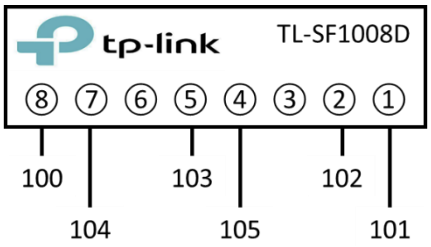
Fluxo	Bytes	Período (*ms*)	De	Para	Protocolo
S1	*1*	-	τ_4	τ_1, τ_2	*Multicast*
S2	*24*	*500*	τ_1	τ_3, τ_3'	*Multicast*
S3	*24*	*500*	τ_2	τ_3, τ_3'	*Multicast*
S4	*8*	-	τ_3	τ_5	*Multicast*
S5	*8*	-	τ_3'	τ_5	*Multicast*

Na Tabela 8.15 apresentam-se os valores máximos e mínimos de resposta dos fluxos S2 e S3 (C1 e C2), receção de mensagens, e respetivas ordenações (ver Figura 8-27). É de salientar que a ordenação das mensagens recebidas nas réplicas encontra-se dentro dos valores esperados e que o valor WCRT enquadra-se no valor médio esperado. Pode verificar-se ainda que o CFB de ordenação, F_DATA_ORDER, garante o determinismo das réplicas e, como tal, os dados são ordenados e processados em função do tempo de validação de cada uma das mensagens recebidas.

Tabela 8.15. Tempos associados ao fluxo de mensagens, sistema replicado

Exper.	Fluxo	Período (*ms*)	WCRT réplicas (*ms*)		Ord. da receção	Consol. votador
			Max.	Min.		
1	*S2*	*500*	*6,325*	*0,398*	*93,80%*	*100%*
	S3	*500*	*3,237*	*0,244*		
2	*S2*	*500*	*7,116*	*0,347*	*95,05%*	*100%*
	S3	*500*	*3,015*	*0,051*		

Alterou-se a implementação física do sistema interligando-se o equipamento a novas portas e, com isso, os pontos de ligação ao *switch* (experiência 2). Estas novas interligações e respetivas posições de ligação ao *switch* podem ser vistas na Figura 8-32.



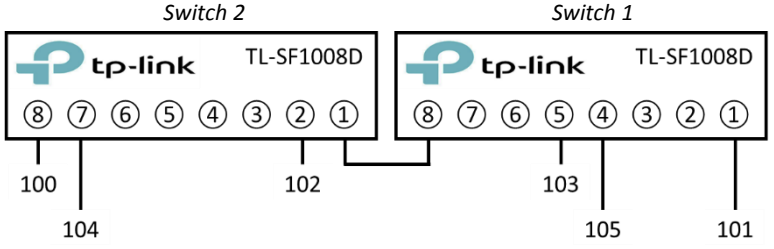
Portas:
1 – RPi2 ip 10.0.1.101
2 – RPi2 ip 10.0.1.102
4 – RPi2 ip 10.0.1.105
5 – RPi2 ip 10.0.1.103
7 – RPi3 ip 10.0.1.104
8 – PC1 ip 10.0.1.100

Figura 8-32. Esquema de ligação do sistema ao switch, novo posicionamento (experiência 2)

Observando-se, novamente, a Tabela 8.15 (experiência 2) poderá verificar-se que o tempo mais elevado de resposta das réplicas ocorreu com S2 (envio de mensagens do componente C1, não replicado), com um tempo de resposta de 7,116 ms. Os resultados obtidos encontram-se em concordância com todos os testes efetuados anteriormente e continua a não se verificar o determinismo das réplicas. Por outro lado, verifica-se que a troca das portas de ligação dos equipamentos não introduziu grandes alterações de comportamentos e de tempos de resposta, experiência 2. No entanto verificou-se que o tempo mínimo de resposta da réplica decresceu substancialmente com S3 (envio de mensagens do componente C2, não replicado). A consolidação dos valores votados situou-se nos 100% de concordância.

8.8.3 Validação comportamental do sistema (dois switchs)

Para este novo teste de sincronização dos valores recebidos o sistema foi reconfigurado. O sistema passou a estar interligado com dois *switchs* de modo a distribuir o mais possível o fluxo de mensagens. A arquitetura final das interligações entre componentes encontra-se representada no esquema apresentado na Figura 8-33. As réplicas foram distribuídas por cada um dos *switchs* (réplicas alocadas aos ips 103 e 104) bem como os "*conveyors*" de alimentação do elemento replicado (componentes alocados aos ips 101 e 102).



Portas switch 1:
1 – RPi2 ip 10.0.1.101
4 – RPi2 ip 10.0.1.105
5 – RPi2 ip 10.0.1.103
Portas switch 2:
2 – RPi2 ip 10.0.1.102
7 – RPi3 ip 10.0.1.104
8 – PC1 ip 10.0.1.100

Figura 8-33. Esquema de interligação dos switchs, novo reposicionamento dos componentes (experiência 3)

Na Tabela 8.16 são apresentados os tempos e a percentagem de sincronismo das réplicas. Os valores obtidos encontram-se em consonância com os restantes.

Tabela 8.16. Tempos do fluxo de mensagens, sistema replicado por dois switchs

Exper.	Fluxo	Período (*ms*)	WCRT réplicas (*ms*)		Ord. da receção	Consol. votador
			Max.	Min.		
3	*S2*	*500*	*6,396*	*0,443*	*97,84%*	*100%*
	S3	*500*	*3,101*	*0,258*		

8.8.4 Validação comportamental do sistema, falhas por colapso

A fim de testar a robustez do sistema realizaram-se testes ao colapso das linhas de comunicação dos diversos dispositivos (falhas por colapso do sistema de comunicação). Nestes testes a simulação do colapso de uma linha de comunicação ou uma possível avaria de uma das portas dos *switch* foi efetuada pelo ato de desligar cada uma das linhas em causa. Assim, desligou-se a linha 1 (ip 10.0.1.101, *switch 1*) associada ao componente C1, religando-a passados alguns segundos, repetiu-se o processo para a linha 2 (ip 10.0.1.102, *switch 2*) associada ao componente C2.

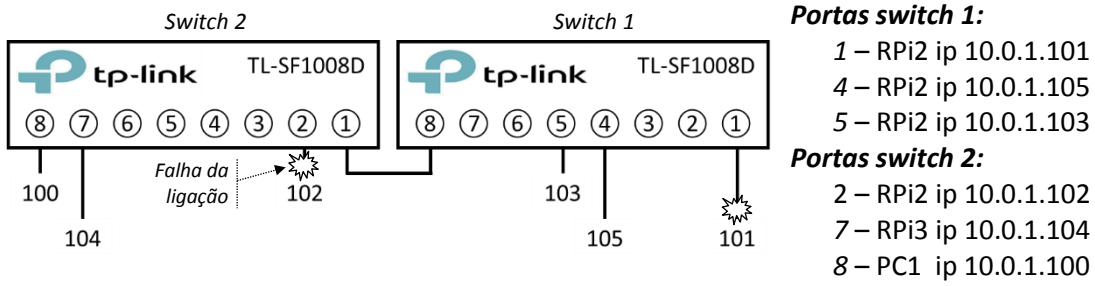


Figura 8-34. Esquema de simulação do colapso das linhas 101 e 102 (experiência 4)

As linhas em questão foram desligadas e ligadas mais que uma vez ao longo do teste e de acordo com o processo anterior, considerando-se os tempos, entre o desligar e o ligar os componentes, aleatórios. Na Figura 8-34 apresentam-se as ligações dos componentes e as linhas sujeitas ao colapso, linhas de alimentação das réplicas. Durante os testes o sistema manteve-se estável e de acordo com o comportamento esperado. Assim sendo, após colapso da linha 101, não se receberam valores provenientes de C1 (valor 1) pelo que as réplicas processaram os valores 2, associado a C2, e o valor 0, associado à ausência do valor 1, componente C1. A inversão do processo manifestou-se na inversão dos resultados pelo que as réplicas deixaram de processar o valor 2, associado a C2, processando 1 e 0, associado ao componente C1. Religando-se todas as linhas o sistema retorna ao estado inicial processando os valores 1 e 2. Na Tabela 8.17 apresentam-se os tempos do fluxo de mensagens trocadas entre C1 e C2 (S2 e S3) e cada uma das réplicas.

Tabela 8.17. Tempos do fluxo de mensagens, sistema replicado por dois switchs (rutura de linhas)

Exper.	Fluxo	Período (*ms*)	WCRT réplicas (*ms*)		Ord. da receção	Consol. votador
			Max.	Min.		
4	*S2*	*500*	*6,279*	*0,381*	*94,98%*	*100%*
	S3	*500*	*2,089*	*0,273*		

Da observação da Tabela 8.17 poderemos verificar que o tempo mais elevado de resposta das réplicas ocorreu com S2 (envio de mensagens do componente C1, não replicado), tempo de resposta de 6,279 ms. Os resultados obtidos neste teste são idênticos aos valores obtidos nos testes efetuados anteriormente, continua a não se verificar o determinismo das réplicas. Por outro lado, verifica-se que o desligar e o ligar das linhas de comunicação dos equipamentos não introduziu alterações significativas no comportamento da rede. Os tempos mínimos de resposta das réplicas encontram-se em consonância com os valores obtidos nos testes anteriores salienta-se, no entanto que, apesar da interrupção das linhas dos componentes, a consolidação dos dados no elemento de votação traduziu-se em 100%. No diagrama temporal apresentado na Figura 8-35 retrata-se o comportamento do sistema replicado nos primeiros segundos de funcionamento (interrupção do envio das mensagens S2 e S3).

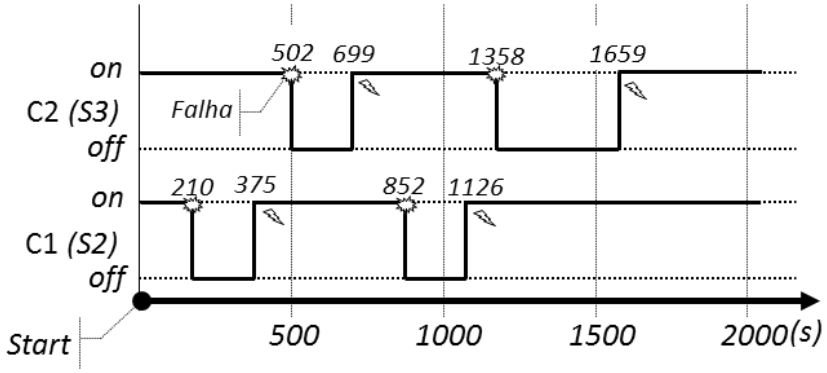


Figura 8-35. Diagrama de rutura das linhas ou de falhas das portas de comunicação dos switchs

Naturalmente que estes testes não poderiam ser dados como concluídos sem testar a capacidade do sistema se manter operacional no caso de uma falha associada a uma das réplicas. Esta nova validação terá de ser feita, de igual modo, recorrendo à ação de desligar e ligar, fisicamente, as linhas de comunicação de cada uma das réplicas C3a e C3b. Na Figura 8-36 apresenta-se o esquema de simulação de falhas das réplicas e das falhas resultantes do colapso das linhas de comunicação ou de avarias das portas do *switch*.

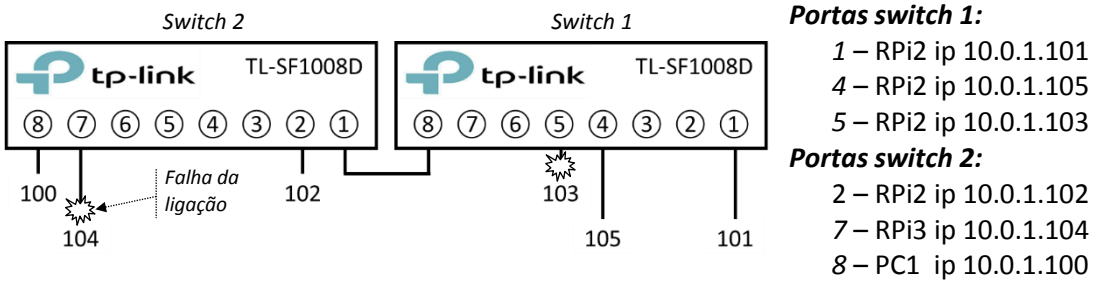


Figura 8-36. Esquema de simulação de rutura das linhas ou de falhas das réplicas (experiência 5)

Na Tabela 8.18 apresentam-se os valores obtidos durante a interrupção do fluxo de mensagens trocadas entre os componentes C1 e C2 (fluxos S2 e S3) e cada uma das réplicas. Os tempos associados à rutura das linhas das réplicas (C3a seguido de C3b) apresentam a tendência verificada anteriormente (S2 apresentas com tempos de resposta mais elevado, ambas as réplicas). Os valores obtidos para a ordenação dos dados recebidos nas réplicas correspondem ao momento em que as respetivas linhas foram desligadas. A ordenação dos dados apresenta um valor de 20,49 % na réplica C3a (R103) o que correspondente ao momento em que esta foi desligada. Por seu lado a ordenação de 25,60 %, associada à réplica C3b (R104), corresponderá aos dados ordenados no momento em que esta réplica foi desligada. Saliente-se, no entanto, que

a votação final no elemento de consolidação, componente C4, apresentou o valor de 100 %. Isto é, apesar do colapso das ligações às réplicas o resultado da consolidação dos dados foi plenamente obtido quer com ambas as réplicas ativas quer com uma só interligada.

Tabela 8.18. Comportamento do sistema replicado rutura das linhas das réplicas

Exper. 5	Fluxo	Período (*ms*)	WCRT réplicas (*ms*)		Ord. da receção	Consol. votador
			Max.	Min.		
R103 (C3b)	*S2*	*500*	*1,500*	*O,104*	*20,49%*	*100%*
	S3	*500*	*2,088*	*0,273*		
R104 (C3a)	*S2*	*500*	*4,397*	*0,411*	*25,60%*	*100%*
	S3	*500*	*2,088*	*0,273*		

Foram realizados diversos testes de simulação do colapso das linhas ou de mau funcionamento das portas do *switch*, tendo como alvos as ligações das duas réplicas do sistema. Nestes, foram verificadas diversas ocorrências que, de algum modo, influenciam não só a perceção de evolução do sistema, mas também a sua operacionalidade. Se por um lado a perceção de evolução do sistema poderá ser um dos fatores menores de funcionamento do mesmo (os módulos de *software* associados às linhas desligadas tornam-se estáticos, perda da perceção de funcionamento) a operacionalidade do mesmo será mais crítica. No decorrer dos testes, realizados com a versão 1.8.4 do *4diac*, pode-se apontar as seguintes anomalias ocorridas aquando da interrupção das linhas de comunicação:

- ***Visualização***: após colapso da linha de comunicação (desligar o cabo) os equipamentos perdem a capacidade de serem monitorizados deixando de ser possível acompanhar a evolução dos elementos de *software*. O componente permanece com todos os seus valores estáticos e a reposição da ligação não altera o estado estático da visualização.
- ***Acesso remoto***: a reposição da ligação não repõe o acesso remoto ao equipamento pelo que se torna impossível controlar o equipamento.
- ***Falha da réplica***: após rutura da linha de comunicação de uma qualquer réplica o último valor processado fica residente o que conduz à produção e consequente votação de dados errados.

O que acaba de se expor não são características intrínsecas de cada uma das réplicas são características comuns, quase na totalidade, a todos os equipamentos que se desligaram da rede, neste caso de estudo. Poderá dizer-se que uma grande parte resolve-se "desligando e ligando o sistema" (*reset*) o que é verdade para a visualização e o acesso remoto. No entanto, deve-se salientar que o acesso remoto poderá ser retomado sem a ação de *reset* do sistema, caso da réplica C3b que utiliza um RPi3, mantendo-se a visualização inalterada. O sistema mantém-se em funcionamento trabalhando quer com uma só réplica quer com um só componente de alimentação do sistema (C1 ou C2).

Por outro lado, no que se refere à falha da réplica, a situação torna-se um pouco mais complicada uma vez que, embora a réplica não esteja a enviar eventos e dados, o último dado enviado permanece disponível para o votador. Isto dever-se-á à forma construtiva dos FBs dado que cada evento gerado disponibilizará um dado de saída para leitura do FB subsequente. Nesta situação o elemento de votação, ou de escolha, dos valores recebidos terá sempre à sua disposição o último valor recebido o que levará a um funcionamento erróneo do módulo de votação. Para obviar este comportamento foi necessário desenvolver um elemento de "filtragem" capaz de eliminar o valor residente da réplica em falha permitindo, no entanto, a sua passagem

sempre que se encontre ativo. O filtro colocado à entrada do votador permite obter, alternadamente, o valor zero (0) ou o valor recebido da réplica, indicação de funcionamento ou de paragem da mesma. Nesta situação o votador escolhe o único valor recebido das réplicas que se encontram em funcionamento.

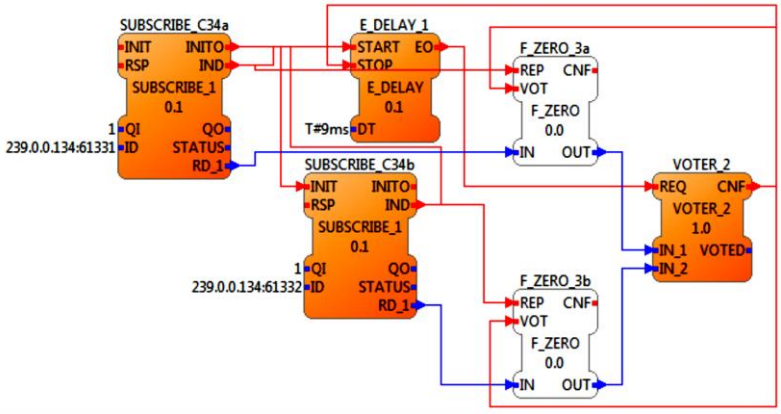


Figura 8-37. Filtragem dos valores residuais dos subscribe do elemento de consolidação

Na Figura 8-37 apresenta-se o sistema de filtragem dos dados enviados pelas réplicas. Este SIFB é, simplesmente, um elemento de isolamento dos valores residuais dos *subscribe* e do votador. O SIFB F_ZERO permite a passagem do valor recebido no *subscribe*, solicitação do *subscribe*, disponibilizando o valor 0, saída OUT, quando solicitado pelo votador. O não envio do evento IND (*subscribe*) implica no bloqueio do valor residual do SIFB F_ZERO e no fornecimento do valor zero ao votador quando solicitado através do evento VOT, sistema *passa-não-passa*.

Algoritmo 6 – Filtragem do valor a votar

```
1:  void FORTE_F_ZERO::setInitialValues(){
2:          IN() = 0;
3:          OUT() = 0; }
4:  void FORTE_F_ZERO::executeEvent(int pa_nEIID){
5:          switch(pa_nEIID){
6:             case scm_nEventREPID:
7:                OUT() = IN();
8:                sendOutputEvent(scm_nEventCNFID);
9:             break;
10:            case scm_nEventVOTID:
11:               OUT() = 0;
12:               sendOutputEvent(scm_nEventCNFID);
13:            break;
14:         }
15:   }
```

O código do SIFB F_ZERO é apresentado no *Algoritmo 6*. As linhas iniciais definem as variáveis associadas ao valor dos dados a transmitir ao votador enquanto as linhas 4 a 13 definem o valor a alocar à variável OUT(). Esta alocação está dependente da solicitação do evento associado, ou seja, de REP através do evento IND do *subscribe* ou VOT através do evento CNF do votador.

Saliente-se ainda que após restauração da rede, mesmo sem ser efetuado um *reset* do sistema, a aplicação funciona como esperado mantendo as suas características e a coerência inicial. As réplicas receberão os valores enviados por C1 e C2 e o votador poderá escolher o valor a utilizar em função dos dados recebidos de cada uma das réplicas (C3a e C3b).

Analisando os resultados obtidos durante a realização destes últimos testes podemos verificar que os valores registados enquadram-se nos valores iniciais quer ao nível dos tempos quer da percentagem de sincronização dos dados recebidos nas réplicas. Na Figura 8-38 apresentam-se as relações entre os dados recebidos em cada uma das réplicas e a ordenação dos dados. Os valores tratados dizem respeito às Tabela 8.15 e Tabela 8.17.

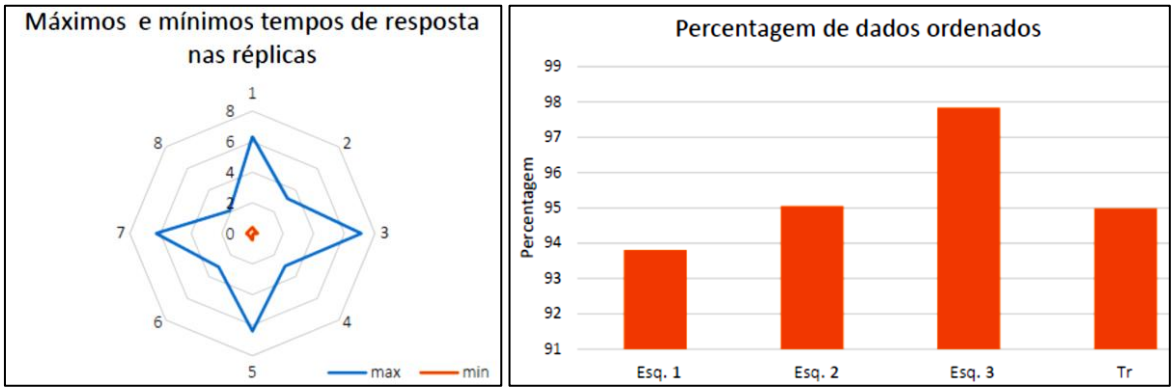
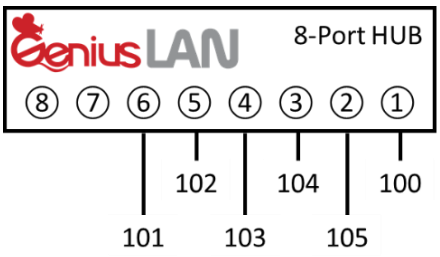


Figura 8-38. Relação dos tempos de resposta e percentagem da sincronização das réplicas

8.9 Validação do sistema replicado utilizando um *HUB*

Com estes testes à rede de comunicações pretendeu-se verificar o comportamento da mesma na presença de um *hub*. Neste sentido, sabendo-se que este equipamento ativo tem como característica principal a disseminação da informação recebida numa porta por todas as outras, pretendia-se indagar o comportamento da aplicação e das comunicações quando sujeitas a um maior tráfego e consequente diminuição de performance da mesma. A nova implementação da rede encontra-se definida na Figura 8-39. Note-se que o esquema de ligações assenta na experiencia número 1 de testes e as tarefas a realizar são as definidas na Tabela 8.13.



Portas:

1 – RPi2 ip 10.0.1.100
2 – RPi2 ip 10.0.1.105
3 – RPi2 ip 10.0.1.104
4 – RPi2 ip 10.0.1.103
5 – RPi3 ip 10.0.1.102
6 – PC1 ip 10.0.1.101

Figura 8-39. Esquema de ligação do sistema ao HUB

O *Hub* de marca Genius possui 8 portas de ligação RJ45 e uma do tipo BNC funcionando segundo o protocolo Ethernet (IEEE 802.3) a uma velocidade de 10 Mbps. A rede foi testada com este novo componente ativo avaliando-se as possíveis perturbações introduzidas pela disseminação da informação para todas as portas, excetuando-se a que recebeu a informação. Na Tabela 8.19 são apresentados os tempos e a percentagem de sincronismo das réplicas sem tráfego adicional na rede de comunicação. O esquema de transferência de mensagens entre os elementos encontra-se representado na Figura 8-40. As interligações e replicação entre componentes do sistema em teste encontram-se esquematizadas na Figura 8-26.

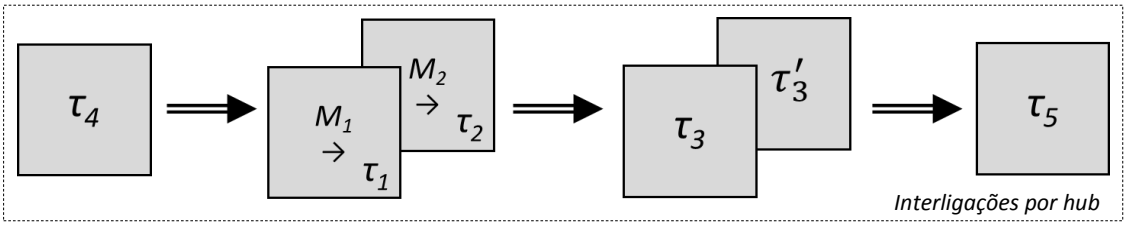


Figura 8-40. Esquema de transmissão de mensagens usando um HUB

Tabela 8.19. Tempos do fluxo de mensagens, interligação por HUB (sem tráfego adicional)

Fluxo	Período	WCRT (*ms*)		Ordem receção	Ordena réplica	Consol. votador
		C3a	C3b			
S2	*1000*	*6,766*	*3,607*	*92,93%*	*100%*	*100%*
S3	*1000*	*6,539*	*3,291*			
S2	*500*	*6,775*	*3,419*	*94,79%*	*100%*	*100%*
S3	*500*	*6,145*	*3,418*			

Da observação da Tabela 8.19 poderá verificar-se que o tempo mais elevado de respostadas das réplicas ocorreu com S2 com um tempo de resposta de 6,766 ms e 6,775 ms (testes efetuados com 1000 ms e 500 ms, respetivamente). Os resultados obtidos encontram-se em concordância com todos os testes efetuados anteriormente e continua a não se verificar, igualmente, o determinismo das réplicas. A introdução do *hub* não introduziu variações significativas no comportamento do sistema mantendo-se a relação percentual de sincronismo idêntica aos testes anteriores. Os tempos médios de reposta das réplicas rondam, aproximadamente, os 6,5 ms na réplica C3a e os 3,4 ms na réplica C3b sendo que, os tempos da réplica C3b são, sensivelmente, metade dos obtidos na réplica C3a. O resultado da ordenação dos dados recebidos nas réplicas C3a e C3b traduziu-se em 100 % de ordenação (F_ORDER) e, como tal, o determinismo das réplicas é conseguido. Os resultados da consolidação dos dados no votador apresentam, também eles, a taxa de assertividade de 100 %, ambas as réplicas enviam os mesmos dados para votação. A utilização de um *hub* como elemento ativo de ligação dos diversos componentes não interferiu nos resultados expectáveis.

Na Tabela 8.20 apresentam-se os tempos de resposta dos fluxos de mensagens referentes ao teste com tráfego adicional na rede segundo esquema apresentado na Figura 8-41.

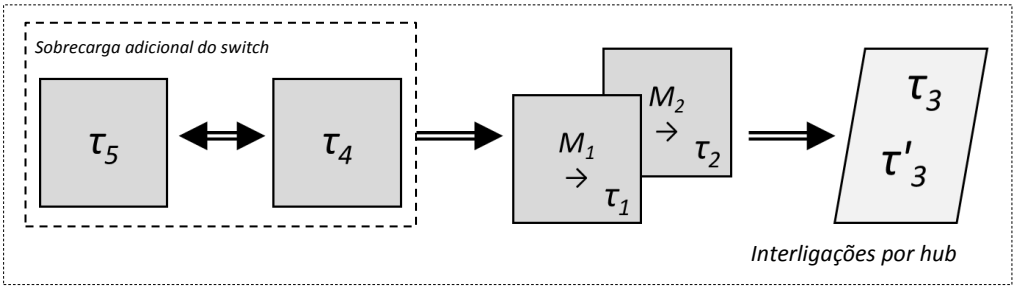


Figura 8-41. Esquema de transmissão de mensagens usando um HUB, tráfego adicional

Os protocolos de difusão e características da aplicação mantiveram-se inalterados. A aplicação foi novamente testada com períodos de 1000 e 500 ms.

Tabela 8.20. Tempos do fluxo de mensagens, interligação por HUB (com tráfego adicional)

Fluxo	Período	WCRT (*ms*)		Ordem receção	Ordena réplica	Consol. votador
		C3a	C3b			
S2	*1000*	*19,360*	*15,538*	*99,70%*	*99,81%*	*34,27%*
S3	*1000*	*19,142*	*15,530*			
S2	*500*	*16,843*	*13,783*	*95,80%*	*91,93%*	*58,11%*
S3	*500*	*15,948*	*13,691*			

Da observação da tabela anterior será possível verificar que os tempos de resposta apresentaram alguma oscilação inter-réplicas. Os tempos mais elevados de resposta registaram-se no fluxo S2 (envio de mensagens do componente C1, não replicado) quer para o período de disseminação de 1000 ms quer para o de 500 ms, alternando-se entre a réplica C3a e a C3b. Os valores de sincronização dos dados recebidos mostraram uma ligeira subida confirmando-se, no entanto, a não existência de sincronismo das réplicas. Os valores médios dos tempos de resposta em cada uma das réplicas rondaram os 17,8 ms em C3a e os 14,6 ms em C3b, aproximadamente. É de salientar que os dados, após ordenação, recebidos nas diferentes réplicas (ordenação dos dados utilizando o FB F_DATA_ORDER) diferem dos 100 % (apresentam um valor médio de 95,9 %, aproximadamente). A consolidação dos mesmos apresenta, também, um desvio significativo do valor 100 apontando para uma média de valores corretos de cerca de 46,2 %. No entanto, a simulação com intervalo de disparo mais reduzido (500 ms) apresenta um comportamento mais próximo da concordância total dos valores.

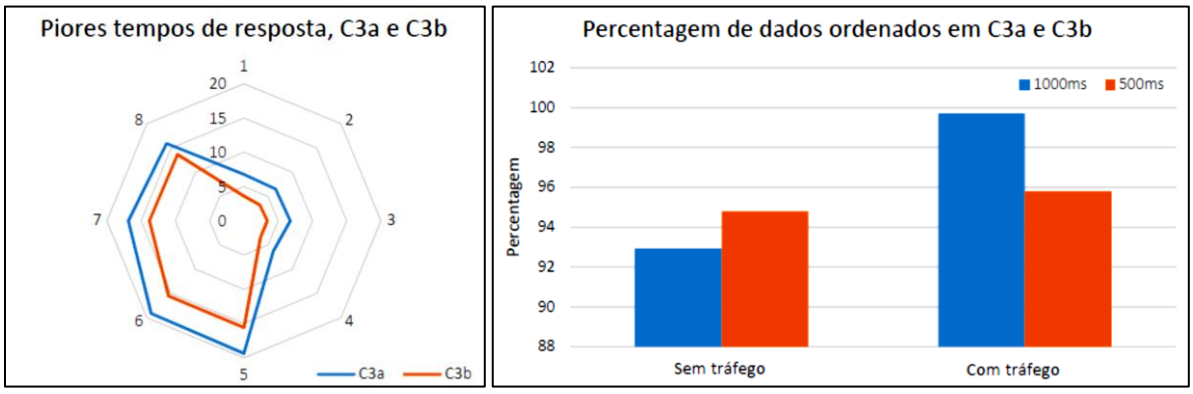


Figura 8-42. Relação dos tempos de resposta e percentagem de dados ordenados numa rede com um HUB.

Analisando graficamente (Figura 8-42) os resultados obtidos com a utilização do *hub* podemos verificar que a ordenação dos dados recebidos nas réplicas ronda um valor médio próximo dos 96 %. É confirmada a tendência da falta de determinismo das réplicas usando unicamente os SIFB de comunicação *Publish/Subscribe,* mesmo para uma rede Ethernet utilizando um *HUB* como elemento de interligação. Os piores tempos de resposta de dados, com tráfego adicional, apresentaram uma subida considerável, no entanto verifica-se que o comportamento de cada uma das réplicas é bastante estável uma vez que os tempos apresentam uma dispersão muito próxima da paralela.

Os valores registados enquadram-se nos valores iniciais quer ao nível dos tempos quer da percentagem de sincronização dos dados recebidos. Os valores tratados dizem respeito à Tabela 8.19 e à Tabela 8.20.

Capítulo 9

Conclusões

9.1 Introdução

Nesta tese propõem-se um modelo de programação genérico e transparente para o desenvolvimento de sistemas tolerantes a falhas e de tempo real, considerando a utilização de componentes de baixo custo. Esta abordagem prima pela possibilidade da aplicação ser desenvolvida, tendo como foco os requisitos de controlo do sistema, abstraindo-se o programador dos detalhes de implementação, da distribuição e da replicação.

Assim sendo, considerou-se que o modelo deveria possuir uma abordagem que permitisse a replicação dos componentes de *software* ou de *hardware* de forma transparente, considerando-se os mecanismos de replicação e de distribuição suportados pela infraestrutura de comunicação. O foco do desenvolvimento da *framework* foi direcionado para a ferramenta de desenvolvimento *Eclipse 4diac™* (4diac, 2019a), desenvolvida para a implementação de aplicações IEC 61499 (IEC 61499, 2012), onde a distribuição poderá ser suportada por interfaces de comunicação (CAN, DeviceNet, Ethernet, PROFInet, entre outras) disponibilizadas no seu repositório de objetos.

A utilização da ferramenta de desenvolvimento *4diac* permite o desenvolvimento de aplicações distribuídas IPMCS portáveis e modulares (Hopsu, 2019) cujo desenvolvimento pode ser efetuado tendo em vista o uso das diversas linguagens de programação da IEC 61131 (IEC 61131, 2013) bem como em C, C++, Java, etc. Se por um lado o *4diac* permite o desenvolvimento de sistemas distribuídos e como tal, o uso pleno dos protocolos de comunicação entre componentes, a replicação de diversos componentes de *software* ou de *hardware* introduzem novos problemas que não foram previstos na IEC 61499 bem como no *Eclipse 4diac™*.

A ferramenta *4diac* foi desenvolvida de acordo com a IEC 61499 especialmente dirigida para o desenvolvimento de aplicações industriais distribuídas. Por este facto, esta disponibiliza, no seu repositório de objetos, uma diversidade de blocos função para a criação de aplicações distribuídas bem como para a implementação de comunicações que garantem as propriedades do protocolo de comunicação *atomic multicast*. As comunicações entre os componentes distribuídos são obtidas pelos standards *Publish/Subscribe* e *Client/Server*, com diferentes interfaces, que possibilitam, num ambiente distribuído, comunicações do tipo *unicast, multicast* ou *broadcast*. No entanto, estas possibilidades de comunicação não são capazes de garantir o determinismo das réplicas pelo que estas poderão processar diferentes conjuntos de dados e em diferentes ordens. Este comportamento torna-se desastroso uma vez que não será possível garantir o determinismo e a respetiva sincronização das réplicas sem que um protocolo adequado ao suporte da replicação e das propriedades de tempo real das mensagens difundidas seja utilizado.

Assim, foi proposta uma *framework* para o desenvolvimento de aplicações tolerantes a falhas e de tempo real garantindo-se a transparência da replicação dos componentes críticos da aplicação. Um conjunto de blocos função foi desenvolvido de modo a realizar as interações básicas do desenvolvimento da aplicação, criando uma interface de implementação entre a aplicação distribuída e a replicação no ambiente distribuído. Os objetivos desenvolvidos são responsáveis pela criação dos requisitos de transparência necessários à implementação da replicação após distribuição da mesma. Neste sentido, o conjunto de objetos propostos poderão garantir não só o sincronismo, mas também, a consolidação dos valores das réplicas. O protocolo de sincronismo proposto garante a tolerância a falhas, em tempo real, a disseminação das mensagens de acordo com o protocolo *multicast* considerando-se a possibilidade de ocorrência de períodos de inoperacionalidade da rede Ethernet.

Na tese são ainda apresentados alguns dos problemas inerentes à implementação desta *framework* tendo em conta a complexidade associada ao suporte de aplicações replicadas e distribuídas, considerando-se uma abordagem genérica e transparente desenvolvida, testada e disseminada no ambiente de desenvolvimento *Eclipse 4diac™*.

9.2 Contributo da investigação

Com o desenvolvimento desta tese pretende-se fornecer alguns contributos que possam ser relevantes para o desenvolvimento de aplicações de tempo real tolerantes a falhas. A implementação de uma abordagem genérica e transparente (considerando o *4diac* e a utilização de componentes de baixo custo) permite que o programador dirija a sua atenção para os requisitos de controlo do sistema, deixando para mais tarde as tarefas de distribuição dos componentes a replicar na aplicação.

A utilização de equipamentos de baixo custo (do tipo Raspberry Pi) com capacidade para correm o FORTE (compilação do desenvolvimento da aplicação no ambiente de desenvolvimento *4diac-IDE*) permitiu desenvolver, num ambiente distribuído e transparente, aplicações replicadas tolerantes a falhas e de tempo real. Com esta abordagem foi possível lidar com requisitos de tempo real, tolerância a falhas, estandardização, transparência, interligação e reutilização, relevantes para o controlo de sistemas computorizados.

A abstração conseguida e aplicada à replicação tem por base o conceito dos blocos função (objetos), considerados como um ou mais componentes, replicados na totalidade ou parcialmente. Este conceito permite que as aplicações sejam configuradas após o seu desenvolvimento, permitindo que estas sejam desenvolvidas sem que os problemas associados à replicação e distribuição sejam considerados. Por outro lado, numa implementação completamente transparente dum sistema replicado e distribuído, há que tomar em conta que a replicação poderá introduzir custos adicionais, dificuldades de implementação e de verificação das propriedades de tempo real e de tolerância a falhas. Assim sendo, o programador deverá desenvolver a aplicação e numa fase posterior, configurar o nível de replicação desejado alocando as tarefas, os recursos ou componentes aos equipamentos do sistema. Isto quererá dizer que durante o desenvolvimento o programador não deve considerar quer a distribuição quer a replicação. A abstração dos problemas de replicação e de distribuição permitem que o foco seja dirigido, unicamente, para os requisitos de controlo do sistema.

Assim, na fase de configuração do sistema a transparência do mesmo só deverá ser considerada ao nível do desenvolvimento da aplicação. O programador tirará partido das ferramentas de configuração do sistema, disponibilizadas pela *framework 4diac*, implementando as interligações entre BFB, CFB, etc. não ficando condicionado pelos detalhes dos mecanismos de implementação dos subníveis de distribuição e de replicação (FB de comunicação e de gestão da replicação). A integração destes blocos função (objetos) são fundamentais para garantir não só a

comunicação entre componentes e o comportamento do sistema, considerando a distribuição e a replicação, mas também, o seu controlo e sua previsibilidade. Os blocos função (FB), constantes do repositório de objetos, possuem diferentes interfaces e requisitos pelo que as aplicações poderão ser reconfiguradas pela substituição ou inserção dos mesmos. Estes fornecerão também, as interfaces intermédias de comunicação e de gestão de dados, da aplicação distribuída, bem como para a implementação dos mecanismos de replicação.

Por outro lado, a replicação e a distribuição, de componentes requererão infraestruturas de suporte *multicast* e mecanismos de consolidação das réplicas. Assim sendo, apresenta-se nesta tese um conjunto de FBs que poderão garantir esses mecanismos enquanto, ao mesmo tempo, a transmissão de mensagens garantirá o determinismo e, como tal, os requisitos de tolerância a falhas e de tempo real. Assim, a difusão das mensagens terá como suporte um conjunto de diferentes protocolos de comunicação que se coadunarão com os diferentes cenários identificados. Por sua vez, para a consolidação das mensagens replicadas é proposto um protocolo de consolidação capaz de implementar os mecanismos de consenso das réplicas.

A garantia das propriedades de tempo real foi assegurada por um conjunto de pré-testes que nos possibilitaram determinar os piores e melhores tempos de resposta das mensagens. Desta análise foi possível definir, *offline*, os tempos de difusão das mensagens geradas durante a execução da aplicação tendo em consideração o *offset* de sincronização dos relógios locais.

9.3 Trabalhos futuros

Os mecanismos de replicação propostos com esta abordagem poderão ser considerados uma abordagem adequada enquanto ferramenta de desenvolvimento de aplicações de controlo distribuído. No entanto, novos desenvolvimentos poderão ser identificados. Assim sendo, poderemos enunciar alguns melhoramentos que poderão ser aplicados aos resultados desta tese bem como apontar trabalhos futuros que poderão ser realizados.

1. Ao longo do seu ciclo de vida o sistema poderá ter a necessidade de sofrer alterações visando a sua operacionalidade, nomeadamente, ao nível das alterações do seu conjunto de requisitos. A *framework* proposta apresenta limitações de reconfiguração das aplicações (ao nível da replicação) uma vez que foi desenvolvida para unicamente duas réplicas. Será interessante prover o repositório de objetos que permitam implementar sistemas mais complexos quer ao nível da replicação quer do consenso. Isto será possível pela adição ao repositório de novos FBs que permitam, pelo menos, implementar a redundância tripla modular (TMR).
2. Os elementos de gestão de réplicas desenvolvidos, para a além de condicionados a dois componentes, poderão ser otimizados. Estes encontram-se implementados segundo CFB que por si só possuem tempos de execução mais elevados do que um *basic* FB (BFB). Assim será interessante desenvolver um módulo de sincronização mais simples, provavelmente com recurso à criação de um ECC que minimize os tempos de execução (WCET) e que disponibilize eventos e dados de saída independentes e de acordo com os tempos de receção e disponibilização das mensagens.
3. Ao nível das comunicações foi necessário realizar uma série de ensaios que definiram os intervalos de tempo de receção das mensagens adotando-se uma postura que é de alguma forma pessimista. Esta abordagem pessimista traduz-se no facto de assumir-se o pior tempo de receção de mensagens (WCRT) em situações em que este tempo pode demonstrar ser demasiado elevado e como tal apresentar elevada probabilidade de não ser atingido. Será interessante analisar os tempos de receção, do ponto de vista de um tempo médio, e, com isso, a probabilidade de perdas de

mensagens durante a fase de difusão, levando à diminuição dos tempos de processamento e de receção da aplicação.

4. Do ponto de vista da configuração da aplicação a replicação e distribuição de um componente é uma tarefa com alguma complexidade uma vez que se pretende uma relação muito estreita entre a eficiência do sistema a sua fiabilidade ou confiabilidade. Por outro lado, deveremos ter ainda em consideração que a distribuição, replicação e alocação dos componentes a diferentes nós se traduzirá numa tarefa complexa uma vez que a simples alteração de um componente poderá alterar todas a características do sistema. Neste sentido, e para além das ferramentas de configuração que a IEC 61499 já prevê, seria interessante desenvolver novos blocos função que pudessem ser englobados num maior número de cenários de replicação. Estas deverão facilitar a reconfiguração dos sistemas mantendo os requisitos de determinismo diminuindo o impacto das alterações no comportamento da aplicação.
5. Poderiam ser estudados outros modos de abordagem aos problemas da replicação em sistemas distribuídos tendo em vista os possíveis ganhos resultantes da aplicação de novos conceitos ou restrições. Nestes deveriam ser analisados os tempos de implementação do sistema, tendo em conta a sua complexidade, tempos de execução e custos associados. Isto poderá ser obtido pela criação de um submenu que possa englobar diversos objetos de replicação e de consolidação que possam ser automaticamente associados ao runtime aquando da compilação do FORTE.

Capítulo 10

Referências Bibliográficas

4diac, Eclipse (2019a). *Open Source for Distributed Industrial Automation* [Online]. [Acedido a 1 mar., 2019]. Disponível em: https://eclipse.org/4diac/.

4diac, Eclipse (2019b). *4diac Repositories* [Online]. [Acedido a 1 mar., 2019]. Disponível em: https://git.eclipse.org/c/4diac/org.eclipse.

4diac, Eclipse (2019c). *Eclipse 4diac Documentation* [Online]. [Acedido a 1 mar., 2019]. Disponível em: https://www.eclipse.org/4diac/en_help.php.

Abdulhameed, Omar Anwer; Resen Israa Abdulameer, Hussain, Saif A.Abd Al (2019). Software Fault Tolerance: A Theoretical Overview. In *International Journal of Simulation Systems, Science & Technology*, Vol. 20(3), pp. 7.1-7.16. DOI 10.5013/IJSSST.a.20.03.07.

Abdulhay; Enas, Elamaran, V.; Arunkumar, N. e Venkataraman, V. (2018). Fault-tolerant medical imaging system with quintuple modular redundancy (QMR) configurations. In *Journal of Ambient Intelligence and Humanized Computing*, Springer. https://doi.org/10.1007/s12652-018-0748-9.

Adelsbach, A. *et al.* (2003). Malicious – and Accidental-Fault Tolerance in Internet Applications: Reference Model and Use Cases. DI/FCUL TR-03-1, Universidade de Lisboa. MAFTIA, Project IST-1999-11583, janeiro.

Ammann, P.E. e Knight, J.C. (1988). Data Diversity: An Approach to Software Fault Tolerance. In *IEEE Transitions on Computers*, Vol. 37, no. 4, pp. 418-425.

April, A., Hayes, J.H., Abran, A. e Dumke, R. (2005). Software Maintenance Maturity Model (SM^{mm}): The software maintenance process model. In *Journal of Software Maintenance and Evolution: Research and Practice,* Vol. 17, pp. 197-223.

Avizienis, A. e Chen, L. (1977). On the Implementation of N-Version Programming for Software Fault-Tolerance During Execution. In *IEEE Computer Society First International Software & Applications Conference (COMPSAC'77)*, Chicago.

Avizienis, A., Laprie, J.-C., Randell, B. e Landwehr Carl (2004). Basic Concepts and Taxonomy of Dependable and Secure Computing. In *IEEE Transactions on Dependable and Secure Computing*, Vol. 1, nº 1, Jan-Mar.

Barrett, P. A., Hilborne, A. M., Bond, P. G., Seaton, D. T., Veríssimo, P., Rodrigues, L. e Speirs, N. A. (1990). The Delta-4 XPA Extra Performance Architecture. In *20th Int. Symposium on Fault-Tolerant Computing Systems*, UK, pp. 481-488.

Beatrice. P. Santos, A. Alberto, T.D.F.M. Lima e F.M.B. Charrua-Santos (2018). INDÚSTRIA 4.0: DESAFIOS E OPORTUNIDADES. In *Revista Produção e Desenvolvimento*, Vol. 4(1), pp. 111-124. [Acedido a 24 fev, 2019]. Disponível em: https://pdfs.semanticscholar.org/5caa/05a3bdfaaee499e68c4ab5c53d5c79d7d9c8.pdf.

Benz, Samuel; Marandi, P. Jalili; Pedone, Fernando e Garbinato, Benoît (2014). Building global and scalable systems with Atomic Multicast. In *Proc. of the 15th International Middleware Conference - Middleware'14*, Bordeaux, France, Dec., pp. 169-180. http://dx.doi.org/10.1145/2578726.2578744.

Bernardi, S., Merseguer, J. and Petriu, D. (2011). A dependability profile within MARTE. In *Software & Systems Modeling*, 10 (3), pp. 313–336.

Berners-Lee, T., e Cailliau, R. (1990). WorldWideWeb: Proposal for a HyperText project [Online]. [Acedido a 11 de jun. 2018]. Disponível em: https://www.w3.org/Proposal.html.

Bhat, Anand; Samii, Soheil e Rajkumar, Ragunathan (2018). Recovery Time Considerations in Real-Time Systems Employing Software Fault Tolerance. In *30th Euromicro Conference on Real-Time Systems (ECRTS 2018)*. No. 23, pp. 23:1–23:22. DOI: 10.4230/LIPIcs.ECRTS.2018.23.

Bijun Li, Nico Weichbrodt, Johannes Behl, Pierre-Louis Aublin, Tobias Distler, and Rüdiger Kapitza1 (2018). Troxy: Transparent Access to Byzantine Fault-Tolerant Systems. In *48th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN-2018),* Luxemburg, pp. 59-70. DOI: 10.1109/DSN.2018.00019.

Black, Geoff e Vyatkin, Valeriy (2010). Intelligent Component-Based Automation of Baggage Handling Systems With IEC 61499. In *IEEE Transactions on Automation Science and Engineering*, Vol. 7, No. 2, pp. 337-351.

Bloomfield, Robin e Lala, Jay (2013). Safety-Critical Systems: The Next Generation. In *IEEE Security & Privacy*, Vol. 11(4), pp. 11-13. DOI**:** 10.1109/MSP.2013.95.

Budhiraja, N., Marzullo, K., Schneider, F.B. and Toueg, S. (1993). The Primary-Backup Approach. In *Distributed Systems*, S. Mullender, ed. Addison-Wesley, pp. 199-216.

Burns, A. (1997). Session Summary: Tasking Profiles. In *8th International Real-Time Ada Workshop*, Ravenscar, England, April. Ada Letters, XVII (5): pp.5-7, ACM Press.

Cabral, J. S. (2006). Organização e Gestão da Manutenção - Conceitos à Prática, 6ª Edição. Lisboa: LIDEL – Edições Técnicas Limitada. ISBN: 9789727574407.

Calleam Consulting Ltd (2008). *Case Study – Denver International Airport Baggage Handling System – An illustration of ineffectual decision making*. [Acedido a 8 jun., 2018]. Disponível em: https://www5.in.tum.de/~huckle/DIABaggage.pdf.

Carzaniga, A., Gorla, A. and Pezz, M. (2009). Handling software faults with redundancy. In R. Lemos, J.-C. Fabre, C. Gacek, F. Gadducci, and M. Beek, editors, *Architecting Dependable Systems VI*. Springer, Vol. 5835 of *Lecture Notes in Computer Science*, pp. 148–171.

Chandra, T. e Toueg, S. (1996). Unreliable failure detectors for reliable distributed systems. In *Journal of the ACM*. Vol. 34(1), pp. 225-267.

Chen, L. e Avizienis, A. (1978). N-Version Programming: A Fault-Tolerance Approach to Reliability of Software Operation. In *8th IEEE International Symposium on Fault tolerand Computing (FTCS-8)*, Toulouse, França.

Chen, Weisheng; Hua, Shaoyong e Zhang, Huaguang (2015). Consensus-Based Distributed Cooperative Learning From Closed-Loop Neural Control Systems. In *IEEE TRANSACTIONS*

ON NEURAL NETWORKS AND LEARNING SYSTEMS, VOL. 26, No. 2, pp. 331-345. DOI: 10.1109/TNNLS.2014.2315535.

Christensen, J. (2018). IEC 61499 - A Standard for Software Reuse in Embedded, Distributed Control Systems [Online]. [Acedido a 13 mai., 2018]. Disponível em: http://www.holobloc.com/papers/iec61499/overview.htm

CMake (2019). O*pen-source build system* [Online]. [Acedido a 4 mar., 2019]. Disponível em: http://www.cmake.org/.

Cristian, F. (1991). Understanding Fault-Tolerant Distributed Systems. In *Communications of the ACM*. Vol. 34(2), pp. 56-78.

Cygwin (2019). [Online]. [Acedido a 4 mar., 2019]. Disponível em: http://cygwin.com/

Dias, G. O. e Melo, Alba C. M. A. (2002). Integrating Optimistic Virtual Synchrony to a CORBA Object Group Service. In *Lecture Notes in Computer Science*, pp. 711–722. DOI: 10.1007/3-540-36124-3_48.

Domokos, P. and Majzik, I. (2005). Design and analysis of fault tolerant architectures by model weaving. In *International Symposium on High-Assurance Systems Engineering (HASE)*, pp. 12-14.

Dubinin, Victor; Voinov, Artem; Senokosov, Ilya e Vyatkin, Valeriy (2018). Implementation of distributed semaphores in IEC 61499 with consensus protocols. In *Proc. of the 16th International Conference on Industrial Informatics – INDIN2018*, Porto, Portugal, pp. 766-771.

Eclipse (2019). The Platform for Open Innovation and Collaboration [Online]. [Acedido a 1 mar., 2019]. Disponível em: https://www.eclipse.org/.

Eletrofun (2019). O que é o Raspberry Pi?. [On line]. [Acedido a 1 mai., 2019]. Disponível em: https://www.electrofun.pt/blog/curso-raspberry-pi-2-o-que-e-o-raspberry-pi/

Elmendorf, W. R. (1972). Fault-Tolerant Programming. In *2nd International Symposium on Fault-Tolerant Computing FTCS-2*, Newton, MA, pp. 79-83.

EN 13306 (2010). European Standard EN 13306:2010. Maintenance - Maintenance terminology, Ref. No. EN13306:2010: E, agosto.

Ferraro (2019). Linux: Instalando, configurando e sincronizando o relógio de servidores e clientes com NTP no Debian, Ubuntu e Windows. [Acedido a 16 de mai. 2019]. Disponível em: http://softwarelivre.org/andre-ferraro/blog/linux-instalando-configurando-e-sincronizando-o-relogio-de-servidores-e-clientes-com-ntp-no-debian-ubuntu-e-windows.

Fiedler, James (2004). Hamming Codes, pp. 1-8. [Acedido a 20 fev., 2019]. Disponível em: https://orion.math.iastate.edu/linglong/Math690F04/HammingCodes.pdf

FIPA (2018). Foundation for Intelligent Physical Agents - Agent Communication Language (FIPA-ACL). [Acedido a 30 mar., 2018]. Disponível em: http://www.fipa.org/specs/fipa00061/SC00061G.pdf

Fischer, M., Lynch, N., e Paterson, M. (1985). Impossibility of Distributed Consensus with One Faulty Process. In *Journal of the ACM*. Vol. 32(2), pp. 374-382.

Gabsi, Wafa e Zalila, Bechir (2015). Towards a Model Level Replication Technique for Fault Tolerant Systems Using AADL. In *16th IEEE/ACIS International Conference on Software Engineering, Artificial Intelligence, Networking and Parallel/Distributed Computing (SNPD2015)*, pp 159-175.

Garcia-Molina, H. and Spauster, A. (1991). Ordered and reliable multicast communication. In *ACM Transactions on Computers Systems*, Vol. 9(3), pp. 242-271, Abril.

Guerraoui, R. e Schiper, A. (1997). Software-based replication for fault tolerance. In *IEEE Computer*, vol. 30(4), pp. 68-74.

Guerraoui, R. e Schiper, A. (1996). Consensus Service: A Modular Approach for Building Agreement Protocols in Distributed Systems. In *26th International Symposium on Fault-Tolerant Computing (FTCS-26)*, pp. 168-177.

Gulay Yalcin, Osman Unsal, Ibrahim Hur, Adrian Cristal, Mateo Valero (2010). FaulTM: Fault-Tolerance Using Hardware Transactional Memory. *Pespma 2010 - Workshop on Parallel Execution of Sequential Programs on Multi-core Architecture*, Jun, Saint Malo, France. ⟨inria-00494285⟩.

Gulland, W. G. (2004). Methods of Determining Safety Integrity Level (SIL), Requirements - Pros and Cons. In *12th Annual Safety-Critical Systems Symposium*, pp. 105–122, Birmingham, UK.

Habinc, Sandi (2002). Functional Triple Modular Redundancy (FTMR). European Space Agency, Report nº: 15102/01/NL/FM(SC) CCN-3.

Hadzilacos, V. e S. Toueg (1993). Fault-Tolerant Broadcasts and Related Problems. In *Distributed Systems*. S. Mullender (Ed.), 2th Ed., ACM Press, New York, ISBN 0-201-62427-3, pp. 97 – 145.

Hall, K. H., Staron, R. J., e Zoitl, A. (2007). Challenges to industry adoption of the IEC 61499 standard on event-based function blocks. In *5th IEEE International Conference on Industrial Informatics (INDIN'07)*, pp. 823–828.

Hallenborg, K. e Demazeau, Y. (2006). Dynamical Control in Large-Scale Material Handling Systems through Agent Technology. In *IEEE/WIC/ACM International Conference on Intelligent Agent Technology (IAT'06)*, pp. 637–645.

Hanisch, H.-M. and Vyatkin, Valeriy (2004). Automation Systems by Using the New International Standard IEC 61499: A Developer's View. *The Industrial Information Technology Handbook*. CRC Press. ISBN: 0-8493-1985-4. Cap. 66. Pp. 66-1 a 66-20.

Hayslip, N., Sastry, S. e Gerhardt, J. (2006). Networked embedded automation. *Assembly Automation*, vol. 26, pp. 235–241.

Hennell, M., Woodcock, J. e Woodward, M. (2006). The safety Integrity Levels of IEC 61508 and a Revised Proposal. In *ESS'06 Embedded Systems Show*.

Hofmann, M., Rooker M. e Zoitl A. (2011). Improved Communication Model for an IEC 61499 Runtime Environment. In *IEEE 16th Conference on Emerging Technologies & Factory Automation (ETFA2011)*, Toulouse, France.

Holanda, J. (2007). Sistemas Distribuídos Tolerantes a Falhas. [Acedido a 6 de abr. 2018]. Disponível em: http://www.lasdpc.icmc.usp.br/disciplinas/pos-graduacao/sistemas-distribuidos/2007/monografias-seminarios/mono_Tolerancia_a_Falhas.doc.

HOLOBLOC, Inc. (2019). *Resources for the New Generation of Automation and Control Software* [Online]. [Acedido a 1 mar., 2019]. Disponível em: http://www.holobloc.com/

Hopsu, Alexander; Atmojo, Udayanto Dwi; Vyatkin, Valeriy (2019). On Portability of IEC 61499 Compliant Structures and Systems. In *IEEE 28th International Symposium on Industrial Electronics (ISIE)*, Vancouver, Canada, pp. 1306-1311. DOI: 10.1109/ISIE.2019.8781290.

Horning J. J. *et al.* (1974). A Program Structure for Error Detection and Recovery. E. Gelenbe and C. Kaiser (eds.), New York: Springer-Verlag, *Lecture Notes in Computer Science*, vol. 16, pp. 171–187.

Hosek, Petr e Cadar, Cristian (2015). VARAN the Unbelievable: An Efficient N-version Execution Framework. In *Twentieth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'15)*, Istambul, pp. 339-353. http://dx.doi.org/10.1145/2694344.2694390.

Hristov, H. e Bo, W. (2015). Safety Critical Computer Systems: Failure Independence and Software Diversity Effects on Reliability of Dual Channel Structures. In *information technologies and control*, Vol 12(2), pp. 9-18. ISSN: 2367-5357. [Acedido a 18 fer., 2019]. Disponivel em: DOI: 10.1515/itc-2015-0011.

Hu, Tingting; Bertolotti, Ivan C. e Navet, Nicolas (2017). Towards seamless integration of N-Version Programming in model-based design. In *22nd IEEE International Conference on Emerging Technologies and Factory Automation (ETFA2017)*, Cyprus, pp. 1-8. DOI: 10.1109/ETFA.2017.8247678.

IEC 61078 (2006). Analysis techniques for dependability – Reliability block diagram and boolean methods, Second edition. International Electrotechnical Commission.

IEC 61131 (2013). Programmable Logic Controllers Part 3 (IEC 61131-3). *International Electrotechnical Commission*, 3th Edition.

IEC 61499 (2012). International Standard IEC 61499-1, Function Block Architecture Part 1, 2th Edition. *International Electrotechnical Commission*.

IEC 61508 (2010). Functional Safety of Electrical, Electronic and Programmable Electronic Safety-Related Systems IEC 61508, part 1 to 7. *International Electrotechnical Commission*.

IEC 62264 (2013). Enterprise-control system integration – Part 1: models and terminology. International Organization for Standardization IEC 62264-1. *International Electrotechnical Commission.*

ISaGRAF (2019). *Rockwell Automation* [Online]. [Acedido a 1 mar., 2019]. Disponível em: http://www.isagraf.com/index.htm.

Jalote, P (1994). Fault tolerance in distributed systems. Prentice Hall, Englewood Cliffs, New Jersey. ISBN: 0-13-301367-7.

Jian Liu; Wenting Li; Ghassan O. Karame e N. Asokan (2019). Scalable Byzantine Consensus via Hardware-Assisted Secret Sharing. In *IEEE Transactions on Computers*, Vol. 68, Issue: 1 , pp. 139-151.

Johnson, G. W. and Jennings, R. (2006). LabVIEW Graphical Programming. New York: McGraw-Hill.

Kagermann, Henning; Wahlster, Wolfgang e Helbig, Johannes (2013). Final report of the Industry 4.0 Working Group. "ACATECH: Recommendations for implementing the strategic initiative INDUSTRIE 4.0".

Khalgui, Mohamed; Rebeuf, Xavier, Simonot-Lion, Françoise (2005). A Schedulability Analysis of an IEC-61499 Control Application. In *6th IFAC International conference on Fieldbus Systems and their Applications (FeT'2005)*, pp. 71-78.

Kim, K. H. (1995). The Distributed Recovery Block Scheme. In *Software Fault Tolerance*, M. R. Lyu (Ed.), New York: John Wiley & Sons.

Kim, K. H. e Welch H. O. (1989). Distributed Execution of Recovery Blocks: An Approach to Uniform Treatment of Hardware and Software Faults in Real-Time Applications. In *IEEE Transactions on Computers*, Vol. 38(5), pp. 626-636.

Kitechenham, Barbara e Neumann, Bernard (1991). Cost Modelling and Estimation. In *Paul Rook - Software Reliability Handbook. 2ª Edição*, Londres. ISBN: 1-85166-400-9, pp. 333-376.

Koenig, Frank; Found, P. Anne e Kumar, Maneesh (2019). Innovative airport 4.0 conditionbased maintenance system for baggage handling DCV systems. In *International Journal of Productivity and Performance Management*, Vol. 68(3), pp. 561-577, https://doi.org/10.1108/IJPPM-04-2018-0136.

Kopetz, H., Damm, A., Koza, C., Mulazzani, M., Schwabl, W., Senft, C. e Zainlinger, R. (1989). Distributed Fault-Tolerant Real-Time Systems: The Mars Approach. In *IEEE Micro*, Vol. 9(1), pp. 25–41.

Koubias, S. A. e Papadopoulos, G. D. (1995). Modern fieldbus communication architectures for real-time industrial applications. In *Computers in Industry*, vol. 26, Elsevier Science, pp. 243-252.

Kuvaiskii, Dmitrii; Oleksenko, Oleskii; Bhatotia, Pramod; Felber, Pascal e Fetzer, Christof (2016). ELZAR: Triple Modular Redundancy using Intel AVX (Practical Experience Report). In *46th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, pp. 646-653. DOI 10.1109/DSN.2016.65

Lamport, L. (1997). How to Make a Correct Multiprocess Program Execute Correctly on a Multiprocessor. In *IEEE Transactions on Computers*. Vol. 46(7), pp. 779-782. Jul.

Lamport, L., Shostak, R., e Pease, M. (1982). The Byzantine Generals Problem. In *ACM Trans. Programming Languages and Systems*, vol. 4, no. 3, pp. 382-401.

Lamport, Leslie (1978). Time, Clocks and the Ordering of Events in a Distributed System. In *Communications of the ACM*, 21(7), pp. 558-565.

Laprie, J. C., Arlat, J., Beounes, C. e Kanoun, K., (1990). Definition and Analysis of Hardware- and Software-Fault-Tolerant Architectures. In *IEEE Computer*. Vol. 23(7), pp. 39-51.

Laprie, J.C., Arlat, Jean; Béounes, Christian e Kanoun, Karma (1995). Architectural Issues in Software Fault Tolerance. In *Software Fault Tolerance*, Michael R. Lyu, editor, Wiley, pp. 47-80.

Lasnier, G., Robert, T., Pautet, L., and Kordon, F. (2010). Behavioral modular description of fault tolerant distributed systems with AADL behavioral annex. In *10th Annual International Conference on New Technologies of Distributed Systems (NOTERE)*, pp. 17-24.

Lednicki, Luka, Carlson, Jan e Sandstrom, Kristian (2013). Model level worst-case execution time analysis for IEC 61499. In *Proc. of the 16th International ACM SIGSOFT Symposium on Component-Based Software Engineering (CBSE'2013)*, Canada.

Lee, P. A. e Anderson, T. (1990). Fault tolerance, principles and practice. 2nd Edition. New York: Springer-Verlag. ISBN 0387820779.

Li H., Feng X., Shi S., Zheng F., Xie X. (2015) A High-Accuracy Clock Synchronization Method in Distributed Real-Time System. In: Xu W., Xiao L., Li J., Zhang C., Zhu Z. (eds) Computer Engineering and Technology. NCCET 2014. Communications in Computer and Information Science, vol 491, pp. 148-157. Springer, Berlin, Heidelberg.

Liew, Daniel; Schemmel, Daniel; Cadar, Cristian; Donaldson, Alastair F.; Zähl, Rafael e Wehrle, Klaus (2017). Floating-Point Symbolic Execution: A Case Study in N-Version Programming. In *32nd IEEE/ACM International Conference on Automated Software Engineering (ASE-2017)*, pp. 601-612.

Mahalik, N.G.P.C. e Lee, S.K. (2002). A study on production line automation with LonWorkskTM control networks. In *Computer Standards & Interfaces*, vol. 24, Elsevier Science, pp. 21–27.

Marik, V., Vrba Pavel, Hall, K. H. e Maturana F. P. (2005). Rockwell automation agents for manufacturing. In *4th Int. Joint Conference on Autonomous Agents and Multiagent Systems (AAMAS'05)*, pp. 107–113.

Marques, J. A. e Guedes, P. (2003). Tecnologia de Sistemas Distribuídos. Lisboa: FCA - Editora de Informática Lda. ISBN: 978-972-722-128-8.

MinGW (2019). MinGW.org - Minimalist GNU for Windows. [Acedido a 4 mar., 2019]. Disponível em: http://www.mingw.org/.

Myers, Glenford J. (1976). Software Reliability Principles and Practices. New York: John Wiley & Sons. ISBN: 0-471-62765-8. pp. 3-14.

Neumann, Peter (2007). Communication in industrial automation - What is going on? In *Control Engineering Practice*, Vol. 15, pp. 1332–1347.

Nguyen, Dong e Liu, Dar-Biau (1998). Recovery Blocks in Real-Time Distributed Systems. In *IEEE Annual Reliability and Maintainability Symposium*, pp. 149–154.

Niz, Dionisio e Feiler, Peter H. (2009). Verification of Replication Architectures in AADL. In *14th IEEE International Conference on Engineering of Complex Computer Systems*. Pp. 365-370. DOI: 10.1109/ICECCS.2009.18.

NP EN 15341 (2009). Norma Portuguesa NP EN 15341:2009. Manutenção – Indicadores de desempenho da manutenção (KPI). Instituto Português da Qualidade, novembro.

NP EN 29000 (1994). Norma Portuguesa NP EN 29000-3:1994. Desenvolvimento, Fornecimento e Manutenção de «Software». Instituto Português da Qualidade, março.

O'Connor, Pratick D. T. (2012). Practical Reliability Engineering, 5rd Edition. Chichester: John Wiley & Sons Ltd. ISBN: 978-0470979815.

OSCAT (2019). *Open Source Community for Automation Technology* [Online]. [Acedido a 1 mar., 2019]. Disponível em: http://www.oscat.de

Ongaro, D. e Ousterhout, J. (2014). In search of an understandable consensus algorithm. In *2014 USENIX Annual Technical Conference*, pages 305–319.

Peng, Hao e Yang, Fan (2015). Fault Tolerant Global Scheduling for Multiprocessor Hard Real Time Systems. In *International Conference on Information Sciences, Machinery, Materials and Energy (ICISMME 2015)*. Pp. 1588-1596.

Pereira, F. J. Didelet (1996). Modelos de Fiabilidade em Equipamentos Mecânicos, Porto: FEUP. Tese de Doutoramento.

Pfeiffer, Olaf, Ayre, Andrew e Keydel, Christian (2008). Embedded networking with CAN and CANopen. 2ª Ediçao. San Clemente, CA: RTC Books. ISBN: 978-0-9765116-2-5.

Pinho, Paulo R., Rech, Luciana de Oliveira, Lung, Lau Cheuk, Correia, Miguel e Camargos, Lásaro Jonas (2016). Replicação de Máquina de Estado Baseada em Prioridade com PRaft. In *XXXIV Simpósio Brasileiro de Redes de Computadores e Sistemas Distribuídos*. Salvador, Brazil.

Pinho, L. (2001). A Framework for the Transparent Replication of Real-Time Applications. Porto: FEUP. Tese de Doutoramento.

Pinho, L., Vasques, F. e Wellings, A. (2004). Replication Management in Reliable Real-Time Systems. In *Real-Time Systems Journal*, vol. 26, pp. 261-269.

Poledna, S. (1994). Replica Determinism in Distributed Real-Time Systems: A Brief Survey. In *Real-Time Systems*, Vol. 6(3), pp. 289-316.

Poledna, S., Burns, A. Wellings, A. e Barret, P (2000). Replica Determinism and Flexible Scheduling in Hard Real-Time Dependable Systems. In *IEEE Transactions on Computers*. Vol. 49(2), pp. 100-111.

Pomales, Wilfredo Torres (2000). Software Fault Tolerance: A Tutorial. Langley Research Center, Hampton, Virgínia. NASA/TM-2000-210616.

Powell, David (Ed.). (2001). A Generic Fault-Tolerant Architecture for Real-Time Dependable Systems. Kluwer Academic Publishers. ISBN 978-1-4757-3353-2.

Powell, David *et al*. (1991). Delta-4 – A Generic Architecture for Dependable Distributed Computing. ESPRIT Research Reports, Springer Verlag.

Proença, Hugo (2003). *MARCS - Sistema Multi-Agente para Controlo de Tráfego Ferroviário*. Dissertação de Mestrado. Faculdades de Ciências, Economia e Engenharia da Universidade do Porto. Disponivel em: http://www.di.ubi.pt/~hugomcp/doc/tese.pdf.

Pteris Global Limited (2019). Our Solutions. http://www.pterisglobal.com/sol_systems.html. [Acedido a 15 junho, 2019].

Pullum, Laura L. (2001). Software Fault Tolerance Techniques and Implementation. Artech House, Inc. ISBN: 1-58053-137-7.

Randell, Brain e Xu, Jie (1994). The Evolution of the Recovery Block Concept. In *Software Fault Tolerance*. Lyu, M. (ed). Michigan: John Wiley & Sons Ltd. ISBN: 0-471-95068-8, pp. 1-25.

Rausand, Marvin e Høyland, Arnljot (2004). System reliability theory: models, statistical methods, and applications – 2nd Ed., John Wiley & Sons, Inc. ISBN: ISBN 0-471-47133-X.

Ribas, Carlos (2017). Indústria 4.0, a quarta revolução industrial. In Casa Eficiente – o futuro hoje. 11 Julho. [Acedido a 24 fev, 2019]. Disponível em: https://casaeficiente.com/2017/07/16/industria-4-0-a-quarta-revolucao-industrial/.

Ribeiro, L., Barata, J., Ferreira, B. e Pires J. (2008). An Architecture for a Fault Tolerant Highly Reconfigurable Shop Floor. In *6th IEEE International Conference on Industrial Informatics (INDIN 2008)*, pp. 1194-1199.

Rierson, L. (2017). Developing Safety-Critical Software: A Practical Guide for Aviation Software and DO-178C Compliance. CRC Press. ISBN: 9781351834056.

RNP (2000). Implementando o serviço NTP na sua rede local, versão 1.0. CAIS - Centro de Atendimento a Incidentes de Segurança, agosto de 2000. [On line].[Acedido a 16 de mai. 2019]. Disponível em: http://www.pop-go.rnp.br/docs/manual_ntp_v1b.pdf.

Rodrigues, J. C. (2008). A Fault Tolerant Distributed Control System for a Manufacturing Cell. In *8th Portuguese Conference on Automatic Control (CONTROLO'08),* Vila Real, Portugal.

Santos, Adriano A. (2001). Análise de Fiabilidade de Sistemas Informáticos, Porto: FEUP. Dissertação de Mestrado.

Santos, Adriano A. e Sousa, Mário de (2008). Framework for Management of Replicated IEC 61499 Applications. In *13th IEEE International Conference on Emerging Technologies and Factory Automation (ETFA 2008)*. Hamburgo, pp. 200-206.

Santos, Adriano A. e Sousa, Mário de (2010). Replication in Distributed Systems using IEC 61499 Standard. In *15th IEEE International Conference on Emerging Technologies and Factory Automation (ETFA 2010).* Bilbao, pp. 1-8.

Santos, Adriano A. e Sousa, Mário de (2018). Replication Strategies for Distributed IEC 61499 Applications. In *Proc. of the 44th Annual Conference of the IEEE Industrial Electronics Society – IECON2018*, Washington, DC, USA, pp. 2225-2230.

Santos, Adriano A., Silva, António F. da, Magalhães, Pessoa e Sousa, Mário de (2018). An IEC 61499 Replication for Distributed Control Applications. In *Proc. of the 16th International Conference on Industrial Informatics – INDIN2018*, Porto, Portugal, pp. 362-367.

Sari, A. and Akkaya, M. (2015). Fault Tolerance Mechanisms in Distributed Systems. In *International Journal of Communications, Network and System Sciences*, Vol. 8, pp. 471-482. DOI: 10.4236/ijcns.2015.812042.

Schneider, F. (1990). Implementing Fault-Tolerant Services Using the State Machine Approach: A Tutorial. In *ACM Computing Surveys*, Vol. 22(4), pp. 299-319.

Schütz, Daniel; Wannagat, Andreas; Legat, Christoph e Vogel-Heuser, Birgit (2013). Development of PLC-Based Software for Increasing the Dependability of Production Automation Systems. In *IEEE Transactions on Industrial Informatics*, Vol. 9, Issue 4, pp. 2397–2406. DOI: 10.1109/TII.2012.2229285.

Siemens (2011). Siemens to extend baggage handling system at Munich Airport T2 [Online]. [Acedido 10 mar., 2018]. Disponível em: http://www.siemens.com/press/en/pressrelease/?press=/en/pressrelease/2011/mobility/imo201108030.htm.

Simulink (2019). Simulation and Model‑Based Design, The MathWorks Inc., Natick [Online]. [Acedido a 12 jun.,2019]. Disponível em: https://www.mathworks.com/products/simulink.html?s_tid=hp_products_simulink.

Sinca, R. e Szász, CS. (2017). FAULT-TOLERANT DIGITAL SYSTEMS DEVELOPMENT USING TRIPLE MODULAR REDUNDANCY. In *International Review of Applied Sciences and Engineering,* Vol. 8, Issue 1, pp. 3–7. DOI: 10.1556/1848.2017.8.1.2.

Sousa, Mário de (2014). Guaranteeing Replica Determinism on IEC 61499. In *Proc. of 19th IEEE International Conference on Emerging Technologies and Factory Automation (ETFA 2014)*, Barcelona, Spain.

Sousa, Mário de e Santos, Adriano A. (2007). Management of Replicated IEC 61499 Applications. In *5th IEEE International Conference on Industrial Informatics (INDIN'07)*, Vienna, Austria, pp 231-236.

Sousa, Mário de; Chrysoulas, Christos e Homay, Aydin E. (2015a). Multiply and Conquer: A Replication Framework for Building Fault Tolerant Industrial Applications. In *13th IEEE International Conference on Industrial Informatics (INDIN'15)*, pp. 1342-1347.

Sousa, Mário de; Chrysoulas, Christos e Honay, Aydin E. (2015b). Building Fault Tolerant Industrial Applications Based on IEC 61499. In *25th International Conference on Flexible Automation and Intelligent Manufacturing (FIAM2015)*, Wolverhampton, UK.

Stemmer, Marcelo Ricardo (2001). Das 5331 – Sistemas Distribuídos e Redes de Computadores para Controle e Automação Industrial. [Acedido a 6 de Jan. 2019]

Disponível em : http://alvarestech.com/temp/simprebal/Relatorios_Tecnicos-Publicacoes-Dissertacoes/docs/cursos/Aula6-Apostila-Sistemas_Distribuidos_E_Redes_De_Computadores_Para_Controle.pdf

Storey, Neil (1996). Safety Critical Computer Systems. Addison-Wesley, Pearson/Prentice Hall. ISBN: 978-0-201-42787-5.

Strasser, T., Rooker, M., Ebenhofer, G., Zoitl, A., Sunder, C., Valentini, A., Martel, A. (2008). Framework for Distributed Industrial Automation and Control (4DIAC), *6th IEEE International Conference on Industrial Informatics (INDIN'08)*, pp.283-288, 13-16 July.

Strasser, T.; Müller, I.; Sünder, C.; Hummer, O.; Uhrmann, H. (2006). Modeling of Reconfiguration Control Applications based on the IEC 61499, Reference Model for Industrial Process Measurement and Control Systems", *IEEE 2006 Workshop on Distributed Intelligent Systems (DIS'06)*, pp. 127-132.

Tabbaa N., Entezari-Maleki R., Movaghar A. (2011) A Fault Tolerant Scheduling Algorithm for DAG Applications in Cluster Environments. In: *Snasel V., Platos J., El-Qawasmeh E. (eds) Digital Information Processing and Communications. ICDIPC 2011. Communications in Computer and Information Science*, vol. 188, pp. 189-199. Springer, Berlin, Heidelberg.

Takaesu, Kohtaro e Yoshida, Takeo (2004). Construction of a Fault-Tolerant Voter for N-Modular Redundancy. In *Electronics and Communications*. Japão, Vol. 87(2), nº 12, pp. 62-71.

Tambe, S., Dabholkar, A. e Gokhale, A. (2009). Fault-tolerance for Component-based Systems - An Automated Middleware Specialization Approach. In *12th IEEE International Symposium on Object/Component/Service-Oriented Real-Time Distributed Computing (ISORC2009)*, pp: 47-54.

Tanembaum, A. S. e Steen, M. V. (2006). Distributed Systems: Principles and Paradigms. 2ª Edição. Prentice Hall. ISBN: 0132392275.

Tian, Jeff (2005). Software Quality Engineering: Testing, Quality Assurance, and Quantifiable Improvement, John Wiley and Sons, pp. 272-275.

Veríssimo, P. (1996). Segurança e Confiabilidade: na ordem do dia dos sistemas distribuídos. In *Jornadas do Colégio de Engenharia Electrotécnica*, Ordem dos Engenheiros, Lisboa-IST.

Veríssimo, P. e Lemos, R. (1989). Confiança no Funcionamento: Proposta para uma terminologia em Português. Technical Report RT48-89, INESC.

Veríssimo, P. e Rodrigues, L. (2001). Distributed Systems for System Architects. Kluwer Academic Publishers. ISBN: 0-7923-7266-2.

Vickers, K e Chinn, R. W. (1998). Passenger terminal baggage handling systems. In *IEEE Colloquium on Systems Engineering of Aerospace Projects*, Digest No. 1998/249, pp. 6/1-6/7.

Visual Studio, VS (2019), Microsoft [Online]. [Acedido a 4 mar., 2019]. Disponível em: https://visualstudio.microsoft.com/pt-br/.

Vrba, Pavel (2003). MAST: Manufacturing agent simulation tool. In *IEEE Conference on Emerging Technology Factory Automation (ETFA'03)*, vol. 1, pp. 282–287.

Vyatkin, Valeriy (2009). The IEC 61499 Standard and Its Semantics. In *IEEE Industrial Electronics Magazine*, Vol. 3(4), pp. 40-48.

Vyatkin, Valeriy (2015). IEC 61499 Function Blocks for Embedded and Distributed Control Systems Design – Third Edition, Instrumentation Society of America (ISA). ISBN: 978-1-941546-72-7.

Weber, Taisy Silva (2009). Um roteiro para exploração dos conceitos básicos de tolerância a falhas [Online]. Universidade Federal do Rio Grande do Sul. [Acedido a 9 de jul. 2019] Disponível em: http://www.inf.ufrgs.br/~taisy/disciplinas/textos/Dependabilidade.pdf.

Wei Luo, FuMin Yang, Gang Tu, LiPing Pang, Xiao Qin (2007). TERCOS: A Novel Technique for Exploiting Redundancies in Fault-Tolerant and Real-Time Distributed Systems. In *13th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA 2007)*. Daegu, South Korea. DOI: 10.1109/RTCSA.2007.70.

Wellings, A. J., Beus-Dukic, Lj., e Powell, D. (1998). Real-Time Scheduling in a Generic Fault-Tolerant Architecture. In *19th IEEE Real-Time Systems Symposium*, Madrid, Spain, pp. 390-298.

Wenger, M., Zoitl, A. and Müller, T. (2018). Connecting PLCs with their Asset Administration Shell for Automatic Device Configuration. In *IEEE 16th International Conference on Industrial Informatics (INDIN2018)*, Porto, Portugal, pp. 74-79.

Xie, Min; Xiong, Chengjie e Ng, Szu-Hui (2014). A study of N-version programming and its impact on software Availability. In *International Journal of Systems Science*, Vol. 45, No. 10, pp. 2145–2157. http://dx.doi.org/10.1080/00207721.2013.763299.

Xie, Zaipeng; Sun, Hongyu e Saluja, Kewal (2009). A Survey of Software Fault Tolerance Techniques. [Acedido a 9 de jun. 2018]. Disponível em: http://www.pld.ttu.ee/IAF0030/Paper_4.pdf.

Yadav, Pooja; Singh, Shilpa e More, Ajay (2016). A Proposed System for Real Time Adaptive N Version Programming. In *International Journal of Computer Applications (0975 – 8887),* Vol. 136, No.4, February 2016.

Yan, Jeffrey and Vyatkin, V. (2011). Distributed Execution and Cyber-Physical Design of Baggage Handling Automation with IEC 61499. In *9th IEEE International Conference on Industrial Informatics (INDIN'11)*, Lisboa, Portugal, pp. 573-578.

Yang, M., Hua, G., Feng, Y. e Gong, J. (2017). Fault-Tolerance Techniques for Spacecraft Control Computers. John Wiley & Sons Singapore. ISBN: 9781119107279.

Zheng, Zibin e Lyu, Michael R. (2015). Selecting an Optimal Fault Tolerance Strategy for Reliable Service-Oriented Systems with Local and Global Constraints. In *IEEE Transactions on Computers,* Vol. 64(1), pp. 219-232. DOI: 10.1109/TC.2013.189.

Zimmermann, Hubert (1980). OSI Reference Model – The ISO Model of Architecture for Open Systems Interconnection. In *IEEE Transactions on Communications*, Vol. 28(4), pp. 425-432, abril.

Zoitl, A.; Strasser, T. and Valentini, A. (2010). Open Source Initiatives as basis for the Establishment of new Technologies in Industrial Automation: 4DIAC a Case Study. *Mechatronics*, pp. 3817-3819.

Capítulo 11

Anexos

Anexo A – Sincronização dos relógios dos RPis (NTP)

Os RPis são minicomputadores que não possuem baterias internas e, como tal, algumas informações como a data e hora não se mantêm atualizadas. Assim, cada vez que um RPi é ligado apresenta a informação que ficou registada na última utilização pelo que, apresentarão discrepâncias temporais entre eles.

A sincronização dos relógios dos RPis obriga à instalação de um servidor local do tipo NTP, ou outro, que definirá a data e hora de todos os equipamentos ligados à rede, condição essencial para a obtenção do determinismo das réplicas. A criação do servidor local de NTP é realizada de acordo com o seguinte procedimento:

Instalação do SERVIDOR NTP (*sistema Linux*):

1 Obter a última versão do NTP a partir de: *www.ntp.org*;
2 Descompactar o arquivo ntp-<versão_atual>.tar.gz e seguir os passos indicados no arquivo *install* contido na distribuição que se resumem aos seguintes comandos (RNP, 2000):

```
$ cd <diretório-de-descompactação-NTP>
$ ./configure
$ make
$ make check
$ make install (como usuário root)
ou simplesmente,
$ apt-get install ntp ntpdate (como usuário root)
```

3 A configuração do server é definida no arquivo `/etc/ntp.conf` que contem as opções de configuração;
4 Criar o arquivo `drift` sem conteúdo. Por defeito o arquivo deverá ser `/etc/ntp.drift` chamado pelo arquivo de configuração;
5 O *daemon ntpd* pode ser inicializado usando os seguintes comandos:

```
$ /etc/init.d/ntp start
ou
$ service ntp start
```

6 Para verificar se o serviço NTP se encontra corretamente inicializado pode-se utilizar o comando `$ ntpq –p`. A inicialização bem-sucedida apresentará, por exemplo, a saída:

```
     Remote           refid      st t when poll reach   delay   offset  jitter
==============================================================================
*t1.time.ir2.yah  212.82.106.33    2 u  648 1024  377  61.607  -2.398  32.090
+ntp1.unixcraft.  193.52.184.106   2 u  539 1024  377  53.176  -1.376   0.933
-165.22.17.7 (ft  194.8.30.16      3 u  350 1024  377  22.527   1.085   9.180
+ntp02.oal.ul.pt  193.190.230.65   2 u  680 1024  377  21.438  -4.026  34.365
```

* – *system peer*, servidor escolhido como principal referência;
\+ – *candidat*, servidor utilizado, porém com menor peso;
\- – *outlier*, servidor funcional, porém geograficamente distante. A distância pode afetar na sincronização;
offset – medidas de deslocamento (em milissegundos), é a diferença de tempo entre nosso servidor NTP e o servidor de referência*;*

Instalação do NTP CLIENTE (*sistema Linux*):

1 A instalação de clientes NTP Linux é igual ao processo de configuração do servidor, ou seja, temos que instalar os pacotes *ntp* e o *ntpdate*:

```
$ apt-get install ntp ntpdate
```

2 Criar o arquivo *ntp.drift* (caso não exista);

```
$ touch /var/lib/ntp/ntp.drif
```

3 Configurar o arquivo */etc/ntp.conf* do cliente. Pode-se remover todas as configurações ou simplesmente acrescentar o necessário, um exemplo de configuração pode ser (Ferraro, 2019):

```
driftfile /var/lib/ntp/ntp.drift
statsdir /var/log/ntpstats/
statistics loopstats peerstats clockstats
filegen loopstats file loopstats type day enable
filegen peerstats file peerstats type day enable
filegen clockstats file clockstats type day enable
server <ip-do-servidor-NTP> maxpoll 4 (4-16s, 10-1024s)
restrict 127.0.0.1
restrict <ip-do-servidor-NTP>
restrict ::1
```

4 Fazer atualização da data e da hora do sistema:

```
$ /etc/init.d/ntp stop
$ ntpdate <ip-do-servidor-NTP>
$ /etc/init.d/ntp start
```

5 Exemplo de ficheiro */etc/ntp.conf* do servidor:

```
driftfile /var/lib/ntp/ntp.drift

statsdir /var/log/ntpstats/
statistics loopstats peerstats clockstats
filegen loopstats file loopstats type day enable
filegen peerstats file peerstats type day enable
filegen clockstats file clockstats type day enable

server 0.debian.pool.ntp.org iburst
server 1.debian.pool.ntp.org iburst
server 2.debian.pool.ntp.org iburst
server 3.debian.pool.ntp.org iburst

restrict -4 default kod notrap nomodify nopeer noquery
restrict -6 default kod notrap nomodify nopeer noquery
restrict 127.0.0.1
restrict ::1
restrict 192.168.123.0 mask 255.255.255.0 notrust
```

O ficheiro de registo ***ntp.drift*** é utilizado para registar a instabilidade na frequência do relógio do seu computador. O *NTP* utiliza este registo para sincronizar corretamente o relógio do seu equipamento com o do servidor de referência.

***Nota**: Para que a instalação dos clientes NTP nos equipamentos remotos possa ser possível estes devem estar ligados a uma rede com acesso à internet.*
*A sincronização dos relógios **só será realizada passados ±5 minutos**. Há a necessidade de aguardar-se por este tempo para que se complete a sincronização de todos os equipamentos.*

Anexo B – Simulação de falha do sistema replicado

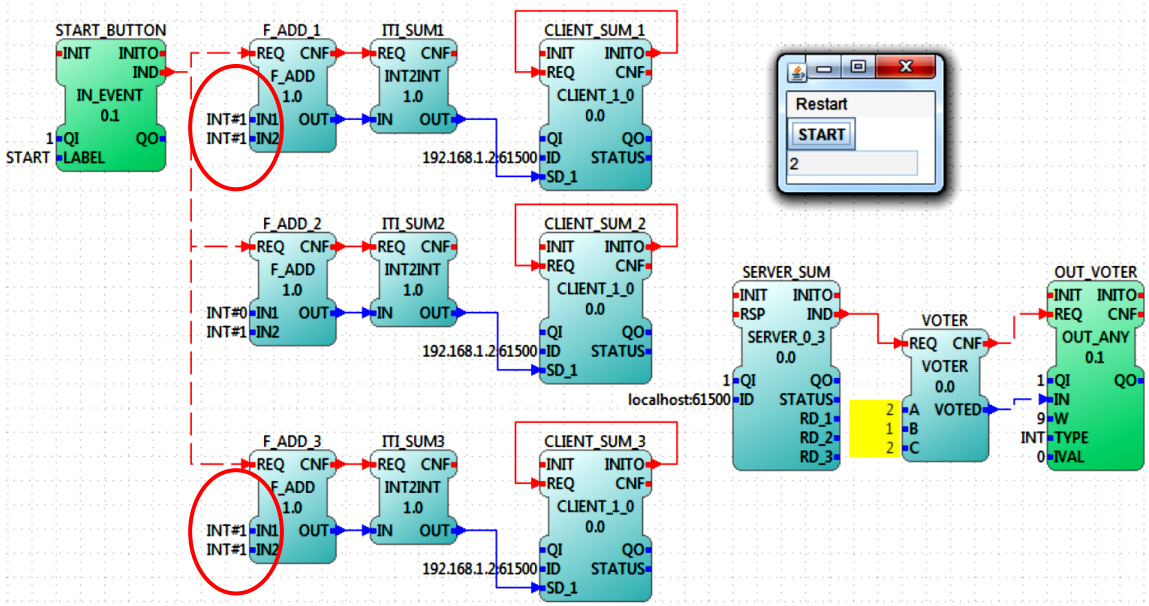


Figura 11-1. Alteração dos valores predefinidos, simulação de falha do algoritmo ADD

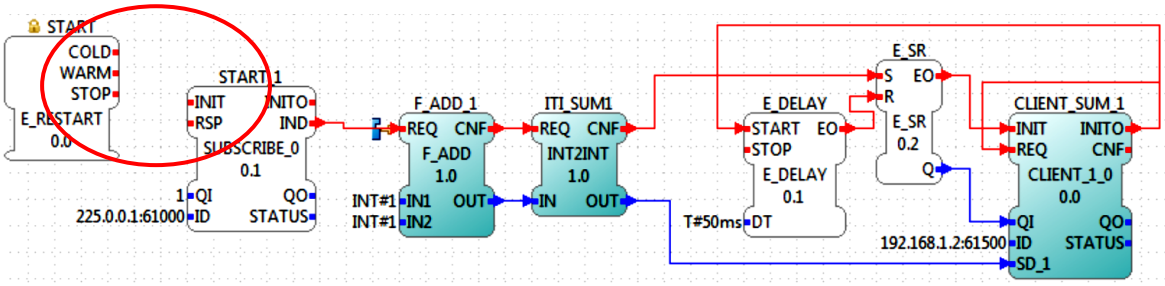


Figura 11-2. Quebra da ligação do HMI/recurso 1, inibição da inicialização do Subscribe START_1

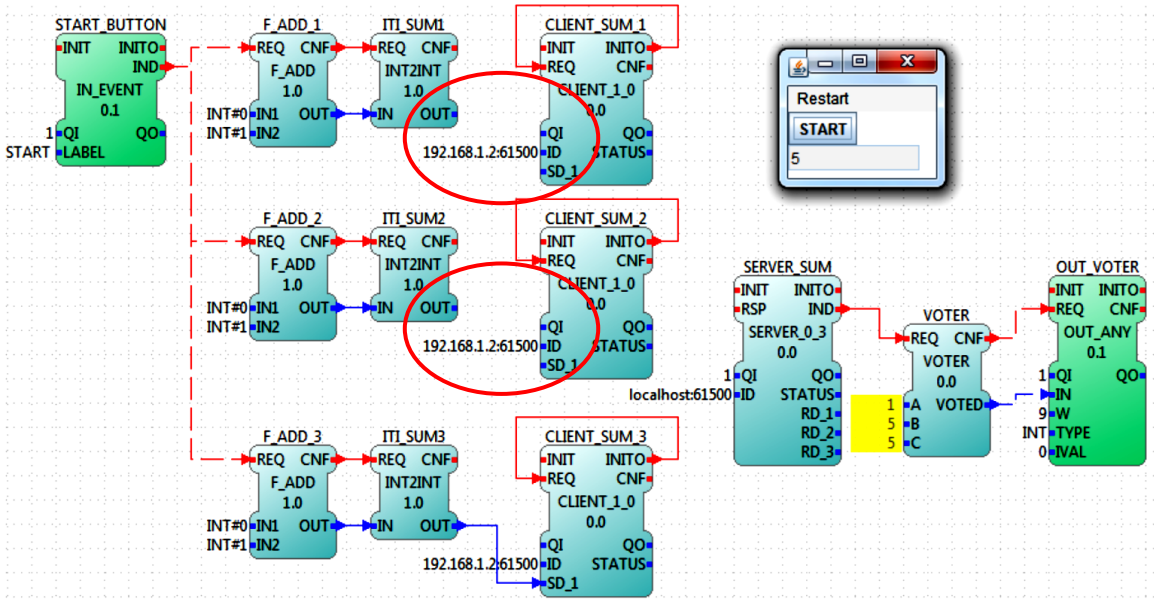


Figura 11-3. Quebra das ligações dos recursos 1 e 2, inibição do envio de dados para o SERVER_SUM

Anexo C – Eclipse 4diac™ - Open Source para Automação Industrial Distribuída

Introdução

A automação industrial é normalmente caracterizada pela proliferação de diferentes plataformas que, embora operando em simultâneo, não garantem interoperabilidade entre as diferentes soluções dos diferentes fornecedores. Se por um lado a norma IEC 61131-3 tenta fornecer bases, que pretendem ser comuns, ao desenvolvimento de sistemas de controlo estas esbarram nas diversas plataformas que não são capazes de se interligarem para operarem em conjunto. Por outro verifica-se que existem várias entidades que desenvolveram esforços no sentido de resolver esta situação apontando como objetivo final a portabilidade, a configurabilidade e a interoperabilidade dos sistemas. Neste sentido, surge a plataforma de desenvolvimento *open-source Eclipse 4diac™* (4diac, 2019a) que visa a criação e execução de aplicações baseadas nos padrões preconizados pela norma IEC 61499.

É com base nos pressupostos enunciados anteriormente que iremos descrever o funcionamento da ferramenta 4DIAC, realçando as suas capacidades e limitações. Neste capítulo realçar-se-á as duas ferramentas de código aberto (*4DIAC-IDE* e *4DIAC-RTE*) que estão no foco da iniciativa *Eclipse 4diac™*.

4DIAC-IDE

O 4DIAC-IDE (*Integrated Development Environment*) é uma ferramenta que permite a edição, o desenvolvimento, importação e exportação de aplicações de controlo distribuído com base na norma IEC 61499. As aplicações podem ser distribuídas pelos diversos recursos e dispositivos de acordo com os meios definidos na norma. Para além destas capacidades a aplicação permite ainda transferir os sistemas desenvolvidos para o *runtime* dos equipamentos onde vão ser executados. O 4diac tem como base de funcionamento a *framework Eclipse* (Eclipse, 2019) o que permite uma fácil integração com outros *plug-ins* que oferecem novas ou adicionais funcionalidades.

O 4DIAC-IDE é obtido gratuitamente através da página oficial da iniciativa (4diac, 2019a), sendo basicamente um editor de componentes (FBs, *Function Blocks*). A versão executável disponível à data de escrita da tese é a Version: 1.10.1, no entanto, outras versões, atualizadas quase diariamente, podem ser encontradas no repositório *4diac* da iniciativa (4diac, 2019b). O código fonte disponível no repositório *4diac* pode ser obtido com a ajuda da ferramenta *TortoiseHG*, para os sistemas operativos Windows, usando-se a opção “clone”, ou através do *Mercurial*, para os sistemas operativos baseados em Linux, por meio da linha de comando "$hg". A instalação de 4DIAC-IDE é independente do sistema operativo utilizado, no entanto, para o executar é necessária a ferramenta Java, versão 8, ou posterior para a versão 1.8.4 (versão utilizada para o desenvolvimento da *framework*), segundo a documentação de apoio *Eclipse 4diac* Help (2019c).

Note-se, no entanto, que para se utilizar o código fonte é necessário instalar a ferramenta *Eclipse* com o "Modeling Tools" (ver Anexo H). Como consequência da execução da aplicação "*eclipse*" é gerada a aplicação 4DIAC-IDE da qual se apresenta na Figura 11-4 uma visão geral da interface de edição, utilização do código fonte da aplicação obtido no repositório.

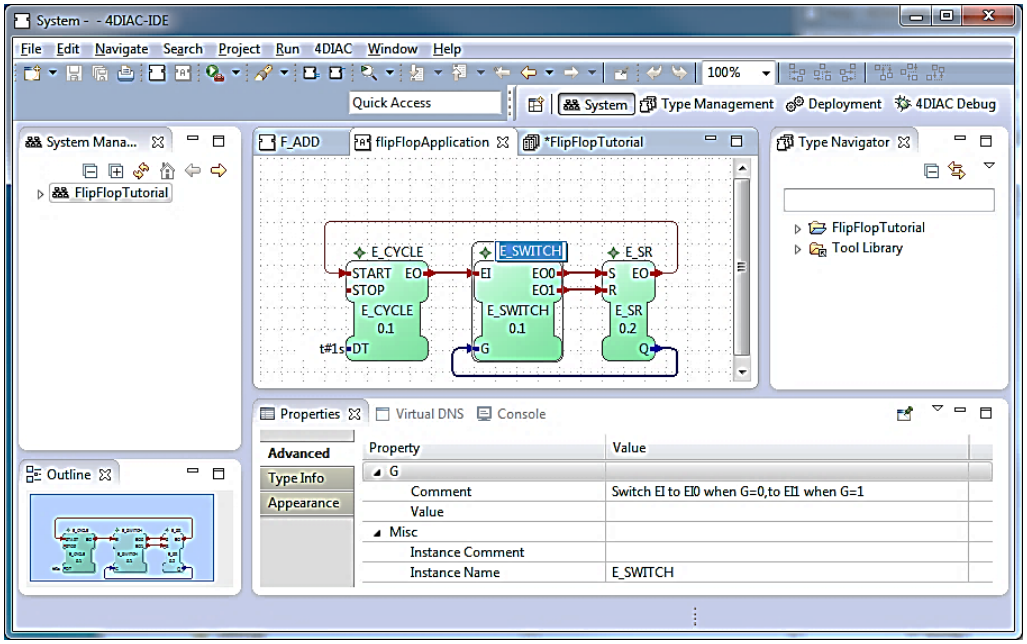


Figura 11-4. Interface do modo de edição da aplicação 4DIAC-IDE (4diac, 2019c)

Embora o 4DIAC-IDE seja uma ferramenta de edição permite a importação e exportação de aplicações e elementos desenvolvidos segundo a norma IEC 61499. Estas novas aplicações e elementos (FBs, recursos, dispositivos e respetivas configurações) deverão ser transferidos para o FORTE (*4DIAC-RTE*) para que possam ser executados. O diagrama apresentado na Figura 11-5 representa o princípio de funcionamento da ferramenta Eclipse 4diac.

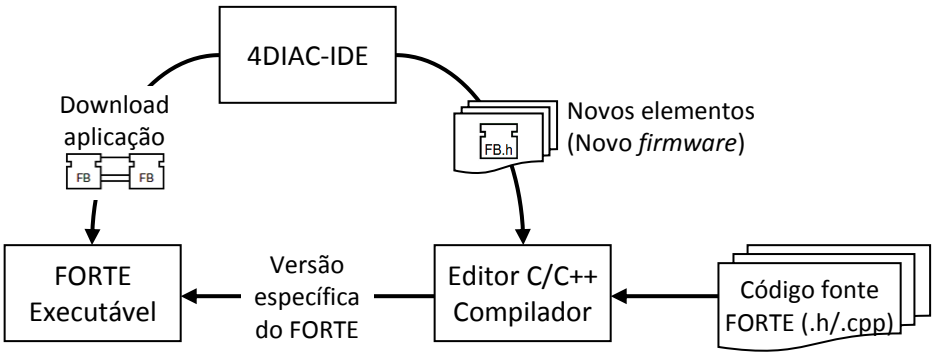


Figura 11-5. Diagrama de funcionamento da Framework Eclipse 4diac

O FORTE nada mais é que um *runtime* que tem como função suportar a execução de sistemas IEC 61499 e por isso, receber as aplicações e elementos que compõe o sistema para que estas possam ser executadas. A compilação do FORTE é um passo crítico e fundamental na arquitetura de funcionamento da ferramenta dado que é necessário compilar o código fonte do FORTE, as configurações próprias da plataforma (*target* para onde se pretende compilar o *runtime*) e os ficheiros exportados pelo 4DIAC-IDE, base de implementação dos novos FBs desenvolvidos pelo utilizador.

O código fonte base do FORTE é implementado em C++ e inclui todos os dispositivos, recursos e FBs disponibilizados pela iniciativa 4diac pelo que, ao serem instanciadas estas implementações é possível criar aplicações que possuam no seu interior, recursos onde são instanciados os FBs necessários para o desenvolvimento de uma qualquer aplicação.

No processo de compilação é necessário definir os parâmetros (*targets*) associados à plataforma de destino de modo que esta possa receber a versão executável do FORTE quer ela

seja um típico PC com Windows quer seja um sistema embebido com Linux. Deste modo, e pelo que já foi referido, todo e qualquer elemento desenvolvido pelo utilizador (FB básico ou composto e de *service interface* ou outros) e exportado pelo IDE deverá ser implementado em C++ para que possa ser incorporado numa nova compilação do FORTE, ou seja, um novo FB implica uma nova compilação, ver diagrama apresentado na Figura 11-5.

4DIAC-IDE, librarias

O 4DIAC-IDE possui uma biblioteca (*typelibrary*) que cobre uma vasta gama de operações pré-definidas e suportadas pelo código fonte base do FORTE (ver Figura 11-4). Desta listagem de bibliotecas destacam-se as seguintes:

- *Interface (FBRT "hmi")* – biblioteca com origem na iniciativa HOLOBLOC, Inc. (HOLOBLOC, 2019). Os SIFBs disponibilizados por esta biblioteca são ferramentas utilizadas no desenvolvimento de interfaces homem-máquina usadas na supervisão e no controlo.
- *Comunicações ("net")* – biblioteca que implementa os modelos de comunicação "*Publish/Subscribe*" e "*Client/Server*". Usada diretamente na aplicação para troca de mensagens entre dispositivos ou recursos.
- *Eventos ("events"* e *"rt_events")* – biblioteca que assegura todas as operações ao nível da manipulação de eventos (*delay, merge, split, etc.*). Os eventos "rt_events" são blocos função que possuem restrições temporais características dos sistemas de tempo real.
- *IEC61131-3* – biblioteca que contem um conjunto de função IEC 61131-3 convertidos para blocos função de acordo com a norma IEC 61499. Disponibiliza ainda funções a utilizar em algoritmos do tipo ST.
- *OCAT* – biblioteca com origem na iniciativa OSCAT (OSCAT, 2019). Conjunto de blocos função adaptado a partir do código-fonte aberto IEC 61131-3 da biblioteca OSCAT.
- *Dispositivos ("Divices")* – biblioteca que contem os dispositivos de processamento (*Remotly Managed Type* - RMT_DEV) e de supervisão (*Remotly Managed Window* - RMT_FRAME). São parametrizáveis, mas não editáveis com a IDE.
- *Recursos ("Resources")* – biblioteca que contem os recursos necessários à implementação das aplicações quer sejam recursos embebidos (EMB_RES), recursos gráficos (PANEL_RESOURCE), recursos remotos (RMT_RES) ou recursos para armazenamento de elementos visuais (VIEW_PANEL).
- *Canais de comunicação ("Segments")* – biblioteca que permite relacionar os diversos dispositivos/recursos entre si por meio dos canais de comunicação predefinidos na ferramenta (CAN, DeviceNet, EIP, EPL, Ethernet e PROFInet).

Naturalmente que não foram apresentadas todas as bibliotecas que a ferramenta disponibiliza na sua versão base. No entanto deve-se referir que alguns dos blocos função contidos na biblioteca podem ser editados criando-se assim novos FBs com origem nos blocos originalmente disponibilizados.

Um outro processo para se obter um novo tipo de FB passa pelo desenvolvimento do mesmo a partir do zero. O editor da ferramenta, 4DIAC-IDE, permite a criação de *Basic*, *Service Interface* e *Composit* FBs com pleno controlo sobre eventos e dados de entrada e de saída bem como a associação entre eventos e dados. Para além desta possibilidade será ainda possível, em alguns casos, definir o ECC e os algoritmos a ele associados e definir as sequências de serviço. Nos CFB é possível criar a sua própria rede de instâncias de FB de acordo com o que se encontra estipulado na norma. De notar que, após a sua criação, todos os novos FBs têm de ser exportados e compilados pelo FORTE para que possam ser executados e utilizados pelas aplicações.

Desenvolvimento de uma aplicação

As aplicações são desenvolvidas no ambiente de edição IDE. De acordo com a aplicação a desenvolver poderá importar-se ou exportar-se sistemas e componentes de modo a facilitar a partilha e o desenvolvimento dos mesmos. Esta importação/exportação não se cinge só ao nível dos FBs uma vez que todos os dispositivos, recursos, segmentos e sistemas podem também ser exportados e importados. Na Figura 11-6 apresenta-se uma das muitas configurações possíveis de uma aplicação desenvolvida com a ferramenta IDE.

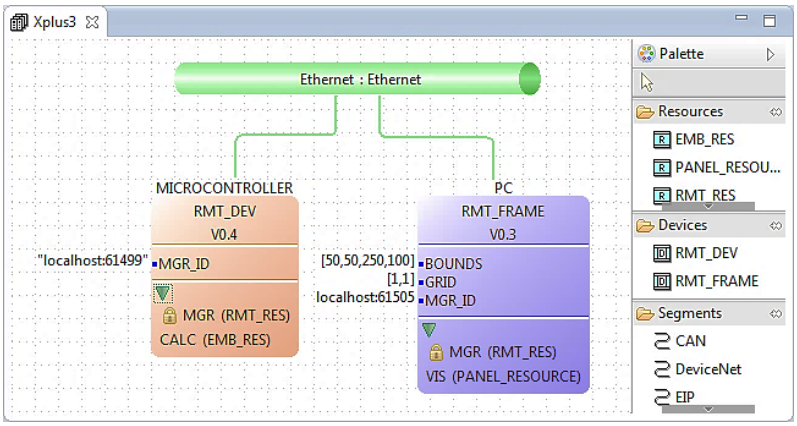


Figura 11-6. Configuração de um sistema, libraria "Segments" (4diac, 2019c)

Nesta configuração é possível identificar as ligações definidas entre o dispositivo de processamento RMT_DEV ("MICROCONTROLLER") e o de supervisão RMT_FRAME ("PC"), cada um com um recurso ("CALC" e "VIS", respetivamente). O parâmetro "MGR_ID" é utilizado para identificar o código IP (*Internet Protocol*) e a porta de escuta do dispositivo. A rede Ethernet é usada para interligar os dispositivos que poderão compor o sistema, o tipo de recursos que cada um dos dispositivos possui e os endereços IP das portas de escuta de cada um.

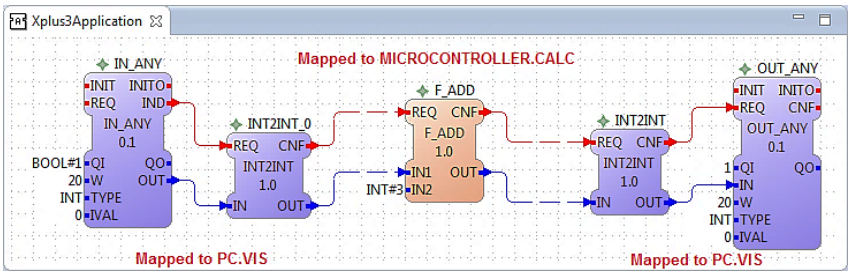


Figura 11-7. Mapeamento dos FBs para os respetivos dispositivos (4diac, 2019c)

A configuração do sistema só ficará completa com o mapeamento dos FBs para o dispositivo onde será executado, Figura 11-7. Cada um dos FBs deve ser associado a um dispositivo e as instâncias, alocadas em diferentes dispositivos, devem estar interligadas por blocos função de comunicação do tipo SIFBs (PUBLISH/SUBSCRIBE ou CLIENT/SERVER), Figura 11-8. O desenvolvimento da parte funcional do sistema é feito de forma independente da configuração final do sistema permitindo ao programador abstrair-se do tipo de comunicação a utilizar na arquitetura das aplicações.

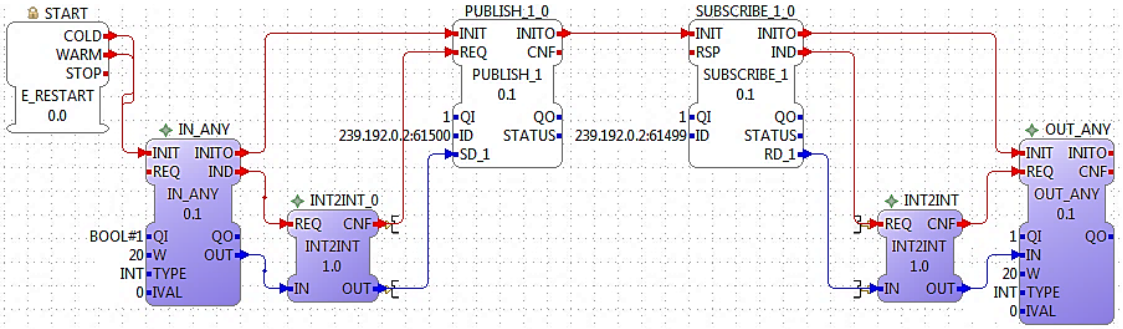


Figura 11-8. Interligação entre instâncias alocadas em diferentes dispositivos (4diac, 2019c)

Transferência da aplicação

Após o desenvolvimento do sistema e respetiva configuração, as aplicações que o compõem devem ser transferidas para o FORTE para que possam ser executadas (efetuar o *download* para o FORTE conforme esquema apresentado na Figura 11-5). O *download* inicia-se com a chamada para execução do *runtime*. O FORTE é executado a partir do menu *Runtime Launcher* da perspetiva *Deployment*, escolhendo-se o código IP do dispositivo (*target*, usualmente *localhost*, equivalente ao endereço IPv4 do equipamento) e respetiva porta de escuta confirmando-se a execução com o botão *Launch FORTE*, Figura 11-9.

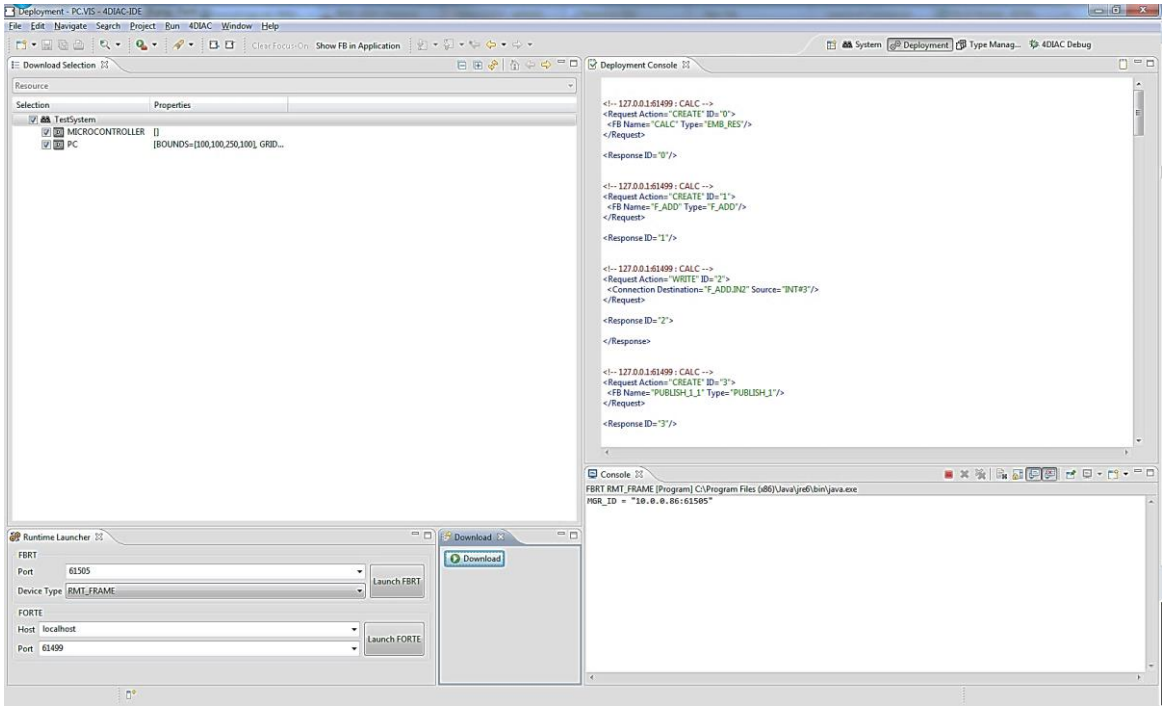


Figura 11-9. Interface do modo deployment da aplicação (4DIAC, 2019c)

A distribuição da aplicação por vários dispositivos obriga à criação de mais execuções do *runtime* com portas de escuta diferentes umas das outras.

4DIAC-RTE (FORTE)

Como já foi referido anteriormente o 4DIAC-RTE (FORTE) é uma aplicação portável, implementada em C++, passível de ser executada num ambiente IEC 61499 capaz de suportar a execução de programas de controlo distribuído. O FORTE consiste num *RunTime Environment*

(RTE) que pode ser executado em dispositivos de controlo quer em Windows, Cygwin (Cygwin, 2019), Linux (i386, PPC) quer em sistemas embebidos (ARM7) (Zoitl *et al.*, 2010) entre outros (4diac, 2019a).

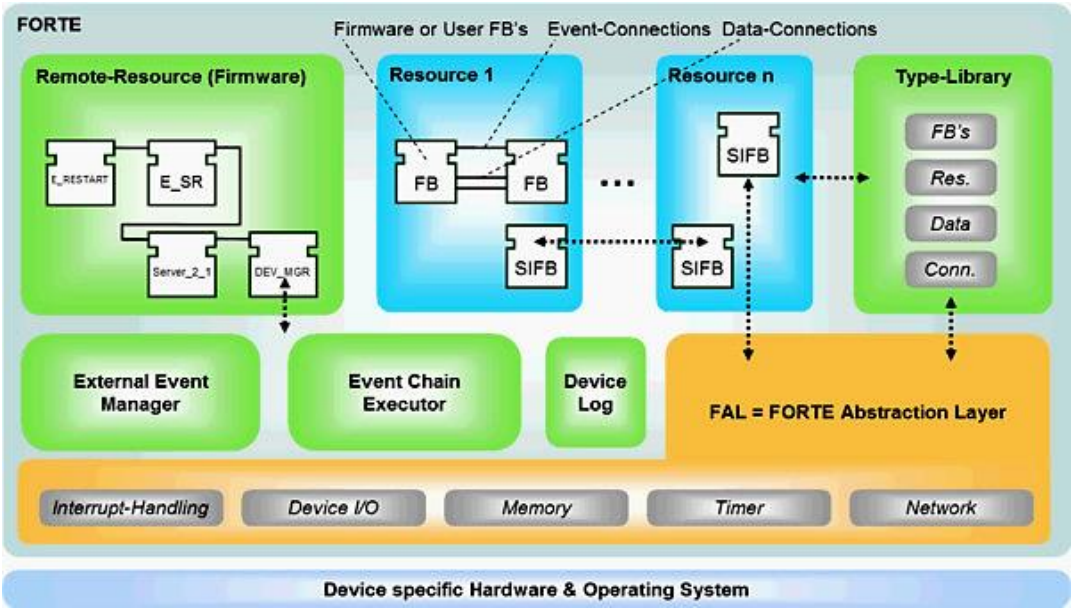


Figura 11-10. Arquitetura FORTE para o controlo de aplicações com base na IEC 61499

Como se depreende da figura anterior o FORTE funciona como um ambiente de suporte à execução da aplicação. É constituído por uma série de componentes que fornecem às aplicações a abstração necessária para execução da aplicação independentemente da plataforma em que são executadas. A componente FAL (FORTE *Abstraction Layer*) é responsável por lidar com todas as funcionalidades de baixo nível fornecidas pelo sistema operativo executando a gestão de comunicações, memória, *divices* I/O, entre outros, conforme esquema apresentado na Figura 11-10. A biblioteca ("*Type-library*"), que contem os diferentes tipos de FBs, recursos, elementos de comunicação, variáveis de dados e outros elementos possíveis de serem instanciados, é utilizada no suporte às aplicações recebidas através do recurso ("*Remote-Resource*") que, com base nos elementos contidos nessa biblioteca, interliga e parametriza, automaticamente, as aplicações recebidas no FORTE. Os restantes módulos são responsáveis pela execução das aplicações ("*Event Chain Executor*"), gestão de *interrupts* ("*External Event Manager*") e registo de funcionamento ("*Device Log*").

Compilação do FORTE

Atualmente o FORTE pode ser compilado para duas grandes arquiteturas suportadas por sistemas operativos que respeitam o standard POSIX, por exemplo Linux, OS X para sistemas Mac e o standard WIN32/64 para sistemas Windows. Existem ainda outras arquiteturas usadas em sistemas embebidos e que correspondem a dispositivos singulares com sistemas operativos próprios como sejam as plataformas da Digi Connect ME e da Lego Mindstorms EV3 (ARM7), da NET+OS 7, da eCos (*open source runtime* suportada pelas ferramentas de desenvolvimento de código aberto GNU) bem como alguns PLCs como sejam o Bachmann M1 PLC e o WAGO PFC200, entre outros (4diac, 2019a).

Embora o FORTE possa ser compilado através do Eclipse, é aconselhado que a sua compilação seja realizada através do CMake (CMake, 2019) de acordo com os passos descritos no Anexo F. Esta plataforma permite compilar e configurar as propriedades a conferir à versão executável do FORTE em função do ambiente de compilação escolhido. Através do CMake GUI é possível

escolher o tipo de plataforma a utilizar para se gerar os *Makefiles* necessários à compilação pelo que, se a plataforma for um PC com sistema operativo Windows basta escolher a opção "FORTE_ARCHITECTURE_WIN32" (4diac, 2019a), Figura 11-11.

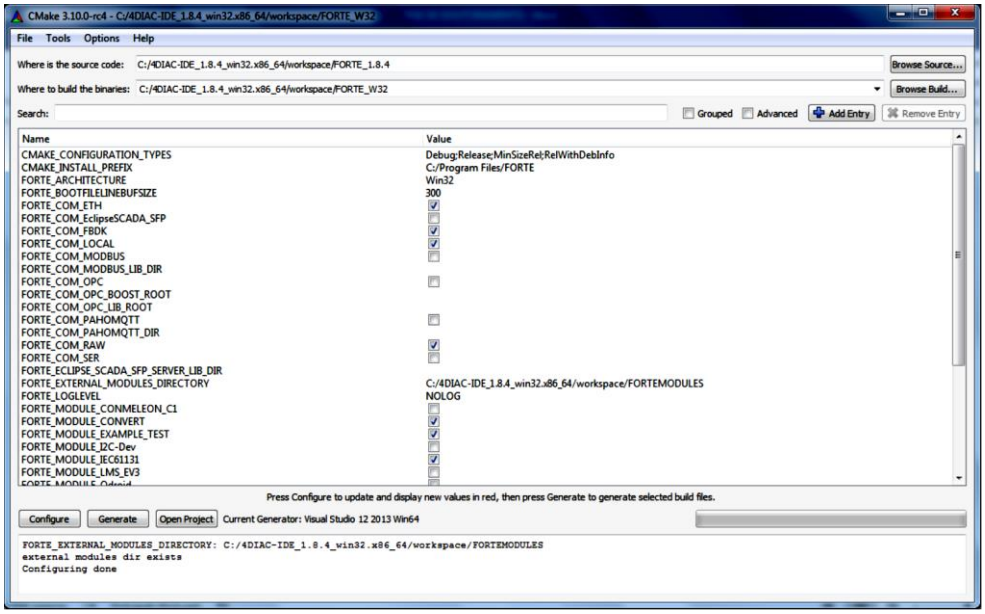


Figura 11-11. Gerar o projeto como o CMake GUI

A compilação do FORTE no Windows pode ser realizada com o Visual Studio (VS, 2019) ou no Eclipse. O uso do Eclipse para o desenvolvimento e a depuração do FORTE no Windows necessita de um ambiente de emulação POSIX como o Cygwin (Cygwin, 2019) ou MinGW (MinGW, 2019).

Arquitetura do FORTE

Em termos de arquitetura o *Eclipse 4diac*, e nomeadamente o FORTE, segue o conceito de programação orientada a objetos fazendo uso de classes para representar entidades pelo que, todos os elementos que constituem a biblioteca da plataforma, disponibilizados em código fonte na pasta "src", encontram-se agrupados nos seguintes três grandes grupos:

- Arquitetura ("*arch*") – pasta que contem o código fonte responsável pelas operações de baixo nível realizadas pelo programa, destacando-se as operações de gestão de mensagens, temporização e de gestão dinâmica da memória. A pasta arquitetura contempla ainda todo o código necessário para a implementação destas operações nas diversas plataformas.
- Elementos base ("*core*") – pasta que inclui todos os ficheiros necessários à implementação dos conceitos inerentes à norma IEC 61499. Nesta pasta estão contidas as classes necessárias à implementação da estrutura dos FBs.
- FBs ("*modules"* e *"stdfblib*") – pasta que agrega os vários tipos de FBs nativos da aplicação (os FBs desenvolvidos pelo programador deverão ser guardados aqui também).

Compreender-se-á que a criação de um novo FB e respetiva exportação para o *runtime* do FORTE obrigará não só à alteração dos ficheiros "CMakeLists.txt" associados ao tipo de FB criados mas também, ao cumprimente de algumas regras (ficheiro "forte_coding_rules.pdf" pasta "doc/coding_rules" do FORTE *source code*) que deverão ser respeitadas aquando da criação de novos elementos ou da edição do código dos elementos fornecidos com a libraria. Assim, a criação

e adição de um novo bloco função à pasta responsável pelas operações com eventos "src/stdfblib/events", por exemplo, passaria pela inserção do nome do novo FB à lista definida nos ficheiros "CMakeLists.txt" associando-se o argumento "forte_add_sourcefile_hcpp" de acordo com o exemplo apresentado na Figura 11-12.

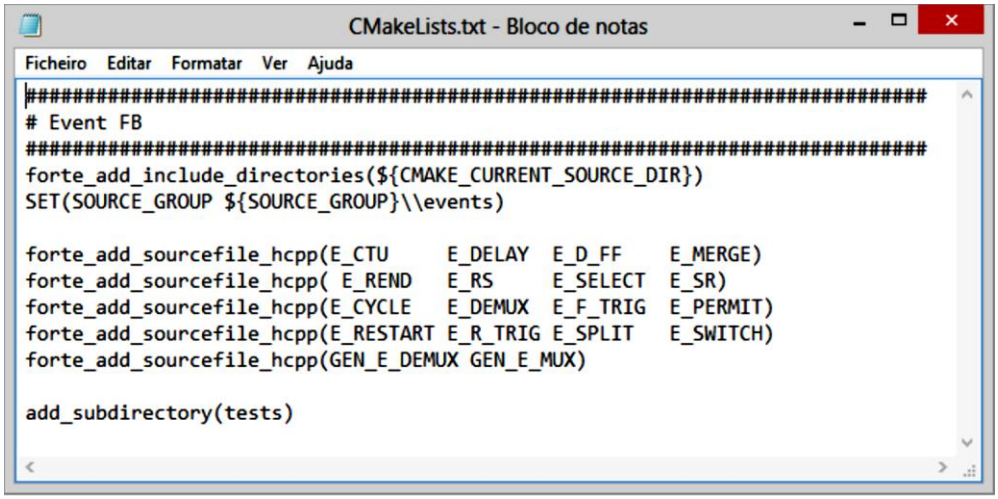


Figura 11-12. Conteúdo do ficheiro CMakeLists.txt da pasta "src/stdfblib/events"

Código fonte, FORTE

Todos os FBs que constituem o código fonte FORTE são definidos segundo o conceito de programação orientada a objetos e como tal, todos os blocos função são definidos como classes. Todas estas classes alteradas ou criadas no IDE terão de ser exportadas e incorporadas no FORTE e por isso, há a necessidade de compreender a estrutura dos diferentes géneros de FBs bem como, o processo de criação de novos blocos função. Na Figura 11-13 apresenta-se a interface do bloco função básico "FB_MUL_INT" (proprietário da aplicação), respetivas entradas e saídas e comentários associados.

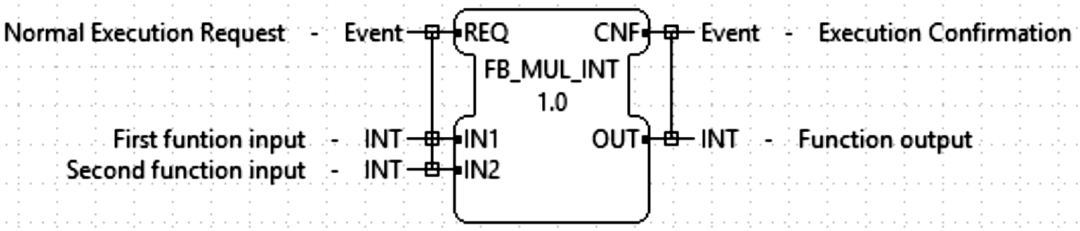


Figura 11-13. Representação do Bloco Função Básico "FB_MUL_INT"

Analisando a figura anterior verifica-se que na interface são indicadas uma série de variáveis estáticas correspondentes às entradas e saídas, aos nomes das entradas e saídas, aos tipos de variáveis utilizadas, à relação entre os eventos e as variáveis, etc. (ver Anexo I). Todas estas variáveis, definidas na IDE durante a fase de desenvolvimento do FB, são parametrizadas durante o processo de exportação gerando-se, automaticamente, dois ficheiros com extensões ".cpp" e ".h" característicos da programação em C++. Durante este processo é ainda criada a função de execução "executeEvent" bem como todas as funções que representam os algoritmos do FB (ver Figura 11-14).

```
63 void FB_MUL_INT::executeEvent(int pa_nEIID){
64   bool bRetVal;
65   do{
66     bRetVal = true;
67     switch(m_nECCState){
68       case scm_nStateSTART:
69         if(scm_nEventREQID == pa_nEIID)
70             enterStateREQ();
71         else
72           bRetVal = false; //no transition cleared
73         break;
74       case scm_nStateREQ:
75         if(1)
76             enterStateSTART();
77         else
78           bRetVal = false; //no transition cleared
79         break;
80       default:
81         DEVLOG_ERROR("The state is not in the valid range! The state value is: %d. The max value can be: 1.",
82             m_nECCState.operator TForteUInt16 ());
83         m_nECCState = 0; //0 is allways the initial state
84         break;
85     }
86     pa_nEIID = cg_nInvalidEventID;  // we have to clear the event after the first check in order to ensure correct behvior
87   }while(bRetVal);
88 }
```

Figura 11-14. Função de execução "executeEvent" da classe "FB_MUL_INT"

Basic Function Block (BFB)

Os *Basic Function Block* (BFBs), blocos função básicos, são automaticamente gerados, por exportação para o FORTE, de acordo com o que foi definido na IDE. O código C++ apresentado na Figura 11-14 é a representação do ECC que comanda o modo de funcionamento do bloco função em causa. A função é ativada aquando da receção de um evento. O código pode ser editado pelo programador, no entanto, deve ter em conta as especificações da norma.

A função "*executeEvent*" (Figura 11-14) recebe como argumento de entrada uma variável inteira "pa_nEIID" que indica qual o evento recebido. A função confronta esta entrada com os códigos dos eventos disponíveis (verificação das condições booleanas das transições do ECC) e, caso se der uma correspondência, é invocada a função responsável pela atualização do estado do ECC. Os algoritmos associados ao novo estado são chamados, são emitidos eventos de saída ligados às ações (funções "*enterStateREQ*" e "*enterStateSTART*"). A função "*executeEvent*" termina quando a variável booleana de retorno "bRetVal" for igual a "false" ou seja, quando nenhumas das transições do estado ativo são satisfeitas.

Composite Function Block (CFB)

Os *Composite Function Block* (CFBs), blocos função do tipo composto, possuem na sua interface, tal como os BFB, variáveis estáticas usadas na identificação dos sinais de entrada e saída.

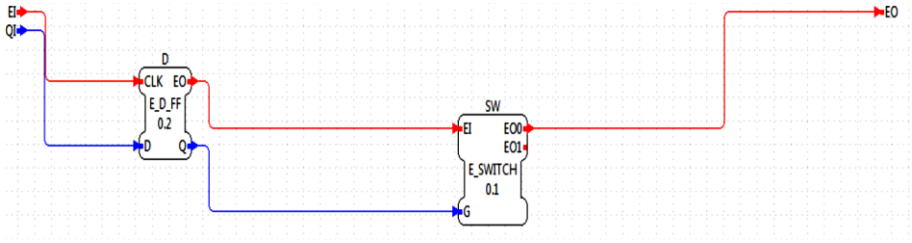


Figura 11-15. Blocos função composto "E_F_TRIG", FBs internos e ligações (4diac, 2019c)

Dada a sua estrutura composta, estes FBs possuem ainda outras variáveis estáticas usadas na interligação dos diversos FBs que os componham e nas ligações de eventos e dados da rede interna. Note-se ainda que para uma correta compilação do novo CFB é necessário que todos os

FBs que constituem a sua rede interna estejam igualmente integrados no *runtime* do FORTE, ver Figura 11-15.

Service Interface Function Blocks (SIFB)

Os *Service Interface Function Blocks* (SIFBs), blocos função de interface especialmente desenvolvidos pelo programador, ao contrário dos FBs descritos anteriormente, não possuem código fonte quando exportados pela IDE para o FORTE. Estes FBs não contêm as funções que representam os algoritmos ou o ECC, pelo simples facto de que os SIFBs não possuem algoritmo nem ECC. Os blocos função de interface possuem única e simplesmente a função "*executeEvent*" desprovida de código, contendo apenas o código necessário para identificar o evento que provocou a sua chamada, Figura 11-16. São geralmente usados em algo que não pode ser feito através do standard IEC 61499 como a comunicação com dispositivos ou segmentos de redes. Os SIFBs são usados para realizar apenas o que não pode ser feito com um BFB ou um CFB.

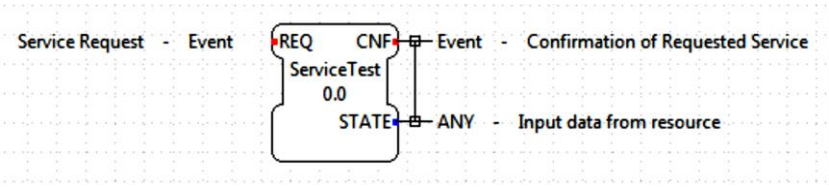


Figura 11-16. Bloco função de interface "ServiceTest", interface desenvolvido pelo utilizador

Neste sentido, é necessário que o programador/utilizador desenvolva o código necessário para que o SIFB cumpra as funcionalidades desejadas garantindo que as novas interfaces interagem corretamente com o processo a controlar e com as redes de comunicação.

```
41
42 void FORTE_ServiceTest::executeEvent(int pa_nEIID){
43   switch(pa_nEIID){
44   case scm_nEventREQID:
45         state= !state;
46         sendOutputEvent(scm_nEventCNFID);
47     break;
48   }
49 }
50
```

Figura 11-17. SIFB "ServiceTest", código desenvolvido após exportação

Uma vez exportado e compilado o FB desenvolvido, o programador pode fazer um teste de verificação do mesmo selecionando-se *FORTE Remote Tester* na configuração de teste. Os passos necessários à realização do teste são apresentados no Anexo J.

Adapter interface function block

Os adaptadores são interfaces de FBs que possuem apenas interface e nenhuma funcionalidade encapsulada. São usados para implementar interfaces complexas entre FB simplificando a aplicação IEC 61499. Estes reduzem o número de elementos da interface (entradas e saídas de eventos e dados) e, por isso, o número de ligações que devem ser criadas entre os FB com que interagem. Estas interfaces são reutilizáveis fornecendo um conceito de interface que permite adotar o conceito polimórfico de programação orientada a objetos e, como tal, podem ser adicionadas à interface de qualquer tipo de FB.

Os adaptadores podem ser usados para definição de tipos específicos de entradas e saídas complexas pelo que poderão ser disponibilizados em dois tipos de interação: de saída chamados de *plugs* e de entrada chamados de *sockets,* Figura 11-18 e Figura 11-19.

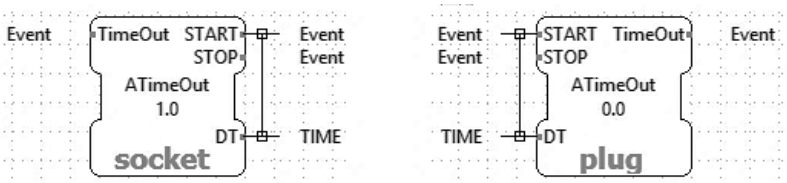


Figura 11-18. Bloco função adaptador, Socket e Plug

- *Socket*: é adicionado como uma entrada de um bloco função. Este é representado no interior da interface do bloco de funções.
- *Plug*: é adicionado como uma saída de um bloco função. Este é representado no interior da interface do bloco de funções.

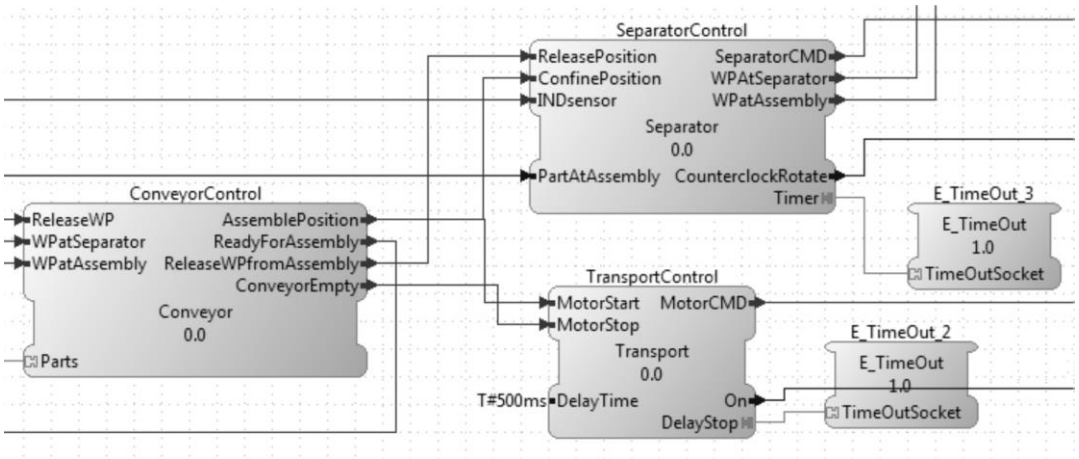


Figura 11-19. Exemplo de aplicação do Socket e do Plug (input e output) em 4diac

A configuração de um sistema distribuído pode incluir vários componentes como sejam, por exemplo, Model, Object, Controller, etc., para cada um dos objetos mecatrônicos do mesmo. Os componentes interagem através de interfaces comuns, por exemplo, o Model interage com o Controller por meio dos mesmos sinais de entrada e saída pelo que, a utilização de adaptadores padronizará as ligações entre funções, simplificando o processo de reconfiguração.

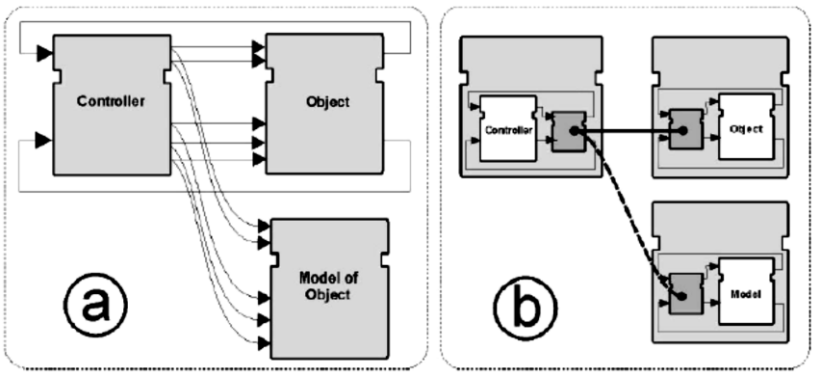


Figura 11-20. Reconfiguração do sistema usando adaptadores (Vyatkin, 2014)

Assim, e no caso de uma reconfiguração, são inúmeras as ligações de eventos e de dados que precisam ser redirecionadas para outro bloco (Figura 11-20a) no exemplo, a interligação de um controlo em circuito fechado. Toda vez que a configuração precisa ser alterada, de Controller-Object para Controller-Model e vice-versa, todas as ligações precisariam ser redirecionadas. A utilização de adaptadores reduz essa rotina apenas à redefinição das ligações do adaptador, conforme se apresenta na Figura 11-20b.

Este encapsulamento produz componentes de aparência mais simples. O preço pago por essa simplicidade é outra camada de hierarquia entre o atual controlador e os blocos função que possuem o adaptador como por exemplo, um *socket* (Vyatkin, 2014). Por outro lado, e no caso em que os FB ligados a adaptador devem ser mapeados para dispositivos diferentes, as interfaces do adaptador podem ser transferidas através de redes de comunicação. Para este propósito, podem ser desenvolvidos CIFB específicos (Figura 11-21) onde os CIFBs personalizados com as interfaces dos adaptadores de entrada e saída, respetivamente, implementam a passagem contínua de informação através do canal de comunicação.

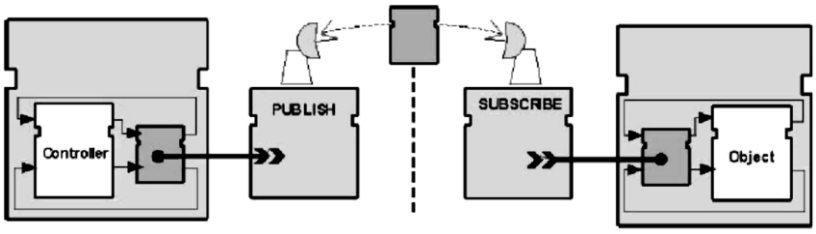


Figura 11-21. Ligação entre dois adaptadores distribuídos por diferentes dispositivos (Vyatkin, 2014)

Funcionalidade de DNS Virtual

As propriedades de visualização do 4diac fornecem um editor de DNS (*Domain Name System*) virtual que permite definir um conjunto de variáveis que poderão ser substituídas por um valor pré-especificado. Os nomes de todas as variáveis que se encontram disponíveis no *Virtual DNS* podem ser usadas para a parametrização de blocos funções e dispositivos sendo substituídas, durante a implementação, pelos valores especificados. Assim, e para que se possam usar as variáveis definidas no DNS, o nome da cada uma das variáveis deve ser escrita entre %.

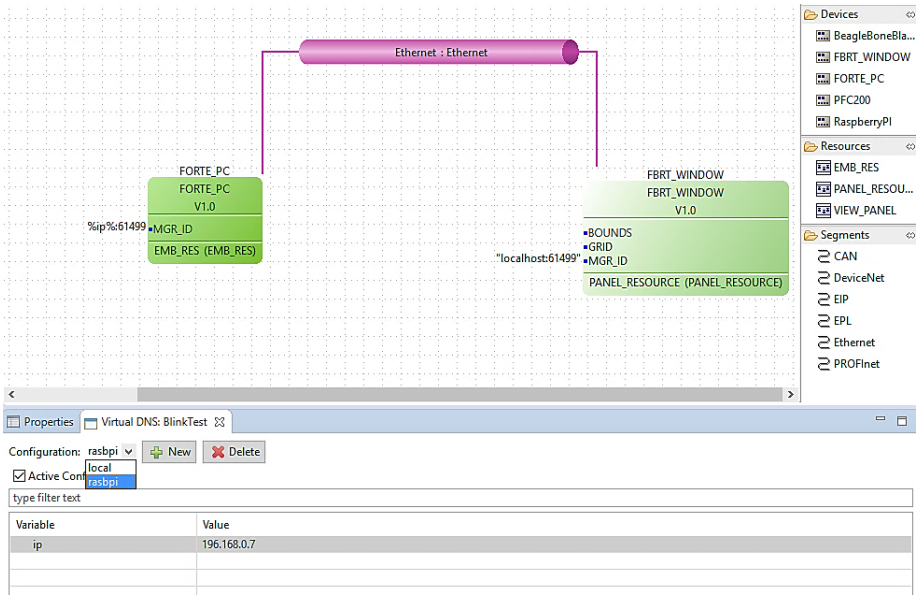


Figura 11-22. Virtual DNS (4diac, 2019c)

No exemplo apresentado na Figura 11-22, foram criadas duas configurações essenciais à definição das variáveis: *local* e *rasbpi*. Neste exemplo, retirado da documentação do 4diac, a configuração rasbpi possui uma variável ip definida para o valor 192.168.0.7 e o local para 127.0.0.1. Na visão principal, a variável ip é usada como o endereço IP do dispositivo FORTE. Quando se quiser usar um FORTE local, basta alterar a configuração e verificar a configuração ativa em vez de alterar o valor do IP, diretamente no FB, cada vez que for necessário alterná-lo.

FORTE Boot-Files

Forte *Boot-Files* são arquivos de inicialização que serão carregados pelo FORTE na sua inicialização. Estes ficheiros assumem um papel importante na configuração da aplicação uma vez que o arquivo contém a configuração de rede de FBs para o dispositivo e será automaticamente instanciado durante a inicialização do FORTE.

Para que estes arquivos de inicialização possam ser criados é necessário que a opção FORTE_SUPPORT_BOOTFILE se encontre habilitada para o FORTE que se está a utilizar, i.e., definida aquando da compilação do FORTE. O arquivo de inicialização deve encontrar-se localizado no mesmo diretório que o FORTE e terá de possuir o nome *forte.fboot*. Assim, e para que se possa criar estes arquivos de inicialização, é necessário selecionar um ou mais dispositivos ou recursos na janela *Download Selection View* do *Deployment*, selecionando-se *Create FORTE boot-files...* no menu de contexto, Figura 11-23.

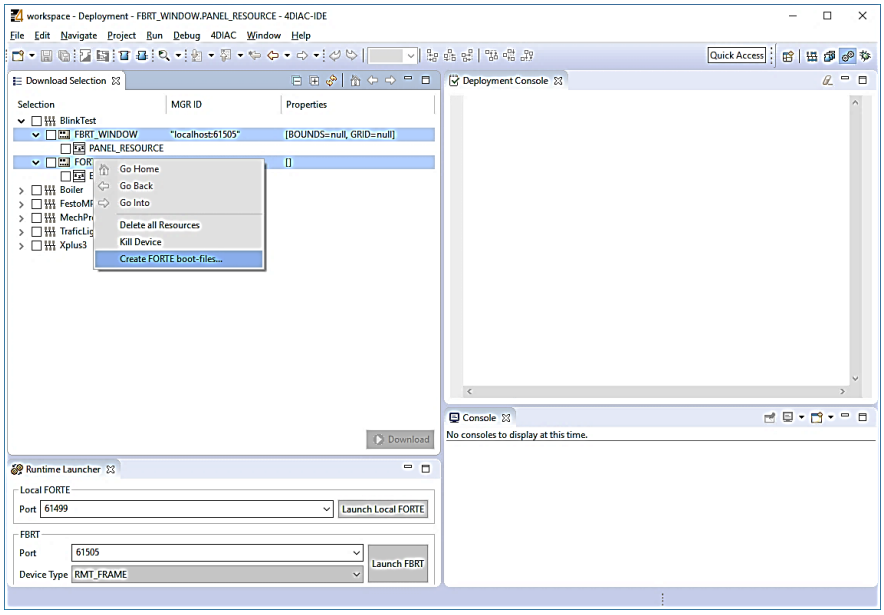


Figura 11-23. Janela Deployment, criação de um arquivo Boot-File de inicialização (4diac, 2019c)

No assistente, seleciona-se os dispositivos e recursos para os quais se deseja criar arquivos de inicialização, bem como o diretório onde colocá-los. Ao concluir a operação será gerado um arquivo de inicialização que conterá os recursos selecionados e as redes de FBs contidas nos recursos, ver Figura 11-24.

O nome do arquivo de inicialização *boot-file* será uma combinação do nome do sistema e dispositivo com *fboot* separado por um ponto. Antes de se utilizar qualquer arquivo de inicialização é necessário renomeá-lo para *forte.fboot*.

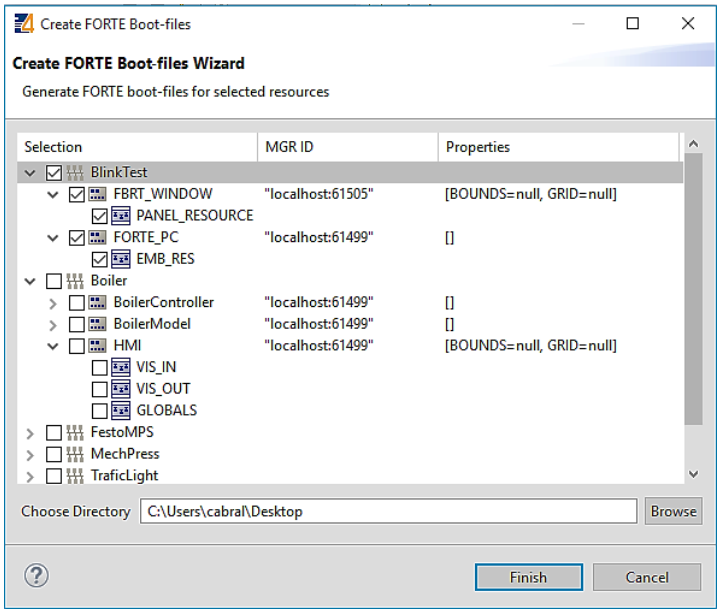


Figura 11-24. Assistente de criação de um arquivo Boot-File de inicialização (4diac, 2019c)

Anexo D – Raspberry Pi 3 B

Explanação do microcomputador Raspberry Pi 3 (1.2GHz 64-bit CPU quad-core ARMv8, 1 GB RAM), conexões e *pinout*.

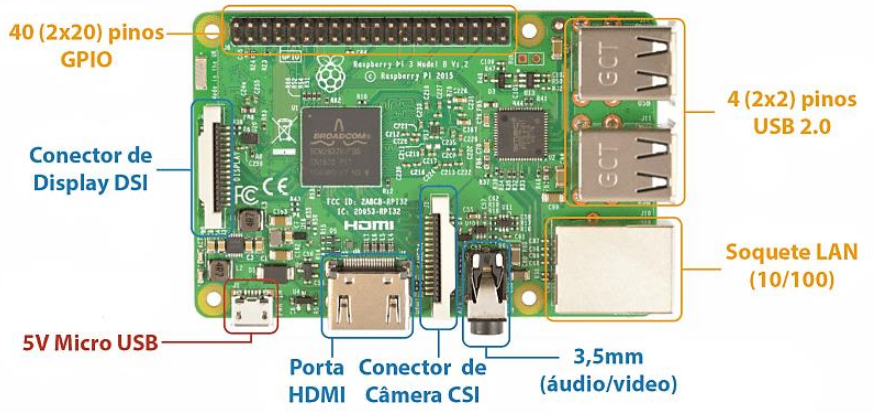


Figura 11-25. Conectores da placa Raspberry Pi (fonte: Eletrofun, 2019)

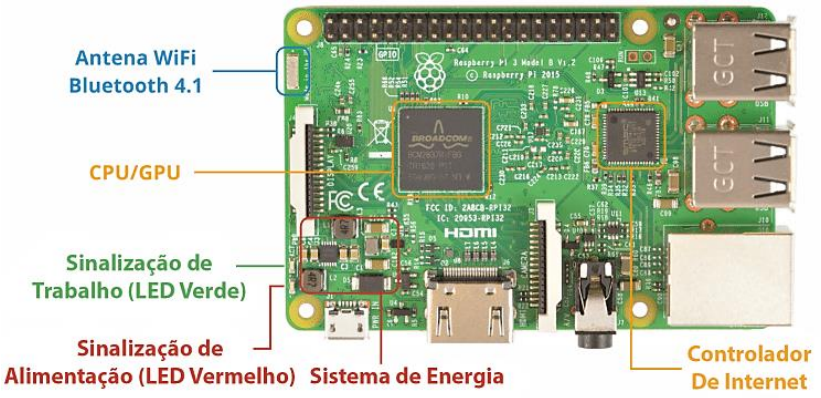


Figura 11-26. Elementos constituintes da placa Raspberry Pi 3 (fonte: Eletrofun, 2019)

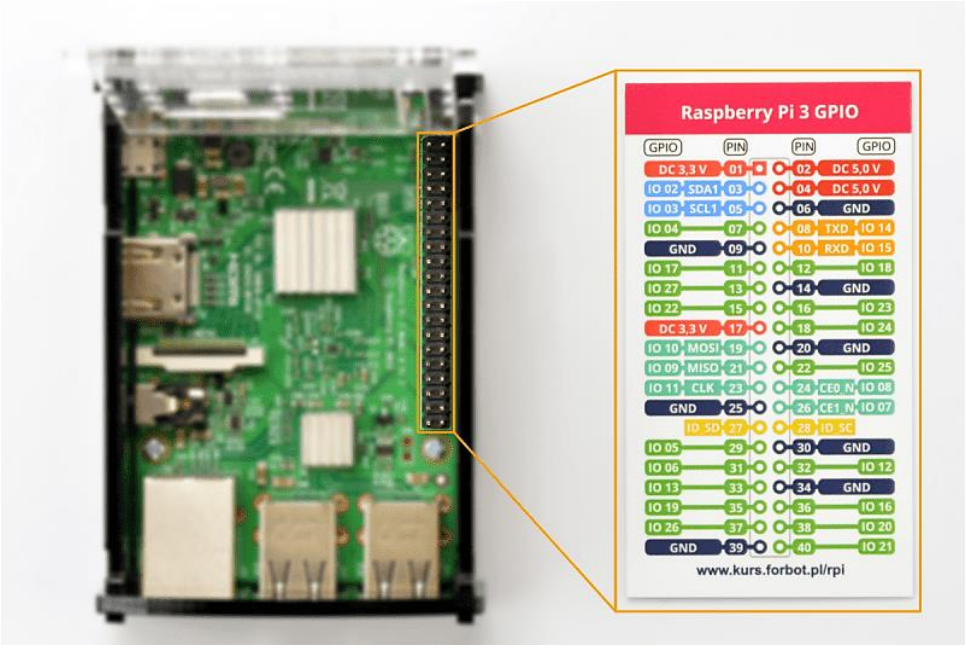


Figura 11-27. Elementos constituintes da placa Raspberry Pi 3 (fonte: Eletrofun, 2019)

Anexo E – Distribuição da aplicação por diversos dispositivos

Neste anexo é apresentado o processo de teste e de distribuição da aplicação, executada localmente, por diversos dispositivos. A janela "Deployment" permite correr o FORTE em distintos equipamentos emulando vários PLCs. Para usar-se esta funcionalidade é necessário que no "System Configuration" se tenha alocado ao dispositivo um *localhost* diferente do standard do 4diac. Na Figura 11-28**Erro! A origem da referência não foi encontrada.** foi definido, para o segundo dispositivo, o localhost para 61500.

Executar os passos seguintes:

1 Segundo as indicações numéricas da figura abaixo, seleciona-se em Local FORTE, a porta pretendida (1) lançando-se, em seguida, o 4diac FORTE local (*localhost:61499*, 2);
2 Executar o procedimento anterior tantas vezes quantas o número de dispositivos usados. Neste exemplo são realizados dois lançamentos do FORTE. Execução dos números 4 e 5, lançamento do FORTE associado ao *localhost:61500*.
3 Executados os passos anteriores deve-se selecionar os elementos da aplicação que pretendemos correr, ou seja, quais os PLCs que pretendemos executar, passo 7.
4 Clicar no botão 8, *Deploy*, e testar a aplicação, Figura 11-29.

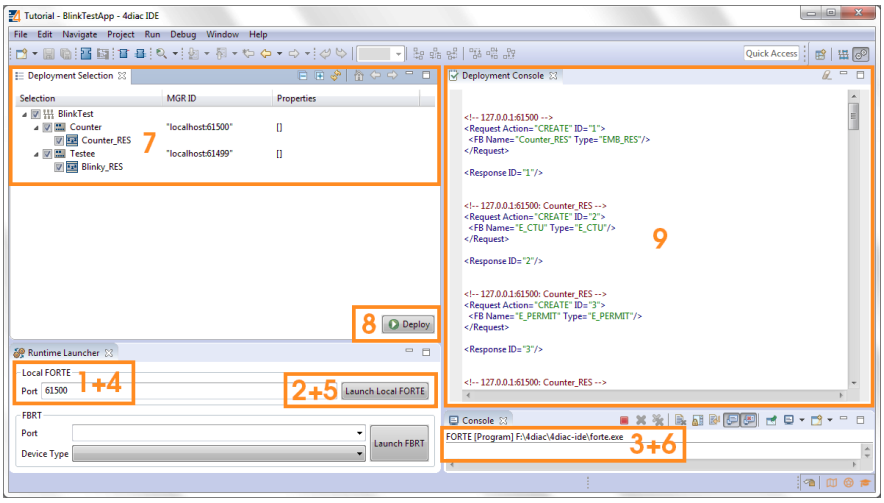


Figura 11-28. Execução local de uma aplicação distribuída por dois dispositivos (4diac, 2019c)

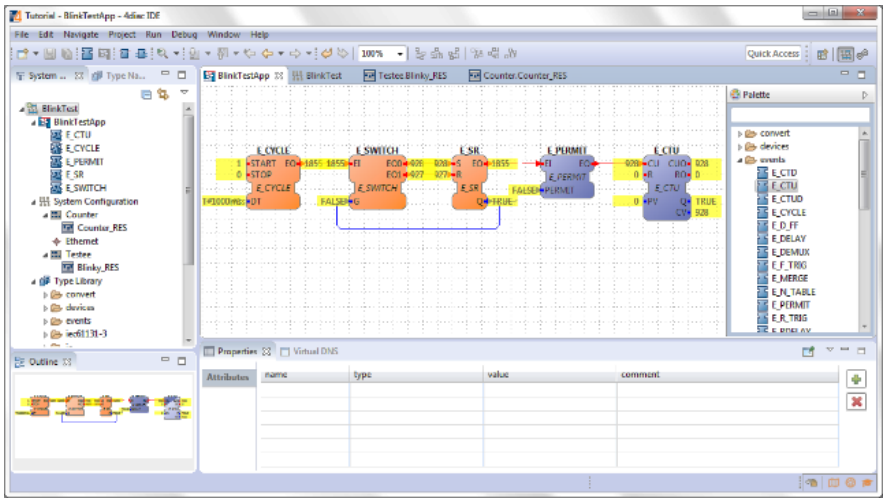


Figura 11-29. Teste da uma aplicação distribuída por dois PLC (4diac, 2019c)

Anexo F – Compilação do Forte (4DIAC-RTE)

Neste anexo serão apresentados alguns passos para a instalação do Forte (4DIAC-RTE) no Windows. Assim, começaremos por executar diferentes procedimentos:

- Criar uma pasta de destino e copiar para esta o conteúdo do repositório mercurial (http://hg.code.sf.net/p/fordiac/forte);
- Instalar o “Cmake” (http://cmake.org/);
- Executar “Cmake-gui”;

- Selecionar a opção "FORTE_ARCHITECTURE_WIN32";
- Clicar em "Configure" novamente, ver Figura 11-11;
- Adicionar todos os módulos desenvolvidos, *external modules directory*;
- Clicar em "Generate";
- Selecionar o diretório com os makefiles;
- Clicar em "Cinfigure" (ver Figura 11-11) e escolher a última versão do "Visual Studio" (VS 2019);
- Selecionar "Use default native compilers", Figura 11-30,

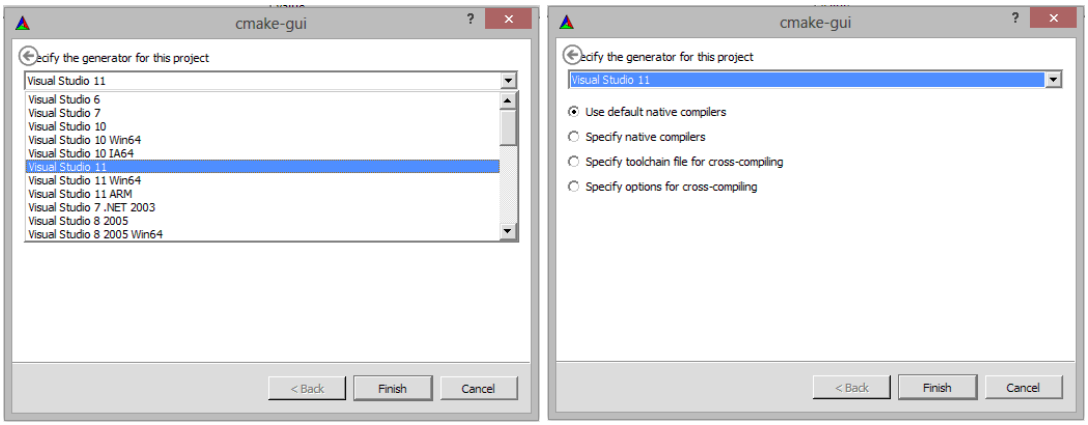


Figura 11-30. Gerador do projeto usando o CMake GUI

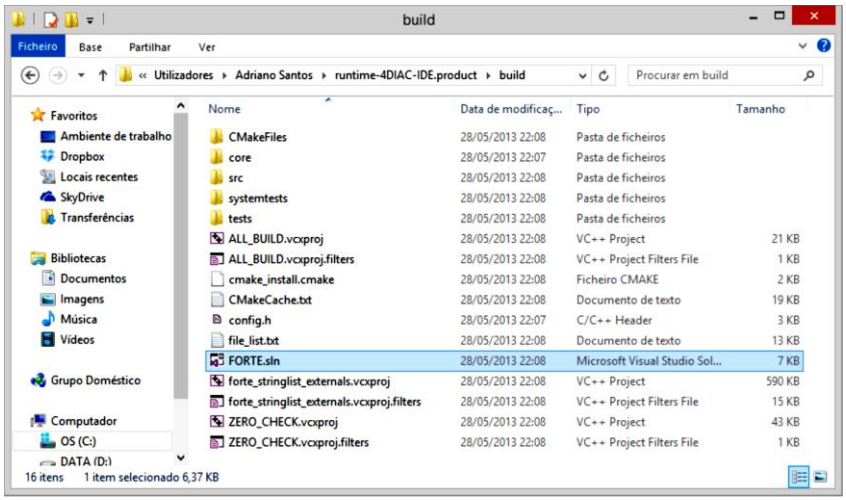


Figura 11-31. Makefiles gerados pelo CMake

- Abrir o ficheiro "FORTE" com extensão ".sln" (Microsoft Visual Studio Solution);
- Selecionar opção “Release”;
- Clicar em "BUILD → Build Solution";

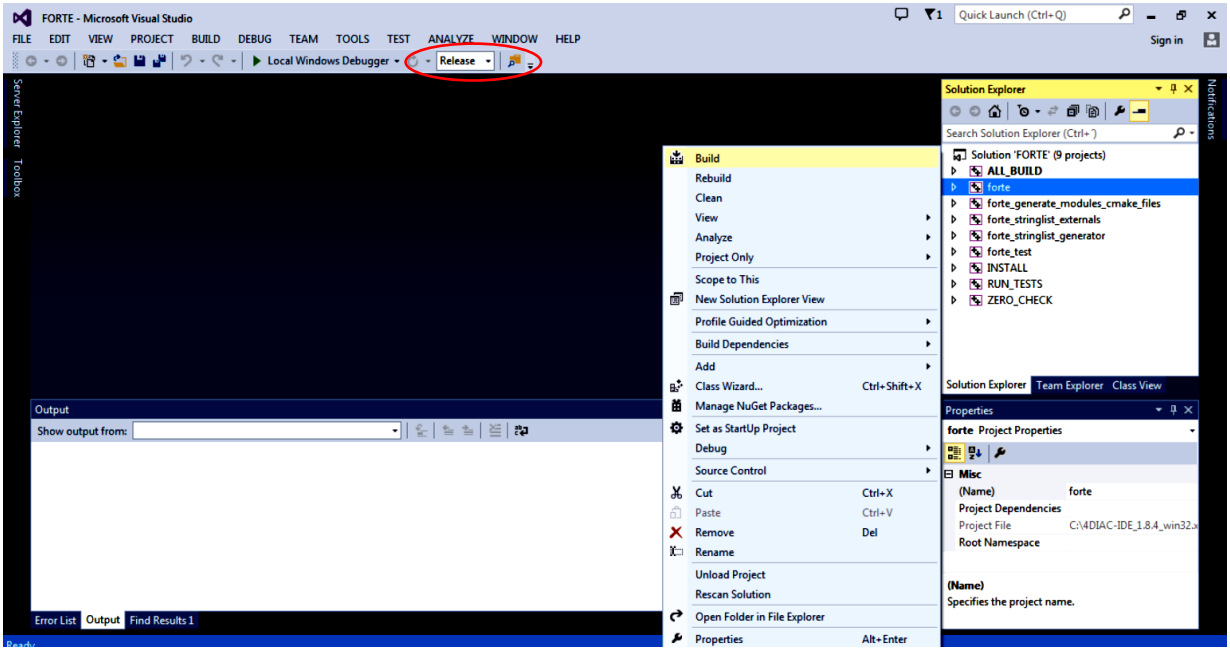


Figura 11-32. Microsoft Visual Studio em execução

O FORTE poderá também ser compilado diretamente do CMke através de um clique no botão “Open Project”, figura seguinte.

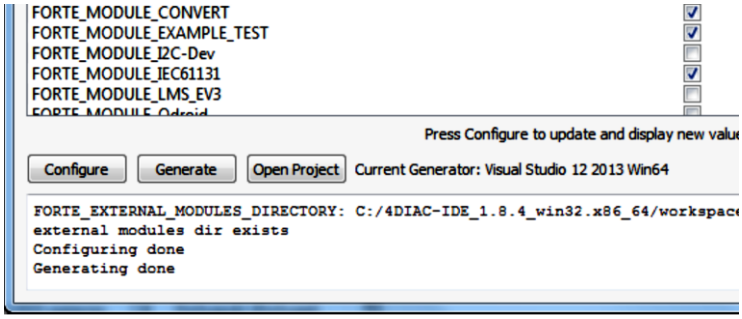


Figura 11-33. Abertura do projeto no Microsoft Visual Studio

Durante a geração do código, o CMake coloca no diretório definido como saída todos os ficheiros necessários para a compilação do FORTE distribuindo-os do seguinte modo:

- *src/modules* – pasta que contém o código-fonte (cpp, h) de todos os blocos de função disponíveis para FORTE;
- *src/Release* – pasta que contém o executável FORTE (.exe) após a compilação;
- *src_gen* – pasta que contém os arquivos de objeto gerados durante a compilação.

Anexo G – Inicialização dos RPis

a) *Socket* de associação a grupo de utilizadores.

```
$ route add -net 224.0.0.0 netmask 240.0.0.0 eth0
```

b) *Script 4diac_st.sh* desenvolvido em *Bash* para inicialização dos Raspberry Pi.

```
1:  #!/bin/bash
2:  sudo route add –net 224.0.0.0 netmask 240.0.0.0 eth0
3:  echo “Adicionado grupo multicast, IP_MEMBERSHIP”
```

Figura 11-34. Script de associação do IP ao grupo

Note-se, no entanto, que a execução do *script* anterior só poderá ser realizada, automaticamente, se adicionarmos no final do ficheiro *autostart* (alocado em */home/pi/.config/lxsession/LXDE-pi*, ficheiro oculto) a seguinte linha, por exemplo:

```
lxterminal -e bash /home/pi/FORTE/4diac_st.sh
```

c) *Script 4diac_st.sh*, ativação do servidor e do cliente NTP

```
4:  echo
5:  sudo /etc/init.d/ntp Start
6:  echo “NTP inicializado …”
```

Figura 11-35. Script 4diac_st.sh de inicialização da sincronização dos relógios (NTP)

Script 4diac_st.sh completo

```
1:  #!/bin/bash
2:  sudo route add –net 224.0.0.0 netmask 240.0.0.0 eth0
3:  echo “Adicionado grupo multicast, IP_MEMBERSHIP”
4:  echo
5:  sudo /etc/init.d/ntp Start
6:  echo “NTP inicializado …”
```

Anexo H – Instalação do 4DIAC-IDE

Neste anexo serão apresentados alguns passos para a instalação do aplicativo 4DIAC-IDE no ambiente Windows a partir do código fonte, bem como de uma versão executável. Assim, começaremos por executar diferentes passos:

- Instalar o “TortoiseHg” (http://tortoisehg.bitbucket.org/

- Criar uma pasta de destino e clonar para esta o conteúdo do repositório mercurial (http://hg.code.sf.net/p/fordiac/fordiac-ide)

- Criar a pasta "typelibrary" e clonar para esta o conteúdo do repositório mercurial (http://hg.code.sf.net/p/fordiac/fordiac-lib)

- Instalar o “Modling Tools” da ferramenta Eclipse (http://eclipse.org/);

- Executar “Eclipse” (Figura 11-36):

  - Importar o projeto 4DIAC-IDE (File→Import→General→Existing Project Into Workspace;
  - Selecionar o *plugin* “org.fordiac.ide”;
  - Copiar para o diretório do "eclipse.exe" a pasta "typelibrary" e "template";
  - Abrir a secção “4DIAC-IDE.product”;
  - Escolher a opção "*features*";
  - Escolher a opção “*Synchronize*” da secção "Testing", pasta Overview, do projeto 4DIAC-IDE.product;
  - Executar na aplicação eclipse “*Launch an Eclipse application*”.

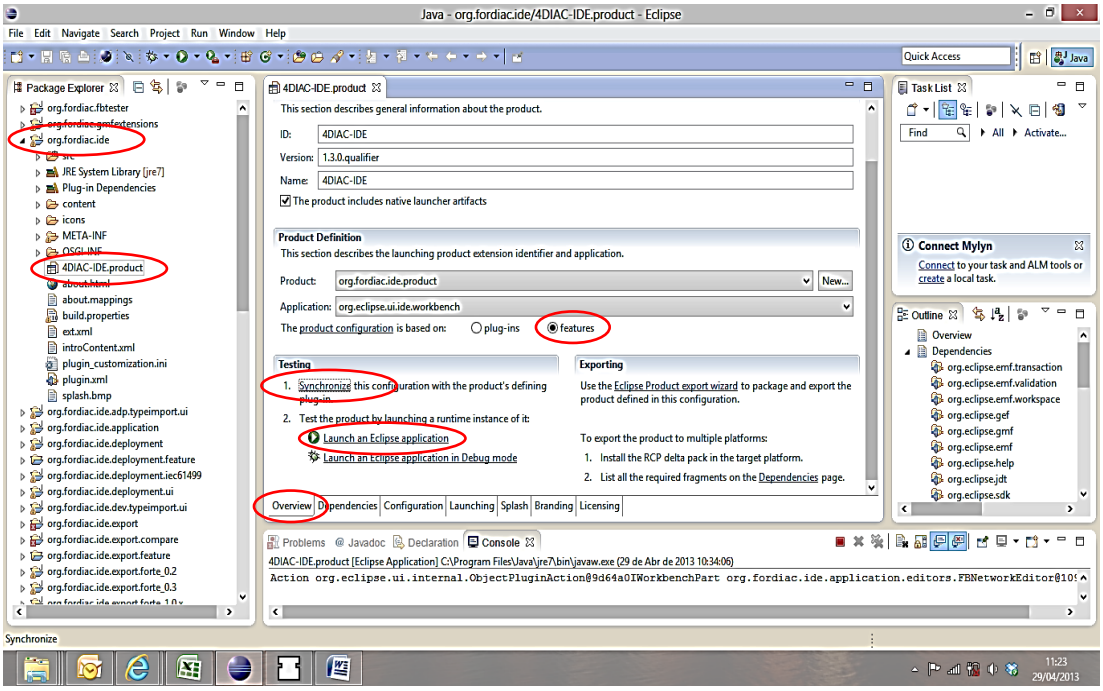


Figura 11-36. Ferramenta Eclipse após importação de projeto 4DIAC-IDE.project

Obtém-se a aplicação de edição 4DIAC-IDE apresentada na Figura 11-4.

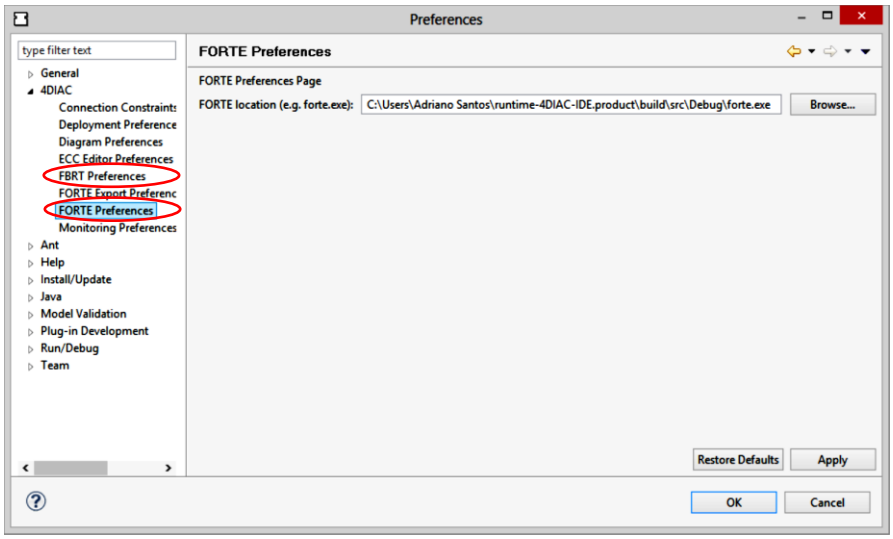


Figura 11-37. Configuração do 4DIAC, menu "Window, Preferences"

- Descarregar de HOLOBLOC, Inc. "FBDK - The Function Block Development Kit" (*http://www.holobloc.com/doc/fbdk/index.htm*)

- Na aplicação 4DIAC (Window→Preferences→4DIAC→FORTE Preferences) indicar:

  - Localização do "forte.exe".
  - Localização do "fbrt.jar"

Anexo I – Código do bloco função "FB_MUL_INT"

C:\4DIAC-T\Mercurial\src_forte\src\modules\math\FB_MUL_INT.cpp 1

```
/*******************************************************************************
 *** FORTE Library Element
 ***
 *** Name: FB_MUL_INT
 *** Description: Multiplies two INT values
 *** Version:
 ***     1.0: 2007-06-26/TS - PROFACTOR GmbH -
 ******************************************************************************/

#include "FB_MUL_INT.h"
#ifdef FORTE_ENABLE_GENERATED_SOURCE_CPP
#include "FB_MUL_INT_gen.cpp"
#endif
#include <devlog.h>

DEFINE_FIRMWARE_FB(FB_MUL_INT, g_nStringIdFB_MUL_INT)

const CStringDictionary::TStringId FB_MUL_INT::scm_anDataInputNames[] = {g_nStringIdIN1, g_nStringIdIN2};

const CStringDictionary::TStringId FB_MUL_INT::scm_anDataOutputNames[] = {g_nStringIdOUT};
const CStringDictionary::TStringId FB_MUL_INT::scm_aunDIDataTypeIds[] = {g_nStringIdINT, g_nStringIdINT};
const CStringDictionary::TStringId FB_MUL_INT::scm_aunDODataTypeIds[] = {g_nStringIdINT};

const TForteInt16 FB_MUL_INT::scm_anEIWithIndexes[] = {0};
const TDataIOID FB_MUL_INT::scm_anEIWith[] = {0, 1, 255};
const CStringDictionary::TStringId FB_MUL_INT::scm_anEventInputNames[] = {g_nStringIdREQ};

const TDataIOID FB_MUL_INT::scm_anEOWith[] = {0, 255};
const TForteInt16 FB_MUL_INT::scm_anEOWithIndexes[] = {0};
const CStringDictionary::TStringId FB_MUL_INT::scm_anEventOutputNames[] = {g_nStringIdCNF};

const SFBInterfaceSpec FB_MUL_INT::scm_stFBInterfaceSpec = {
  1,
  scm_anEventInputNames,
  scm_anEIWith,
  scm_anEIWithIndexes,
  1,
  scm_anEventOutputNames,
  scm_anEOWith,
  scm_anEOWithIndexes,
  2,
  scm_anDataInputNames, scm_aunDIDataTypeIds,
  1,
  scm_anDataOutputNames, scm_aunDODataTypeIds,
  0,
  0
};

void FB_MUL_INT::alg_REQ(void){
OUT() = static_cast<TForteInt16>(IN1()*IN2());
}

void FB_MUL_INT::enterStateSTART(void){
  m_nECCState = scm_nStateSTART;
}

void FB_MUL_INT::enterStateREQ(void){
  m_nECCState = scm_nStateREQ;
  alg_REQ();
  sendOutputEvent( scm_nEventCNFID);
}

void FB_MUL_INT::executeEvent(int pa_nEIID){
  bool bRetVal;
  do{
    bRetVal = true;
    switch(m_nECCState){
      case scm_nStateSTART:
        if(scm_nEventREQID == pa_nEIID)
            enterStateREQ();
        else
          bRetVal = false; //no transition cleared
        break;
      case scm_nStateREQ:
        if(1)
            enterStateSTART();
        else
          bRetVal = false; //no transition cleared
        break;
      default:
        DEVLOG_ERROR("The state is not in the valid range! The state value is: %d. The max value can be: 1.", m_nECCState.operator TForteUInt16 ↙
     ());
        m_nECCState = 0; //0 is allways the initial state
        break;
    }
    pa_nEIID = cg_nInvalidEventID;  // we have to clear the event after the first check in order to ensure correct behvior
  }while(bRetVal);
}

FB_MUL_INT::FB_MUL_INT(const CStringDictionary::TStringId pa_nInstanceNameId, CResource *pa_poSrcRes) :
       CBasicFB(
           pa_poSrcRes,
           &scm_stFBInterfaceSpec, pa_nInstanceNameId,
            0, m_anFBConnData, m_anFBVarsData) {
}

FB_MUL_INT::~FB_MUL_INT(){
}
```

Figura 11-38. Código em C++ do bloco função básico "FB_MUL_INT"

Anexo J – Testar FB desenvolvidos com o FBTester

Neste anexo é apresentado o modo de teste dos FB (FBTester). Esta janela permite definir valores de interface para testar um único bloco função. Para isso é necessário que estes tenham já sido exportados e compilados para o FORTE. Para usar esta funcionalidade é necessário que no CMake se selecione a opção FORTE_SUPPORT_MONITORING.

Abrir o bloco função que pretende testar e selecionar o separador FBTester executando-se os passos seguintes de acordo com a figura:

1 Selecionar no menu dropdown Test Configuration a opção *FORTE Remote Tester*;
2 Inicie o Forte dentro da perspetiva de implantação ou como uma ferramenta externa;
3 Pressione o botão *Start Testing FB*;
4 Inicie o teste inserindo entradas e pressionando o evento de entrada associado.

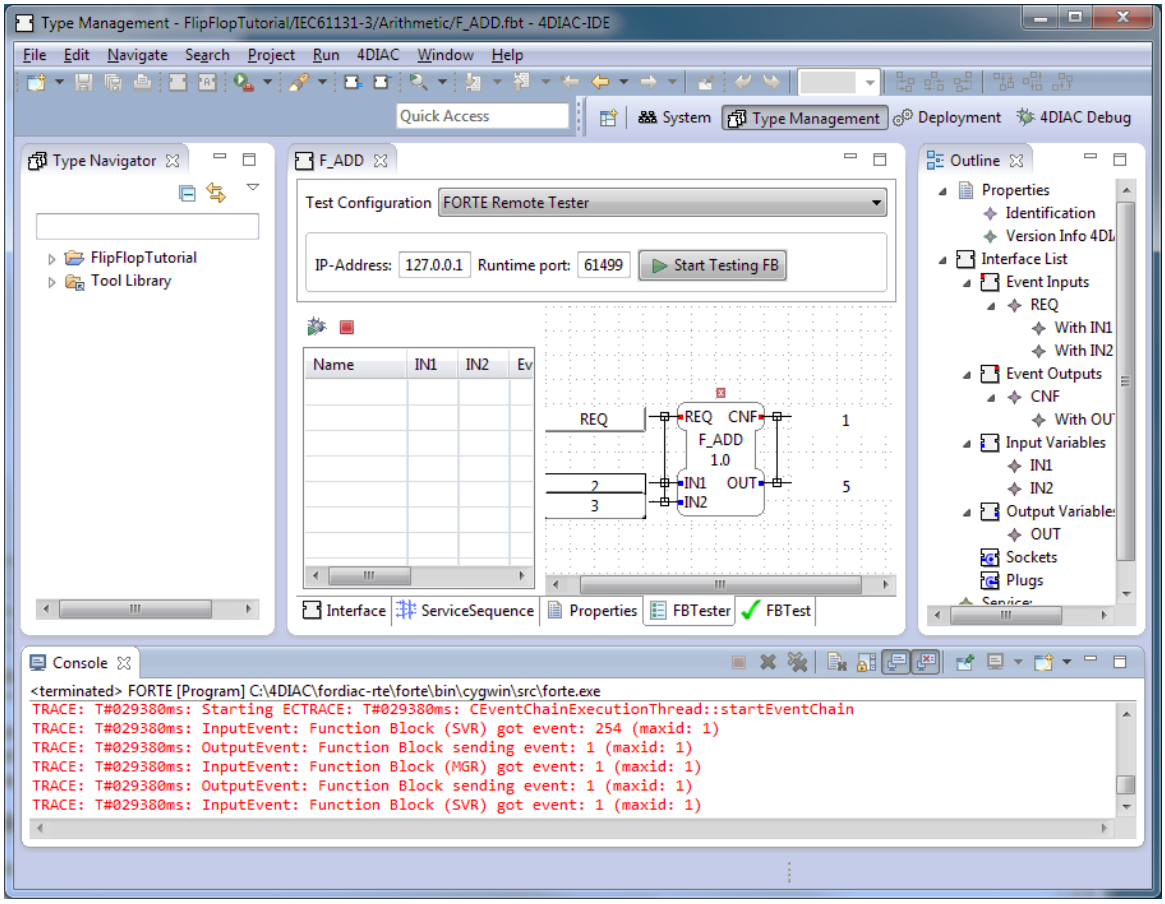


Figura 11-39. Interface FBTester usada no teste de FB desenvolvidos pelo programador

A. Publicações Relacionadas com a Tese

A.1 Publicadas em *proceedings* de conferências

1. Mário de Sousa, Adriano A. Santos (2007). Management of Replicated IEC 61499 Applications. In *Proc. of the 5th IEEE International Conference on Industrial Informatics – INDIN 2007*, Vienna, Áustria, pp 231-236. *http://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=4384761*.
2. Adriano A. Santos, Mário de Sousa (2008). Framework for Management of Replicated IEC 61499 Applications. In *Proc. of the 13th IEEE International Conference on Emerging Technologies and Factory Automation – ETFA 2008*, Hamburgo, Alemanha, pp 200-206. *http://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=4638393.*
3. Adriano A. Santos, Mário de Sousa (2010). Replication in Distributed Systems using IEC 61499 Standard. In *Proc. of the 15th IEEE International Conference on Emerging Technologies and Factory Automation – ETFA 2010*, Bilbao, Espanha. *http://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=5641331.*
4. Adriano A. Santos, Mário de Sousa, Pessoa Magalhães, António F. da Silva (2016). IEC 61499 Replication for Fault Tolerant System. In *Proc. of the 5th International Conference on Integrity-Reliability-Failure – IRF2016*, Porto, Portugal, pp. 849-850. *https://pdfs.semanticscholar.org/fc97/5abfe8561685ad16fb1e4ea99bc9b1c757d8.pdf.*
5. Adriano A. Santos, Mário de Sousa, Pessoa Magalhães, António F. da Silva (2018). An IEC 61499 Replication for Distributed Control Applications. In *Proc. of the 16th International Conference on Industrial Informatics – INDIN2018*, Porto, Portugal, pp. 362-367. *https://ieeexplore.ieee.org/abstract/document/8471958.*
6. Adriano A. Santos, Mário de Sousa (2018). Replication Strategies for Distributed IEC 61499 Applications. In *Proc. of the 44th Annual Conference of the IEEE Industrial Electronics Society – IECON'18,* Washington D.C., USA, pp. 2225-2230. *https://ieeexplore.ieee.org/document/5641331.*

A.2 Publicadas em revistas internacionais

B. Citações relacionadas com a Tese

B.1 Dissertações e Teses

Mestrados

1. Marcelo Vladimir García Sánchez (2013). Implementación de Sistemas Empotrados de Control Distribuidos bajo el Estándar IEC-61499. Universidad del País Vasco, Escuela Técnica Superior de Ingeniería, setembro *(Tese de Mestrado)*. *http://repositorio.educacionsuperior.gob.ec/bitstream/28000/1535/1/T-SENESCYT-00668.pdf*, (accessed 14 fevereiro, 2019).

Doutoramentos

1. Li Hsien Yoong (2010). Modelling and Synthesis of Safety-critical Software with IEC 61499. University of Auckland, DEEC, junho *(Tese de Doutoramento)*. *https://researchspace.auckland.ac.nz/bitstream/handle/2292/6691/whole.pdf?sequence=2*, (accessed 14 fevereiro, 2019).
2. Yan, Jeffrey (2015). Provisions for Adaptability in Distributed Intelligent Cyber Physical Systems. University of Auckland, junho *(Tese de Doutoramento).* *https://researchspace.auckland.ac.nz/handle/2292/25014*, (accessed 14 fevereiro, 2019)

B.2 Publicadas em capítulos de livros

1. Li Hsien Yoong, Partha S. Roop, Zeeshan E. Bhatti, Matthew M. Y. Kuo (2015). Efficient Code Synthesis from Function Blocks. In «Model-Driven Design Using IEC 61499 - A Synchronous Approach for Embedded and Automation Systems». Springer. ISBN 978-3-319-10520-8. Cp. 5, pp 93-122. *https://link.springer.com/chapter/10.1007/978-3-319-10521-5_5.*

B.3 Publicadas em *proceedings de conferências*

1. Guadalupe Morán, Federico Pérez, Darío Orive, Elisabet Estévez, Marga Marcos (2011). Automatic Composition of IEC 61499 Distributed Control Applications. In *Proc. of the 16th IEEE International Conference on Emerging Technologies and Factory Automation – ETFA 2011*, Toulouse, pp. 1-7, setembro. *http://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=6059116.*
2. Priego, R., Armentia, A.; Orive, D.; Marcos, M. (2013). Supervision-based reconfiguration of industrial control systems. In *Proc. of the IEEE 18th Conference on Emerging Technologies & Factory Automation – ETFA 2013*, Cagliari, pp. 1-4, setembro. *https://ieeexplore.ieee.org/abstract/document/6648130.*

3. Mário de Sousa, Christos Chrysoulas, Aydin E. Homay (2015). Building Fault Tolerant Industrial Applications Based on IEC 61499. In *Proc. of the 25th International Conference on Flexible Automation and Intelligent Manufacturing – FAIM 2015,* Wolverhampton, junho.
   https://www.researchgate.net/publication/280314802_Building_Fault_Tolerant_Industrial_Applications_Based_on_IEC_61499 (accessed 14 fevereiro, 2018).
4. Glatz, B., Schuster, H., Horauer, M., Rauscher, T. e Obermaisser, R. (2016). Fault injection for IEC 61499 applications. In *21 IEEE International Conference on Emerging Technologies and Factory Automation – ETFA 2016*, Berlin, pp. 1-3, September.
   https://ieeexplore.ieee.org/document/7733676, (accessed 14 fevereiro, 2019).
5. Prenzel, Laurin; Provost, Julien (2019). FBBeam: An Erlang-based IEC 61499 Implementation. In *17th International Conference on Industrial Informatics (INDIN 2019)*, Helsinki, Finland, Finland, pp. 629-634, July.
   https://ieeexplore.ieee.org/document/8972123/authors#authors (accessed 1 fevereiro, 2020).

B.4 Publicadas em *Jornais Internacionais*

1. Anupam Singh, Raghuraj Suryavanshi, Divakar Yadav (2019). Formal Development and Verification of Quorum Based Static Voting Replica Control Protocol Using Event-B. In *International Journal of Advanced Science and Technology*. Vol. 28, No. 20, (2019), pp. 133-144.
   http://sersc.org/journals/index.php/IJAST/article/view/2705/1905, (accessed 1 janeiro, 2020).