UNIVERSIDADE FEDERAL DO RIO GRANDE DO SUL
INSTITUTO DE INFORMÁTICA
CURSO DE ENGENHARIA DE COMPUTAÇÃO

LUIZ GUSTAVO AIRES GOMES

Injeção de falhas de comunicação sobre implementações do protocolo EtherCAT

Monografia apresentada como requisito parcial para a obtenção do grau de Bacharel em Engenharia de Computação.

Orientadora: Profª. Dra. Taisy Silva Weber
Co-orientador: Prof. Dr. Sérgio Luis Cechin

Porto Alegre
2016

“Se vi mais longe, foi por estar de pé sobre ombros de gigantes”

— ISAAC NEWTON

AGRADECIMENTOS

Primeiramente, gostaria de agradecer aos meus pais Jorge Gomes e Maria da Glória Gomes e minha irmã Mônica Cristina Gomes pelo apoio incondicional e incentivo desde o primeiro dia neste curso, que sem isto, nada disto seria possível.

Gostaria de agradecer aos professores Taisy Weber e Sérgio Cechin por todas as críticas construtivas durante os últimos meses que ajudaram a moldar este trabalho e às discussões e argumentações quanto a escrita de artigos e deste trabalho, que me ajudaram significativamente a ter uma visão mais crítica e objetiva. Além disso gostaria de expressar minha gratidão a todos os demais professores e funcionários da Universidade Federal do Rio Grande do Sul pelo ensino de qualidade e apoio que de maneira direta ou indireta também contribuíram para este momento

Também gostaria de agradecer ao amigo Rodrigo Dobler, onde seu trabalho de Mestrado serviu de base inicial para que este trabalho pudesse ser iniciado e a todos os colegas do Laboratório de Automação e Integração de Sistemas, em especial a João de Moraes, por todos os momentos de discussão e construção de ideias ao longo deste trabalho.

Por fim, gostaria de agradecer aos colegas e grandes amigos que seja por alguma ajuda ou pelas boas risadas entre aulas tornaram esse período prazeroso: Alexandre John, Eduardo Cavichioli, Lucas Alpi, Matheus Souza, Pablo Soares, Pablo Bodmann, Thiago Martins, Vicente Guedes, Victoria Ramos e Willian Vidal.

RESUMO

Este trabalho tem como objetivo o desenvolvimento do injetor de falhas TANATTOS para o protocolo EtherCAT, que por sua vez, é baseado no protocolo Ethernet. Para o protocolo EtherCAT, a cada novo dispositivo desenvolvido, é necessário que os desenvolvedores façam suas próprias implementações e as submetam para o processo de validação conforme a finalidade. De posse de um injetor de falhas, tais implementações serão testadas frente à ocorrência de falhas, auxiliando no processo da validação de sua dependabilidade. O injetor proposto neste trabalho usa como base o injetor de falhas FITT (*Fault Injection Test Tool*), desenvolvido para protocolos seguros que seguem a norma IEC 61508, em especial o protocolo PROFIsafe. FITT intercepta a comunicação entre dispositivos mestre e escravo, alterando o comportamento normal da comunicação entre estes dispositivos. Testes realizados demonstram que além de a adaptação de FITT para o protocolo EtherCAT ter sido bem-sucedida, o injetor de falhas TANATTOS desenvolvido ao final deste trabalho é capaz de alterar o fluxo de comunicação entre um mestre e um escravo, provando que a injeção de falhas foi efetuada com sucesso.

Palavras-chave: Injeção de falhas. Protocolos de comunicação. Indústria 4.0. Tolerância a falhas.

Communication fault injection on EtherCAT protocol implementations

ABSTRACT

This work aims to develop the TANATTOS fault injector for the EtherCAT protocol, which in turn is based on the Ethernet protocol. For the EtherCAT protocol, for each new device to be developed, it is necessary for developers to make their own implementations and submit them to a validation process. With a fault injector, such implementations will be tested against the occurrence of failures, helping in the process of validating their dependability. The injector proposed in this work is based on the *Fault Injection Test Tool* (FITT), developed for safe protocols that follow the IEC 61508 standard, especially the PROFIsafe protocol. FITT intercepts the communication between master and slave devices, changing the normal behavior of their communication. Tests carried out demonstrate that in addition to the positive results in FITT's adaptation to the EtherCAT protocol, the TANATTOS fault injector developed at the end of this work is able to change the communication flow between a master and a slave, proving that the injection of failures was successful.

Keywords: Fault injection. Communication protocols. Industry 4.0. Fault-tolerance.

LISTA DE FIGURAS

LISTA DE TABELAS

LISTA DE ABREVIATURAS E SIGLAS

ASIC	Application Specific Integrated Circuits
CPU	Central Unit Processing
CRC	Cyclic Redundancy Check
CSV	Comma-Separated Values
DNA	Direct NIC Access
DWORD	Double Word
EEPROM	Electrically-Erasable Programmable Read-Only Memory
ETG	EtherCAT Technology Group
EtherCAT	Ethernet for Control Automation Technology
FCS	Frame Check Sequence
FIFO	First In First Out
FITT	Fault Injection Test Tool
FMMU	Fieldbus Memory Management Unit
FPGA	Field Programmable Gate Array
GB	Giga Bytes
GHz	Gigahertz
IEC	International Electrotechnical Commission
IP	Internet Protocol
IRQ	Interrupt Request
LRD	Logical Read
LTS	Long Term Support
LVDS	Low-Voltage Differential Signaling
LWR	Logical Write
MTBF	Mean Time Between Failures

MTTF	Mean Time to Failure
NAPI	New API
NIC	Network Interface Card
OSI	Open Systems Interconnection
ProfiBUS	Process Field Bus
PROFINET	Process Field Net
PROFIsafe	PROFINET Safety
RAM	Random Access Memory
SWIFI	Software-Implemented Fault Injection
UCP	Unidade Central de Processamento
UDP	User Datagram Protocol
XML	eXtensible Markup Language
VHDL	Very High-speed integrated circuit Hardware Description

SUMÁRIO

1 INTRODUÇÃO

Este capítulo apresenta as motivações deste trabalho e uma contextualização para o desenvolvimento de um injetor de falhas para o protocolo EtherCAT, através da apresentação de conceitos a ele relacionados e quais os objetivos a serem alcançados levando em conta tais conceitos.

1.1 Motivação

Com sistemas de controle digital se comportando cada vez mais como uma grande rede interconectada e o custo reduzido de equipamentos com interface Ethernet, surgiu um interesse econômico em introduzir sistemas de comunicação baseadas em Ethernet no domínio industrial (NEUMANN, 2007). Portanto, soluções em comunicação baseadas em Ethernet foram desenvolvidas especialmente para atender requisitos industriais como tempo e sincronismo.

Uma destas soluções é o protocolo EtherCAT (“EtherCAT Technology Group | HOME”, 2016), planejado para dar suporte ao processamento de informações, monitoramento e controle de sistemas de controle para diversos setores industriais de diferentes domínios, de maneira simples. Como o padrão EtherCAT não define uma implementação padrão pré certificada (ZHOU; HU, 2011) para qualquer dispositivo em geral, o protocolo deve ser implementado pelos desenvolvedores e sua validação feita pelo grupo órgão certificador responsável (EtherCAT Technology Group). Infelizmente não estão disponíveis ferramentas que auxiliem os desenvolvedores a testar seus códigos de forma automatizada frente à ocorrência de falhas previstas na especificação. Por conta disto, cada nova implementação deve ser enviada e validada pelo órgão certificador. Com o auxílio de uma ferramenta de injeção de falhas, o processo de validação se torna mais rápido, visto que a implementação já terá sido testada na ocorrência de falhas antes de ser submetida ao órgão certificador.

Atualmente existem ferramentas que realizam monitoramento de dispositivos que já executam o protocolo EtherCAT, porém com nenhuma delas é possível determinar o comportamento destes dispositivos sob falhas de comunicação.

1.2 Objetivos

Um dos objetivos deste trabalho era avaliar a possibilidade de usar o injetor de falhas FITT (DOBLER, 2016) para validar implementações de EtherCAT. O injetor FITT foi desenvolvido para validar PROFIsafe (“PROFIsafe”, 2016) um protocolo de comunicação

segura que executa sobre ProfiBUS ou ProfiNET e cuja especificação é diferente de EtherCAT, exceto por alguns tipos de falhas que tanto PROFIsafe como EtherCAT devem tolerar. Assim foi criado um ambiente experimental onde equipamentos já validados e que executam EtherCAT foram utilizados como mestre e escravo e nesses equipamentos foram injetadas falhas usando FITT. Este cenário serviu para que a portabilidade do injetor de falhas FITT fosse avaliada, ou seja, quais funcionalidades presentes podem ser utilizadas e quais precisam ser modificadas a fim de que falhas possam ser injetadas no protocolo EtherCAT.

O segundo objetivo é aproveitar os resultados do experimento de portabilidade para criar um novo injetor específico para EtherCAT que será utilizado posteriormente como parte de uma arquitetura de validação de uma implementação do protocolo EtherCAT sendo desenvolvido pelo grupo de pesquisa. Tal implementação está sendo desenvolvida em um *Field Programmable Gate Array* (FPGA), e futuramente originará um Circuito Integrado (ASIC).

1.3 Organização do texto

Este trabalho apresenta o processo de desenvolvimento do injetor de falhas TANATTOS, que serve de prova de conceito para uma possível extensão / adaptação do injetor FITT para o protocolo EtherCAT.

O capítulo 2 apresenta uma visão geral do conceito de injeção de falhas, relacionando-o com outro conceito importante: o de dependabilidade. Em seguida são apresentados os objetivos da injeção de falhas e seus diferentes tipos e técnicas. No capítulo 3 é introduzido o protocolo EtherCAT com suas principais características e princípio de funcionamento bem como uma breve descrição da norma IEC 61158, na qual o protocolo se baseia.

No capítulo 4 é feito um apanhado de trabalhos relacionados, onde foram realizadas pesquisas de diferentes ferramentas que possuem características a serem levadas em consideração no desenvolvimento do trabalho.

O capítulo 5 é dedicado exclusivamente a falar sobre o injetor de falhas FITT. Por servir de base para este trabalho, uma descrição mais aprofundada é feita, discutindo seu funcionamento e características relevantes que foram levadas em conta. Já o capítulo 6 aborda o desenvolvimento e questões de projeto do injetor de falhas TANATTOS, com uma análise crítica das ferramentas pesquisadas e como o injetor aplica as falhas ao sistema. No capítulo 7 é descrito o ambiente experimental utilizado para a validação do injetor de falhas desenvolvido e a metodologia de testes utilizada. No capítulo final são apresentadas as conclusões a partir

dos resultados obtidos e possíveis melhorias a serem realizadas no injetor desenvolvido em trabalhos futuros.

2 INJEÇÃO DE FALHAS

Este capítulo apresenta conceitos referentes à técnica de injeção de falhas, como uma visão geral dos principais aspectos, seus objetivos e por fim são apresentadas as técnicas de injeção de falhas existentes.

2.1 O conceito de Dependabilidade

A definição original de dependabilidade (AVIŽIENIS *et al.*, 2004) é a *habilidade de entregar um serviço que pode ser garantidamente confiável*. Tal definição, porém depende do que pode ser entendido como confiável. Uma definição alternativa é de que *a dependabilidade de um sistema é a habilidade de evitar que falhas de serviço sejam mais frequentes e mais severas do que o aceitável* (AVIŽIENIS et al., 2004). O conceito de confiança pode ser derivado da dependabilidade de um sistema, mais especificamente, da *dependência* do sistema. Por exemplo, a dependência de um sistema A em um outro sistema B representa a extensão da dependabilidade do sistema A que é afetada pelo sistema B (AVIŽIENIS et al., 2004).

Com o passar dos anos, dependabilidade se tornou um conceito integrado que engloba outros atributos: *disponibilidade* (prontidão para realizar um serviço de maneira correta), *confiabilidade* (continuidade de um serviço correto), *segurança funcional* (*safety* - ausência de consequências catastróficas para o(s) usuário(s) e o ambiente), *integralidade* (ausência de alterações impróprias no sistema) e *manutenibilidade* (habilidade de passar por modificações e reparos).

Dependabilidade pode ser medida de maneira empírica através de *life testing* (CLARK; PRADHAN, 1995), que é um processo feito sob condições controladas para determinar como e quando um produto irá falhar no seu ambiente de destino. No entanto, o tempo necessário para se obter um número estatisticamente significativo de falhas faz com que o *life testing* se torne impraticável para a maior parte dos sistemas tolerantes a falhas. Já com injeção de falhas é possível acelerar a ocorrência de falhas, podendo se aplicar as falhas sobre um protótipo do sistema, não só sobre modelos.

2.2 Visão geral

Injeção de falhas pode ser visto como uma técnica para validar a dependabilidade de um sistema, onde o seu comportamento é observando através de experimentos controlados, com falhas sendo deliberadamente introduzidas neste sistema (YU; BASTIEN; JOHNSON, 2005; ZIADE; AYOUBI; VELAZCO, 2004). É um processo que complementa e não compete ou exclui outras abordagens de validação, como modelagem e simulação.

Aplicada inicialmente a sistemas centralizados, como arquiteturas tolerantes a falhas no início dos anos 1970, injeção de falhas era utilizada quase que exclusivamente no meio industrial para medir a cobertura e latência de falhas em sistemas de alta confiabilidade. A partir da década seguinte, o meio acadêmico começou a conduzir experimentos de injeção de falhas ativamente em pesquisas relacionadas a teste de sistemas, tendo foco principalmente em entender a propagação dos erros e analisar a eficiência de novos métodos de detecção de falhas. Com o passar dos anos, injeção de falhas passou a ser usada em sistemas distribuídos e na *Internet* (YU; BASTIEN; JOHNSON, 2005).

Em um experimento de injeção de falhas, inicialmente se assume que o sistema está em um estado correto. Assim que a falha é introduzida e uma carga de trabalho é aplicada, dois comportamentos podem ser observados (NATELLA; COTRONEO; MADEIRA, 2016). Primeiro, a falha não é ativada e permanece *dormente*. Neste caso, nenhum defeito é produzido. Segundo, a falha é ativada e causa um erro. Neste ponto, tal erro pode se *propagar*, corrompendo outras partes do sistema, até que um defeito ocorra, pode ficar *latente* no sistema, isto é, presente, mas não sinalizado ou pode ser *mascarado* ou *corrigido* por mecanismos de tolerância a falhas (AVIŽIENIS *et al.*, 2004).

Com base nos resultados obtidos em um experimento de injeção de falhas, alguns parâmetros podem ser inferidos, como a probabilidade de uma falha causar um erro, a probabilidade de que o sistema irá realizar as ações necessárias para se recuperar deste erro (*cobertura* de falhas) (KOREN; KRISHNA, 2007), a *latência* de falhas (diferença temporal entre a ocorrência da falha e seu correspondente erro) (DREBES, 2005) o tempo médio entre falhas (*Mean Time Between Failures* – MTBF) e o tempo médio para uma falha ocorrer (*Mean Time to Failure* – MTTF (CLARK; PRADHAN, 1995).

Além de fornecer tais parâmetros, injeção de falhas também pode avaliar diretamente métricas de dependabilidade, sendo particularmente útil para medir atributos do sistema que

são difíceis de modelar analiticamente, como por exemplo a influência de uma dada carga de trabalho na dependabilidade (CLARK; PRADHAN, 1995).

2.3 Objetivos da injeção de falhas

Além de fornecer parâmetros para medição da cobertura e latência de falhas, injeção de falhas pode avaliar diretamente métricas de dependabilidade. É particularmente útil para medir atributos do sistema que são difíceis de modelar analiticamente, por exemplo, a influência que uma dada carga de trabalho exerce na dependabilidade.

Injeção de falhas também busca determinar quando a resposta de um sistema corresponde com sua especificação, na presença de um dado conjunto de falhas. Os tipos de falhas a serem injetadas, onde e quando estas falhas serão injetadas são parâmetros decididos após uma análise inicial do sistema (YU; BASTIEN; JOHNSON, 2005).

Outros dois grandes objetivos complementares também podem ser identificados: *validação* e *auxílio no projeto* (ARLAT et al., 1990).

2.3.1 Validação

Neste processo, a injeção de falhas está voltada para o conceito de cobertura, ou seja, como se ter convicção nos métodos e mecanismos usados para garantir a confiança do sistema (ARLAT *et al.*, 1990).

Neste contexto, existem duas grandes preocupações que dizem respeito à validação dos processos de verificação: revelar falhas durante todas as fases do processo de desenvolvimento e a validação dos mecanismos de tolerância a falhas, destinados a atingir a dependabilidade do sistema em sua fase operacional (ARLAT *et al.*, 1990).

Ainda no processo de validação, injeção de falhas tem papel ativo em outras duas atividades: *prevenção* e *remoção de falhas* (ARLAT et al., 1990). Prevenção de falhas busca quantificar a confiança que pode ser atribuída a um sistema, estimando o número e as consequências de possíveis falhas no sistema. Em um aspecto qualitativo, a prevenção de falhas visa identificar, classificar e ordenar os modos de falhas, ou identificar combinações de eventos que possam levar à eventos não desejados (YU; BASTIEN; JOHNSON, 2005). Quanto à remoção de falhas, o objetivo é reduzir a presença de falhas no projeto e implementação do sistema. Neste cenário, consequências seriam notadas na forma de um comportamento diferente do esperado quando falhas deveriam ser tratadas (YU; BASTIEN; JOHNSON, 2005). Assim,

na remoção de falhas a análise é puramente qualitativa, voltada a checar os procedimentos de verificação e mecanismos de tolerância a falhas (ARLAT *et al.*, 1990).

Na prática, prevenção e remoção de falhas são frequentemente utilizadas em sequência (ZIADE; AYOUBI; VELAZCO, 2004). Por exemplo, após um resultado negativo obtido em uma prevenção de falhas, um ou mais testes de remoção de falhas devem ser realizados. Tais testes ajudam a melhorar o sistema, para então uma nova bateria de testes de prevenção de falhas ser realizado, dando início a um novo ciclo.

2.3.2 Auxílio no projeto

Injeção de falhas pode ser aplicada em diferentes etapas de um processo de desenvolvimento e os resultados negativos obtidos são usados como realimentação para iniciar iterações de melhorias nos procedimentos de teste e mecanismos de tolerância a falhas (ARLAT *et al.*, 1990). Neste contexto, o teste funcional com injeção de falhas realizado em um protótipo durante o desenvolvimento de um sistema pode identificar erros nos mecanismos de tolerância a falhas e fornecer um retorno sobre a eficiência destes mecanismos. Quando o sistema estiver pronto, injeção de falhas pode ser usada para observar sintomas de erros ou falhas associadas com cada componente faltoso (CLARK; PRADHAN, 1995).

Outro objetivo da injeção de falhas como auxílio no projeto é a criação de dicionários de falhas, que podem ser usados no desenvolvimento de procedimentos de diagnóstico (ARLAT *et al.*, 1990).

Em resumo, injeção de falhas fornecem meios para entender como sistemas computacionais se comportam na presença de falhas. Tal conhecimento irá levar a sistemas melhor projetados e com alta dependabilidade (CLARK; PRADHAN, 1995).

2.4 Tipos de falhas

Uma *falha* é definida como sendo a possível causa de um estado incorreto em um sistema. Este desvio no funcionamento correto é chamado de *erro* (AVIŽIENIS et al., 2004). Um *defeito* é um evento que ocorre quando um determinado serviço é entregue de maneira incorreta, ou seja, um estado errado é percebido pelos usuários externos do sistema. Um *modelo de falhas* descreve as falhas que são esperadas e previstas pelo sistema durante sua operação (NATELLA; COTRONEO; MADEIRA, 2016).

Falhas podem ocorrer em qualquer estágio do desenvolvimento de um sistema. A maioria das falhas que ocorrem antes da finalização do sistema como um todo são descobertas e eliminadas através de teste. Falhas que não são removidas podem diminuir a dependabilidade de um sistema (ZIADE; AYOUBI; VELAZCO, 2004).

Falhas de *hardware* ocorrem durante sua operação e podem ser classificadas de acordo com sua duração: falhas *permanentes*, *transientes* e *intermitentes* (ZIADE; AYOUBI; VELAZCO, 2004). Falhas permanentes são causadas por dano irreversível a algum componente físico do sistema. Falhas transientes causam um mal funcionamento durante um certo período de tempo, desaparecendo em seguida fazendo com que o funcionamento normal seja retomado (KOREN; KRISHNA, 2007). Podem ser causadas por condições do ambiente, possuem ocorrência maior do que falhas permanentes e são mais difíceis de se detectar (ZIADE; AYOUBI; VELAZCO, 2004). Falhas intermitentes, ao contrário das falhas transientes, nunca desaparecem por completo, oscilando entre períodos de repouso e atividade (KOREN; KRISHNA, 2007). Podem ser causadas por *hardware* instável e são reparadas com substituição de componentes ou alterações no projeto (ZIADE; AYOUBI; VELAZCO, 2004).

Outras categorias de falhas em *hardware* são *benignas* e *maliciosas* (KOREN; KRISHNA, 2007). Uma falha que faça o sistema deixar de funcionar por exemplo é chamada benigna, pois além de ser um comportamento errôneo evidente, não deixa o sistema chegar a um estado incorreto de funcionamento. Já falhas maliciosas, ou Bizantinas (KOREN; KRISHNA, 2007), fazem com que o sistema apresente alguma das seguintes características: (1) produza saídas erradas que não são tão evidentes; (2) produza saídas que parecem razoavelmente corretas mas que na verdade são incorretas ou (3) fazem com que o sistema se comporte de maneira maliciosa, enviando valores de saída diferentes para seus todos ou alguns de seus destinatários, quando deveria enviar os mesmos valores a todos.

Falhas em *software* por outro lado, são causadas por erros em sua especificação, projeto ou codificação (CLARK; PRADHAN, 1995). Mesmo que o *software* não danifique fisicamente um sistema após ser instalado, falhas ainda podem se manifestar posteriormente durante sua operação, geralmente sob cargas de trabalho pesadas ou incomuns (CLARK; PRADHAN, 1995).

2.5 Técnicas de injeção de falhas

Com injeção de falhas se consolidando ao passar dos anos como um método necessário para validar a dependabilidade de um sistema, diversas técnicas foram desenvolvidas para se injetar falhas em um protótipo ou modelo do sistema, podendo ser divididas em cinco categorias: baseadas em *hardware*, *software*, simulação, emulação e híbrida (ZIADE; AYOUBI; VELAZCO, 2004).

2.5.1 Injeção de falhas baseada em *Hardware*

Se utiliza de *hardware* adicional para introduzir falhas no *hardware* já existente no sistema-alvo. Este *hardware* adicional é projetado especialmente para injetar determinados tipos de falhas no sistema e avaliar o resultado obtido (ZIADE; AYOUBI; VELAZCO, 2004). Dependendo das falhas e do local onde serão injetadas, a abordagem baseada em *hardware* pode se encaixar em duas categorias: injeção em *hardware* com e sem contato (HSUEH; TSAI; IYER, 1997).

Em injeção de falhas em *hardware* com contato, o injetor possui contato físico com o sistema-alvo, produzindo mudanças na sua tensão ou corrente (HSUEH; TSAI; IYER, 1997). Com esta abordagem é possível injetar falhas do tipo *stuck-at* (onde um terminal ou pino tem seu valor fixado em nível lógico alto ou baixo) (NATELLA; COTRONEO; MADEIRA, 2016), inversão do valor dos sinais dos pinos e operações lógicas do tipo AND e OR entre pinos adjacentes (ZIADE; AYOUBI; VELAZCO, 2004).

Como vantagens tem-se a habilidade de modelar falhas permanentes a nível de pinos e dispensa um modelo previamente desenvolvido. Por outro lado, este tipo de injeção de falhas pode acarretar danos ao sistema se feita de maneira descuidada e pode possuir baixa portabilidade entre diferentes sistemas-alvo adjacentes (ZIADE; AYOUBI; VELAZCO, 2004).

2.5.2 Injeção de falhas baseada em *Software*

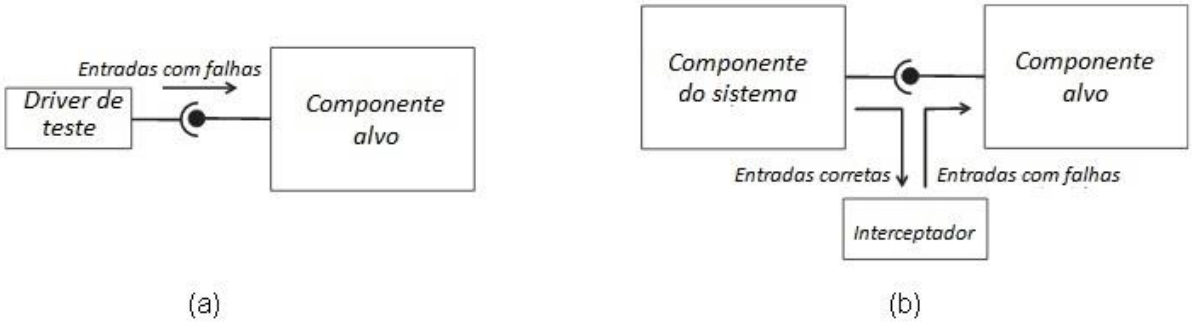
Ao longo dos anos, pesquisadores passaram a demonstrar maior interesse em desenvolver ferramentas de injeção de falhas feitos em *software* (HSUEH; TSAI; IYER, 1997) por diferentes motivos, como o fato de que falhas oriundas de erros de *software* são provavelmente a maior causa de interrupções em um sistema (ZIADE; AYOUBI; VELAZCO,

2004). É uma técnica que não necessita de *hardware* específico e pode ser utilizada para aplicações e sistemas operacionais, o que é difícil de se alcançar com injeção de falhas baseada em *hardware* (HSUEH; TSAI; IYER, 1997).

Também conhecida como SWIFI (*Software-Implemented Fault Injection*) (NATELLA; COTRONEO; MADEIRA, 2016), é uma técnica voltada para detalhes da implementação do sistema (ZIADE; AYOUBI; VELAZCO, 2004), simulando erros de *hardware* ao se injetar falhas em *software*. Pode ser subdividida em *injeção em tempo de compilação* e *injeção em tempo de execução* (HSUEH; TSAI; IYER, 1997). Falhas injetadas em tempo de compilação são inseridas diretamente no código-fonte ou código de montagem (*assembly*) do programa-alvo antes que este seja carregado e executado. O código alterado afeta as instruções do programa, dando início à injeção. Isto gera uma imagem errônea do *software*, que quando executada pelo sistema irá disparar a falha. Em falhas injetadas em tempo de execução, um mecanismo é necessário para ativar a injeção de falhas. Mecanismos frequentemente usados são contadores que depois de decorrido certo tempo (*time-out*) geram uma interrupção que ativa a falha; um desvio na exceção que transfere o controle para o injetor, a fim deste injetar a falha ou ainda inserção de código, onde instruções são adicionadas ao programa-alvo permitindo que a falha seja injetada antes de uma determinada instrução (HSUEH; TSAI; IYER, 1997).

Outra abordagem utilizada se dá através de mecanismos externos ao sistema, chamada de *injeção de erros de interface* (NATELLA; COTRONEO; MADEIRA, 2016). Esta é uma forma de injeção de falhas que corrompe as entradas e saídas que um componente de um sistema maior troca com outros componentes. Ao se injetar erros nas entradas (Figura 2.1 (a)), é possível emular falhas externas ao alvo e avaliar sua habilidade de detectar e lidar com estas falhas. De modo similar, corrupção de dados de saída (Figura 2.1(b)) é adotada para emular componentes defeituosos e como estas falhas impactam no restante do sistema (NATELLA; COTRONEO; MADEIRA, 2016). Injeção de erros de interface pode ser alcançada através de um *driver* de teste que está ligado ao componente alvo e altera suas entradas e saídas ou através de um módulo que intercepta e corrompe as interações entre o componente alvo e outras partes do sistema (NATELLA; COTRONEO; MADEIRA, 2016).

Figura 2.1 – Injeção de erros de interface



Fonte: Adaptado de (NATELLA; COTRONEO; MADEIRA, 2016).

Como vantagens pode-se citar o fato de que não se necessita de *hardware* específico, diminuindo o custo do experimento (HSUEH; TSAI; IYER, 1997) e é um método que pode ser estendido para novas classes de falhas, não se restringindo apenas àquelas causadas por *hardware*, por exemplo códigos com funções cooperativas ou de comunicação (ZIADE; AYOUBI; VELAZCO, 2004). Porém com esta técnica não se pode alcançar trechos ou locais inacessíveis por *software* (HSUEH; TSAI; IYER, 1997) e por algumas vezes exigir modificações no código fonte, faz com que o código executado durante o experimento não seja o mesmo código utilizado na aplicação de campo (ZIADE; AYOUBI; VELAZCO, 2004).

2.5.3 Injeção de falhas baseada em simulação

Este método envolve a construção de um modelo de simulação baseado no sistema sob análise (ZIADE; AYOUBI; VELAZCO, 2004), onde os erros ou defeitos do sistema simulado ocorrem de acordo com uma distribuição pré-determinada. Tais modelos geralmente são desenvolvidos utilizado linguagens de descrição de *hardware*, como VHDL (*Very high speed integrated circuit Hardware Description Language*) (ZIADE; AYOUBI; VELAZCO, 2004). Nesta abordagem, falhas são injetadas no modelo e são ativadas através de um conjunto único de entradas.

Existem várias técnicas para se injetar falhas em simulação (ZIADE; AYOUBI; VELAZCO, 2004), que podem ser divididas em duas grandes categorias, técnicas que exigem alteração do código VHDL e técnicas que usam funções prontas do simulador. Uma primeira abordagem baseada em modificação do código, adiciona módulos específicos chamados *sabotadores* (componente inativo durante a execução normal, mas que altera diversos sinais e temporizações durante a injeção de falhas), e *mutantes* (trecho de código interno ao modelo que

permanece dormente durante a execução normal e alteram o comportamento do modelo durante a injeção de falhas).

Uma segunda abordagem (ZIADE; AYOUBI; VELAZCO, 2004), que não altera o código do modelo, utiliza ferramentas próprias do simulador VHDL que dão suporte a injeção de falhas e monitoramento dos sinais alterados. Apesar de não necessitar de alterações no código do modelo, depende muito da existência de simuladores que contam com tais recursos.

Entre as vantagens desta técnica (ZIADE; AYOUBI; VELAZCO, 2004), está o controle total sobre o modelo de falhas e dos mecanismos de injeção. Tal flexibilidade em alterar o valor de cada sinal e fácil observação dos efeitos permite que qualquer falha em potencial seja modelada com precisão.

A principal desvantagem está no fato de que a exatidão dos resultados depende do quão próximo o sistema modelado é do modelo real (ZIADE; AYOUBI; VELAZCO, 2004). Além disso, para ativar a injeção de falhas, parâmetros de entrada devem ser precisamente selecionados, que são difíceis de prever devido a mudanças na tecnologia e no projeto (HSUEH; TSAI; IYER, 1997).

2.5.4 Injeção de falhas baseada em emulação

Ao se utilizar injeção de falhas baseada em simulação, é comum um experimento levar um longo tempo para ser concluído. Para lidar com isto, foi proposta uma técnica onde o sistema modelado é implementado em um sistema baseado em *Field Programmable Gate Array* (FPGA) (ZIADE; AYOUBI; VELAZCO, 2004). Tal FPGA é conectado a um computador que define e controla os experimentos de injeção de falhas e exibe os resultados.

Esta técnica se aproveita da velocidade de um protótipo em *hardware* para rodar vários experimentos em um intervalo de tempo menor, comparado ao tempo que se levaria para rodar a mesma quantidade de experimentos utilizando simulação (ZIADE; AYOUBI; VELAZCO, 2004).

O ganho de tempo de execução que se obtém em relação à técnica de simulação permite que baterias de teste sejam executadas em um tempo menor, porém a descrição em VHDL inicial deve ser sintetizada de maneira ótima para não usar mais componentes internos do FPGA do que o necessário (ZIADE; AYOUBI; VELAZCO, 2004).

2.5.5 Injeção de falhas híbrida

Uma abordagem híbrida combina duas ou mais técnicas de injeção de falhas mencionadas anteriormente, combinando a versatilidade das técnicas de injeção baseadas em *software* com a precisão do monitoramento feito por *hardware* (ZIADE; AYOUBI; VELAZCO, 2004).

Esta técnica é mais adequada quando se deseja medir latências extremamente baixas (ZIADE; AYOUBI; VELAZCO, 2004). Porém, dado que as técnicas de injeção de falhas baseadas em simulação fornecem um ganho significativo no controle e observação dos experimentos, pode ser vantajosa uma combinação da abordagem baseada em simulação com alguma(s) da(s) outra(s) para testar plenamente o sistema sob análise (ZIADE; AYOUBI; VELAZCO, 2004).

2.5 Falhas de comunicação

Com sistemas computacionais se conectando através de redes de comunicação, sua distância física traz preocupações relacionadas à dependabilidade mais sérias do que as existentes em sistemas autocontidos. Caso a comunicação seja feita através de ondas eletromagnéticas, interferências do próprio ambiente podem levar à perda ou alterações na informação, mesmo que nenhuma falha seja apresentada no *hardware* do sistema. Para estes casos, o uso de técnicas de tolerância a falhas deixa de ser um valor simplesmente agregado ao sistema para ser algo indispensável para que o mesmo funcione de maneira aceitável (DREBES, 2005).

Para falhas de comunicação, o modelo de falhas está focado nas mensagens sendo trocadas entre dois sistemas ou dois componentes de um sistema. Mensagens podem ser perdidas, alteradas ou atrasadas (HAN; SHIN; ROSENBERG, 1995). Mensagens perdidas simplesmente não são entregues ao(s) seu(s) destinatário(s), podendo ser especificado se apenas mensagens em um único sentido (origem para destino ou destino para origem) ou se mensagens em ambos os sentidos serão descartadas. Além disso, podem ser descartadas algumas mensagens de maneira intermitente, com uma probabilidade escolhida perlo usuário, ou todas as mensagens (HAN; SHIN; ROSENBERG, 1995). Mensagens podem ser alteradas de uma forma análoga a erros de memória, ou seja, alterando o valor de um ou mais *bits.* O usuário pode especificar onde injetar o erro, seja no corpo (para alterar os dados transmitidos) ou no cabeçalho (para alterar informações referentes a roteamento) da mensagem (HAN; SHIN;

ROSENBERG, 1995). Para atraso de mensagens, deve ser especificado um método para definir por quanto tempo cada mensagem será atrasada. Tal tempo pode ser determinístico ou seguir uma distribuição probabilística (HAN; SHIN; ROSENBERG, 1995).

Para realizar injeção de falhas de comunicação pode-se utilizar qualquer uma das técnicas apresentadas anteriormente. O ponto chave é que o injetor precisa ter acesso às mensagens sendo trocadas no sistema (ou entre sistemas), ou seja, é necessário que o injetor possa manipular essas mensagens para que o modelo de falhas seja explorado com maior eficácia.

3 O PROTOCOLO ETHERCAT

Neste capítulo será apresentada uma breve descrição da norma IEC 61158, na qual o protocolo EtherCAT é baseado. Em seguida é apresentada uma visão geral do protocolo, suas principais características de funcionamento, seu modelo de falhas e os respectivos mecanismos de detecção e correção de falhas.

3.1 Norma IEC 61158

IEC 61158 é uma norma com criação nos anos de 1980 para padronizar o grande número de sistemas *Fieldbus* que estavam disponíveis no mercado à época, sendo um esforço em conjunto entre a *International Society of Automation* (ISA) (DURANTE; VALENZANO, 1999) e a *International Electrotechnical Commission* (IEC). Devido a diversos conflitos comerciais entre empresas concorrentes de sistemas *Fieldbus* e pressões políticas (FELSER; SAUTER, 2002), o padrão 61158 levou aproximadamente 15 anos para ser finalizado.

O padrão IEC 61158 prevê um número reduzido de camadas em sua pilha de protocolos, sendo apenas três: Física, Enlace e Aplicação, podendo oferecer serviços tanto na camada de Enlace quanto na camada de Aplicação. Serviços da camada de Enlace são compatíveis com suas contrapartes no modelo OSI, mesmo que sistemas *Fieldbus* não sejam naturalmente compatíveis com o modelo OSI (DURANTE; VALENZANO, 1999).

Um sistema 61158 pode possuir várias sub redes, baseadas em uma topologia de barramento compartilhado, com as várias estações destas sub redes se interconectando através de *bridges*. *Bridges* podem ser usadas tanto como simples interfaces para a rede quanto para conectar dispositivos que devem ser capazes de satisfazer certos requisitos temporais impostos pelas regras de comunicação (DURANTE; VALENZANO, 1999).

3.2 Protocolo EtherCAT – Principais características

EtherCAT é um protocolo industrial baseado em Ethernet desenvolvido para oferecer alta performance em tempo real com baixos *jitter* e tempos de ciclo, aliados a uma alta utilização da banda disponível. Desenvolvido pela empresa *Beckoff* e mantido pelo *EtherCAT Technology Group* (ETG), faz parte da norma 61158 e é largamente utilizado no controle de servo motores, coleta de dados de forma sincronizada e na área de automação em geral (ZHOU; HU, 2011).

Uma rede EtherCAT é composta de um mestre EtherCAT e até 65535 escravos que podem estar conectados sem qualquer restrição de topologia, sendo linha, árvore, anel, estrela ou qualquer combinação destas, permitindo que quaisquer 2 dispositivos estejam distantes em até 100m entre si.

Os princípios, metodologias e modelo de protocolo adotados pelo EtherCAT são baseados no modelo OSI (ETHERCAT TECHNOLOGY GROUP, 2013a), com camadas que podem ser modeladas e desenvolvidas de forma independente. A grande diferença é que algumas camadas do modelo OSI (mais precisamente, as camadas 3 a 6) são condensadas em uma mesma camada EtherCAT, como mostra a Tabela 3.1.

Tabela 3.1 – Comparação entre as camadas do Modelo OSI e Modelo EtherCAT

Camada no modelo OSI	Camada equivalente em EtherCAT
7 - Aplicação	Aplicação
6 - Apresentação*	
5 - Sessão*	
4 - Transporte*	Enlace
3 - Rede*	
2 - Enlace	
1 - Física	Física

Fonte: Próprio autor

As camadas de Apresentação e Sessão do modelo OSI, quando existentes, podem estar presentes na camada de Aplicação do protocolo EtherCAT, mas não na camada de Enlace. Já as camadas de Transporte e Rede do modelo OSI podem estar presentes tanto nas camadas de Aplicação quanto de Enlace do modelo EtherCAT (ETHERCAT TECHNOLOGY GROUP, 2013a).

A camada Física do protocolo EtherCAT trata de requisitos e configurações de rede e meio físico para garantir integridade de dados antes da checagem de erros da camada de enlace e interoperabilidade entre dispositivos do ponto de vista do meio físico (ETHERCAT TECHNOLOGY GROUP, 2013b).

A camada de Enlace fornece suporte básico à comunicação de tempo crítico entre dispositivos (ETHERCAT TECHNOLOGY GROUP, 2013c). Por “tempo crítico” entende-se aplicações que precisem de uma janela de tempo restrita, onde uma ou mais ações específicas devem ser realizadas neste tempo com algum nível de certeza. As funções da camada de Enlace

são: calcular, comparar e verificar o *frame check sequence* dos *frames* Ethernet e extrair e inserir dados nestes *frames* (ETHERCAT TECHNOLOGY GROUP, 2013c).

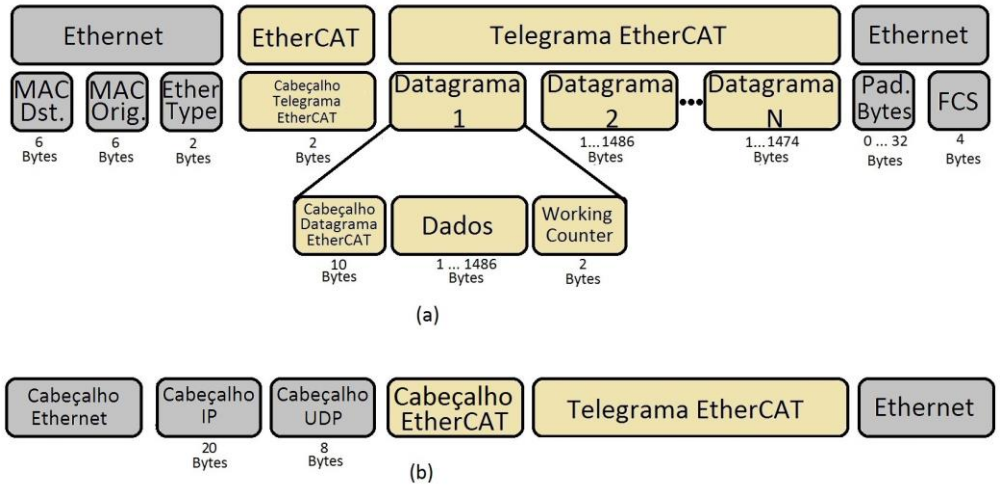
A camada de Aplicação permite que programas de usuário acessem o ambiente de comunicação *Fieldbus*. Neste contexto, a camada de Aplicação pode ser vista como "uma janela entre programas de aplicações correspondentes" (ETHERCAT TECHNOLOGY GROUP, 2013d).

3.3 Princípio de funcionamento

Por utilizar *frames* Ethernet do padrão IEEE802.3 para troca de dados, um controlador de rede comum pode ser utilizado sem a necessidade de *hardware* especial, ao contrário de outros protocolos de tempo real baseados em Ethernet (KANG *et al.*, 2011). Por conta disto, tanto o mestre quanto o escravo EtherCAT podem, em princípio, ser implementados em qualquer PC com interface de rede que implemente as camadas física e de enlace do modelo OSI. Porém na prática, os escravos EtherCAT são implementados com *hardware* especial, seja em FPGAs ou A*pplication Specific Integrated Circuits* (ASICs) para diminuir os atrasos ao se enviar pacotes. O mestre EtherCAT por outro lado pode ser implementado normalmente utilizando apenas componentes padrão (PRYTZ, 2008).

Dados EtherCAT são encapsulados dentro de um *frame* Ethernet padrão (Figura 3.1 (a)), também podendo se adotar os protocolos IP e UDP (Figura 3.1 (b)), sendo chamado de *telegrama* EtherCAT. Telegramas EtherCAT são diferenciados de outros *frames* Ethernet através de seu *EtherType* (0x88A4), fazendo com que dados EtherCAT possam ser transmitidos em paralelo com outros protocolos (KANG *et al.*, 2011). Um telegrama EtherCAT contém um ou mais *datagramas* EtherCAT, onde cada datagrama é atribuído a um ou mais escravos. Com isto, pode-se acessar cada escravo separadamente através de um datagrama específico ou múltiplos escravos com apenas um datagrama (ETHERCAT TECHNOLOGY GROUP, 2015).

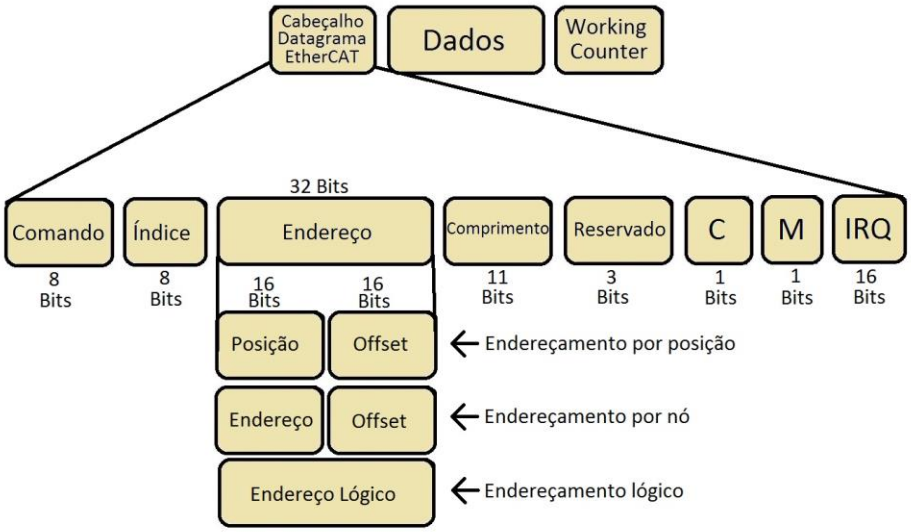
Figura 3.1 – Estrutura do *frame* EtherCAT



Fonte: Próprio autor

Cada datagrama EtherCAT possui um cabeçalho independente, que contém as seguintes informações: tipo de comando a ser executado; um índice que serve de identificador numérico utilizado pelo mestre para identificar datagramas duplicados ou perdidos e que não é modificado pelos escravos; o endereço do escravo, podendo ser um endereço direto ou implícito; um identificador numérico que informa o comprimento (em *bytes*) de dados que estão sendo transmitidos; o *Circulating Bit*; o indicador se há ou não mais datagramas dentro do telegrama e um registrador de eventos dos escravos (IRQ) (Figura 3.2).

Figura 3.2 – Estrutura do cabeçalho de um datagrama EtherCAT



Fonte: Próprio autor

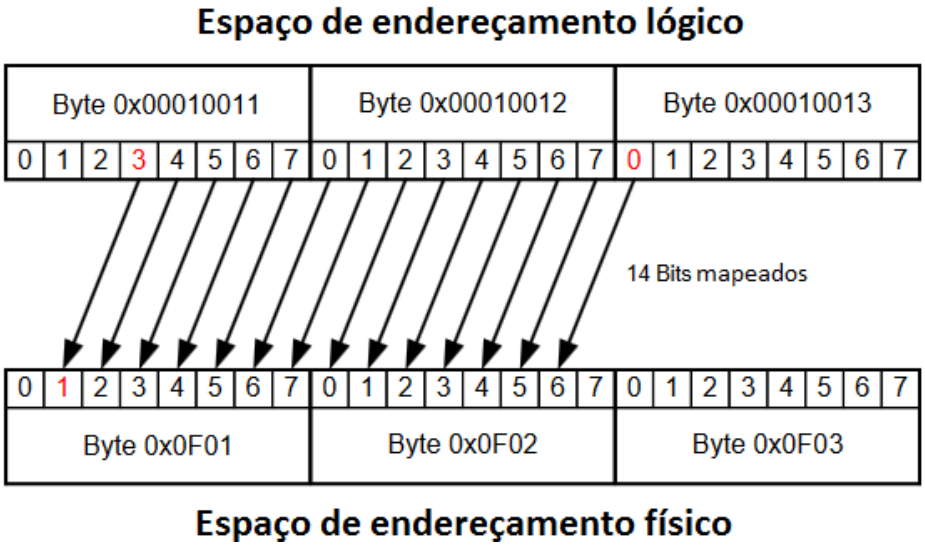
O início do ciclo de comunicação EtherCAT se dá com o mestre enviando um telegrama que passa por todos os nós escravos da rede. Cada escravo lê e insere seus respectivos dados (identificados pelo endereçamento) no seu datagrama designado dentro do *frame*, enquanto este está em trânsito. Como cada *frame* não é colocado em um buffer para leitura ou escrita de dados, este é atrasado apenas pelos tempos de propagação do *hardware*. O último escravo a receber o *frame* detecta que não possui escravos subsequentes a quem possa repassar o *frame* e envia o mesmo *frame* no sentido contrário, em direção ao mestre, utilizando a característica de *full duplex* da tecnologia Ethernet (ETHERCAT TECHNOLOGY GROUP, 2015).

Do ponto de vista da aplicação (ORFANUS *et al.*, 2013), o mestre EtherCAT é o único nó de um segmento permitido a enviar *frames* ativamente; todos os demais nós apenas repassam os *frames* adiante. Este conceito previne atrasos imprevistos e garante a capacidade de comunicação em tempo real (ETHERCAT TECHNOLOGY GROUP, 2015).

No modo de endereçamento direto (*Device Addressing*) (ETHERCAT TECHNOLOGY GROUP, 2014), dispositivos podem ser endereçados através de sua posição, pelo endereço do nó ou por um *broadcast*. Utilizando endereçamento por posição, ou *Auto Increment Address*, um mesmo datagrama é enviado a todos os escravos no início da operação do sistema, com um valor negativo como endereço. Cada escravo ao ler este valor o incrementa. O escravo que ler um valor igual a zero é endereçado e irá executar o comando apropriado. Normalmente é utilizado para varrer a rede no início da operação e ocasionalmente detectar novos escravos adicionados (ETHERCAT TECHNOLOGY GROUP, 2014). No endereçamento por nó cada escravo tem seu endereço definido pelo mestre individualmente durante o início da operação. Endereçamento por *broadcast* é utilizado para a inicialização e checagem do estado de todos os escravos (ETHERCAT TECHNOLOGY GROUP, 2014).

No endereçamento lógico (*Logical Addressing*) (ETHERCAT TECHNOLOGY GROUP, 2014), todos os dispositivos realizam suas operações de leitura e escrita no mesmo espaço de endereçamento de 4 GB (campo de endereço de 32 *bits* dentro do datagrama EtherCAT). Escravos utilizam uma unidade de mapeamento chamada *Fieldbus Memory Management Unit* (FMMU) para converter dados do espaço lógico para seu espaço físico através de um mapeamento. Durante a inicialização do sistema, o mestre configura as FMMUs dos escravos, para que estes saibam de antemão quais partes do espaço de endereçamento lógico serão mapeados em seus espaços de endereçamento físico (ETHERCAT TECHNOLOGY GROUP, 2014). A lógica de funcionamento está ilustrada na Figura 3.3.

Figura 3.3 – Lógica de funcionamento da FMMU



Fonte: Adaptado de (ETHERCAT TECHNOLOGY GROUP, 2014).

3.4 Ordem de processamento dos *frames*

Cada escravo EtherCAT possui no mínimo duas e no máximo quatro portas (interfaces de rede), sendo a porta 0 definida como a porta de entrada. Cada porta possui dois estados: *aberto* e *fechado* (ETHERCAT TECHNOLOGY GROUP, 2014). Se uma porta está aberta, ela está apta a receber e transmitir *frames*. Uma porta fechada não irá trocar *frames* com outros dispositivos, ao invés disso, os *frames* serão encaminhados internamente para a próxima porta, até uma porta aberta ser encontrada. Para controlar o estado de cada porta, escravos suportam quatro modos de configuração, sendo dois automáticos e dois manuais, definidos pelo mestre: *Manual Open*, *Manual Closed*, *Auto* e *Auto Close* (ETHERCAT TECHNOLOGY GROUP, 2014).

No modo *Manual Open* a porta está sempre aberta independente do estado do enlace. Em caso de enlace defeituoso ou inexistente, *frames* serão perdidos.

Já no modo *Manual Close*, a porta está fechada independente do estado do enlace, ou seja, mesmo com um enlace funcional presente, *frames* não serão transmitidos.

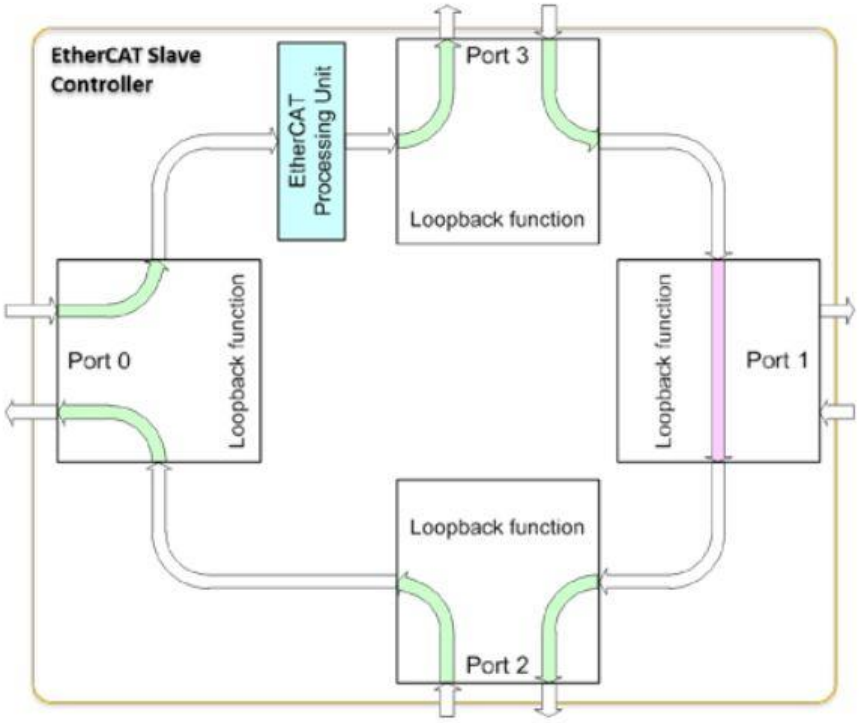
Com o modo *Auto* o estado do enlace define o estado da porta. Assim a porta está aberta se um enlace está presente e estará fechada na ausência de um enlace.

No modo *Auto Close,* o estado da porta também é definido pelo estado do enlace, mas de maneira diferente da adotada no modo *Auto*. Aqui, em caso de perda de enlace, a porta é fechada. Se o enlace é reestabelecido, a porta não é aberta de imediato, permanecendo fechada. Tipicamente a porta deverá ser aberta pelo mestre através de uma instrução explícita. Outro

método de abertura da porta é através do recebimento de um *frame* Ethernet válido nesta porta, que será aberta após a validação do CRC deste *frame*.

Do ponto de vista físico, qualquer topologia EtherCAT sempre forma um anel lógico (ETHERCAT TECHNOLOGY GROUP, 2012), já que o processamento de um *frame* em um escravo funciona como uma rótula (Figura 3.4). Escravos estão conectados ao nível superior (mestre) sempre pela porta 0 e aos níveis inferiores pelas portas 1 a 3. Cada *frame* é processado apenas uma vez em cada escravo através da *EtherCAT Processing Unit*, que está localizada logo após a porta 0. Com isso, *frames* que retornarem não serão processados novamente, mas apenas transmitidos para a próxima porta ou devolvidas à porta 0.

Figura 3.4 – Escravo com 4 portas e sua ordem de processamento de *frames*



Fonte: (ETHERCAT TECHNOLOGY GROUP, 2012).

3.5 SyncManager

A memória de um escravo EtherCAT pode ser usada para troca de dados entre um mestre EtherCAT e uma aplicação local sem qualquer restrição. No entanto, usar a memória para este tipo de comunicação possui alguns riscos: a consistência de dados não é garantida, dependendo de semáforos implementados em *software* para coordenar a troca de dados; não há garantias para assegurar a segurança dos dados, para isso mecanismos de segurança também

devem ser implementados em *software*; tanto o mestre quanto o escravo devem realizar *polling* (verificar o estado de forma ativa) na memória para se certificar que o acesso do outro lado acabou.

Para lidar com estes problemas existe o *SyncManager* (ETHERCAT TECHNOLOGY GROUP, 2014), que são dispositivos internos aos escravos e configurados pelo mestre que garantem consistência e troca segura de dados entre o mestre e a aplicação no escravo, gerando interrupções para informar ambos os lados.

A direção de comunicação é configurável, bem como o modo que a comunicação ocorre (*buffered* ou *mailbox*). *SyncManagers* utilizam um *buffer* localizado em memória para a troca de dados, com acesso sendo controlado via *hardware*.

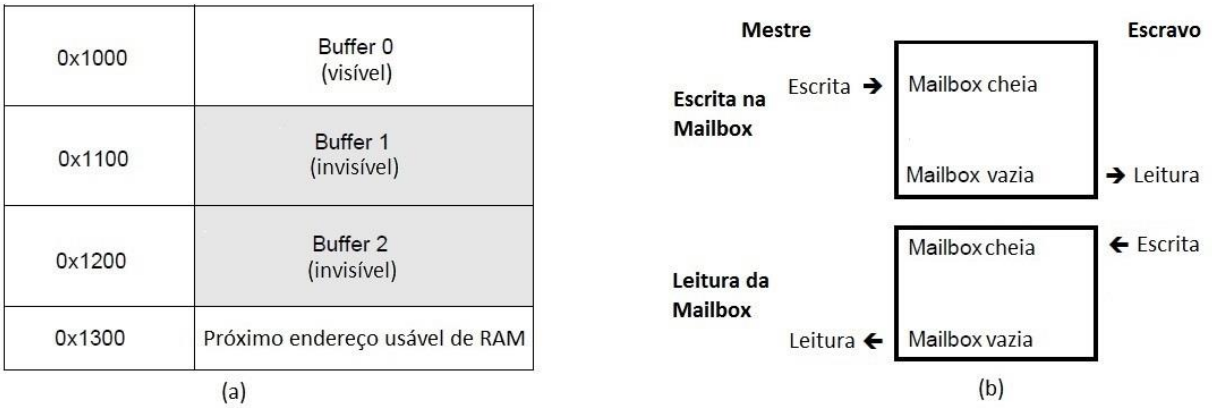
Para acessar o *buffer* é necessário se iniciar pelo primeiro endereço do mesmo, caso contrário o acesso é negado. Após o endereço inicial ser acessado, todo o *buffer* é liberado para acesso, seja em uma ou várias vezes. O acesso ao *buffer* é encerrado após seu último endereço ser acessado, o estado do *buffer* é alterado após isto e uma interrupção é gerada para avisar ambos os lados (ETHERCAT TECHNOLOGY GROUP, 2014).

SyncMangers suportam dois modos de comunicação entre o mestre e um escravo: modo *buffered*, controlado via *hardware* e modo *mailbox*, gerenciado através de semáforos (ETHERCAT TECHNOLOGY GROUP, 2014).

O modo *buffered* (ETHERCAT TECHNOLOGY GROUP, 2014) (também chamado *3-buffer-mode*), permite escrita e leitura de dados de forma simultânea sem interferência. Se uma escrita é feita no *buffer* de forma mais rápida que a leitura, os dados mais antigos serão descartados. Fisicamente, três *buffers* de tamanho idêntico são utilizados, (Figura 3.5 (a)), sendo um alocado para o produtor (escrita), outro para o consumidor (leitura) e o terceiro que mantém o último dado consistente escrito pelo produtor. Neste modo mestre e escravo podem acessar o *buffer* de comunicação do escravo a qualquer momento, com o consumidor dos dados sempre tendo acesso ao conteúdo mais recente vindo do produtor, enquanto o produtor sempre pode atualizar seus dados livremente sem se preocupar com consistência dos dados já postos no *buffer*.

Já o modo *mailbox* implementa um mecanismo de *handshake* para lidar com troca de dados através de um único *buffer*. Cada lado, mestre ou escravo, apenas terá acesso ao *buffer* após o lado oposto ter encerrado seu acesso (Figura 3.5 (b)). Assim, após uma escrita no *buffer* ser finalizada, o acesso à escrita é bloqueado e o *buffer* é liberado para leitura pelo outro lado. Somente após o último *byte* ser lido o *buffer* é liberado para escrita novamente (ETHERCAT TECHNOLOGY GROUP, 2014).

Figura 3.5 – Modos *mailbox* e *buffered*



Fonte: Adaptado de (ETHERCAT TECHNOLOGY GROUP, 2014).

3.6 Clock distribuído

Em aplicações espacialmente distribuídas, com processos tendo que realizar ações simultâneas, sincronização é um aspecto importante. Devido a isso, um baixo *jitter* é necessário para que cada escravo possa realizar ações locais ou realizar alguma medição baseado em sua cópia local do tempo do sistema.

Para atender estes requisitos, EtherCAT conta com o *clock distribuído*, onde é possível alcançar um *jitter* máximo de 11ns (ORFANUS *et al.*, 2013). Para que todos os dispositivos estejam sincronizados no mesmo *clock*, a calibração dos *clocks* dos nós da rede é completamente feita via *hardware* (ETHERCAT TECHNOLOGY GROUP, 2014). Um tempo de referência é ciclicamente distribuído a todos os dispositivos no sistema, assim, os *clocks* locais podem ser ajustados com base neste *clock* de referência.

O processo de sincronização do *clock* distribuído possui duas fases (ORFANUS *et al.*, 2013). Na primeira fase, seja no início da operação do sistema ou continuamente durante o funcionamento (ETHERCAT TECHNOLOGY GROUP, 2014), os atrasos entre todos os escravos são medidos. Com base nestas medidas, o mestre calcula tempos de compensação para cada escravo e escreve esta informação em registradores específicos de cada escravo, através de telegramas. Na segunda fase, chamada *Time Control Loop,* o mestre envia periodicamente telegramas de compensação de *drift* (*Drift Compensation Telegram*) que auxilia os escravos a compensarem os *drifts* em suas cópias locais do tempo do sistema (ORFANUS *et al.*, 2013).

Tipicamente a fonte do tempo de referência é o primeiro escravo, mas na teoria qualquer escravo ou o mestre podem ser esta fonte. O motivo de se escolher escravos para tomar o tempo

de referência se deve ao fato de o *jitter* causado pelo processamento do mestre ser muito maior que o de qualquer escravo (ORFANUS *et al.*, 2013).

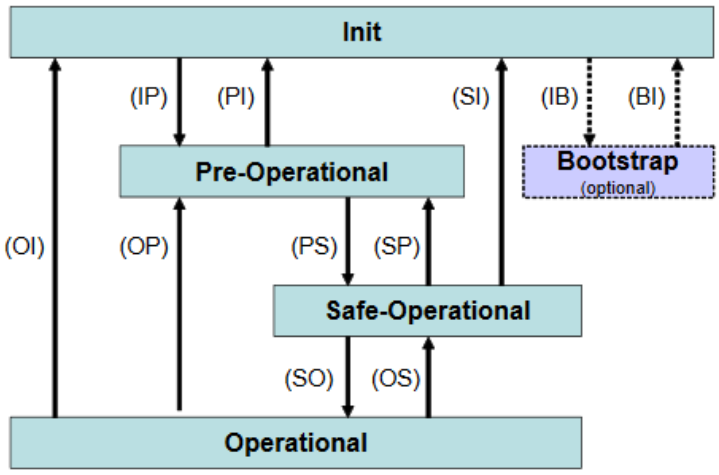
3.7 Máquina de estados

Cada escravo EtherCAT possui uma máquina de estados que é responsável por coordenar aplicações do mestre e escravos no início ou durante a operação e regular as funções do escravo a cada momento, já que o atual estado define quais funções estão disponíveis.

Tipicamente, mudanças de estados são iniciadas por requisições do mestre e são respondidas pelo escravo após a inicialização bem-sucedida de um conjunto específico de registradores. Quando a inicialização de um ou mais parâmetros falha, o escravo não responde ao mestre e a transição de estado não acontece. No caso de uma falha interna, o escravo retorna ao estado anterior. Em ambos os casos, o escravo informa o mestre sobre o comportamento da máquina de estados através de um *bit* de indicação e escrevendo um código de erro correspondente no registrador *Application Layer Status Code* (ETHERCAT TECHNOLOGY GROUP, 2014). Erros típicos que afetam a operação da máquina de estados incluem configuração incorreta de áreas de memória usadas para troca de dados, perda de sincronismo entre mestre e escravo ou falhas internas de aplicações específicas do escravo.

Para esta máquina de estados, existem quatro estados obrigatórios e um opcional (Figura 3.6): *Init*, *Pre-Operational*, *Safe-Operational*, *Operational* e *Bootstrap* (opcional) (ETHERCAT TECHNOLOGY GROUP, 2014). Uma breve descrição de cada estado é apresentada a seguir.

Figura 3.6 – Máquina de estados de um escravo EtherCAT



Fonte: (ETHERCAT TECHNOLOGY GROUP, 2014).

O estado *Init* define as relações de comunicação entre mestre e escravo na camada de Aplicação, porém, nenhuma comunicação direta no nível de Aplicação entre mestre e escravo está disponível (ETHERCAT TECHNOLOGY GROUP, 2013e). Neste estado o mestre inicializa um conjunto de registradores do escravo. Caso o escravo suporte o modo *mailbox*, as configuraçẽos de seu *SyncManager* são feitas neste estado. Neste estado também são designados os endereços de cada escravo e inicialização dos *clocks* distribuídos.

No estado *Pre-Operational*, a *mailbox* é ativada, caso o escravo suporte este modo, para que sejam usados os protocolos apropriados para trocar parâmetros específicos de inicialização da aplicação (ETHERCAT TECHNOLOGY GROUP, 2013e). Neste estado não está disponível nenhuma comunicação relativa à dados de processamento (ETHERCAT TECHNOLOGY GROUP, 2012).

No estado *Safe-Operational* a aplicação do escravo deve fornecer dados de entrada reais sem manipular os dados de saída (ETHERCAT TECHNOLOGY GROUP, 2013e), ou seja, apenas dados de entrada são avaliados, com os dados de saída sendo mantidos em "modo seguro" (ETHERCAT TECHNOLOGY GROUP, 2012).

No estado *Operational* a aplicação do escravo deve entregar dados reais de entrada e a aplicação do mestre deve entregar dados reais de saída (ETHERCAT TECHNOLOGY GROUP, 2013e). Em outras palavras, tanto dados de entrada quanto dados de saída são considerados válidos (ETHERCAT TECHNOLOGY GROUP, 2012).

Já o estado *Bootstrap* é opcional, porém recomendado caso sejam necessárias atualizações de *firmware* no escravo. Comunicação é disponível apenas via *mailbox*, não sendo permitidos dados de processamento (ETHERCAT TECHNOLOGY GROUP, 2012).

3.8 Modelo de falhas de comunicação

Erros de comunicação podem ser causados pelos mais variados motivos, desde falhas de *hardware* até interferências do ambiente externo. As falhas consideradas na especificação do EtherCAT serão apresentadas a seguir.

Falhas de perda de pacotes ocorrem quando um ou mais pacotes não conseguem alcançar seu destino. Podem ser causadas, por exemplo, por congestionamento do enlace, fazendo com que o protocolo descarte pacotes. Entre outras causas, pode-se citar mal desempenho de equipamentos físicos (roteadores, switches, etc.), mau funcionamento do software utilizado, rompimento do meio físico de transmissão, entre outros.

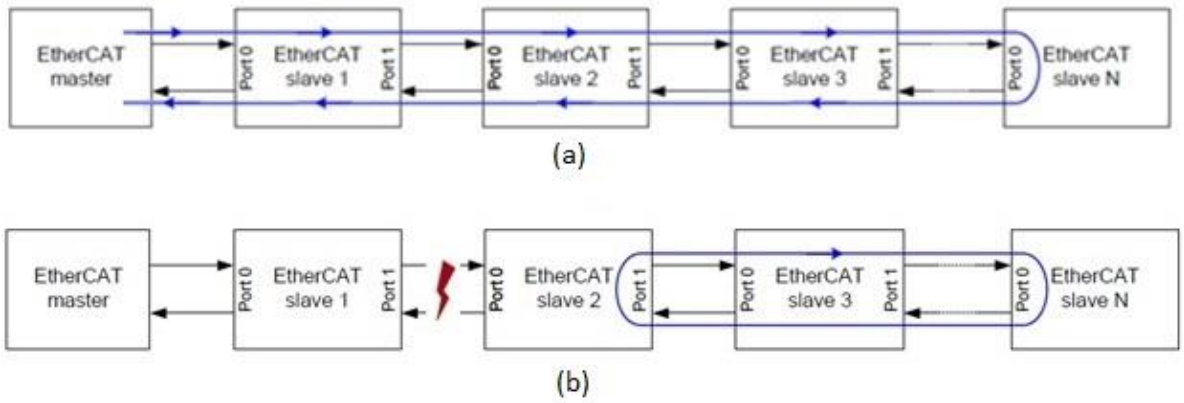
Falhas por corrupção de dados são ocasionadas quando a mensagem que sai do dispositivo de origem é diferente da mensagem que chega ao dispositivo de destino. Pode ser causada por falhas em elementos físicos do sistema ou por erros de software, como operações malsucedidas de leitura ou escrita.

Atraso de mensagens é observado quando restrições temporais não são respeitadas, isto é, o tempo esperado para um pacote ir de sua origem até seu destino é maior do que o esperado.

Repetição ou duplicação de pacotes acontece quando mensagens já transmitidas são reenviadas. Um dispositivo defeituoso pode achar que sua mensagem não foi recebida e reenviar.

Frame(s) que circula(m) pelo sistema indefinidamente pode acontecer no caso de ruptura do enlace ou outra falha que faça com que uma topologia em linha seja dividida em duas, um *frame* viajando através dos dispositivos pode começar a circular, ou seja, ser transportado apenas entre os nós de um lado da ruptura sem nunca alcançar o último escravo ou o mestre. Tal situação pode ser observada a seguir. Na Figura 3.7 (a) é exibido o caminho normal que um pacote percorre. Ao ocorrer uma falha de enlace entre os escravos 1 e 2 (Figura 3.7 (b)), estes fecham suas portas 1 e 0 respectivamente e frames à direita do escravo 2 seguirão circulando indefinidamente, pois estão impossibilitados de alcançar o nó mestre.

Figura 3.7 – Caminho normal percorrido por um *frame* e caminho percorrido após uma falha de enlace



Fonte: Adaptado de (ETHERCAT TECHNOLOGY GROUP, 2014).

3.9 Mecanismos de detecção e correção de falhas

Por ser um protocolo focado em velocidade de comunicação, EtherCAT não possui muitos mecanismos de detecção e correção de erros. Porém é capaz de lidar com alguns dos tipos mais comuns de falhas existentes.

A primeira medida é o *Frame Check Sequence* (FCS) presente no *frame* Ethernet. Este é checado tanto pelo mestre quanto pelos escravos para determinar se um *frame* foi recebido corretamente. Como vários escravos podem alterar o frame durante seu percurso pela topologia, o FCS é calculado por cada nó escravo na recepção (vindo do mestre) do *frame* e recalculado na retransmissão (indo para o mestre). Se um erro de *checksum* é encontrado, o escravo não repara o FCS, mas sinaliza o mestre incrementando um contador de erros interno, para que o ponto de falha seja precisamente localizado na topologia. É capaz de detectar falhas de perda e repetição de pacotes (ETHERCAT TECHNOLOGY GROUP, 2013f).

O *Working Counter* é o último campo do datagrama EtherCAT. Possui um valor esperado calculado pelo mestre e que é incrementado pelos escravos a cada leitura ou escrita bem-sucedida realizada no datagrama, ou seja, se ao menos um *byte* ou um *bit* em todo o datagrama foi lido ou escrito com sucesso. Escravos que apenas estão encaminhando o datagrama, não alteram o *Working counter*. Ao comparar o *Working counter* com o número esperado de escravos que deveriam acessar dados, o mestre pode saber quantos escravos processaram seus dados correspondentes. Se o *Working Counter* presente no datagrama vindo dos escravos possui um valor diferente do esperado, o mestre não envia os dados deste datagrama para a aplicação de controle (ETHERCAT TECHNOLOGY GROUP, 2015).Detecta falhas de perda e repetição de pacotes (ETHERCAT TECHNOLOGY GROUP, 2014).

Escravos EtherCAT contam com dois *watchdogs* internos para monitorar comunicações malsucedidas: um *watchdog* monitora acessos de escrita feitos no escravo e outro *watchdog* monitora leituras e escritas bem-sucedidas feitas pelo escravo. Caso não ocorra nenhuma comunicação dentro do tempo previsto, o respectivo *watchdog* tem seu contador incrementado e seus sinais de saída não são entregues (ETHERCAT TECHNOLOGY GROUP, 2014).

Em uma comunicação utilizando o modo *mailbox*, existe um contador (*counter*), que é incrementado pelos escravos a cada novo serviço de *mailbox*. Tanto o mestre quando os escravos possuem contadores independentes, sendo que os escravos não verificam se a sequência do contador está correta. É checado pelo mestre para detecção de perda de serviços de *mailbox* e pelos escravos para detecção de repetição de um serviço de escrita. Também detecta falhas de perda e repetição de pacotes (ETHERCAT TECHNOLOGY GROUP, 2013c)

Ao se utilizar o modo *Ethernet over EtherCAT* (EoE), onde *frames* Ethernet são transmitidos através do protocolo EtherCAT (ETHERCAT TECHNOLOGY GROUP, 2013d), são armazenados os tempos de envio e recebimento de um dado *frame* (ETHERCAT TECHNOLOGY GROUP, 2013e). É utilizado para detectar falhas relacionadas a atraso de mensagens.

Para lidar com frames que podem circular indefinidamente pelo sistema, existe o *Circulating Bit*, que é um *bit* que informa aos dispositivos da topologia quando um *frame* está circulando pela topologia na ocorrência de uma falha de enlace que cause uma situação semelhante à da Figura 3.7 (b). Neste exemplo, os escravos 1 e 2 detectam a falha no enlace e fecham suas portas (porta 1 no escravo 1 e porta 0 no escravo 2). Com isso o *frame* irá circular indefinidamente pelo anel formado pelos escravos 2 a N. Para prevenir isto, um escravo sem um enlace em sua porta 0 fará o seguinte: se o *Circulating Bit* de um datagrama EtherCAT é 0, muda seu valor para 1 e o envia adiante. Se o *Circulating Bit* é 1, o *frame* não é processado e sim descartado. No caso exemplo, um frame circulante dará no máximo uma volta antes de ser detectado e destruído (ETHERCAT TECHNOLOGY GROUP, 2014).

A Tabela 3.2 sumariza os tipos de falhas suportados pelo protocolo e suas respectivas medidas para detecção e correção que foram apresentados anteriormente.

Tabela 3.2 – Falhas previstas e seus mecanismos de detecção e correção

Tipos de falhas	**Mecanismos de detecção ecorreção**					
	FCS	***Working Counter***	***Counter* no modo *mailbox***	***Watchdogs***	***Timestamps* no modo EoE**	***Circulating Bit***
Perda		X	X			
Corrupção	X					
Atraso				X	X	
Repetição		X	X			
Frames circulando indefinidamente						X

Fonte: Próprio autor

Para testar uma implementação do protocolo EtherCAT, uma ferramenta do tipo injetor de falhas que exercitasse os mecanismos de detecção e correção de erros desta implementação de acordo com as falhas previstas seria de grande auxílio.

4 TRABALHOS RELACIONADOS

Este capítulo apresenta um conjunto não exaustivo, mas significante, de ferramentas voltadas à verificação e teste em geral de implementações do protocolo EtherCAT. Por ser um protocolo aberto e sem uma implementação padrão para todos os dispositivos, ferramentas deste tipo são desejáveis para se avaliar a conformidade de uma implementação frente a especificação. As ferramentas descritas a seguir foram escolhidas por servirem, mesmo que parcialmente, a este propósito.

4.1 beSTORM

beStorm (“beSTORM® Software Security Testing Tool”, 2016) é uma ferramenta voltada para avaliação de segurança que realiza uma análise exaustiva para descobrir novas vulnerabilidades em aplicações que utilizam algum protocolo de rede durante sua fase de desenvolvimento. Ao testar automaticamente combinações de ataque, beSTORM busca a segurança de produtos antes que estes atinjam o mercado.

Como cobrir todas as combinações de entradas teóricas não é uma tarefa trivial, *beSTORM* conta com uma técnica é chamada de *Smart Fuzzing,* onde algoritmos de priorização são utilizados para habilitar a cobertura de todas as entradas que possuem maior chance de ativar uma falha de segurança. Para fazer isto, um conjunto de testes é gerado com base na especificação do protocolo de rede escolhido e exercita este protocolo com ênfase em casos de testes errôneos, porém tecnicamente legais, como por exemplo, teste de entradas até que um estouro de *buffer* ocorra.

Todos os testes são executados sobre a aplicação pronta, ou seja, sobre seu binário já compilado, permitindo uma abordagem independente da linguagem de programação utilizada no desenvolvimento da aplicação.

4.2 EC-Lyser

Para monitoramento e análise de sistemas de barramento EtherCAT, existe a ferramenta EC-Lyser (“EC-Lyser - EtherCAT Diagnosis Tool”, [s.d.]). Com essa ferramenta, é possível analisar a “saúde” do sistema a partir de detecção de erros.

Também é possível visualizar ou alterar o estado de um determinado escravo modificando valores internos de seus registradores ou de sua EEPROM, além de fornecer uma

visão geral da topologia da rede, através de uma interface gráfica que mostra, entre outras coisas, a qualidade da rede em determinado ponto.

4.3 EC-STA

É um *framework* (“EC-STA [EtherCAT Slave Test Application]”, [s.d.]) desenvolvido para realização de testes durante e após o desenvolvimento de escravos EtherCAT. Possui como componente principal uma aplicação baseada em .NET e escrita em linguagem C#. Também é fornecido o código fonte completo, permitindo ao usuário adicionar suas próprias funções. Seu funcionamento se dá configurando um computador com um mestre EtherCAT (fornecido separadamente) e conectando este ao escravo que será avaliado.

Com esta ferramenta é possível alterar o atual estado tanto do mestre quanto do escravo, exibição dos dados sendo transmitidos entre o mestre e o escravo, testar o modo de comunicação *mailbox,* entre outros testes.

4.4 Elmo Motion Control EtherCAT Network Diagnostics

É uma ferramenta voltada para visualização de diagnósticos de redes EtherCAT (“Elmo Motion Control EtherCAT Network Diagnostics”, 2016). Funções de diagnóstico também contam com modos para ler e escrever no dicionário de objetos do escravo. Este dicionário é uam estrutura de dados que contém parâmetros, dados de aplicação e informações de mapeamento da interface entre dados do processo e dados da aplicação do escravo (ETHERCAT TECHNOLOGY GROUP, 2013d). É usado principalmente no modo *mailbox*.

Outros diagnósticos presentes incluem erros de configuração e inicialização, perda de enlace, erros relativos a envio e recebimento de mensagens, *frames* inválidos e erros relativos aos *working counters*.

4.5 EtherCAT Conformance Test Tool

É uma ferramenta desenvolvida pelo próprio ETG (“EtherCAT Technology Group | EtherCAT Conformance Test Tool (CTT)”, 2016) para realizar testes em dispositivos escravos EtherCAT a fim de verificar sua conformidade.

Nesta ferramenta, *frames* EtheCAT são enviados ao dispositivo sob teste através de uma porta Ethernet padrão de modo a estimulá-lo. Casos de teste são lidos a partir de um arquivo

XML, de modo que alterações possam ser feitas sem que se necessite alterar a ferramenta em si. Resultados dos testes e possíveis observações são exportados posteriormente em um formato reconhecível por editores de planilhas eletrônicas (CSV).

Dentre os testes realizados, estão o teste da máquina de estados do escravo, comunicação pelo modo *mailbox*, consistência dos dados do dicionário de objetos e o funcionamento da memória EEPROM.

Esta ferramenta por ter suporte e ser licenciada pelo grupo responsável pelo protocolo EtherCAT, é utilizada pelo órgão verificador como última etapa dos processos de validação e certificação e é tida como mandatória para que produtos que implementam EtherCAT possam ser certificados e assim liberados para atingir o mercado.

5 O INJETOR DE FALHAS FITT

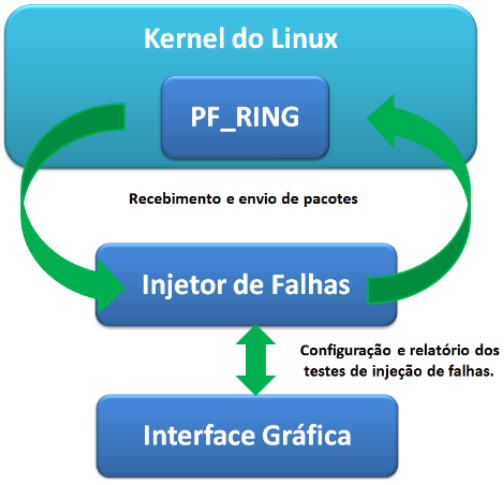
O injetor de falhas FITT (*Fault Injection Test Tool*) (DOBLER *et al.*, 2016) é um injetor desenvolvido para protocolos seguros baseados na norma IEC 61508, em especial para o protocolo seguro PROFIsafe (“PROFIsafe”, 2016). Esta ferramenta endereça a maioria dos tipos de falhas previstas na norma: endereçamento, corrupção, atraso, sequência incorreta, inserção, perda e repetição de pacotes.

Mesmo que tenha sido desenvolvida para um protocolo diferente de EtherCAT, o modo com que a ferramenta intercepta os pacotes para injeção das falhas e por ser de funcionamento simples, a tornaram atrativa a ponto de servir de base para estre trabalho, por isto uma descrição mais detalhada será apresentada a seguir.

5.1 Características

FITT é um injetor de falhas de comunicação para protocolos seguros feito para ser utilizado em sistemas *Linux* com *kernel* a partir da versão 2.6.32 e *hardware* genérico. A escolha do sistema *Linux* para desenvolvimento se deu porque o injetor utiliza uma biblioteca extra para captura de pacotes em alta velocidade, chamada PF_RING (“PF_RING”, 2011), que demanda um sistema *Linux* para seu correto funcionamento. O injetor conta ainda com mais duas partes: a aplicação que implementa as funções de injeção de falhas e uma interface gráfica que envia parâmetros para esta aplicação, com base no tipo de falha a ser injetada, como ilustrado pela Figura 5.1.

Figura 5.1 – Arquitetura do injetor FITT



Fonte: (DOBLER, 2016).

FITT é executada em um computador próprio, de modo a ser uma entidade separada do resto do sistema, sendo um intermediário entre dois dispositivos que troquem mensagens entre si, chamados de *F-Host* e *F-Device* (Figura 5.2). Deste modo as falhas são injetadas em ambos os sentidos de comunicação sem que os dispositivos saibam da existência do injetor. Por conta disto, o computador que irá rodar FITT deve contar com ao menos duas interfaces de rede para que os pacotes possam ser interceptados e enviados a seus destinos.

Figura 5.2 - Ligação entre o *F-Host*, Injetor de Falhas e *F-Device*.

Fonte: (DOBLER, 2016).

5.2 Funcionamento

FITT trabalha sobre pacotes Ethernet IP / UDP, tipo 0x8000, para pacotes do protocolo PROFIsafe utilizados em sua implementação. Somando estes dados juntamente com o número de porta 5100, um filtro de pacotes foi criado, de modo que apenas pacotes pertinentes ao protocolo PROFIsafe fossem enviados ao injetor. Os demais pacotes são enviados automaticamente para seu destino.

Falhas são injetadas com base em tempos informados pelo usuário. Para falhas de endereçamento, corrupção, sequência incorreta e perda de pacotes, um único tempo é informado, o intervalo de tempo em que a falha deve ocorrer. Por exemplo, informar um tempo de 2 segundos para a injeção de corrupção de dados significa que a cada 2 segundos um pacote terá seus dados corrompidos. Já falhas de atraso, inserção e repetição de pacotes necessitam que dois parâmetros de tempo sejam informados: um tempo para ativar a falha e um tempo de duração da falha. Por exemplo, para falhas de atraso de pacotes, informando um primeiro tempo de 3 segundos e um segundo tempo de 4 segundos significa que a cada 3 segundos um pacote será atrasado por 4 segundos.

Em sua estrutura principal, são utilizadas *threads* para controlar os vários mecanismos necessários para seu funcionamento, como direção de injeção de falhas e tempo para injetar a falha, dependendo do tipo de falha a ser injetada.

A primeira e segunda *threads* controlam o envio e recebimento de pacotes, de forma independente, nas interfaces de rede. A injeção de falhas pode ser aplicada em ambos os sentidos de comunicação entre o *F-Host* e *F-Device*, porém em apenas um sentido por vez.

A terceira *thread* é usada para contar o tempo de injeção de falhas informado pelo usuário. Após decorrer este tempo, esta *thread* sinaliza o injetor que o próximo pacote a chegar deve ser o alvo da injeção da falha. Após a falha ser injetada, o contador é reiniciado e o ciclo se repete até o programa ser encerrado.

A quarta *thread* é utilizada apenas nas funções de atraso, inserção e repetição de pacotes e também serve de contador de tempo, como a terceira *thread*. Neste cenário, esta quarta *thread* controla o tempo que deve decorrer antes que o pacote resultante do processo de injeção de falhas seja enviado a seu destino.

Após decorrida o processo de injeção de falhas e o programa encerrado, informações relativas ao processo são salvas em um arquivo texto para análise posterior. Dentre as informações salvas estão o número total de pacotes que trafegaram pelas interfaces de rede e o total de pacotes que foram afetados pela injeção de falhas.

O processo de injeção de falhas de fato se inicia com a biblioteca PF_RING enviando os pacotes que chegam por uma das interfaces de rede para a aplicação responsável. Após as falhas serem injetadas nos pacotes, estes são enviados de volta à PF_RING para que possam ser encaminhados para a outra interface de rede em direção a seu destino original.

5.3 A biblioteca PF_RING

PF_RING é uma biblioteca desenvolvida para captura e envio de pacotes em alta velocidade, sendo apta a ser usada em análise e manipulação de pacotes e tráfego ativo na rede. Esta biblioteca possui um módulo (também chamado PF_RING) que deve ser carregado no *kernel* do *Linux*, permitindo que aplicações do espaço do usuário se comuniquem diretamente com as interfaces de rede sem o intermédio do *kernel*. Este módulo possui dois modos de operação: *Vanilla* e *DNA* (*Direct NIC Access*).

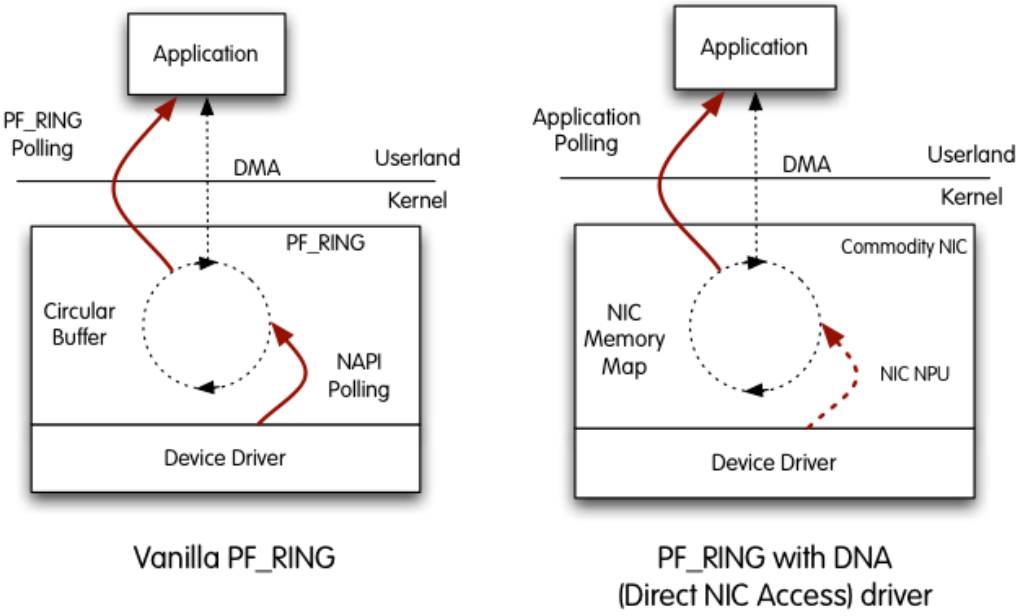
No modo *Vanilla*, PF_RING captura os pacotes das interfaces de rede através da *New API* (NAPI) do *Linux*. NAPI é um mecanismo de captura de pacotes integrado ao *kernel* do *Linux*, desenvolvido para dar suporte ao tráfego *Gigabit*, aliviando o problema de *livelock*, onde em tráfego intenso o sistema não entra em *deadlock*, mas os ciclos de CPU são utilizados para processar interrupções de alta prioridade, fazendo com que processos de usuário entrem em *starvation* (SALAH; QAHTAN, 2008). Com isto, NAPI copia os pacotes da interface de rede

para o *buffer* circular do PF_RING, de onde a aplicação retira os pacotes. Neste modo o PF_RING apenas recebe pacotes.

No modo *DNA*, a memória e registradores da interface de rede são mapeados diretamente para o espaço do usuário. Assim os pacotes são trocados entre a interface de rede e a aplicação sem o intermédio da NAPI, fazendo com que ciclos de CPU sejam usados apenas para consumir pacotes e não os mover da interface de rede. Neste modo operações de envio e recebimento de pacotes são suportadas.

A Figura 5.3 ilustra os dois modos descritos, deixando claro a diferença entre ambos. A esquerda, o modo *Vanilla* com o *buffer* circular servindo de intermédio entre a aplicação e o adaptador de rede, utilizando o NAPI. Na direita, o modo *DNA* onde ocorre o mapeamento direto entre a aplicação e o adaptador de rede.

Figura 5.3 – Modos *Vanilla* e *DNA* de PF_RING

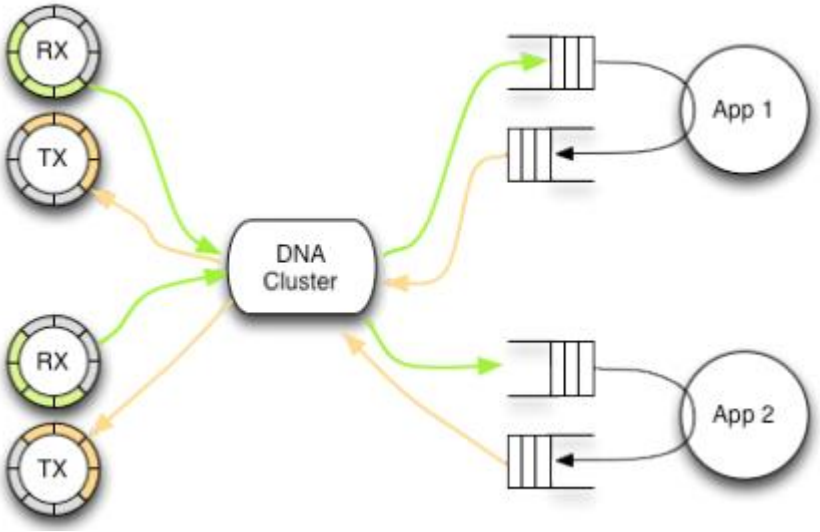


Fonte: (NTOP, 2013).

Para permitir esta troca de pacotes entre o adaptador de rede e aplicações do espaço do usuário, PF_RING conta com uma blibioteca extra, chamada *libzero*, que distribui pacotes entre *threads* e processos, fazendo com que se possa trabalhar com pacotes de variados tamanhos na velocidade máxima da transmissão do adaptador de rede. Para isso são utilizados *drivers* específicos para determinados tipos de adaptadores de rede, que realizam a troca de pacotes entre o adaptador de rede e a aplicação do usuário. *Libzero* possui dois componentes principais: *DNA Cluster* e *DNA Bouncer*.

DNA Cluster implementa agrupamento de pacotes, de modo que todas as aplicações que pertençam ao mesmo grupo (ou *cluster*), compartilhem pacotes que chegam da(s) interface(s) de rede utilizando uma função de balanceamento e transmitindo pacotes em modo de "cópia-zero". Por modo "cópia-zero", entende-se o modo que permite a aplicações ler pacotes na memória sem intermediário envolvido, seja o *kernel* ou mesmo uma cópia do pacote contida em memória. O *cluster* permite que usuários definam suas funções de despacho através de filtragem, distribuição e duplicação de pacotes através de múltiplas *threads (NTOP, 2013)*. Tal esquema é ilustrado pela Figura 5.4.

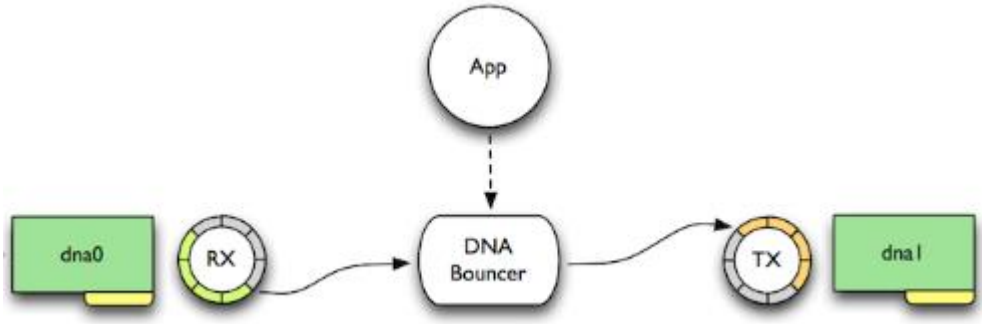
Figura 5.4 – Modo de funcionamento do PF_RING no modo *DNA Cluster*



Fonte: (NTOP, 2013).

DNA Bouncer (Figura 5.5) troca pacotes entre duas interfaces, também em modo "cópia-zero", deixando a cargo do usuário especificar uma função que decida pacote a pacote quais devem ser encaminhados para a próxima interface ou não (NTOP, 2013). No desenvolvimento do injetor FITT, foi utilizado PF_RING *DNA* com seu componente *DNA Bouncer*.

Figura 5.5 - Modo de funcionamento do PF_RING no modo *DNA Bouncer*



Fonte: (NTOP, 2013).

6 O INJETOR DE FALHAS TANATTOS

Neste capítulo é apresentado primeiramente uma análise das ferramentas pesquisadas anteriormente bem como uma breve justificativa para a não adoção das mesmas. Em seguida é detalhado o processo de desenvolvimento do injetor de falhas TANATTOS (*Tool for fAult iNjection on ethercAT implementaTiOnS*), juntamente com uma descrição do funcionamento de cada uma das funções de injeção de falhas desenvolvidas.

6.1 Análise das ferramentas pesquisadas

Dentre as ferramentas apresentadas, poucas podem ser usadas para validar de forma 100% eficaz uma implementação do protocolo EtherCAT, seja por ser uma ferramenta de monitoramento ou por ser uma ferramenta voltada para testes nos estágios finais de desenvolvimento. Outro ponto importante a se destacar é o fato de todas as ferramentas apresentadas serem comerciais, logo, existe um custo extra a ser considerado em todo o processo de validação e pior, seu código fonte pode não estar disponível para criação de casos de teste customizados.

A ferramenta *beSTORM* é voltada a encontrar falhas de segurança e seu uso para uma verificação do protocolo EtherCAT é válido, porém pode não ser considerado mandatório quando se procura testar o correto funcionamento de uma dada implementação. Além disso, por ser uma ferramenta do tipo “caixa-preta”, não é sabido quais destes testes de segurança podem ser utilizados para identificar erros na implementação. Outra desvantagem diz respeito à análise dos resultados dos testes, já que sua versão de demonstração não conta com um recurso para salvar um arquivo de *log* para conferência dos resultados posteriormente.

Elmo Motion Control EtherCAT Network Diagnostics apenas realiza monitoramento e informa diagnósticos de redes EtherCAT, por isso seu uso nas fases de desenvolvimento é dispensável.

As ferramentas *EC-Lyser*, *EC-STA* e *EtherCAT Conformance Test Tool* são capazes de testar completamente uma implementação do protocolo EtherCAT de acordo com a especificação. Porém além de serem ferramentas comerciais e pagas, são ferramentas voltadas para o produto em seu estágio final de desenvolvimento, quando este está prestes a ser lançado no mercado, podendo não se adequar a testes realizados durante o processo de desenvolvimento. Além disso, os testes são realizados seguindo uma abordagem simples, através de estímulos nos dispositivos e comparação das respostas recebidas com resultados esperados, não atuando nas mensagens de comunicação.

6.2 Questões de projeto do injetor TANATTOS

O injetor TANATTOS foi desenvolvido tendo como base o injetor FITT e utiliza sua arquitetura proposta, com a biblioteca PF_RING servindo para captura de pacotes, um módulo com as funções de injeção de falhas, uma interface gráfica que envia parâmetros para as funções de injeção de falhas e sendo executado em um computador único com duas interfaces de rede a fim de interceptar pacotes sendo trocados entre um dispositivo mestre e um dispositivo escravo (Figura 6.1).

Figura 6.1 – Arquitetura do injetor TANATTOS

Fonte: Próprio autor

O primeiro passo no desenvolvimento de injetor TANATTOS foi adaptar FITT para se adequar ao fluxo de pacotes do protocolo EtherCAT (GOMES *et al.*, 2016). Originalmente se filtrava qualquer pacote que não fosse de acordo com o modelo imposto pelo protocolo PROFIsafe. Com isso qualquer pacote que fosse diferente deste modelo não era enviado para o injetor, sendo encaminhado automaticamente. Para contornar esta particularidade do injetor FITT, foi feita uma alteração neste filtro, fazendo com que apenas pacotes pertinentes ao mestre e escravo, ou seja, pacotes com o *EtherType* igual a 0x88a4 fossem enviados para o injetor.

Outra mudança significativa feita foi quanto ao método de escolha dos pacotes escolhidos para sofrerem a falha. Na especificação do FITT, os pacotes eram escolhidos com base em tempos informados pelo usuário. Esta abordagem é bastante determinística, não condizendo com situações reais, já que para as mesmas condições de testes os mesmos resultados são obtidos. Por conta disso, foi adotado um método onde os pacotes que irão sofrer a injeção de falhas são escolhidos de maneira aleatória: o usuário informa um valor entre 0,0 e 1,0, que corresponde à chance percentual de um pacote sofrer a falha (0,0 sendo equivalente a 0% e 1,0 sendo equivalente a 100%). Em seguida, dentro da função de injeção de falhas escolhida, cada vez que um pacote é recebido e enviado ao injetor, um valor também entre 0,0

e 1,0 é escolhido de maneira aleatória seguindo uma distribuição uniforme. Caso este valor aleatório seja menor que o valor informado pelo usuário, o pacote sofrerá a injeção de falhas, caso contrário, o pacote será encaminhado normalmente. Desta forma têm-se uma metodologia mais condizente com a realidade, onde falhas podem acontecer de maneira imprevista a qualquer momento e dificilmente um teste retornará os mesmos resultados caso seja executado várias vezes com os mesmos parâmetros de entrada. Assim, é possível obter uma maior cobertura nos testes de detecção de falhas.

Para garantir que a escolha de números aleatórios seja de fato uniformemente distribuída, foi utilizada a biblioteca científica GSL – GNU (FREE SOFTWARE FOUNATION, 2016), que conta com diversas funções matemáticas já implementadas.

Por conta da substituição de tempos para probabilidades, o módulo de injeção de falhas conta agora com duas e não 4 *threads* como anteriormente, pois as *threads* que gerenciavam os tempos de injeção de falhas não são mais utilizadas. Com isso apenas as *threads* que controlam o envio e recebimento de pacotes nas duas interfaces de rede são utilizadas.

Uma limitação existente em FITT que foi corrigida diz respeito à validação dos dados de entrada fornecidos pelo usuário através da interface gráfica. As janelas responsáveis por receber as configurações dos cenários de injeção de falhas, não realizam qualquer tipo de verificação na consistência dos dados informados. Com isso, se um determinado campo receber um valor não esperado pela aplicação, um erro será gerado durante a execução, visto que valores indefinidos serão passados como parâmetros de entrada para a função de injeção de falhas correspondente. Isto foi corrigido fazendo com que a própria interface gráfica faça a consistência dos dados e informe o usuário através de uma janela *pop-up* quando algum dos dados informados foge do formato esperado. Tais janelas *pop-up* continuarão sendo exibidas enquanto existirem dados sendo informados de maneira incorreta.

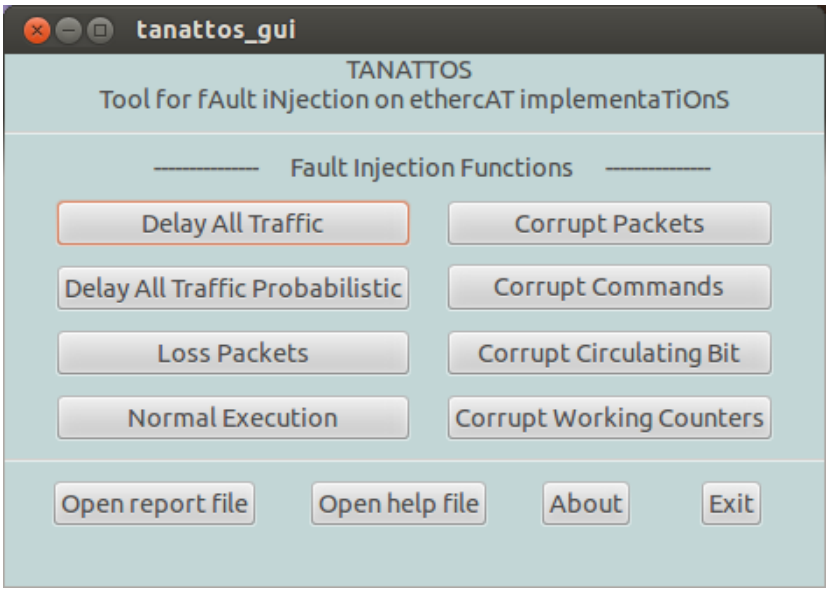
Por exemplo, é comum a todas as funções de injeção de falhas implementadas um valor entre 0,0 e 1,0 correspondente à probabilidade de cada pacote sofrer a falha. Caso seja informado um valor negativo, um valor maior do que 1,0, um caractere ou outro símbolo qualquer, uma janela *pop-up* será exibida informando o usuário sobre o erro.

Optou-se por fazer a consistência dos dados através da interface gráfica para deixar a aplicação que gerencia as funções de injeção de falhas mais coesa e se preocupar apenas com as tarefas relativas à injeção de falhas em si. Além disso, por questões de implementação, fazer com que a aplicação responsável pelas funções de injeção de falhas disparasse mensagens de erro para exibir uma janela via interface gráfica se mostrou uma tarefa bastante complexa.

A interface gráfica foi desenvolvida utilizando-se a ferramenta *Glade* (“Glade - A User

Interface Designer", 2016) e foi planejada para ser a mais intuitiva possível, com botões autoexplicativos para cada funcionalidade. Em sua janela principal (Figura 6.2) conta com botões referentes às funções de injeção de falhas, para abrir o arquivo de *log* com resultados das execuções das funções de injeção de falhas, outro botão que abre o arquivo de ajuda para informar ao usuário instruções de uso da ferramenta, um botão com informações referentes ao desenvolvedor e finalmente um botão para encerrar a ferramenta.

Figura 6.2 – Janela principal do injetor TANATTOS



Fonte: Próprio autor

6.3 Funções de injeção de falhas

Com base no modelo de falhas levantado a partir da especificação do protocolo, foram criadas funções de injeção de falhas que exercitem os mecanismos de detecção e correção de falhas já descritos. Assim, o injetor TANATTOS conta com as funções de atraso de pacotes, atraso de pacotes intermitente, perda de pacotes, corrupção de pacotes, corrupção de comandos de escrita e leitura, corrupção do *Circulating Bit*, corrupção do *Working Counter* e uma função de execução normal. Com exceção da função de execução normal e parte da função de corrupção de pacotes, todas as demais funções foram criadas especialmente para o protocolo EtherCAT, não se baseando nas funções que já existiam em FITT. A seguir é apresentada uma descrição detalhada do funcionamento de cada função.

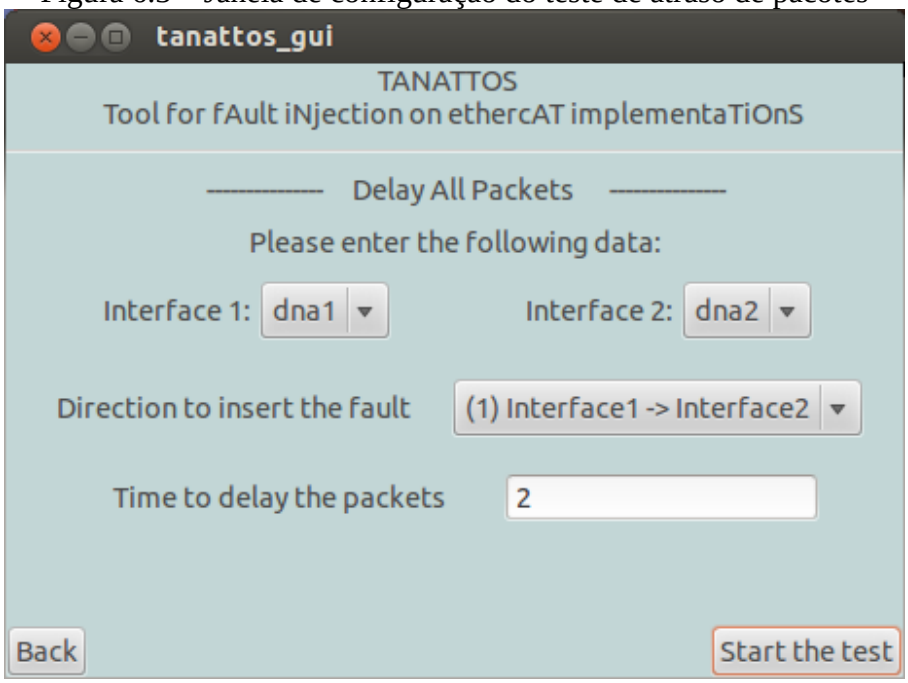
6.3.1 Função de atraso de pacotes

A função de atraso de pacotes realiza o atraso de todo o fluxo de dados entre dois dispositivos. Este teste simula uma situação onde este fluxo de dados excede a capacidade do enlace de comunicação ou quando um dispositivo qualquer da topologia apresenta problemas que façam suas mensagens apresentem demora na transmissão.

Este teste tem dois objetivos principais: o primeiro é avaliar o correto funcionamento dos *watchdogs* do escravo ao reproduzir uma situação de "silêncio" na comunicação por um período de tempo maior que o contador interno dos *watchdogs*. O segundo objetivo é obter uma métrica da margem de operacionalidade do dispositivo mestre através da seguinte questão: se é sabido que um dispositivo escravo introduz um atraso de X milissegundos na comunicação do sistema como um todo, qual o maior atraso que um mestre pode suportar antes de entrar em colapso? Ou ainda, quantos dispositivos escravos um mestre pode gerenciar antes que erros decorrentes de atrasos de comunicação se tornem críticos? Tais perguntas podem ser respondidas a partir de sucessivos testes de atraso de comunicação onde se aumenta gradativamente o tempo em que os pacotes ficam em espera antes de serem enviados.

Na janela de configuração do teste (Figura 6.3), o usuário deve informar os seguintes parâmetros: as duas interfaces *DNA* do PF_RING correspondentes às interfaces de rede físicas utilizadas, a direção onde serão introduzidas as falhas (sentido interface 1 para interface 2 ou interface 2 para interface 1) e um tempo para atrasar todo o fluxo de dados. É importante salientar que para descobrir quais interfaces *DNA* o PF_RING está utilizando, pode-se usar o comando *ifconfig* em um terminal *Linux*. Adicionalmente, os pacotes que trafegam na direção oposta (direção que não foi escolhida pelo usuário) não sofrem qualquer interferência do injetor, sendo transmitidos normalmente.

Figura 6.3 – Janela de configuração do teste de atraso de pacotes



Fonte: Próprio autor

Assim que o primeiro pacote vindo no sentido informado anteriormente chegar à interface de rede, este fica retido no *buffer* circular criado pelo PF_RING durante o tempo especificado pelo usuário e sendo enviado ao seu destino após decorrido este tempo. Como a ordem de transmissão de pacotes deve ser mantida, os demais pacotes que chegam posteriormente também ficarão retidos neste *buffer* circular pelo tempo informado, respeitando a ordem de chegada seguindo uma lógica FIFO (*First In, First Out*). Este buffer circular tem capacidade configurável no momento do carregamento da biblioteca no *kernel* do *Linux*, podendo suportar até 65534 pacotes. Como EtherCAT trabalha com muitos pacotes sendo transmitidos em um intervalo pequeno de tempo, não é recomendável informar um tempo de atraso muito alto, pois experimentalmente comprovou-se que assim que a capacidade do *buffer* circular se esgota, ocorre o descarte de novos pacotes que chegam na interface de rede, causando um erro indesejável que foge do escopo deste teste.

6.3.2 Função de atraso intermitente de pacotes

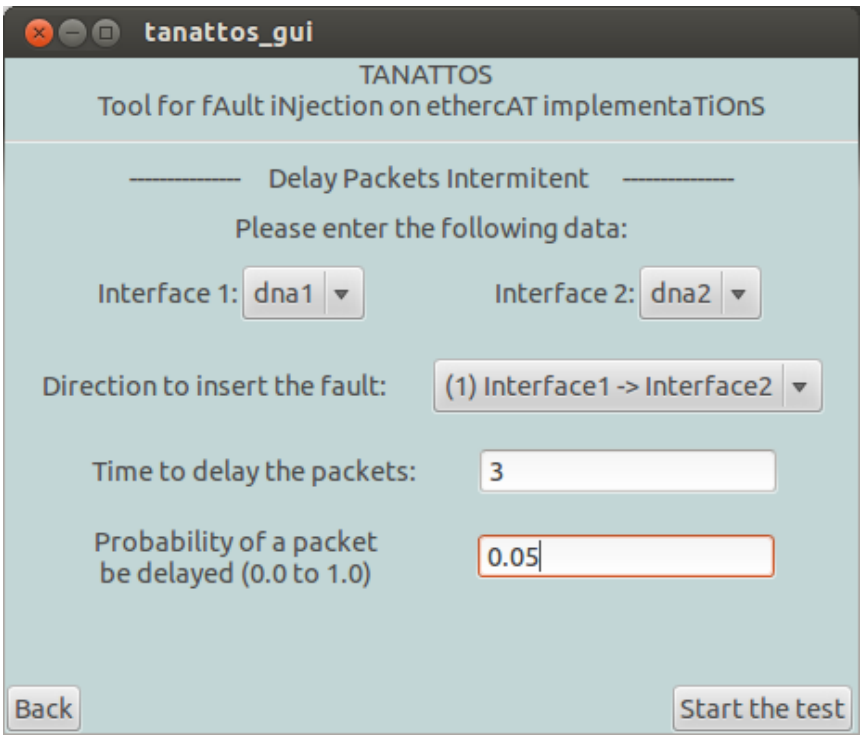
Já nesta função, que é uma variante da função de atraso de pacotes descrita anteriormente e com os mesmos objetivos. O diferencial está no modo com que o fluxo de dados é atrasado: neste caso, o fluxo de pacotes é atrasado de maneira intermitente, ou seja, simula

uma situação onde a comunicação entre dois dispositivos alterna entre períodos de funcionamento normal e períodos onde o fluxo de pacotes inteiro é atrasado.

Esta abordagem foi adotada seguindo o modelo de falhas de comunicação sugerido em (HAN; SHIN; ROSENBERG, 1995) onde falhas transientes, intermitentes e permanentes são consideradas quando se deseja injetar falhas nas mensagens trocadas entre sistemas distribuídos.

Na Figura 6.4 é apresentada a janela de configuração deste teste. Ela segue o mesmo princípio da janela de configuração do teste de atraso de pacotes, porém conta com um parâmetro extra, uma probabilidade de um pacote, e os pacotes subsequentes, serem atrasados. Deste modo, assim que um pacote chega à interface e é repassado ao injetor, um número aleatório entre 0,0 e 1,0 é selecionado seguindo uma distribuição uniforme. Caso este valor selecionado seja menor que o valor informado pelo usuário, este pacote fica retido no buffer circular durante o tempo especificado. Seguindo a mesma lógica do teste de atraso convencional, os demais pacotes que forem chegando também serão atrasados. Após passado o tempo, o primeiro pacote a sofrer a falha é enviado ao seu destino e o processo de escolha de um valor aleatório para injetar ou não a falha se repete para o próximo pacote da fila.

Figura 6.4 – Janela de configuração do teste de atraso intermitente de pacotes



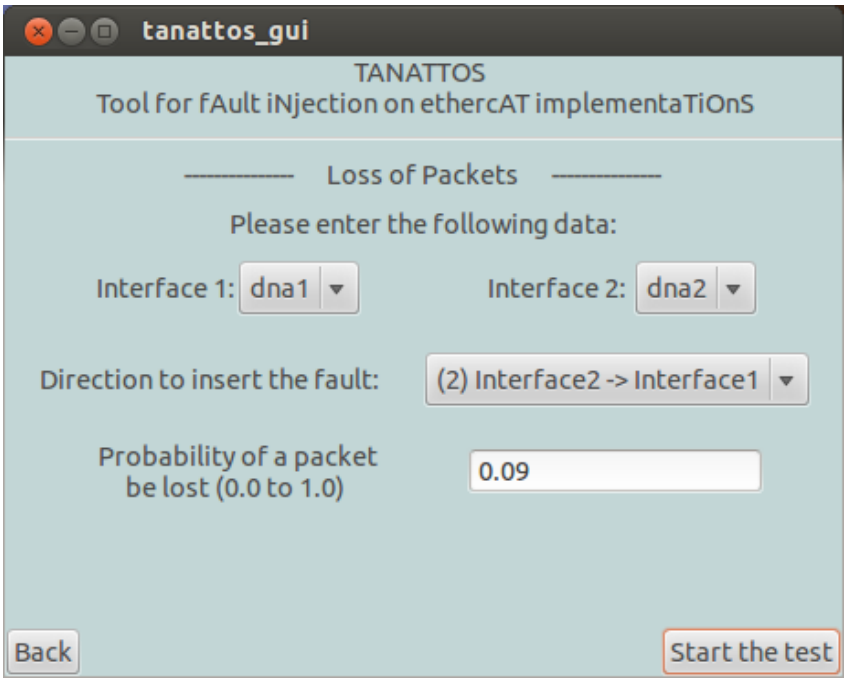
Fonte: Próprio autor

6.3.3 Função de perda de pacotes

Na função de perda de pacotes, o injetor age não enviando pacotes que estiverem sendo recebidos em uma das interfaces de rede. Este cenário simula um barramento ou dispositivo com problemas de conexão, onde pacotes são perdidos durante a comunicação e tem como finalidade avaliar a robustez do dispositivo.

Para este teste o usuário precisa definir (Figura 6.5) as interfaces de rede correspondentes às interfaces *DNA* do PF_RING, o sentido onde as falhas serão inseridas e a probabilidade de que um dado pacote seja descartado.

Figura 6.5 – Janela de configuração do teste de perda de pacotes



Fonte: Próprio autor

A escolha do pacote a ser descartado segue a mesma metodologia empregada nos testes anteriormente descritos: assim que um pacote é enviado ao injetor pela biblioteca PF_RING, um valor numérico entre 0,0 e 1,0 é escolhido aleatoriamente seguindo uma distribuição uniforme. Caso este valor seja menor que o valor informado pelo usuário, o pacote não é enviado para seu destino. A mesma lógica é aplicada a cada pacote subsequente que chegar.

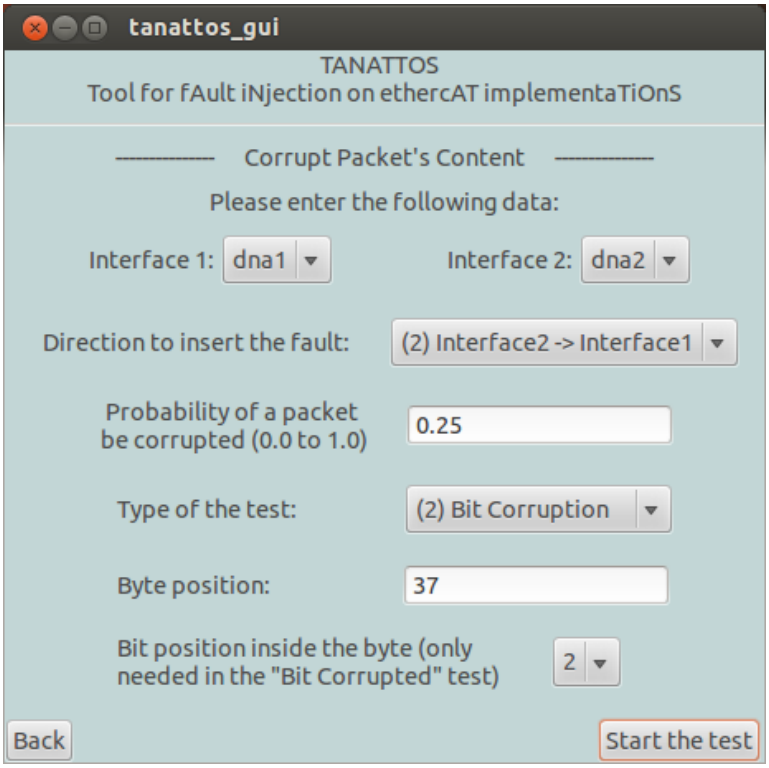
6.3.4 Função de corrupção de pacotes

Com esta função pode-se alterar o valor de um *byte* inteiro do pacote ou um *bit* específico dentro de um *byte*. A alteração é feita através de um *bit-flip* (inversão do valor atual) do(s) *bit(s)* especificado(s) pelo usuário e pode realizada em qualquer *byte* pertencente ao pacote, incluindo valores do cabeçalho.

A partir desta função torna-se possível simular um ambiente onde interferências externas causam falhas na comunicação, alterando os dados sendo transmitidos e tem por objetivo forçar erros na checagem do FCS por parte do escravo a fim de verificar o correto funcionamento das funções de cálculo de CRC da camada de Enlace do escravo através do contador interno responsável por armazenar o número de erros deste tipo.

Pela Figura 6.6 observa-se um aumento significativo na quantidade de parâmetros a serem informados. Além das interfaces de rede que correspondem às interfaces definidas pelo PF_RING (*DNAs*) e de um sentido para injetar as falhas, é necessário informar um valor numérico entre 0,0 e 1,0 correspondente a uma probabilidade de que um dado pacote sofra a falha, o tipo de teste, corrupção de um *byte* completo ou corrupção de um *bit* dentro de um *byte* e a posição deste *bit* dentro do *byte*.

Figura 6.6 – Janela de configuração do teste de corrupção de pacotes



Fonte: Próprio autor

Caso o tipo de teste escolhido seja corrupção de *byte*, todos os *bits* deste *byte* sofrerão um *bit-flip* e o último campo (*Bit position*) não precisa ser informado. De fato, mesmo que algum valor seja informado neste campo, este valor será ignorado caso o teste escolhido seja corrupção de *byte*. Por exemplo, uma corrupção de um *byte* que possua o valor $9D_{16}$ (10011101_2), este acaba se tornando 62_{16} (01100010_2). Caso o tipo de teste escolhido seja corrupção de *bit*, um único *bit* dentro de um *byte* específico sofrerá um *bit-flip*. Neste caso o preenchimento do último campo se torna obrigatório. Ainda sobre a corrupção de *bit*, estão disponíveis valores de 0 a 7, sendo 7 o *bit* mais significativo e 0 o *bit* menos significativo. Por exemplo, ao se aplicar um *bit-flip* no *bit* 7 no *byte* que possui o conteúdo $A9_{16}$ (10101001_2), este se torna 29_{16} (00101001_2).

A escolha do pacote a ser corrompido se dá na mesma forma que nos outros testes apresentados, se o valor de probabilidade escolhido aleatoriamente for menor que o valor informado pelo usuário, o pacote é afetado. Quando um pacote é escolhido para sofrer a falha, o valor do *byte* ou *bit* informado é alterado. Após isto o pacote é enviado ao seu destino.

Um detalhe importante diz respeito aos limites para se escolher a posição do *byte* a ser afetado. Valores negativos ou iguais a zero não serão aceitos (uma janela pop-up informa o usuário). Caso o valor informado exceda o tamanho do pacote selecionado, um novo valor de *byte* para sofrer a injeção de falhas é escolhido aleatoriamente, também seguindo uma distribuição uniforme.

6.3.5 Função de corrupção de dados comandos

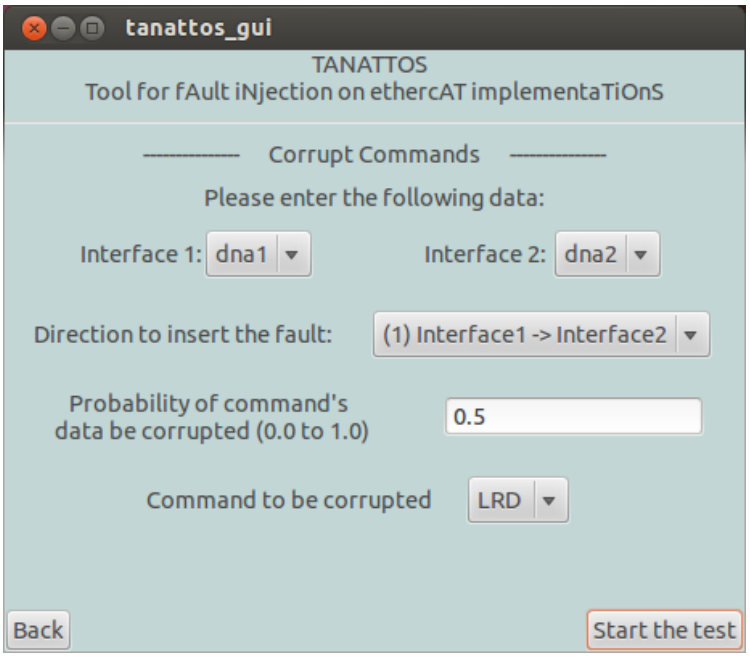
Esta função é um caso específico da função de corrupção. Neste caso o alvo não são apenas pacotes e sim datagramas EtherCAT dentro dos pacotes. Aqui o interesse está em datagramas que carregam comandos de leitura (*Logical Reading – LRD*) ou escrita (*Logical Writing – LWR*).

Neste contexto, o objetivo está em corromper os dados transmitidos por estes datagramas EtherCAT, ou seja, dados escritos no escravo e / ou lidos pelo mestre. Para isto, é feita uma varredura em cada pacote, passando por cada datagrama e verificando se ele contém um comando *LRD* ou *LWR* (escolhido pelo usuário). Com base no tamanho fixo do cabeçalho de cada datagrama (10 *bytes*), no tamanho do campo de *Working Counter* (2 *bytes*), no valor do campo de comprimento dos dados e no *bit* responsável por informar se existem mais datagramas na sequência, é possível passar por todos os datagramas do pacote sem uma grande complexidade.

Após a varredura no pacote em todos os seus datagramas ser concluída, se os dados dos comandos *LRD* ou *LWR* (se existirem no pacote) forem alterados, o CRC do pacote não é calculado para também causar uma falha na verificação do FCS pelo dispositivo que irá recebe-lo.

Na janela de configuração (Figura 6.7), o usuário deve informar os seguintes parâmetros: as duas interfaces *DNA* do PF_RING correspondentes às interfaces de rede físicas utilizadas, a direção onde serão introduzidas as falhas (sentido interface 1 para interface 2 ou interface 2 para interface 1), uma probabilidade para um dado datagrama com o comando especificado ter seus dados alterados e o comando alvo propriamente dito (*LRD* ou *LWR*).

Figura 6.7 – Janela de configuração do teste de corrupção de dados de comandos



Fonte: Próprio autor

6.3.6 Função de corrupção do *Circulating Bit*

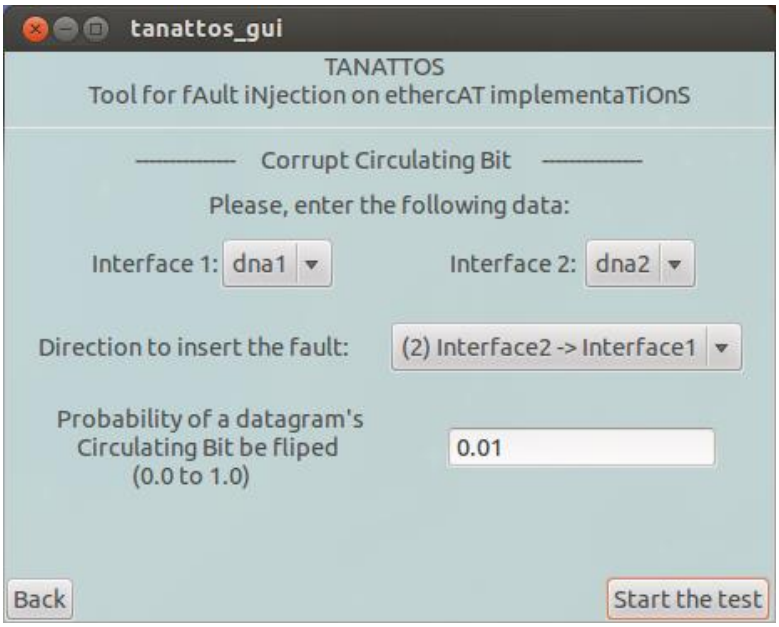
Nesta função, o injetor busca alterar o valor do *Circulating Bit* dos datagramas contidos dentro de cada pacote. Com isso, alterar seu valor de 0 (padrão) para 1 fará com que o pacote não seja processado e sim descartado ao chegar no dispositivo escravo.

A forma como se verificam todos os datragramas EtherCAT dentro do pacote se dá na mesma forma que no teste de corrupção de comandos, tomando o comprimento fixo de 10 *bytes* do cabeçalho de cada datagrama EtherCAT juntamente com os 2 *bytes* do campo *Working Counter*, o comprimento em *bytes* do campo de dados correspondente ao datagrama e o *bit* indicador de um próximo datagrama existente, é possível visitar todos os datagramas existentes.

Como a posição do *Circulating Bit* dentro de seu respectivo *byte* é fixa, a tarefa de encontrá-lo e alterar seu valor é relativamente fácil.

A Figura 6.8 mostra as configurações necessárias para este teste. Os parâmetros necessários não são muito diferentes dos já apresentados, como as interfaces de rede correspondentes às interfaces *DNA* do PF_RING e a direção para ocorrer a injeção de falhas. A diferença está no valor percentual correspondente à chance de a falha ser injetada. Neste caso a chance é relativa a cada datagrama EtherCAT individualmente e não ao pacote como um todo. Logo um valor relativamente alto fará com que vários datagramas sejam atingidos pela falha, enquanto valores baixos irão causar a falha em poucos datagramas.

Figura 6.8 – Janela de configuração do teste de corrupção do *Circulating Bit*



Fonte: Próprio autor

Quando um pacote chega através da biblioteca PF_RING e é encaminhado ao injetor, começa a varredura pelo pacote. Para cada pacote um número aleatório é sorteado para definir se a falha será injetada ou não. Se o valor sorteado for maior que o valor informado pelo usuário, não ocorre a injeção, avança-se para o próximo datagrama e um novo valor é sorteado repetindo o processo. Já se o valor sorteado for menor que o valor informado, o *Circulating Bit* deste datagrama sofre um *bit-flip*.

Caso ao menos um datagrama EtherCAT tenha tido seu *Circulating Bit* alterado, é calculado o CRC32 do pacote, seguindo o modelo adotado no padrão Ethernet e este valor é anexado ao final do pacote para que o mesmo não seja descartado por erro de FCS e sim pelo *Circulating Bit* igual a 1.

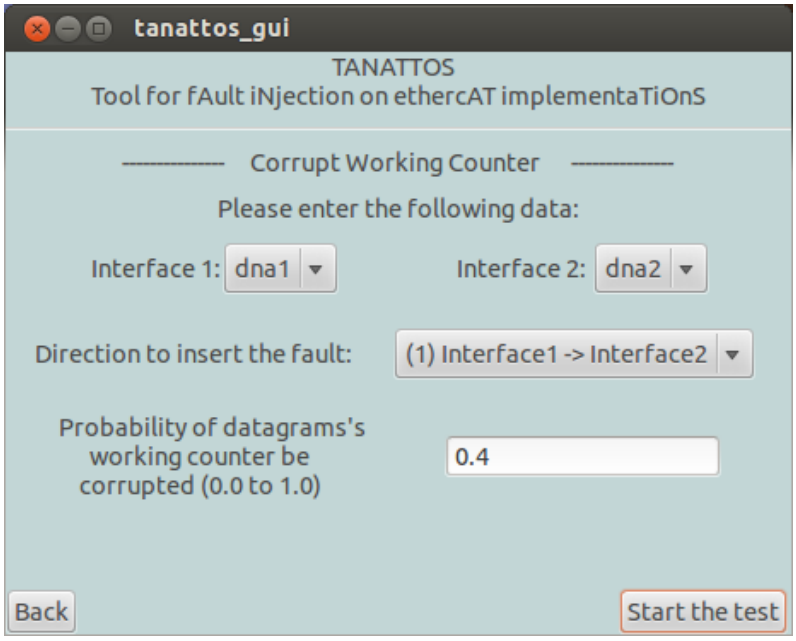
6.3.7 Função de corrupção do *Working Counter*

Já a função de corrupção do *Working Counter* é a única função desenvolvida que não tem como alvo testar a funcionalidade de um dispositivo escravo e sim observar como o mestre se comporta ao receber um valor inesperado de *Working Counter*.

A lógica por trás da injeção de falhas neste cenário é a mesma adotada nos testes de corrupção do *Circulating Bit* e dados dos comandos *LRD* e *LWR*, passando por cada datagrama EtherCAT utilizando os 10 *bytes* de cabeçalho de cada datagrama, o comprimento (em *bytes*) dos dados transmitidos e os 2 *bytes* do campo do próprio *Working Counter* como *offset* para passar por todos os datagramas.

Os parâmetros de configuração (Figura 6.9) também seguem o mesmo padrão da função de corrupção do *Circulating Bit*, com as interfaces de rede usadas, a direção para injetar as falhas e a probabilidade de cada *Working Counter* ser corrompido. Também, neste caso, a chance de injeção de uma falha é com relação a cada datagrama presente dentro do pacote, não ao pacote como um todo.

Figura 6.9 – Janela de configuração do teste de corrupção do *Working Counter*



Fonte: Próprio autor

Assim que um pacote chega ao injetor, um valor numérico entre 0,0 e 1,0 é escolhido aleatoriamente seguindo uma distribuição uniforme. Caso este valor seja menor que o valor informado pelo usuário, o *Working Counter* deste datagrama será corrompido da seguinte

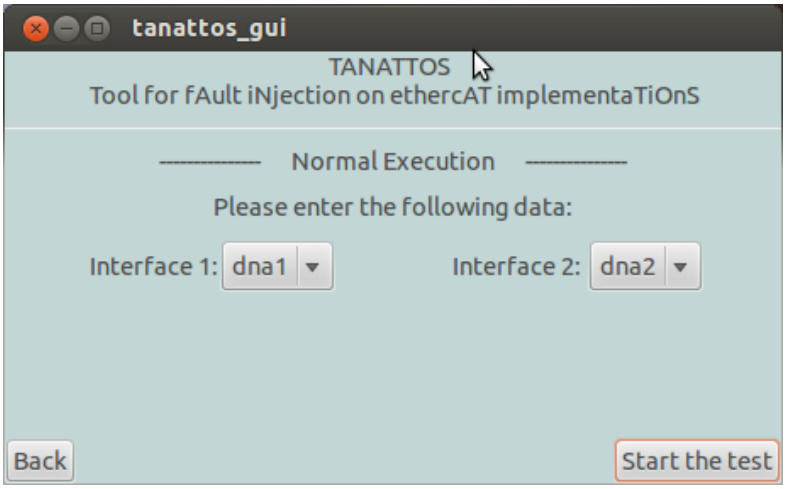
forma: um novo valor aleatório entre 0 e 7 é escolhido seguindo também uma distribuição uniforme. Este valor representa a posição do *bit* do *Working Counter* que sofrerá um *bit-flip*. Após isto o *bit* escolhido irá sofrer a inversão de valor. Caso algum datagrama presente no pacote tenha sido alterado, se calcula o CRC32 deste pacote e este é anexado ao pacote antes de enviá-lo para seu destino.

6.3.8 Função de execução normal

Esta função oferece a opção de uma comunicação normal, isto é, sem que a injeção de falhas ocorra, entre dois dispositivos. Esta função tem o objetivo de verificar se o ambiente de testes montado está configurado de maneira correta, ou seja, se a comunicação entre dois dispositivos está funcionando adequadamente antes que os testes de injeção de falhas possam ser iniciados.

Para este teste, o usuário precisa configurar apenas quais as interfaces de rede usadas correspondem às interfaces *DNA* do PF_RING (Figura 6.10).

Figura 6.10 – Janela de configuração do teste de execução normal



Fonte: Próprio autor

6.4 Geração do relatório pós injeção de falhas

Após o encerramento da execução do injetor de falhas (através de um comando “ctrl + c”), informações pertinentes à função de injeção de falhas escolhida serão salvas em um arquivo de texto, podendo ser acessado pelo botão correspondente na janela principal do injetor, sendo exibido em uma janela do editor de texto padrão do sistema (*gedit* por exemplo).

Neste relatório são exibidos a data e hora de início do teste, a quantidade de pacotes recebidas e enviadas em cada interface e direção bem como uma indicação de qual direção foi escolhida para a injeção de falhas e a quantidade de falhas que foram injetadas. Neste caso, para os testes de atraso e atraso intermitente de pacotes é informado quantos pacotes foram atrasados e o tempo de atraso, sendo que no teste de atraso intermitente é exibido a porcentagem de pacotes afetados. Para o teste de corrupção de pacotes, além da quantidade de pacotes transmitidos entre as interfaces é exibida a quantidade (absoluta e percentual) de pacotes que foram corrompidos. Para os testes de corrupção de dados dos comandos *LRD* e *LWR*, corrupção de *Circulating Bit* e corrupção de *Working Counter* é exibida a quantidade de datagramas afetados e a média de datagramas por pacote. Para o teste de execução normal é exibida apenas a quantidade de pacotes transmitidos entre as interfaces. É comum a todos os testes (com exceção dos testes de atraso e execução normal) a exibição da probabilidade de ocorrência de falha informada pelo usuário, a fim de se ter um comparativo com a real porcentagem de pacotes ou datagramas que foi afetado pela injeção de falhas.

Nos testes que envolvem corrupção dos pacotes (corrupção, *Circulating Bit*, *Working Counter* e dados dos comandos *LRD* e *LWR*), quando um pacote é alterado pela respectiva função, é exibida em uma tela de terminal o conteúdo do pacote antes e depois da falha ser injetada, para se verificar se o resultado do teste condiz com o esperado. Optou-se por exibir tais informações apenas no terminal e não no arquivo de texto, pois como se tem uma quantidade muito grande de pacotes transitando por segundo, a tendência é que o tamanho do arquivo de texto cresça de forma abrupta com o passar do tempo, tornando sua análise difícil e cansativa.

Vale ressaltar que a cada novo resultado escrito no arquivo, os resultados anteriores permanecem intactos, já que a escrita se dá apenas no fim do arquivo, anexando os novos resultados aos já existentes.

7 AMBIENTE EXPERIMENTAL E TESTES REALIZADOS

Este capítulo descreve o ambiente experimental utilizado para validar o injetor de falhas TANATTOS mostrando seu funcionamento de acordo com a especificação apresentada no capítulo anterior. Primeiramente é mostrada a arquitetura do ambiente utilizado, detalhando todos os componentes utilizados e sua função. Em seguida a metodologia de testes é discutida, com explicações referentes a estes testes. E finalmente, são apresentados os resultados obtidos das execuções destes testes.

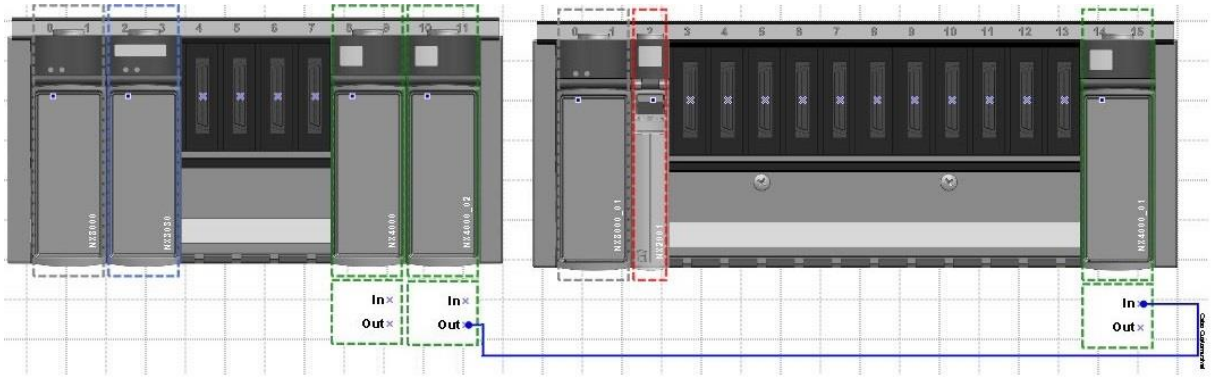
7.1 Arquitetura do ambiente experimental

O ambiente experimental apresentado na seção 6.2, conta com dispositivos que executam o protocolo EtherCAT cedidos pela empresa parceira no projeto: um mestre EtherCAT que é monitorado em tempo real por um computador separado, um escravo EtherCAT que recebe comandos vindos do mestre e um segundo computador que executa o injetor, sendo colocado entre o mestre e o escravo, interceptando os pacotes trocados.

O computador onde o injetor foi executado conta com processador *Intel Core* i7-3770 3.4 GHz com 16 GB de memória RAM, com sistema operacional *Ubuntu* 12.04 LTS 64 *bits* e duas placas de rede *Intel Gigabit* modelo e1000e 82574L que são responsáveis por interligar os dispositivos mestre e escravo ao injetor de falhas. Já o computador que monitorava o mestre EtherCAT conta com um processador *Intel Core 2 Duo* E4000 2.4 GHz, 4 GB de memória RAM e sistema operacional *Windows*® 7 32 *bits.*

Antes que o sistema fosse fisicamente montado, primeiramente foi feito em *software* um modelo deste sistema, definindo quais dispositivos seriam utilizados, como seriam conectados e como seria sua disposição física nos barramentos, de modo que fosse possível o monitoramento via *software*. A Figura 7.1 exibe uma captura de tela que mostra a modelagem final do sistema.

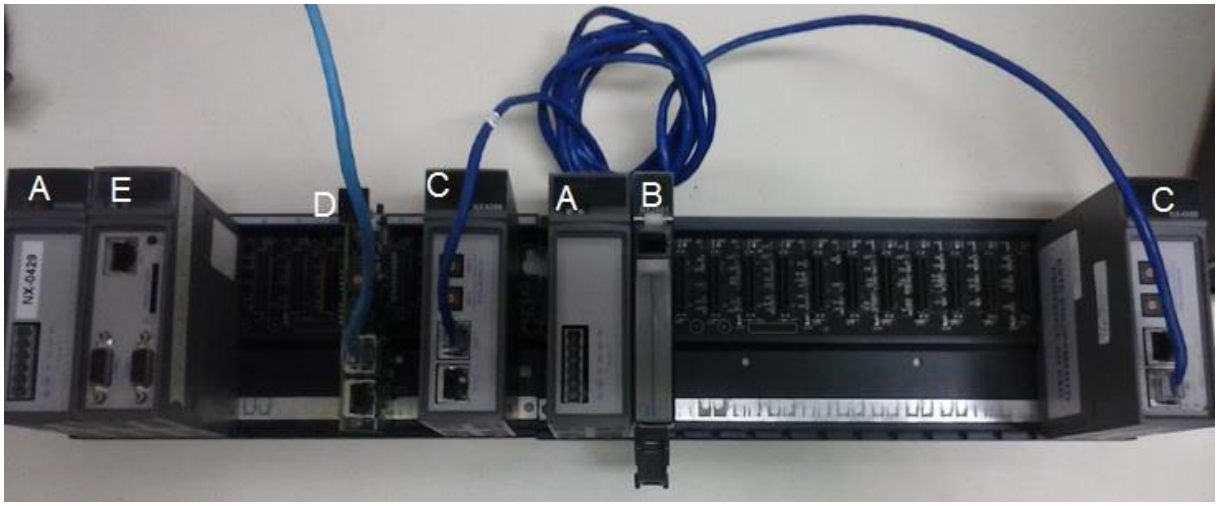
Figura 7.1 – Arquitetura do sistema modelada via *sotfware*



Fonte: Próprio autor

Em seguida, o ambiente de testes final foi montado seguindo o modelo desenvolvido em *software.* A Figura 7.2 mostra o sistema (sem o computador que irá rodar o injetor de falhas e o computador monitor do sistema) com os seguintes componentes: dispositivos "A" são fontes de alimentação 30 W e 24 Vdc. O dispositivo "B" é um módulo a transistor servindo como escravo EtherCAT. Dispositivos "C" são expansores de barramento que convertem sinais LVDS (*Low-Voltage Differential Signaling*) em pacotes Ethernet 100BASE-TX. Estes dispositivos "C" serão conectados cada um a uma das placas de rede do computador executando o injetor de falhas. O dispositivo "E" é um controlador programável que age como mestre. E o dispositivo "D" é um expansor de barramento adaptado apenas para monitorar o fluxo de pacotes que transita pelo mestre e está conectado ao computador monitor para fins de depuração, onde o *software Wireshark* é executado.

Figura 7.2 – Arquitetura do ambiente utilizado durante os experimentos



Fonte: Próprio autor

7.2 Testes preliminares do injetor de falhas

Antes que os experimentos com o injetor fossem realizados com os dispositivos cedidos, tais dispositivos foram dispostos como na Figura 7.2. Em seguida, o fluxo de pacotes no mestre foi observado e salvo com o auxílio do *software Wireshark,* para se ter um modelo real de como os pacotes EtherCAT são estruturados a nível de *bytes*, para que possam ser manipulados posteriormente. De posse deste modelo, foi desenvolvida uma aplicação simples, chamada de *FakeCAT,* em linguagem C que, inicialmente, cria e envia pacotes no formato EtherCAT contendo comandos *LRD* ou *LWR* seguindo a mesma estrutura dos pacotes observados via *Wireshark.*

Um dos principais motivos para criação de tal aplicação foi para poder testar e analisar o resultado das funções de injeção de falhas à medida que estas funções eram desenvolvidas. Como os dispositivos cedidos trocam muitas mensagens em um pequeno intervalo de tempo, a tarefa de analisar o resultado da injeção de falhas apenas pelo *Wireshark,* tornou-se impraticável, visto que os pacotes afetados pela injeção se perdiam em meio aos demais pacotes sendo exibidos.

Esta etapa é chamada de teste “caixa-branca” ou teste estrutural (SOMMERVILLE, 2011), onde o injetor foi testado tendo acesso ao código fonte. Aqui os testes são realizados de modo a exercitar todas as decisões lógicas do programa / função e as entradas possuem valores limites dentro do domínio de valores considerado válido (PRESSMAN, 2011)..

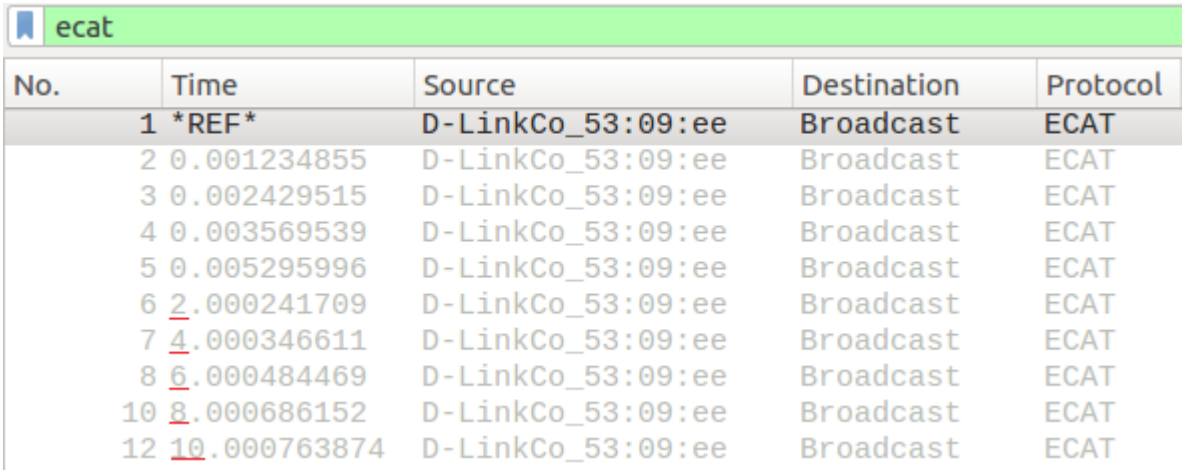
Através de *FakeCAT,* é possível controlar a cadência de pacotes transmitidos por segundo, tornando-se possível, por exemplo, verificar se as funções de atraso de pacotes estavam funcionando de forma correta.

Posteriormente foi adicionada a possibilidade de controlar o número de datagramas EtherCAT enviados em cada pacote. Com isso foi possível testar as funções de corrupção de dados de comandos, corrupção do *Circulating Bit* e corrupção do *Working Counter* para, principalmente, avaliar se os cálculos de *offset* para passar de um datagrama EtherCAT para o outro estavam corretos.

FakeCAT foi executada em um computador separado com *Intel Core 2 Duo* E4000 2.4 GHz, 4 GB de memória RAM e sistema operacional *Ubuntu* 16.04 LTS 32 *bits* e os testes eram realizados enviando pacotes para o computador executando o injetor de falhas, que por sua vez, devolvia os pacotes após a injeção pela mesma interface, se aproveitando da propriedade *full-duplex* da tecnologia Ethernet.

Para os testes de atraso, foram enviados 5 pacotes com um atraso de 2 segundos entre cada um. É possível ver na Figura 7.3 que os 5 primeiros pacotes (que saem em direção ao injetor) possuem um intervalo de 1 milissegundo entre si, enquanto os 5 seguintes possuem um intervalo de 2 segundos entre si.

Figura 7.3 – Análise do resultado do teste preliminar de atraso de pacotes via *Wireshark*



Fonte: Próprio autor

O arquivo de saída que contém o relatório (Figura 7.4) também mostra que 5 pacotes foram atrasados por 2 segundos cada.

Figura 7.4 – Relatório do teste preliminar de atraso de pacotes

```
########################################  Start of the Report ########################################

Delay all traffic started at: Thu Nov 17 11:27:12 2016

Direction 1: Interface 1 -> Interface 2
   ==> Number of packets received at Interface 1 == 5
   ==> Number of packets sent to Interface 2 == 5
   ==> All packets were delayed for 2.00 seconds

Direction 2: Interface 2 -> Interface 1
   ==> Number of packets received at Interface 2 == 0
   ==> Number of packets sent to Interface 1 == 0
   ==> No Fault Injection was applied in this Direction

########################################  End of the Report ########################################
```

Fonte: Próprio autor

Para testar a função de perda de pacotes, o injetor foi configurado a descartar 25% dos pacotes recebidos e *FakeCAT*, enviou 15 pacotes, logo é esperado que algo em torno de 21 ou 22 pacotes sejam exibidos pelo *Wireshark*. Neste caso, foram descartados 4 pacotes (Figura 7.5), totalizando 26,7% do total, o que para um número pequeno de pacotes é uma porcentagem aceitável. A Figura 7.6 exibe o resultado capturado via *Wireshark*.

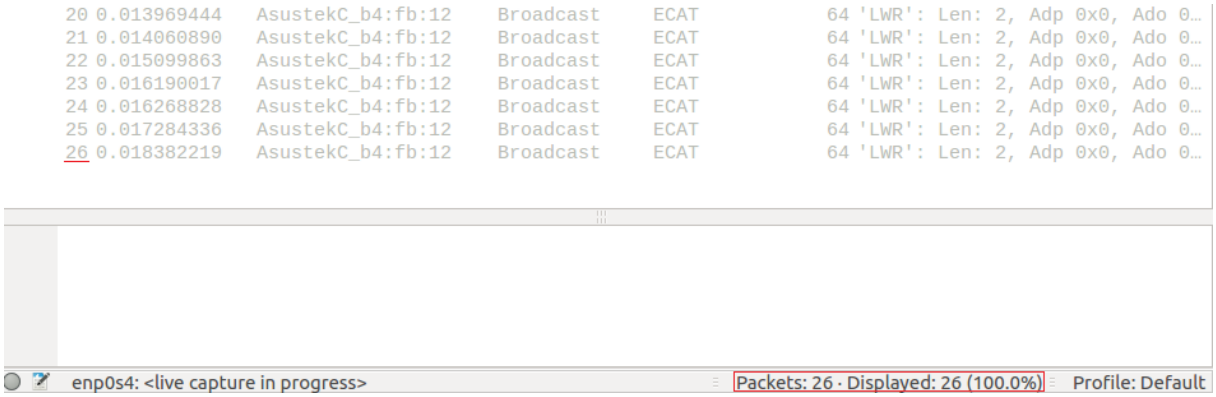
Figura 7.5 – Resultado do teste preliminar de perda de pacotes

```
Loss Packets started at: Thu Nov 17 11:37:25 2016

Direction 1: Interface 1 -> Interface 2
   ==> Number of packets received at Interface 1 == 15
   ==> Number of packets sent to Interface 2 == 11
   ==> 4 (26.67% of the total) packets were discarded according to a probability of 25.00%
```

Fonte: Próprio autor

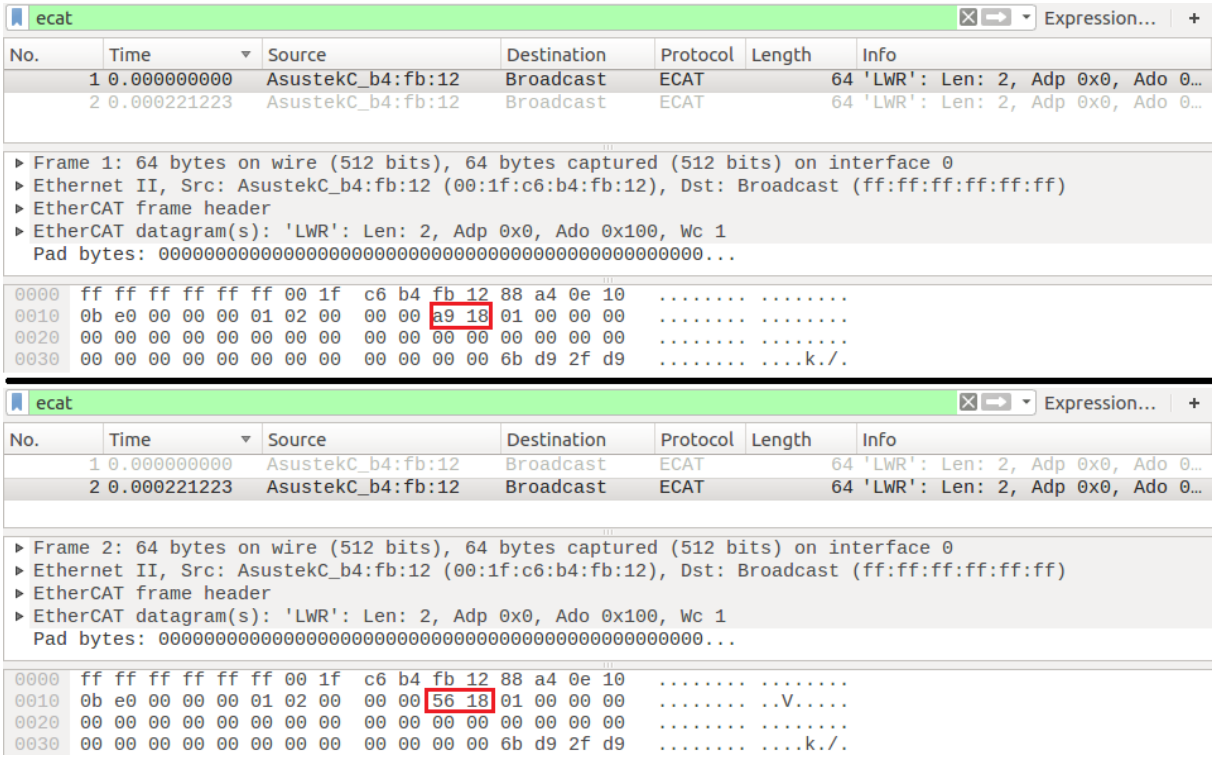
Figura 7.6 – Resultado do teste preliminar de perda de pacotes via *Wireshark*



Fonte: Próprio autor

Nos testes de corrupção de pacotes, neste primeiro momento, o objetivo era testar apenas a lógica de funcionamento, isto é, se a corrupção dos dados ocorreria no ponto esperado ou não. Logo os cálculos de CRC para mascarar a injeção de falhas frente aos mecanismos de *checksum* do *hardware* não eram uma preocupação. Neste contexto, um pacote era enviado ao injetor que era configurado de modo a corromper o 27º *byte* do pacote com uma probabilidade alta, para que a injeção de falhas se manifestasse. Em seguida o pacote alterado era reenviado ao computador de origem para uma comparação (Figura 7.7).

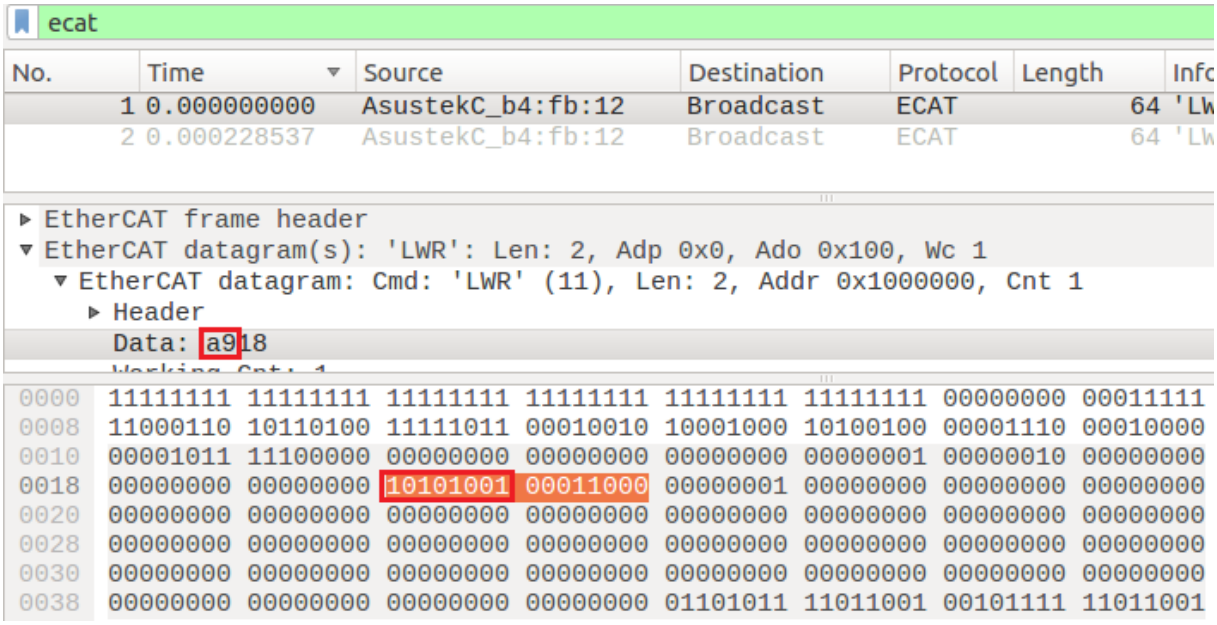
Figura 7.7 – Resultado do teste preliminar de corrupção de pacotes



Fonte: Próprio autor

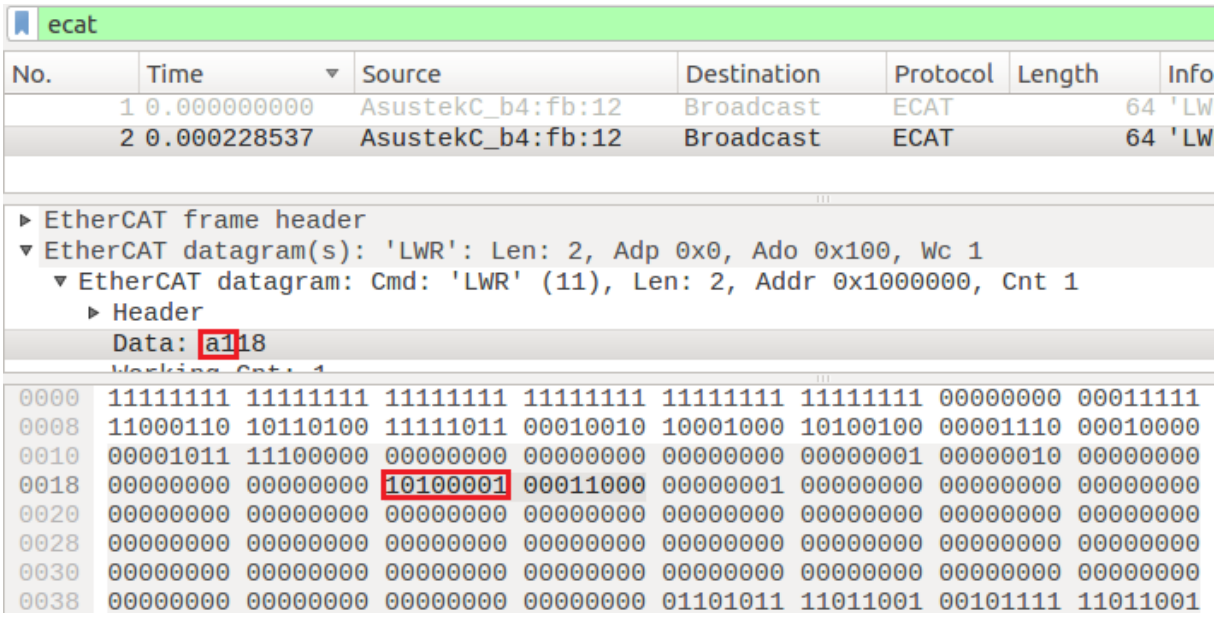
No caso da corrupção de um único *bit* o mesmo *byte* foi mantido, mas desta vez foi escolhido o terceiro *bit* para sofrer um *bit-flip*. Como mencionado na seção 6.3.4, o *bit* mais significativo está localizado à esquerda (*bit* 7) e o *bit* menos significativo à direita (*bit* 0). As Figuras 7.8 e 7.9 exibem o conteúdo dos pacotes antes e depois da corrupção, desta vez exibindo o conteúdo dos pacotes em representação binária ao invés de hexadecimal para facilitar a visualização do *bit-flip*.

Figura 7.8 – Conteúdo de um pacote antes da corrupção de um *bit*



Fonte: Próprio autor

Figura 7.9 - Conteúdo de um pacote após a corrupção de um *bit*

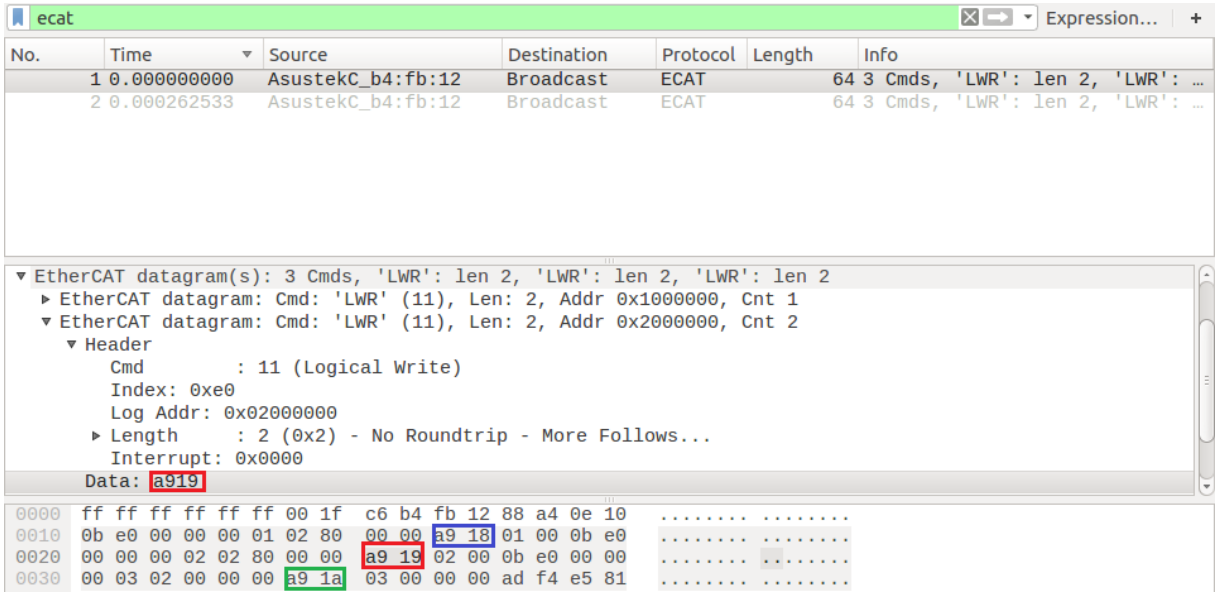


Fonte: Próprio autor

Para a corrupção de dados de comandos, o injetor recebeu um pacote contendo 3 datagramas EtherCAT com comandos *LWR* e dados distintos. Aqui o objetivo principal é avaliar a capacidade do injetor em passar por todos os datagramas EtherCAT, parando no último e não avançando para uma área de memória inapropriada. O segundo objetivo neste contexto é observar se os *bytes* corrompidos pelo injetor eram de fato correspondentes aos dados

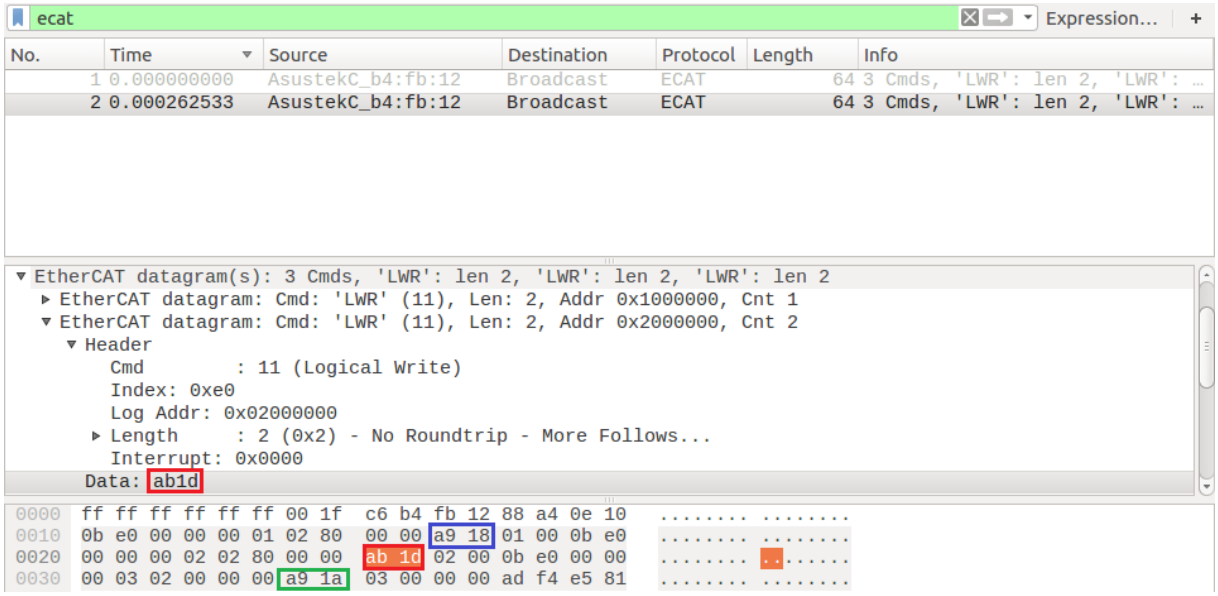
carregados pelos datagramas EtherCAT. As Figuras 7.10 e 7.11 mostram o conteúdo dos pacotes recebidos através do *Wireshark* comparando o pacote original enviado e o recebido após a corrupção. A mudança ocorreu no segundo datagrama EtherCAT do pacote, enquanto os demais permaneceram intactos. Em azul está destacado o dado carregado pelo primeiro datagrama, em vermelho o dado carregado pelo segundo datagrama e em verde o dado carregado pelo terceiro datagrama.

Figura 7.10 – Conteúdo dos dados do pacote antes da corrupção



Fonte: Próprio autor

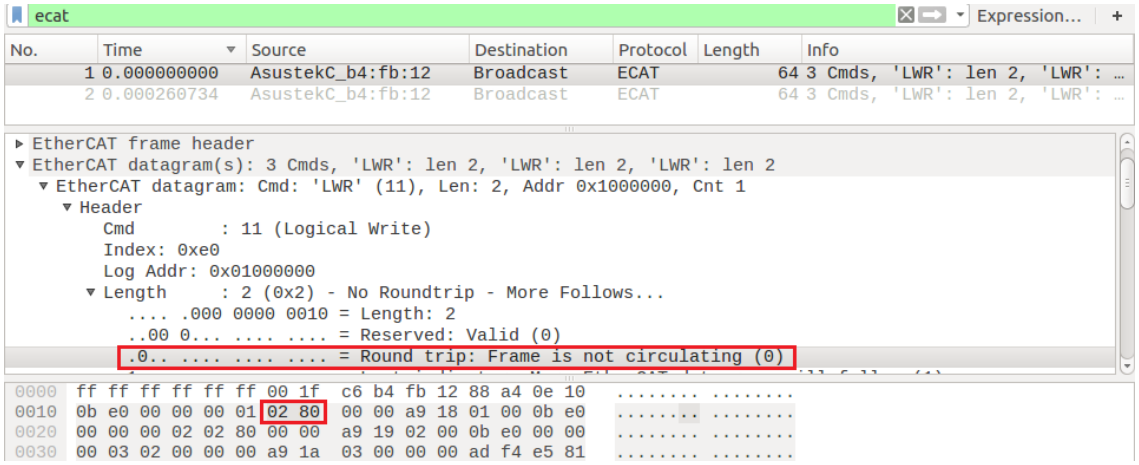
Figura 7.11 – Conteúdo dos dados do pacote após a corrupção



Fonte: Próprio autor

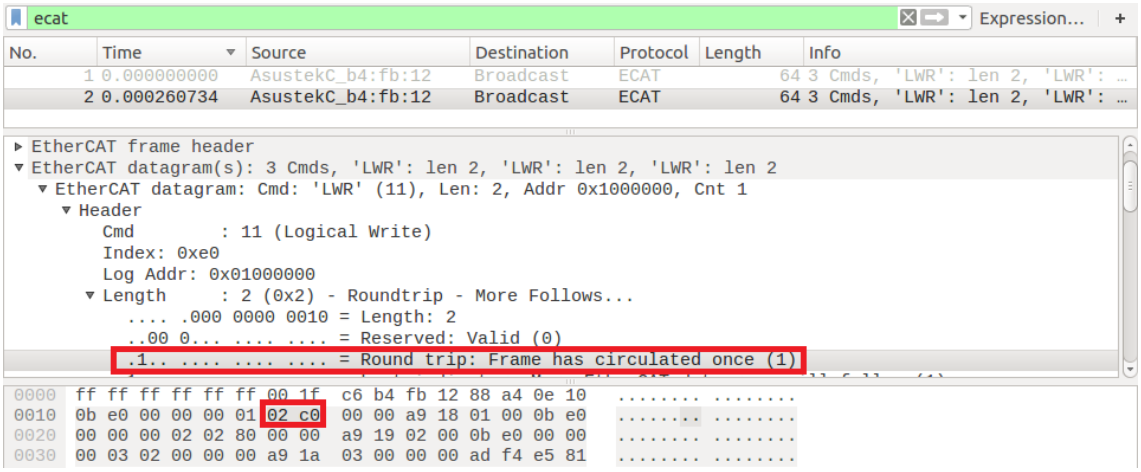
No caso da função de corrupção do *Circulating Bit*, o procedimento foi o mesmo mencionado anteriormente, com um pacote contendo 3 datagramas EtherCAT sendo recebido pelo injetor, para avaliar se a função era capaz de percorrer todos os datagamas, parando quando não houvessem mais e se o *bit-flip* ocorreria como o esperado. As Figuras 7.12 e 7.13 exibem o resultado observado através do Wireshark.

Figura 7.13 – Datagrama EtherCAT antes da alteração do *Circulating Bit*



Fonte: Próprio autor

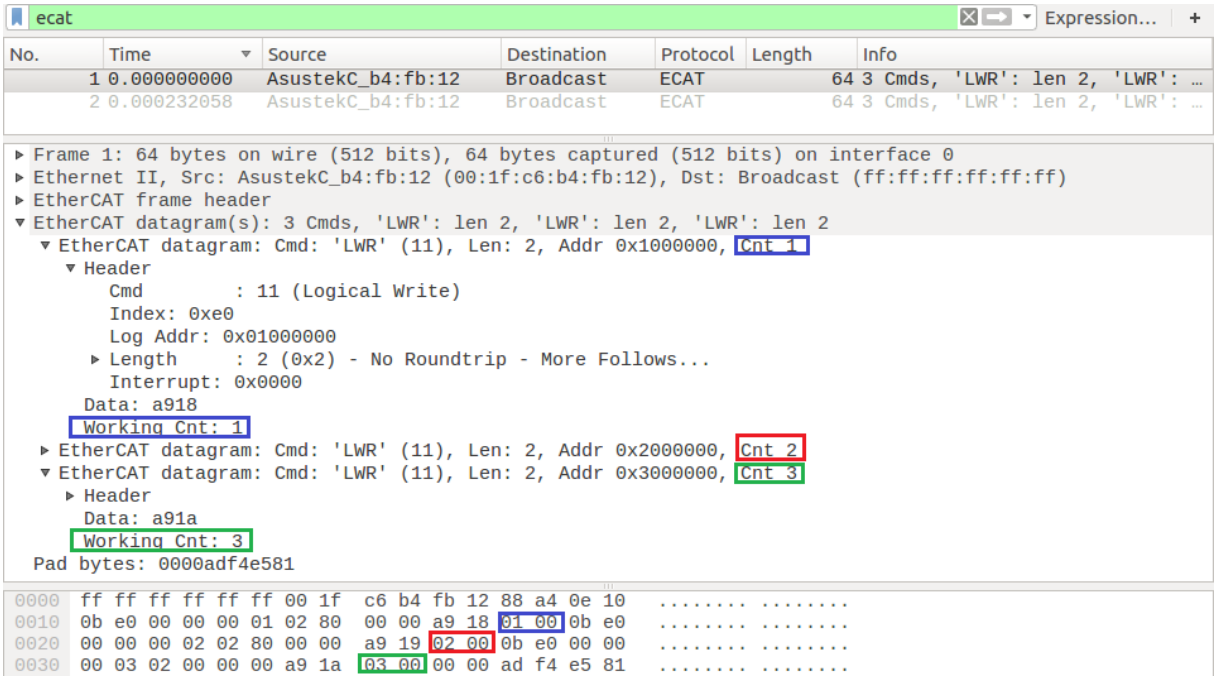
Figura 7.14 - Datagrama EtherCAT após a alteração do *Circulating Bit*



Fonte: Próprio autor

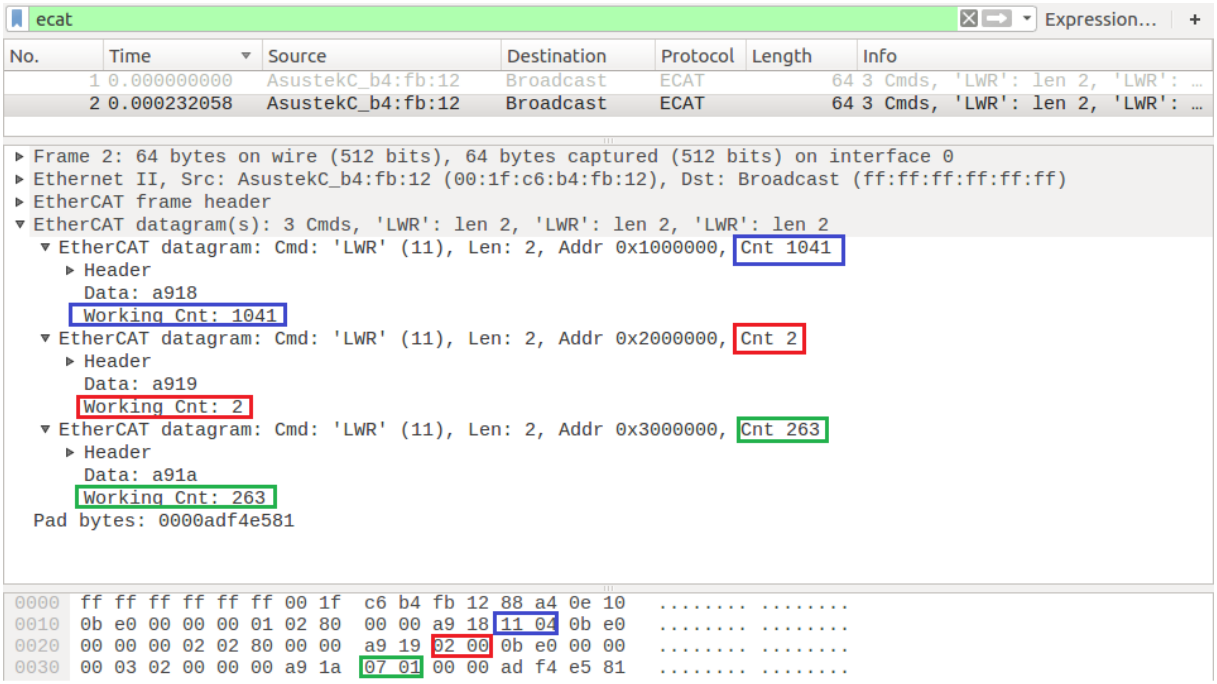
Finalmente, para o teste de corrupção do *Working Counter*, os métodos utilzados foram os mesmos citados nos dois últimos testes apresentados. Novamente 1 pacote com 3 datagramas EtherCAT foi enviado ao injetor, que deveria procurar em cada um destes datagramas seu *Working Counter* correspondente e alterá-lo realizando *bit-flips* em *bits* selecionados de forma aleatória. O resultado pode ser observado nas Figuras 7.15 e 7.16 onde o *Working Counter* do primeiro e terceiro datagramas EtherCAT foi alterado.

Figura 7.15 – *Working Counter* dos datagramas EtherCAT antes da corrupção



Fonte: Próprio autor

Figura 7.16 - *Working Counter* dos datagramas EtherCAT após corrupção



Fonte: Próprio autor

Em azul está apresentado o *Working Counter* relativo ao primeiro datagrama EtherCAT, em vermelho, o *Working Counter* do segundo datagrama e em verde, o *Working Counter* do terceiro datagrama. Ao se comparar o valor interpretado pelo *Wireshark* com os valores em

hexadecimal no corpo do pacote, pode-se causar uma certa confusão pelos valores não estarem iguais. Isto se dá pela ordem com que EtherCAT transfere e recebe os *bytes*. Assumindo que $b = b_0 ... b_{n-1}$ seja uma sequência de *bits* e k um inteiro não negativo tal que $8(k-1) < k < 8k$. Então b é transferido em k *bytes*, na seguinte sequência: $b_7 ... b_0$ seguido de $b_{15} ... b_8$ até $b_{8k-1} ... b_{8k-8}$ (ETHERCAT TECHNOLOGY GROUP, 2013c). Desta forma, o valor 1104 mostrado no corpo do pacote, deve ser interpretado como 0411_{16} ou 411_{16} que é igual a 1041_{10}. Pelo mesmo motivo, 0701 deve ser lido como 0107_{16} ou 107_{16} que é igual a 263_{10}.

7.3 Testes para validação do injetor de falhas

Após o injetor ser testado seguindo o modo "caixa-branca", foi aplicada a técnica de teste "caixa-preta" ou teste comportamental (PRESSMAN, 2011), onde o ponto de interesse está apenas nas entradas do sistema e suas respectivas saídas. Neste caso não há acesso ao código fonte, casos de teste são derivados apenas com base na especificação (SOMMERVILLE, 2011) e busca-se encontrar erros de interface, erros em estruturas de dados, erros de comportamento ou desempenho e erros de inicialização e término (PRESSMAN, 2011).

Neta etapa foram utilizados os dispositivos cedidos pela empresa parceira, onde o mestre foi conectado a uma das placas de rede do computador que executava o injetor de falhas e o escravo foi conectado à outra placa do mesmo computador. O sistema foi diagnosticado em tempo real através de uma aplicação *MasterTool*, para programar e monitorar o dispositivo mestre, que também foi cedida pela empresa.

A aplicação utilizada para o monitoramento conta com uma série de variáveis para diferentes diagnósticos. As variáveis utilizadas para avaliar o funcionamento dos dispositivos incialmente são: *byWHSBBusErrors* que é um contador de falhas no barramento variando de 0 a 255. A variável *adwModulePresenceStatus*, é um *array* de DWORDS representando o *status* de presença dos dispositivos de entrada e saída declarados nos barramentos individualmente. Cada DWORD deste *array* representa um bastidor, cujas posições são representadas pelos *bits* dessas DWORDs. Por exemplo, o *bit* 0 da DWORD 0 equivale a posição zero do bastidor zero. Já *adwRackIOErrorStatus* é o diagnóstico de erros nos dispositivos dos bastidores e assim como *adwModulePresenceStatus*, é um *array* de DWORDs representando os dispositivos em seus respectivos bastidores. A variável *byHotSwapAndStartupStatus* informa através de códigos de erro, situações anormais no barramento responsáveis pela parada da aplicação.

7.3.1 Testes de execução normal

Inicialmente foi aplicada a função de execução normal para saber o estado normal das variáveis apresentadas anteriormente, ou seja, qual valor é exibido por elas nas condições normais de funcionamento. Assim, são obtidos os valores correspondentes a uma execução sem falhas como parâmetro de comparação para testes futuros.

Após um breve período de operação a execução foi terminada e o resultado do relatório do injetor de falhas pode ser visto na Figura 7.17, onde todos os pacotes que chegaram à interface 1 foram repassados para a interface 2 e vice-e-versa, dando indícios que a comunicação entre mestre e escravo ocorreu sem problemas.

Figura 7.17 – *Log* resultante da operação normal dos dispositivos mestre e escravo

```
########################################  Start of the Report ########################################

Normal Execution started at: Wed Nov 16 11:42:13 2016

Direction 1: Interface 1 -> Interface 2
   ==> Number of packets received at Interface 1 == 21539
   ==> Number of packets sent to Interface 2 == 21539
   ==> No Fault Injection was applied in this Direction

Direction 2: Interface 2 -> Interface 1
   ==> Number of packets received at Interface 2 == 21539
   ==> Number of packets sent to Interface 1 == 21539
   ==> No Fault Injection was applied in this Direction

########################################  End of the Report ########################################
```

Fonte: Próprio autor

Em seguida as variáveis de diagnóstico foram observadas para se saber qual seu valor em uma execução normal. A Figura 7.18 mostra o estado das variáveis *byHotSwapAndStartupStatus* e *byWHSBBusErrors*. A variável *byHotSwapAndStartupStatus* exibe o estado NORMAL_OPERATING_STATE, que segundo o manual do fabricante, quer dizer que a aplicação está em modo *Run*, ou seja, a aplicação rodando no mestre está operando sem maiores problemas e o contador *byWHSBBusErrors* está marcando 0 erros, o que significa que a comunicação de fato ocorreu sem problemas.

Figura 7.18 – Variáveis *byHotSwapAndStartupStatus* e *byWHSBBusErrors* após execução normal

WHSB	T_DIAG_WHSB	
byHotSwapAndStartupStatus	EN_HOT_SWAP	NORMAL_OPERATING_STATE
adwRackIOErrorStatus	ARRAY [0..31] OF DWORD	
adwModulePresenceStatus	ARRAY [0..31] OF DWORD	
byWHSBBusErrors	BYTE	0

Fonte: Próprio autor

No caso da variável *adwModulePresenceStatus*, ela exibe os valores 1024 na sua posição 0 e 16388 na posição 1 (Figura 7.19).

Figura 7.19 – Variável *adwModulePresenceStatus* após execução normal

adwModulePresenceStatus	ARRAY [0..31] OF DWORD	
adwModulePresenceStatus[0]	DWORD	1024
adwModulePresenceStatus[1]	DWORD	16388
adwModulePresenceStatus[2]	DWORD	0
adwModulePresenceStatus[3]	DWORD	0

Fonte: Próprio autor

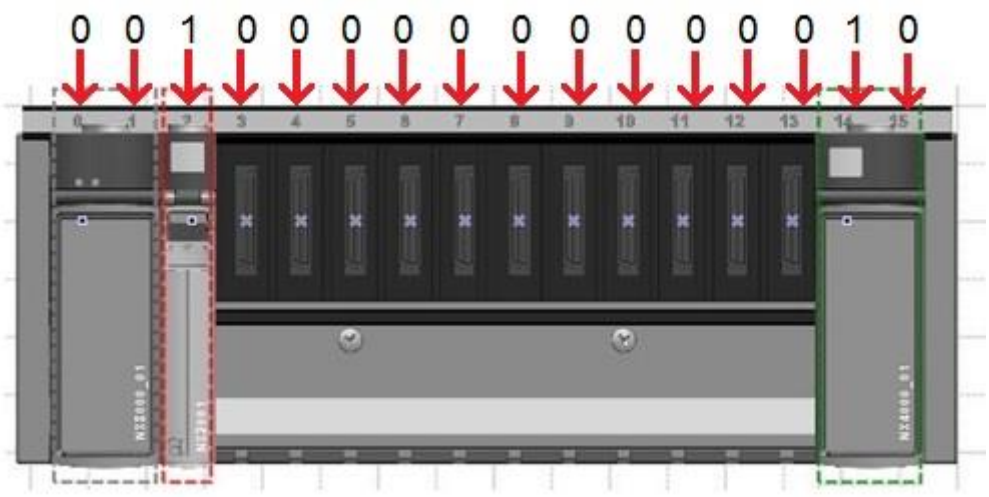
O valor 1024 da posição 0 pode não fazer sentido quando exibido em decimal, porém quando convertido para binário (0100 0000 0000), indica quais dispositivos estão presentes no bastidor 0, que foi configurado como sendo o bastidor contendo o mestre. Quando este valor é lido da direita para a esquerda (0000 0000 0010) e comparado com a montagem do bastidor nota-se que as posições em "1" realmente correspondem aos dispositivos montados nos bastidores. É importante ressaltar que esse diagnóstico é válido para todos os dispositivos do fabricante, exceto fontes de alimentação, dispositivos presentes nos bastidores, mas não declarados via *software* e Unidades Centrais de Processamento – UCPs (categoria onde o mestre EtherCAT cedido se enquadra). No caso do sistema montado, a Figura 7.20 sumariza a explicação, onde apenas o expansor de barramento para troca dos pacotes é reconhecido, enquanto a fonte de alimentação, o dispositivo mestre e o expansor adaptado como *sniffer* não tem sua presença informada.

Figura 7.20 – *Status* do bastidor 0

Fonte: Próprio autor

Seguindo a mesma lógica, para o valor 16388 apresentado em decimal, ao convertê-lo para binário (0100 0000 0000 0100), tomando ele da direita para a esquerda (0010 0000 0000 0010) e comparando-o com o bastidor 1 que contém o dispositivo escravo, observa-se que os dispositivos presentes (escravo e expansor de barramento) correspondem a cada “1”, como mostra a Figura 7.21.

Figura 7.21 – *Status* do bastidor 1



Fonte: Próprio autor

Finalmente, a variável *adwRackIOErrorStatus* apresenta valores iguais a 0 para as duas primeiras posições (Figura 7.22), pois nenhum dispositivo nos bastidores apresenta erros a serem reportados.

Figura 7.22 – Variável *adwRackIOErrorStatus* após execução normal

adwRackIOErrorStatus	ARRAY [0..31] OF DWORD	
adwRackIOErrorStatus[0]	DWORD	0
adwRackIOErrorStatus[1]	DWORD	0
adwRackIOErrorStatus[2]	DWORD	0
adwRackIOErrorStatus[3]	DWORD	0

Fonte: Próprio autor

7.3.2 Testes de perda de pacotes

Para o teste de perda de pacotes, o injetor foi configurado a descartar 25% dos pacotes na direção do mestre para o escravo. Este valor foi escolhido porque probabilidades menores do que 25% não foram capazes de gerar erros na operação normal dos dispositivos.

Pela entrada gerada no arquivo de log do injetor (Figura 7.23), fica evidente que de fato

houve perda de pacotes, onde chegaram 10768 pacotes na interface 1 (vindos do mestre) e foram repassados 8012 pacotes para a interface 2 (indo em direção ao escravo). Seguindo o funcionamento do protocolo EtherCAT, onde os pacotes ao chegarem ao final da topologia retornam ao mestre, estes 8012 pacotes retornaram para a interface 2 (vindos do escravo) e foram repassados à interface 1 (indo em direção ao mestre).

Figura 7.23 - *Log* resultante da perda de pacotes entre os dispositivos mestre e escravo

```
########################################  Start of the Report ########################################

Loss Packets started at: Wed Nov 16 11:57:15 2016

Direction 1: Interface 1 -> Interface 2
   ==> Number of packets received at Interface 1 == 10768
   ==> Number of packets sent to Interface 2 == 8012
   ==> 2756 (25.59% of the total) packets were discarded according to a probability of 25.00%

Direction 2: Interface 2 -> Interface 1
   ==> Number of packets received at Interface 2 == 8012
   ==> Number of packets sent to Interface 1 == 8012
   ==> No Fault Injection was applied in this Direction

########################################  End of the Report ########################################
```

Fonte: Próprio autor

Com relação às variáveis de diagnóstico apresentadas anteriormente, a variável *byHotSwapAndStartupStatus* agora apresenta o *status* APPL_STOP_MODULES_NOT_READY, que indica que a aplicação se encontra em modo *Stop* e todas as consistências foram realizadas com sucesso, mas os dispositivos de E/S não estão aptos para a partida do sistema. Isto é um indicativo que a perda de pacotes pode ter sido significativa a ponto de os dispositivos mestre e escravo não conseguirem executar os procedimentos necessários para o início do funcionamento.

A variável *adwRackIOErrorStatus* por sua vez, apresentava os valores 1024 para o bastidor 0 e 16388 para o bastidor 1. Como explicado no teste de execução normal, pela lógica por trás destes valores, fica claro que os expansores de barramento em ambos os bastidores e o dispositivo escravo apresentaram problemas devido à perda de pacotes. Os dados coletados podem ser vistos na Figura 7.24.

Figura 7.24 – Variável *adwRackIOErrorStatus* após a ocorrência de perda de pacotes

WHSB	T_DIAG_WHSB	
byHotSwapAndStartupStatus	EN_HOT_SWAP	APPL_STOP_MODULES_NOT_READY
adwRackIOErrorStatus	ARRAY [0..31] OF DWORD	
adwRackIOErrorStatus[0]	DWORD	1024
adwRackIOErrorStatus[1]	DWORD	16388
adwRackIOErrorStatus[2]	DWORD	0
adwRackIOErrorStatus[3]	DWORD	0

Fonte: Próprio autor

7.3.3 Testes de atraso de pacotes

Para este teste, os pacotes seriam atrasados por 1,5 segundos, pois os *watchdogs* internos do escravo estavam configurados para aguardar por pacotes por até 1 segundo. Desta vez cada pacote tinha uma probabilidade de 1% de chance de ser atrasado. O resultado do experimento é mostrado na Figura 7.25.

Figura 7.25 - *Log* resultante da perda de pacotes entre os dispositivos mestre e escravo

```
########################################  Start of the Report ########################################

Delay All Traffic Intermitent started at: Wed Nov 16 12:15:58 2016

Direction 1: Interface 1 -> Interface 2
   ==> Number of packets received at Interface 1 == 1567
   ==> Number of packets sent to Interface 2 == 1566
   ==> 18 (1.15% of the total) packets were delayed for 1.50 secs according to a probability of 1.00%

Direction 2: Interface 2 -> Interface 1
   ==> Number of packets received at Interface 2 == 1513
   ==> Number of packets sent to Interface 1 == 1513
   ==> No Fault Injection was applied in this Direction

########################################  End of the Report ########################################
```

Fonte: Próprio autor

Desta vez o número de pacotes enviados na direção mestre para escravo é diferente da quantidade de pacotes enviados na direção escravo para mestre. Uma possível explicação é que ao acontecerem erros devido à falta de comunicação detectados pelos *watchdogs*, o escravo deixa de entregar seus sinais de saída, como explicado na seção 3.9, o que explicaria o número de pacotes enviados pelo escravo ser menor.

Ainda, as variáveis *byHotSwapAndStartupStatus* e *byWHSBBusErrors* mostraram valores diferentes dos exibidos durante do teste de execução normal (Figura 7.26), o que também mostra que a comunicação entre mestre e escravo foi perturbada. Novamente a variável *byHotSwapAndStartupStatus* apresenta o *status* APPL_STOP_MODULES_NOT_READY, que mostra que os dispositivos não estão aptos a se comunicar. Já a variável *byWHSBBusErrors* apresenta uma contagem de 215 erros detectados. Este valor alto e diferente de 18 (número de pacotes atrasados) se deve ao fato de que o contador *byWHSBBusErrors* ser de certa forma genérico e contabilizar outros erros no barramento além do “silêncio” na comunicação.

Figura 7.26 – Variáveis *byHotSwapAndStartupStatus* e *byWHSBBusErrors* após atraso de pacotes

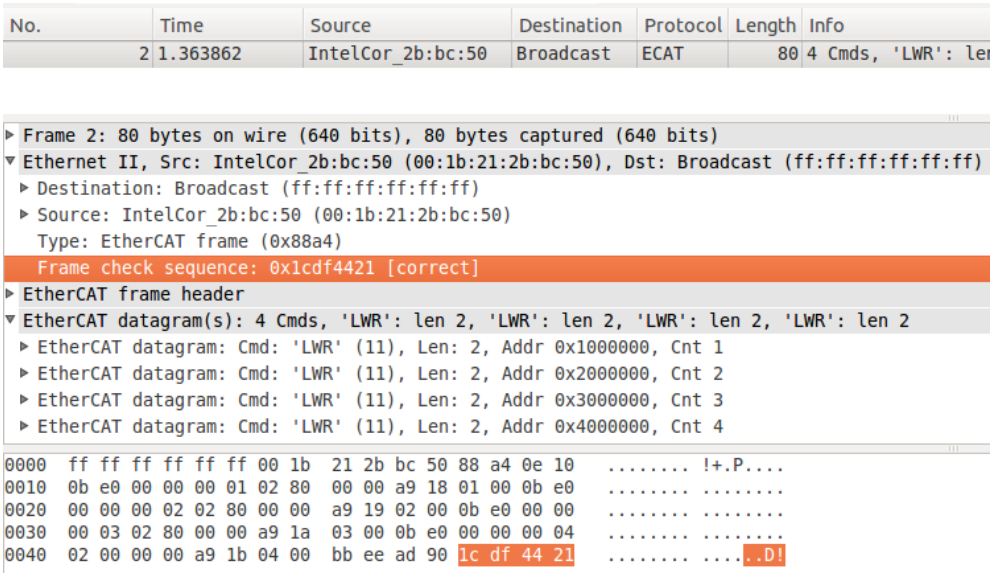
Fonte: Próprio autor

7.3.4 A questão dos testes envolvendo corrupção de pacotes

Para os testes que envolvem algum tipo de alteração no conteúdo dos pacotes (corrupção de dados de comandos, corrupção do *Circulating Bit* e corrupção do *Working Counter*), a tarefa de injetar as falhas se tornou complexa devido ao fato de que pacotes com erros de CRC normalmente são descartados pelas interfaces de rede na camada física antes de serem enviados às camadas superiores.

Inicialmente, buscou-se uma maneira de fazer com que o *driver* / *hardware* das interfaces de rede fizesse duas coisas: passar o campo de FCS dos pacotes para as camadas superiores e repassar pacotes com erros de CRC para as camadas superiores ao invés de descartá-los. Isto é obtido com o uso do utilitário *ethtool* presente no próprio *Linux*. Através dos comandos *sudo ethtool -K <interface> rx-fcs on* e *sudo ethtool -K <interface> rx-all on* (onde “interface” corresponde ao nome da interface desejada, visto através do comando *ifconfig*). Com estas duas linhas de comando executadas em um terminal, é possível utilizar *softwares* como *Wireshark* ou *tcpdump* para visualizar o campo FCS dos pacotes, mesmo que estes não estejam corretos. A Figura 7.27 mostra um exemplo de pacote exibido pelo *Wireshark* onde o campo FCS é enviado às camadas superiores.

Figura 7.27 – Campo FCS capturado pelo *Wireshark*

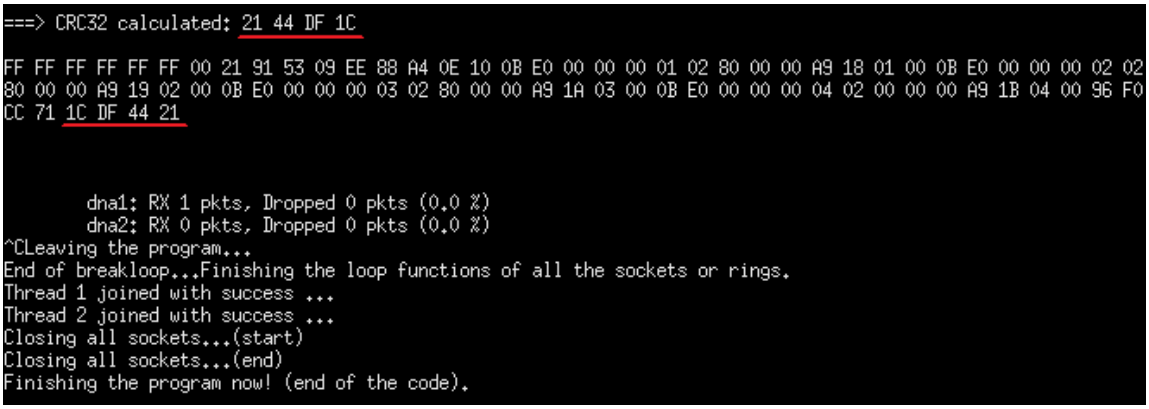


Fonte: Próprio autor

Porém, quando estes mesmos comandos são aplicados às interfaces *DNA* quando o PF_RING é carregado ao *kernel* do *Linux*, o funcionamento do injetor de falhas como um todo é afetado. Ao se executarem os comandos *sudo ethtool -K dna1 rx-fcs on* e *sudo ethtool -K dna1 rx-all on* ou *sudo ethtool -K dna2 rx-fcs on* e *sudo ethtool -K dna2 rx-all on*, ainda é possível receber o campo de FCS dos pacotes e manipulá-los, mas o próprio PF_RING se mostrou incapaz de enviar os pacotes de uma interface para outra.

Para se ter certeza que os pacotes recebidos no injetor continham o campo de FCS, foi desenvolvida uma função para calcular e exibir o CRC32 dos pacotes. De posse desta função, basta comparar os quatro últimos *bytes* do pacote com o resultado do cálculo de CRC. Como mostrado na Figura 7.28, ao se utilizar os argumentos *rx-fcs on* e *rx-all on* para o *ethtool*, é possível notar que os quatro últimos *bytes* correspondem ao valor de CRC32 calculado.

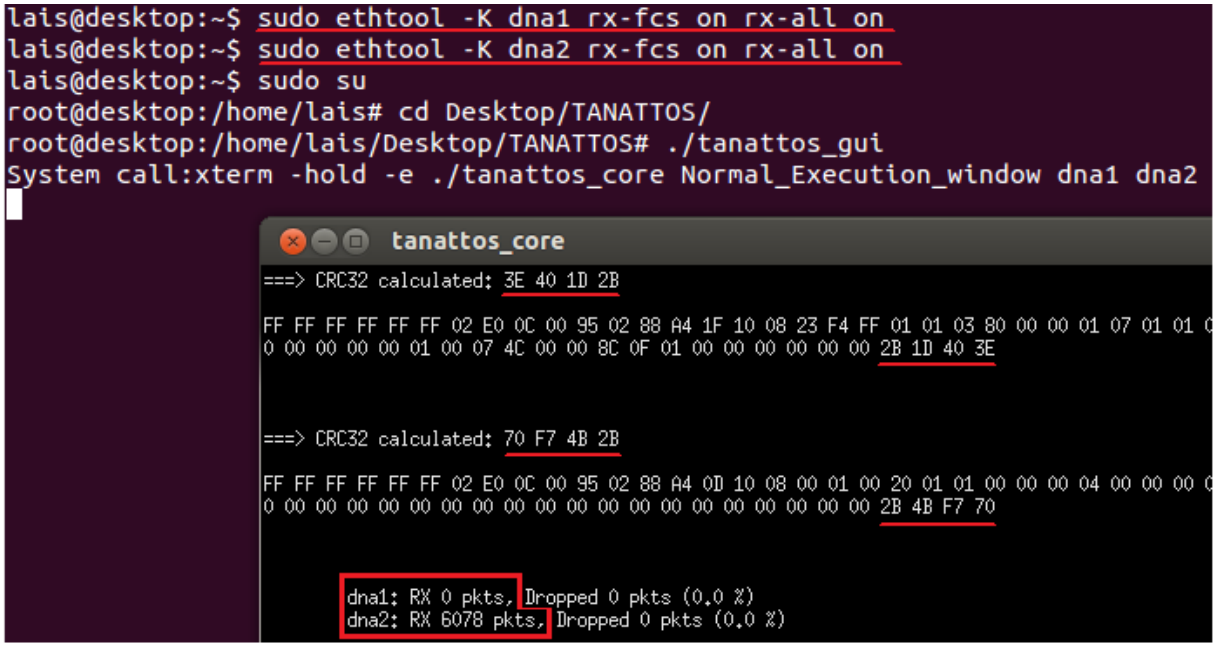
Figura 7.28 – Campo FCS capturado pelo injetor de falhas e CRC calculado



Fonte: Próprio autor

Ao se tentar executar o teste de execução normal, por exemplo, nota-se que os pacotes não puderam ser trocados entre as interfaces de rede. A Figura 7.29 mostra o resultado de uma execução de aproximadamente 30 segundos onde não houve comunicação bi-direcional.

Figura 7.29 – Execução normal do injetor de falhas ao capturar o campo FCS dos pacotes



Fonte: Próprio autor

Por conta disto, os testes do injetor TANATTOS tiveram de ser interrompidos neste ponto e os dispositivos não puderam ser testados frente às funções de corrupção de pacotes, corrupção de dados de comandos *LWR* e *LRD*, corrupção do *Circulating Bit* e corrupção do *Working Counter*. No momento busca-se uma solução para que seja possível realizar a troca de pacotes entre as interfaces quando houver um erro de FCS.

8. CONCLUSÃO

No decorrer deste trabalho foram apresentados conceitos relativos ao protocolo de comunicação EtherCAT e injeção de falhas neste protocolo. Primeiramente foi visto que EtherCAT é um protocolo de comunicação industrial baseado no tradicional protocolo Ethernet, sendo bastante usado em áreas relacionadas à automação industrial. Tal protocolo possui como componentes principais um mestre e vários escravos conectados trocando mensagens. O mestre pode ser implementado em qualquer computador com uma interface de rede, enquanto para um escravo recomenda-se um *hardware* específico (FPGA ou ASIC). Por não haver uma implementação base pré-certificada, para um novo dispositivo a ser desenvolvido, uma nova implementação deve ser feita e esta deve ser validada pelo órgão responsável. Neste contexto, uma ferramenta que seja capaz de testar estas implementações perante falhas durante seu processo de desenvolvimento se torna interessante.

Um dos objetivos deste trabalho foi analisar a viabilidade de se adaptar o injetor de falhas FITT, desenvolvido para o protocolo PROFIsafe, de modo que este possa ser utilizado para se injetar falhas no protocolo EtherCAT. Outro objetivo seria utilizar os resultados obtidos deste processo de adaptação para criar injetor de falhas próprio para o protocolo EtherCAT.

De posse de dispositivos mestre e escravo EtherCAT cedidos pela empresa parceira no projeto, as modificações necessárias para se adaptar FITT para o fluxo de dados EtherCAT foram feitas e renderam bons resultados, já que a partir de mudanças mínimas no código fonte, FITT foi capaz de reconhecer um tráfego de pacotes do protocolo EtherCAT. Com base nestes resultados positivos, partiu-se para o segundo objetivo, que seria criar um injetor de falhas próprio para EtherCAT, tendo como base alguns aspectos de FITT.

Durante a fase de desenvolvimento deste novo injetor de falhas, chamado de TANATTOS, foram criadas funções para injetar falhas nos pacotes sendo transmitidos entre dois dispositivos, de modo a perturbar a comunicação entre eles. Tais funções foram criadas de acordo com os mecanismos de detecção e correção de erros descritos na especificação do protocolo EtherCAT. Durante esta fase, constatou-se que para testar o injetor o uso dos dispositivos cedidos impôs uma dificuldade, pois estes trocam muitas mensagens por segundo e avaliar o resultado da injeção de falhas em um único pacote através do *software Wireshark* se tornou uma tarefa complexa devido ao pacote com falhas se perder muito facilmente na listagem exibida com os demais pacotes em trânsito. Para isto, foi desenvolvida outra aplicação que envia um modelo simples de pacotes EtherCAT a uma cadência e quantidade menores, para que se possa verificar mais facilmente o resultado da injeção de falhas.

O desenvolvimento desta nova aplicação forneceu subsídios para entender melhor a estrutura dos pacotes EtherCAT a nível de *bytes*. De posse do *log* gerado pelo *Wireshark* após uma execução normal dos dispositivos comunicando-se entre si, a aplicação foi desenvolvida para imitar um pacote EtherCAT com todos os campos observados via *Wireshark.* Assim, a tarefa de alterar dados específicos nos pacotes pelas funções de injeção de falhas se tornou mais fácil, pois já se conhecia a posição exata de *bytes* específicos dentro do pacote.

Após esta etapa, foram conduzidos os primeiros experimentos com os dispositivos cedidos, aplicando as funções de atraso e perda de pacotes e a partir de uma aplicação que monitorava o dispositivo mestre, foi constatado que a comunicação conseguiu ser perturbada e *flags* de erro foram acionadas, o que mostra que como prova de conceito, os objetivos de testar a adaptabilidade de FITT e a criação de um injetor de falhas para EtherCAT foram alcançados.

A maior dificuldade encontrada foi para testar as funções de injeção de falhas que se baseam em alteração do conteúdo do pacote e consequentemente causam alteração no campo FCS do pacote. Para que o pacote não seja descartado pelas interfaces de rede por conta de um erro de CRC do pacote, é necessário que duas condições sejam satisfeitas: (1) o campo FCS dos pacotes deve ser mantido pelas interfaces de rede e enviado às camadas superiores para posterior manipulação e (2) as interfaces não devem descartar pacotes com o campo de FCS inválido. Com o utilitário *ethtool* presente no *Linux*, é possível fazer isto, porém, quando a biblioteca PF_RING (utilizada pelo injetor para captura dos pacotes) é carregada no *kernel* do *Lilnux*, as interfaces de rede não são capazes de trocar pacotes entre si quando as condições (1) e (2) são habilitadas pelo *ethtool*, possivelmente devido a um conflito interno causado no próprio PF_RING. Por conta deste fator, algumas funções de injeção de falhas propostas não puderam ser testadas até o presente momento, enquanto se buscam alternativas ou soluções para que pacotes com o campo FCS alterado possam ser enviados e recebidos pelas interfaces de rede sem o risco de descarte.

Como trabalho futuro está a resolução da questão que envolve as funções de injeção de falhas que lidam com corrupção dos pacotes, de modo a encontrar um meio de realizar o envio e recebimento de pacotes após alterar seu conteúdo sem que estes sejam descartados pelas interfaces de rede.

Outra alteração a ser considerada está no modo com que o injetor abre o arquivo de ajuda e o relatório com o resultado das injeções de falha. Atualmente é utilizada a chamada de sistema *system* presente no próprio sistema *Linux*, com um parâmetro que abre o arquivo desejado no editor de texto *gedit*. Porém, para tornar a ferramenta mais portável e não depender de chamadas de sistema, uma alternativa interessante seria abrir os arquivos desejados em uma

nova janela do próprio injetor, de maneira similar com que janelas correspondentes às funções de injeção de falhas são abertas e exibir o conteúdo dos arquivos nestas janelas.

REFERÊNCIAS

ARLAT, J. *et al.* Fault injection for dependability validation: a methodology and some applications. **Ieee transactions on software engineering**, 1990. v. 16, n. 2, p. 166–182.

AVIŽIENIS, A. *et al.* Basic concepts and taxonomy of dependable and secure computing. **Ieee transactions on dependable and secure computing**, 2004. v. 1, n. 1, p. 11–33.

Bestorm® software security testing tool. [S.l.], 2016. Disponível em: <http://www.beyondsecurity.com/bestorm.html>. Acesso em: 26 out. 2016.

CLARK, J. A.; PRADHAN, D. K. Fault injection: a method for validating computer-system dependability. **Computer**, 1995. v. 28, n. 6, p. 47–56.

DOBLER, R. J. **Fitt: uma ferramenta de injeção de falhas para validar protocolos de comunicação seguros**. Brasil: Universidade Federal do Rio Grande do Sul, 2016. Dissertação (Mestrado).

DOBLER, R. J. *et al.* A software fault injector to validate implementations of a safety communication protocol. *In*: LATIN-AMERICAN SYMPOSIUM ON DEPENDABLE COMPUTING, 2016, Colômbia. **Anais**... Colômbia: [s.n.], 2016. p. 34–43.

DREBES, R. J. **Firmament: um módulo de injeção de falhas de comunicação para linux**. Brasil: Universidade Federal do Rio Grande do Sul, 2005. Dissertação (Mestrado).

DURANTE, L.; VALENZANO, A. On the performance of the iec 61158 fieldbus. **Computer standards & interfaces**, 1999. n. 21, p. 241–250.

Ec-lyser - ethercat diagnosis tool. [S.l.], [s.d.]. Disponível em: <http://www.acontis.com/eng/products/ethercat/ec-lyser/index.php>. Acesso em: 26 out. 2016.

Ec-sta [ethercat slave test application]. [S.l.], [s.d.]. Disponível em: <http://www.acontis.com/eng/products/ethercat/ec-sta/index.php>. Acesso em: 26 out. 2016.

Elmo motion control ethercat network diagnostics. [S.l.], 2016. Disponível em: <http://www.elmomc.com/capabilities/3%20.GMAS%20EtherCAT%20Field%20Bus%20Communication/1.EtherCAT%20Network%20Diagnostics/EtherCAT-Network-Diagnosticsl.html#page=page-1>. Acesso em: 26 out. 2016.

ETHERCAT TECHNOLOGY GROUP. **Ethercat slave implementation guide**. EtherCAT Technology Group.

ETHERCAT TECHNOLOGY GROUP. **Ethercat specification – part 1**. EtherCAT Technology Group.

ETHERCAT TECHNOLOGY GROUP. **Ethercat specification – part 2**. EtherCAT Technology Group.

ETHERCAT TECHNOLOGY GROUP. **Ethercat specification – part 4**. EtherCAT Technology Group.

ETHERCAT TECHNOLOGY GROUP. **Ethercat specification – part 5**. EtherCAT Technology Group.

ETHERCAT TECHNOLOGY GROUP. **Ethercat specification – part 6**. EtherCAT Technology Group.

ETHERCAT TECHNOLOGY GROUP. **Ethercat specification – part 3**. EtherCAT Technology Group.

ETHERCAT TECHNOLOGY GROUP. **Ethercat slave controller – hardware data sheet section 1**. Disponível em: <http://fs1.gongyeku.com/data/default/201212a/20121212142659998.pdf>. Acesso em: 18 maio 2016.

ETHERCAT TECHNOLOGY GROUP. **Ethercat – the ethernet fieldbus**. EtherCAT Technology Group. Disponível em: <https://www.ethercat.org/en/downloads.html>.

ETHERCAT TECHNOLOGY GROUP | Ethercat Conformance Test Tool (ctt). [S.l.], 2016. Disponível em: <https://www.ethercat.org/en/ctt.htm>. Acesso em: 27 out. 2016.

ETHERCAT TECHNOLOGY GROUP | Home. [S.l.], 2016. Disponível em: <https://www.ethercat.org/default.htm>. Acesso em: 30 maio 2016.

FELSER, M.; SAUTER, T. The fieldbus war: history or short break between battles? [S.l.]: IEEE, 2002. p. 73–80. Disponível em: <http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=1159702>. Acesso em: 13 out. 2016.

FREE SOFTWARE FOUNATION. GSL - GNU scientific library. [S.l.], 2016. Disponível em: <https://www.gnu.org/software/gsl/>. Acesso em: 30 nov. 2016.

Glade - a user interface designer. [S.l.], 2016. Disponível em: <https://glade.gnome.org/>. Acesso em: 8 nov. 2016.

GOMES, L. G. *et al.* Injeção de falhas de comunicação sobre implementações do protocolo ethercat. **14ª escola regional de redes de computadores**, 2016. p. 53–60.

HAN, S.; SHIN, K. G.; ROSENBERG, H. A. Doctor: an integrated software fault injection environment for distributed real-time systems. [S.l.]: IEEE, 1995. p. 204–213. Disponível em: <http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=395831>. Acesso em: 25 out. 2016.

HSUEH, M.-C.; TSAI, T. K.; IYER, R. K. Fault injection techniques and tools. **Computer**, abr. 1997. v. 30, n. 4, p. 75–82.

KANG, C. *et al.* Design of ethercat slave module. *In*: INTERNATIONAL CONFERENCE ON MECHATRONICS AND AUTOMATION, 2011, Beijing. **Anais**... Beijing: IEEE, 2011. p. 1600–1604.

KOREN, I.; KRISHNA, C. M. **Fault-tolerant systems**. Amsterdam ; Boston: Elsevier/Morgan Kaufmann, 2007.

NATELLA, R.; COTRONEO, D.; MADEIRA, H. S. Assessing dependability with software fault injection: a survey. **Acm computing surveys**, 2016. v. 48, n. 3, p. 1–55.

NEUMANN, P. Communication in industrial automation—what is going on? **Control engineering practice**, 2007. v. 15, n. 11, p. 1332–1347.

NTOP. **Pf_ring user's guide**. NTOP.

ORFANUS, D. *et al.* Ethercat-based platform for distributed control in high-performance industrial applications. *In*: IEEE 18TH CONFERENCE ON EMERGING TECHNOLOGIES & FACTORY AUTOMATION, 2013, Cagliari. **Anais eletrônicos**... Cagliari: IEEE, 2013. p. 1–8. Disponível em: <http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=6647972>. Acesso em: 5 maio 2016.

Pf_ring. **Ntop**, [S.l.], 4 ago. 2011. Disponível em: <http://www.ntop.org/products/packet-capture/pf_ring/>. Acesso em: 18 out. 2016.

PRESSMAN, R. **Engenharia de software : uma abordagem profissional**. 7. ed. [S.l.]: McGraw-Hill, 2011.

Profisafe. [S.l.], 2016. Disponível em: <http://www.profibus.com/technology/profisafe>. Acesso em: 30 maio 2016.

PRYTZ, G. A performance analysis of ethercat and profinet irt. [S.l.]: IEEE, 2008. p. 408–415. Disponível em: <http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=4638425>. Acesso em: 11 out. 2016.

SALAH, K.; QAHTAN, A. Boosting throughput of snort nids under linux. [S.l.]: IEEE, 2008. p. 643–647. Disponível em: <http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=4781733>. Acesso em: 18 out. 2016.

SOMMERVILLE, I. **Software engineering**. 9. ed. USA: Pearson, 2011.

YU, Y.; BASTIEN, B.; JOHNSON, B. W. A state of research review on fault injection techniques and a case study. [S.l.]: IEEE, 2005. p. 386–392. Disponível em: <http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=1408393>. Acesso em: 3 out. 2016.

ZHOU, T.; HU, J. Design and realization of ethercat master. *In*: EMEIT 2011 INTERNATIONAL CONFERENCE, 2011, Harbin, Heilongjiang, China. **Anais eletrônicos**... Harbin, Heilongjiang, China: IEEE, 2011. V. 1, p. 173–177. Disponível em: <http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=6022890>. Acesso em: 29 jan. 2016.

ZIADE, H.; AYOUBI, R. A.; VELAZCO, R. A survey on fault injection techniques. **International arab journal of information technology**, 2004. v. 1, n. 2, p. 171–186.

ANEXO A – TRABALHO DE GRADUAÇÃO I

Estudo sobre injeção de falhas de comunicação sobre implementações do protocolo EtherCAT

Luiz Gustavo A. Gomes[1], Taisy S. Weber[1], Sérgio L. Cechin[1]

[1]Instituto de Informática – Universidade Federal do Rio Grande do Sul (UFRGS)
Caixa Postal 15.064 – 91.501-970 – Porto Alegre – RS – Brasil

{lgagomes, taisy, cechin}@inf.ufrgs.br

Abstract. *Ethernet-based protocols are frequently used in different industrial areas. Among these protocols, EtherCAT is a popular used one. For this protocol source codes are available, but only for specific devices, thus for new equipments, such protocol must be specially implemented and submitted to a validation process. This work proposes to adapt an existent fault injector to the EtherCAT protocol, to provide support in this task. Thereby, such implementations will be tested against fault occurrences, helping in the process of validating its dependability.*

Resumo. *Protocolos baseados em Ethernet são usados com frequência em diferentes áreas industriais. Dentre estes protocolos, o EtherCAT é um popularmente usado. Para este protocolo estão disponíveis códigos prontos, porém apenas para dispositivos específicos, assim é necessário que os desenvolvedores façam suas próprias implementações e as submetam para o processo de validação conforme a finalidade. Neste contexto, o trabalho propõe adaptar um injetor de falhas existente para o protocolo EtherCAT, de modo a servir de apoio neste processo. Com isto, tais implementações serão testadas frente à ocorrência de falhas, auxiliando no processo da validação de sua dependabilidade.*

1. Introdução

Após passar por três grandes revoluções: mecânica, via máquinas a vapor, elétrica, através de linhas de montagem para produção em massa e informatizada, com controladores lógicos e sistemas eletrônicos, a indústria passa por uma nova revolução graças aos avanços tecnológicos que deram origem a componentes eletrônicos que aliam baixo consumo a alto desempenho. Tais avanços têm sido chamados de 4ª Revolução Industrial, ou ainda Indústria 4.0 [Kagermann et al. 2013]. Neste cenário, indústrias estão conectando seus dispositivos físicos entre si de modo que estes possam trocar dados de uma maneira organizada, disparar ações e controlar um ao outro [Danielis et al. 2014].

Por conta destes avanços, dispositivos digitais passaram gradativamente a substituir dispositivos analógicos em alguns pontos de plantas industriais de processamento, o que criou a necessidade de novos protocolos de comunicação entre estes dispositivos e seus controladores [Galloway e Hancke 2013], os chamados protocolos *fieldbus,* fazendo com que sistemas de controle digital passassem a se comportar cada vez mais como uma grande rede interconectada (Figura 1). Paralelamente, Ethernet se tornou o padrão dominante em comunicação corporativa e residencial, resultando em componentes baratos devido à sua produção em massa. Somando isto ao fato de que redes industriais se comportam cada vez mais como redes convencionais [Galloway e Hancke 2013], Ethernet tendendo a possuir maior largura de banda do que protocolos *fieldbus* tradicionais e maior variedade de topologias possíveis [Orfanus et al. 2013], surgiu um interesse econômico em introduzir sistemas de comunicação baseadas em Ethernet

no domínio industrial [Neumann 2007]. Portanto, soluções em comunicação foram desenvolvidas especialmente para atender requisitos industriais como tempo e sincronismo.

Uma destas soluções é o protocolo EtherCAT [EtherCAT Technology Group 2016], planejado para dar suporte a processamento de informações, monitorar e controlar sistemas de controle para diversos setores industriais de diferentes domínios, de maneira simples. Ao se utilizar EtherCAT, pode-se adotar uma topologia simples (em linha) ou em árvore, fugindo do risco de se ter um único ponto de falhas presente na topologia estrela tradicional, tudo isto sem o requerimento de componentes ou infraestrutura custosos. Devido a isto, equipamentos com uma interface Ethernet padrão podem fazer parte de uma rede EtherCAT.

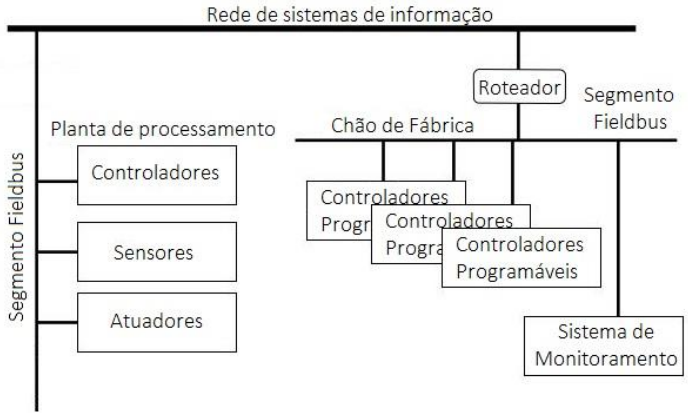


Figura 0.1. Rede fieldbus genérica (Fonte: Adaptado de EtherCAT Technology Group, 2013)

Apesar de existirem implementações prontas, [Open EtherCAT Society 2016] e [IgH EtherCAT Master for Linux 2016], estas são para dispositivos pré-determinados. Como o padrão EtherCAT não define uma implementação padrão pré-certificada [Zhou e Hu 2011] para qualquer dispositivo em geral, uma implementação do protocolo deve ser feita e validada pelo grupo responsável (EtherCAT Technology Group) de acordo com a norma IEC 61158 [Winkel 2006]. Por ser um processo custoso, é desejável que a implementação a ser validada esteja conforme a especificação. Infelizmente não estão disponíveis ferramentas que auxiliem os desenvolvedores a testar seus códigos de forma automatizada frente à ocorrência de falhas previstas na especificação. Diante deste cenário, é objetivo deste trabalho o estudo da viabilidade para o desenvolvimento de um injetor de falhas de comunicação próprio para o protocolo EtherCAT a fim de que implementações do mesmo sejam verificadas diante da ocorrência de falhas.

2. Injeção de falhas

Segundo [Arlat et al. 1990], injeção de falhas pode ser vista como uma técnica para testar a tolerância de um sistema com respeito a uma classe de entradas específicas para este sistema, isto é, as falhas em si. É um processo que complementa e não compete com outras abordagens, como modelagem e simulação.

Entre seus objetivos, pode-se citar o aumento da confiança do sistema no processo de validação [Arlat et al. 1990] e assegurar que este sistema é robusto e confiável perante condições imprevistas [Natella et al. 2016]. Injeção de falhas também pode ser aplicada em várias etapas do processo de desenvolvimento, onde os resultados negativos podem ser usados para melhorias nos procedimentos de teste e mecanismos de tolerância a falhas. Assim sendo, injeção de falhas pode ser considerada uma ferramenta de auxílio no desenvolvimento de um sistema, quando aplicada nos estágios iniciais, em modelos empíricos ou axiomáticos [Arlat et al. 1990].

Injeção de falhas também proporciona um meio de remover as falhas (corrigindo potenciais deficiências na tolerância a falhas do sistema) e prever falhas (avaliação da eficiência dos procedimentos de teste) [Ziade et al. 2004].

Ao se introduzir uma falha em um experimento deste tipo, dois comportamentos podem ocorrer. Primeiro, a falha não é ativada, permanecendo *dormente*. Neste caso, o experimento não produziu nenhum defeito. Segundo, a falha é ativada causando um erro. Neste ponto um erro pode se *propagar* corrompendo outras partes do sistema até que um defeito seja notado; pode permanecer *latente*, isto é, está presente, mas não é sinalizado; ou pode ser *mascarado* ou *corrigido* [Avižienis et al. 2004].

2.1. Tipos de falhas

Um erro pode ser definido como parte do estado global do sistema que pode conduzir a um defeito. Um defeito é um desvio de função desejada em *hardware* ou *software*. Uma falha é a causa de um erro, podendo ser interna ou externa ao sistema [Avižienis et al. 2004]. Falhas podem ocorrer em qualquer estágio do desenvolvimento de um sistema. A maioria das falhas que ocorrem antes da finalização do sistema como um todo são descobertas e eliminadas através de teste. Falhas que não são removidas podem diminuir a dependabilidade de um sistema [Ziade et al. 2004].

Segundo [Ziade et al. 2004] falhas podem ser classificadas de acordo com o ambiente em que ocorrem, *hardware* ou *software*. Falhas em *hardware* podem ser classificadas de acordo com sua duração: *permanentes, transientes e intermitentes*. Falhas *permanentes* são causadas por dano irreversível de componentes internos seja por superaquecimento ou erro de fabricação. Este tipo de falha só pode se corrigida reparando ou substituindo o componente. Falhas *transientes* são ativadas por condições do ambiente, tais como interferência eletromagnética ou radiação. Além de serem mais comuns que falhas permanentes, também são mais difíceis de detectar. Falhas *intermitentes* podem ser causadas por instabilidades no *hardware* e são corrigidas através de um reprojeto ou substituição de componentes.

Falhas em *software* podem ser criadas em qualquer etapa do desenvolvimento, desde a concepção inicial do projeto, passando pela sua especificação e podendo chegar até a codificação. Como exemplos disto, pode-se citar uma implementação incorreta ou ausente que exija uma mudança no projeto, falhas decorrentes do compartilhamento de recursos, validação errada de dados ou assinalamento incorreto de valores [Ziade et al. 2004].

2.2. Técnicas de injeção de falhas

Injeção de falhas já é reconhecida como um método necessário para validar a dependabilidade de um sistema. Portanto, foram desenvolvidas diversas técnicas para se injetar falhas em um protótipo ou modelo de um sistema, podendo ser divididas em cinco categorias: baseadas em *hardware*, *software*, simulação, emulação e híbrida [Ziade et al. 2004].

Injeção de falhas baseadas em *hardware* é feita diretamente no dispositivo físico, perturbando o *hardware* com parâmetros do ambiente (radiação, interferência eletromagnética, etc.), através de alterações na alimentação do *hardware*, ou modificando configurações da pinagem do circuito [Ziade et al. 2004].

Injeção de falhas baseadas em *software*, também conhecida como SWIFI (*Software-Implemented Fault Injection*) [Natella et al. 2016], é uma técnica onde erros que normalmente seriam produzidos por falhas ocorridas no *hardware* são reproduzidos no nível de *software* [Ziade et al. 2004]. Pode ser atingida corrompendo um endereço de memória ou um registrador que seriam afetados no caso de uma falha no *hardware* [Natella et al. 2016].

Injeção de falhas baseadas em simulação consiste em injetar falhas em modelos do sistema, por exemplo, VHDL. Permite uma avaliação inicial da dependabilidade do sistema quando apenas um modelo do mesmo está disponível. É possível utilizar diferentes níveis de abstração neste processo, porém um ambiente coerente deve ser previsto a fim de que haja interoperabilidade entre os níveis e para integrar a validação do processo [Ziade et al. 2004].

Injeção de falhas baseada em emulação é uma alternativa para reduzir tempo gasto com a abordagem via simulação. Utiliza-se principalmente de *Field Programmable Gate Arrays* (FPGAs). Esta técnica permite que se estude o comportamento do circuito no ambiente real da aplicação, levando em consideração interações de tempo real [Ziade et al. 2004].

Injeção de falhas híbrida: Abordagem que combina injeção de falhas baseadas em *software* com monitoramento de *hardware* [Ziade et al. 2004].

De modo geral, técnicas de injeção de falhas podem ser divididas em invasivas, que deixam para trás um rastro durante os testes e não invasivas, que são capazes de mascarar sua presença, de modo que seu único efeito no sistema são as falhas injetadas.

3. Protocolo EtherCAT

EtherCAT [EtherCAT Technology Group 2016] é um protocolo industrial de tempo real baseado em Ethernet. Não possui restrições quanto à topologia: linha, árvore, estrela ou uma combinação destas. Possui um mestre EtherCAT (ECM) e até 65535 escravos EtherCAT (ESC), onde apenas o mestre pode criar e enviar um *frame* e os escravos apenas podem realizar leituras e escritas nos *frames* [EtherCAT Technology Group 2015].

O mestre EtherCAT envia um telegrama que passa por cada nó pertencente à rede. Cada dispositivo escravo lê o dado que lhe é endereçado e insere seus dados enquanto o telegrama está passando pelo dispositivo [EtherCAT Technology Group 2015]. Um *frame* EtherCAT não para enquanto passa por um escravo, apenas sendo levemente desacelerado enquanto a leitura ou escrita são realizadas [Orfanus et al. 2013]. O último escravo EtherCAT a receber o *frame* é responsável por enviar o mesmo no sentido inverso, que será interpretado pelo mestre como um *frame* de resposta [EtherCAT Technology Group 2013a].

EtherCAT encapsula seus dados (telegrama EtherCAT) dentro de um *frame* Ethernet padrão, podendo-se optar por ser encapsulado em um *frame* UDP / IP (Figura 2). É diferenciado de um *frame* convencional através de um identificador (0x88A4) no campo *EtherType* [EtherCAT Technology Group 2015]. Um telegrama EtherCAT é subdividido em um cabeçalho EtherCAT e um ou mais datagramas EtherCAT [EtherCAT Technology Group 2014] que indicam que tipo de acesso o mestre deseja executar (leitura e / ou escrita) e quais escravos devem realizar tal tarefa [EtherCAT Technology Group 2015].

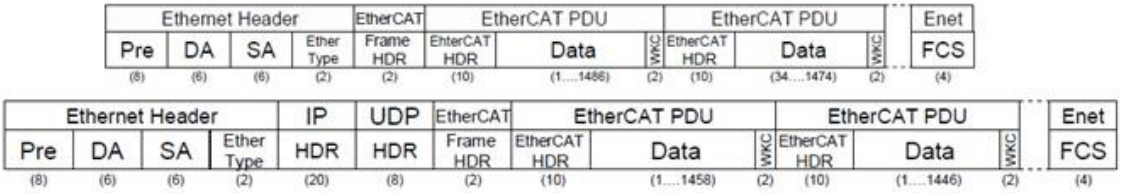


Figura 0.2. Dados EtherCAT encapsulados em um *frame* Ethernet padrão e UDP / IP (Fonte: Adaptado de EtherCAT Technology Group, 2013)

Vários nodos podem ser endereçados individualmente através de um único *frame* Ethernet, carregando diversos *Protocol Data Unit* (PDUs) [EtherCAT Technology Group 2013a] – estrutura utilizada para configurar os escravos e transmitir dados [Orfanus et al. 2013].

EtherCAT conta com dois modos de comunicação entre o mestre e um escravo, modo *buffered* e modo *mailbox* [EtherCAT Technology Group 2014]. No modo *buffered*, mestre e escravo podem acessar o buffer de comunicação do escravo a qualquer momento, com o consumidor dos dados sempre tendo acesso ao conteúdo mais recente vindo do produtor, enquanto o produtor sempre pode atualizar seus dados. Este modo conta com três *buffers* de tamanho igual, sendo um alocado para o produtor (escrita), outro para o consumidor (leitura) e o terceiro que mantém o último dado consistente escrito pelo produtor. Já o modo *mailbox* implementa um mecanismo de *handshake* para lidar com troca de dados. Cada lado, mestre ou escravo, apenas terá acesso ao buffer após o lado oposto ter encerrado seu acesso. Assim, após uma escrita no *buffer*, o acesso à escrita é bloqueado e o *buffer* é liberado para leitura. Somente após o último *byte* ser lido o *buffer* é liberado para escrita novamente.

Outra característica importante a se destacar é o *Clock Distribuído*, modo com que o EtherCAT consegue reduzir o *jitter* do sistema para a casa de 1μs, fazendo com que o relógio de cada escravo seja ajustado conforme um relógio de referência [EtherCAT Technology Group 2015]. O processo de sincronização do Clock Distribuído possui duas fases. Na primeira fase os atrasos entre todos os escravos são medidos. Baseado nestes atrasos, o mestre calcula a compensação de tempo necessária para cada escravo e escreve tal informação em registradores específicos de cada escravo, usando telegramas PDU. Na segunda fase – chamada *Time Control Loop* – o mestre periodicamente envia telegramas de compensação de *drift* – *Drift Compensation Telegram (DCT)* – que ajuda os escravos a detectarem e compensarem seu *drift* local [Orfanus et al. 2013]. Para se tomar o tempo de referência tipicamente se escolhe o primeiro escravo. Na teoria qualquer escravo (ou o próprio mestre) pode ser o escolhido, porém um escravo é escolhido devido ao fato de o mestre possuir um *jitter* de processamento muito maior do que os escravos [Orfanus et al. 2013].

3.1. Modelo de falhas

Erros de comunicação podem ser causados por diversos motivos, desde falhas de *hardware* até interferências do ambiente externo. Esta seção abrange quais tipos de falhas são previstas pela especificação do protocolo [EtherCAT Technology Group 2013a, 2013b, 2013c, 2013d, 2014].

Falhas de perda de pacotes ocorrem quando um ou mais pacotes não conseguem alcançar seu destino. Podem ser causadas, por exemplo, por congestionamento do enlace, fazendo com que o protocolo descarte pacotes. Entre outras causas, pode-se citar mal desempenho de equipamentos físicos (roteadores, *switches*, etc.), mau funcionamento do *software* utilizado, rompimento do meio físico de transmissão, entre outros.

Repetição ou duplicação de pacotes acontece quando mensagens já transmitidas são reenviadas. Um dispositivo defeituoso pode achar que sua mensagem não foi recebida e reenviar.

Falhas por corrupção de dados são ocasionadas quando a mensagem que sai do dispositivo de origem é diferente da mensagem que chega ao dispositivo de destino. Pode ser causada por falhas em elementos físicos do sistema ou por erros de *software*, como operações mal sucedidas de leitura ou escrita.

Erro de sequenciamento de mensagens acontece quando a ordem cronológica dos dados enviados e recebidos é alterada.

Atraso de mensagens é observado quando restrições temporais não são respeitadas, isto é, o tempo esperado para um pacote ir de sua origem até seu destino é maior do que o esperado.

Frame(s) que circula(m) pelo sistema indefinidamente pode acontecer no caso de ruptura do enlace ou outra falha que faça com que uma topologia em linha seja dividida em duas, um *frame* viajando através dos dispositivos pode começar a circular, ou seja, ser transportado apenas entre os nós de um lado da ruptura sem nunca alcançar o último escravo ou o mestre. Tal situação pode ser observada a seguir. Na Figura 3 é exibido o caminho normal que um pacote percorre. Ao ocorrer uma falha de enlace entre os escravos 1 e 2 (Figura 4), estes fecham suas portas 1 e 0 respectivamente e *frames* à direita do escravo 2 seguirão circulando indefinidamente, pois estão impossibilitados de alcançar o nó mestre.

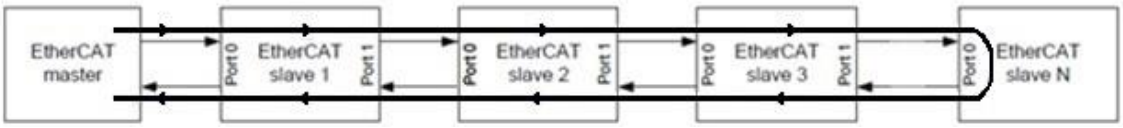


Figura 0.3. Caminho normal percorrido por um pacote (Fonte: Adaptado de EtherCAT Technology Group, 2013)

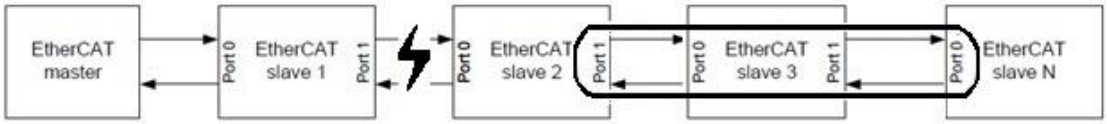


Figura 0.4. Falha de um enlace fazendo com que um pacote circule indefinidamente (Fonte: EtherCAT Technology Group, 2013)

3.2. Mecanismos de detecção / correção de erros do protocolo

Por ser um protocolo focado em velocidade de comunicação, EtherCAT não possui muitos mecanismos de detecção e correção de erros. Porém são capazes de lidar com alguns dos tipos mais comuns de falhas existentes.

A primeira medida é *o Frame Check Sequence* (FCS) presente no *frame* Ethernet. Este é checado tanto pelo mestre quanto pelos escravos para determinar se um *frame* foi recebido corretamente. Como vários escravos podem alterar o *frame* durante seu percurso pela topologia, o FCS é calculado por cada nó escravo na recepção (vindo do mestre) do *frame* e recalculado na retransmissão (indo para o mestre). Se um erro de *checksum* é encontrado, o escravo não repara o FCS, mas sinaliza o mestre incrementando um contador de erros interno, para que o ponto de falha seja precisamente localizado na topologia. É capaz de detectar falhas de perda, repetição e erro de sequenciamento de pacotes [EtherCAT Technology Group 2013c].

O *Working Counter* é o ultimo campo do datagrama EtherCAT. Possui um valor esperado calculado pelo mestre e que é incrementado pelos escravos a cada leitura ou escrita bem sucedida realizada no datagrama. Escravos que apenas estão encaminhando o datagrama, não alteram o *working counter*. Ao comparar o *working counter* com o número esperado de escravos que deveriam acessar dados, o mestre pode saber quantos escravos processaram seus dados correspondentes. Detecta falhas de perda, repetição e erro de sequenciamento de pacotes [EtherCAT Technology Group 2014].

Para uma comunicação utilizando o modo *mailbox*, existe um contador (*counter*), que é incrementado pelos escravos a cada novo serviço de *mailbox*. Tanto o mestre quando os escravos possuem contadores independentes, sendo que os escravos não verificam se a sequência do contador está correta. É checado pelo mestre para detecção de perda de serviços de *mailbox* e pelos escravos para detecção de repetição de um serviço de escrita. Também

detecta falhas de perda, repetição e erro de sequenciamento de pacotes [EtherCAT Technology Group 2013a, 2013b]

Ao se utilizar o modo *Ethernet over EtherCAT* (EoE), onde *frames* Ethernet são transmitidos através do protocolo EtherCAT, armazenam-se os tempos de envio e recebimento de um dado *frame* [EtherCAT Technology Group 2013d]. É utilizado para detectar falhas relacionadas a atraso de mensagens.

Para lidar com *frames* que podem circular indefinidamente pelo sistema, existe o *Circulating Bit*, que é um *bit* que informa aos dispositivos da topologia quando um *frame* está circulando pela topologia na ocorrência de uma falha de enlace que cause uma situação semelhante a da Figura 4. Neste exemplo, os escravos 1 e 2 detectam a falha no enlace e fecham suas portas (porta 1 no escravo 1 e porta 0 no escravo 2). Com isso o *frame* irá circular indefinidamente pelo anel formado pelos escravos 2 a N. Para prevenir isto, um escravo sem um enlace em sua porta 0 fará o seguinte: se o *Circulating Bit* do datagrama EtherCAT é 0, muda seu valor para 1. Se o *Circulating Bit* é 1, não processa o *frame* e o descarta. No caso exemplo, um *frame* circulante dará no máximo uma volta antes de ser detectado e destruído [EtherCAT Technology Group 2014].

4. Proposta

Este trabalho tem como base as ideias apresentadas em [Dobler 2016], onde um injetor de falhas para protocolos seguros baseados na norma IEC 61508, em particular para o protocolo *PROFIsafe* [PROFIsafe 2016], foi desenvolvido. Neste contexto propõe-se um estudo da viabilidade de adaptá-lo para uso no protocolo EtherCAT, bem como a realização de alterações necessárias para isto. De posse de uma ferramenta deste tipo, o processo de validação de uma dada implementação do EtherCAT pelo grupo responsável, terá maiores chances de ser aprovada, visto que já terá sido submetido a uma bateria de testes com falhas previstas pela especificação.

A linguagem de programação a ser utilizada para a implementação do injetor será a linguagem C e a arquitetura do sistema será como mostrada na Figura 5. Nesta arquitetura o injetor atuará entre a comunicação entre um mestre e um escravo, interceptando os pacotes trocados por eles e injetando as falhas nos mesmos. Injeção de falhas em mensagens trocadas no sistema é uma técnica já utilizada para injeção de falhas de comunicação [Natella et al. 2016]. O injetor irá rodar em um computador com sistema operacional Linux, que a fim de interceptar os pacotes em alguma das duas direções disponíveis (mestre para escravo ou escravo para mestre), contará com duas placas de rede.

Para a realização de testes serão utilizados dispositivos de controle cedidos pela empresa parceira no projeto de pesquisa, que já implementam o protocolo EtherCAT. Futuramente planeja-se realizar avaliações de implementações do protocolo feitas pelo grupo de pesquisa do projeto em questão, em especial um Circuito Integrado (CI) que implementa o protocolo EtherCAT que está sendo desenvolvido pelo grupo.

O injetor proposto neste trabalho irá se limitar a injetar apenas alguns tipos de falhas, a serem definidos, servindo como uma prova de conceito para que a implementação de um injetor completo seja realizada no futuro.

Figura 0.5. Arquitetura proposta

5. Cronograma

As atividades, para realização do Trabalho de Graduação II (TG-II), são divididas em quatro etapas principais: implementação, testes e validação, escrita e apresentação.

A etapa de implementação consiste em realizar adaptações e alterações na ferramenta apresentada em (DOBLER, 2016) para que esta possa ser usada para o protocolo EtherCAT. Os testes e a validação serão realizados paralelamente a fim de se verificar se o protótipo desenvolvido atende às funcionalidades pretendidas com o auxílio dos dispositivos cedidos pela empresa parceira. Simultaneamente aos testes e validação será iniciada a escrita da monografia. E por fim preparativos para a apresentação.

Tabela 1. Plano de atividades proposto para o Trabalho de Graduação II

Etapa	**Julho**	**Agosto**	**Setembro**	**Outubro**	**Novembro**	**Dezembro**
Implementação	**X**	**X**	**X**			
Testes e validação			**X**	**X**		
Escrita			**X**	**X**	**X**	
Apresentação						**X**

6. Bibliografia

Arlat, J. et al. (1990). “Fault injection for dependability validation: A methodology and some applications”, *IEEE Transactions on Software Engineering*, v. 16, n. 2, p. 166–182.

Avižienis, A. et al. (2004). “Basic concepts and taxonomy of dependable and secure computing”, *IEEE Transactions on Dependable and Secure Computing*, v. 1, n. 1, p. 11–33.

Danielis, P. et al. (2014). “Survey on real-time communication via Ethernet in industrial automation environments”, Em *Proceedings of the 2014 IEEE Emerging Technology and Factory Automation*. IEEE.

Dobler, R. J. (2016). “FITT: Uma Ferramenta de Injeção de Falhas para Validar Protocolos de Comunicação Seguros”, Brasil, Dissertação (Mestrado), Universidade Federal do Rio Grande do Sul.

EtherCAT Technology Group (2013a). “EtherCAT Specification – Part 4”, EtherCAT Technology Group.

EtherCAT Technology Group (2013b). “EtherCAT Specification – Part 1”, EtherCAT Technology Group.

EtherCAT Technology Group (2013c). “EtherCAT Specification – Part 3”, EtherCAT Technology Group.

EtherCAT Technology Group (2013d). “EtherCAT Specification – Part 6”, EtherCAT Technology Group.

EtherCAT Technology Group (2014). “EtherCAT Slave Controller – Hardware Data Sheet Section 1”.

EtherCAT Technology Group (2015). “EtherCAT – The Ethernet Fieldbus”, EtherCAT Technology Group. Disponível em: <https://www.ethercat.org/en/downloads.html>.

EtherCAT Technology Group | HOME (2016). Disponível em: <https://www.ethercat.org/default.htm>, Acesso em 30 Mai. 2016.

Galloway, B. e Hancke, G. P. (2013). “Introduction to industrial control networks”, *IEEE Communications Surveys & Tutorials*, v. 15, n. 2, p. 860–880.

IgH EtherCAT Master for Linux (2016). Disponível em: <http://etherlab.org/en/ethercat/>, Acesso em 30 Mai. 2016.

Kagermann, H. et al. (2013). “Recommendations for implementing the strategic initiative INDUSTRIE 4.0.”, acatech—National Academy of Science and Engineering. Disponível em: <http://www.acatech.de/fileadmin/user_upload/Baumstruktur_nach_Website/Acatech/root/de/Material_fuer_Sonderseiten/Industrie_4.0/Final_report__Industrie_4.0_accessible.pdf>, Acesso em 13 Mai. 2016.

Natella, R. et al. (2016). “Assessing Dependability with Software Fault Injection: A Survey”, *ACM Computing Surveys*, v. 48, n. 3, p. 1–55.

Neumann, P. (2007). “Communication in industrial automation—What is going on?”, *Control Engineering Practice*, v. 15, n. 11, p. 1332–1347.

Open EtherCAT Society (2016). Disponível em: <http://openethercatsociety.github.io/>, Acesso em 30 Mai. 2016.

Orfanus, D. et al. (2013). “EtherCAT-based platform for distributed control in high-performance industrial applications”, Em *Emerging Technologies & Factory Automation*. IEEE.

PROFIsafe (2016). Disponível em: <http://www.profibus.com/technology/profisafe>, Acesso em 30 Mai. 2016.

Winkel, L. (2006). “Real-Time Ethernet in IEC 61784-2 and IEC 61158 series”, Em *4th IEEE International Conference on Industrial Informatics*.

Zhou, T. e Hu, J. (2011). “Design and realization of EtherCAT master”, Em *Electronic and Mechanical Engineering and Information Technology*. IEEE.

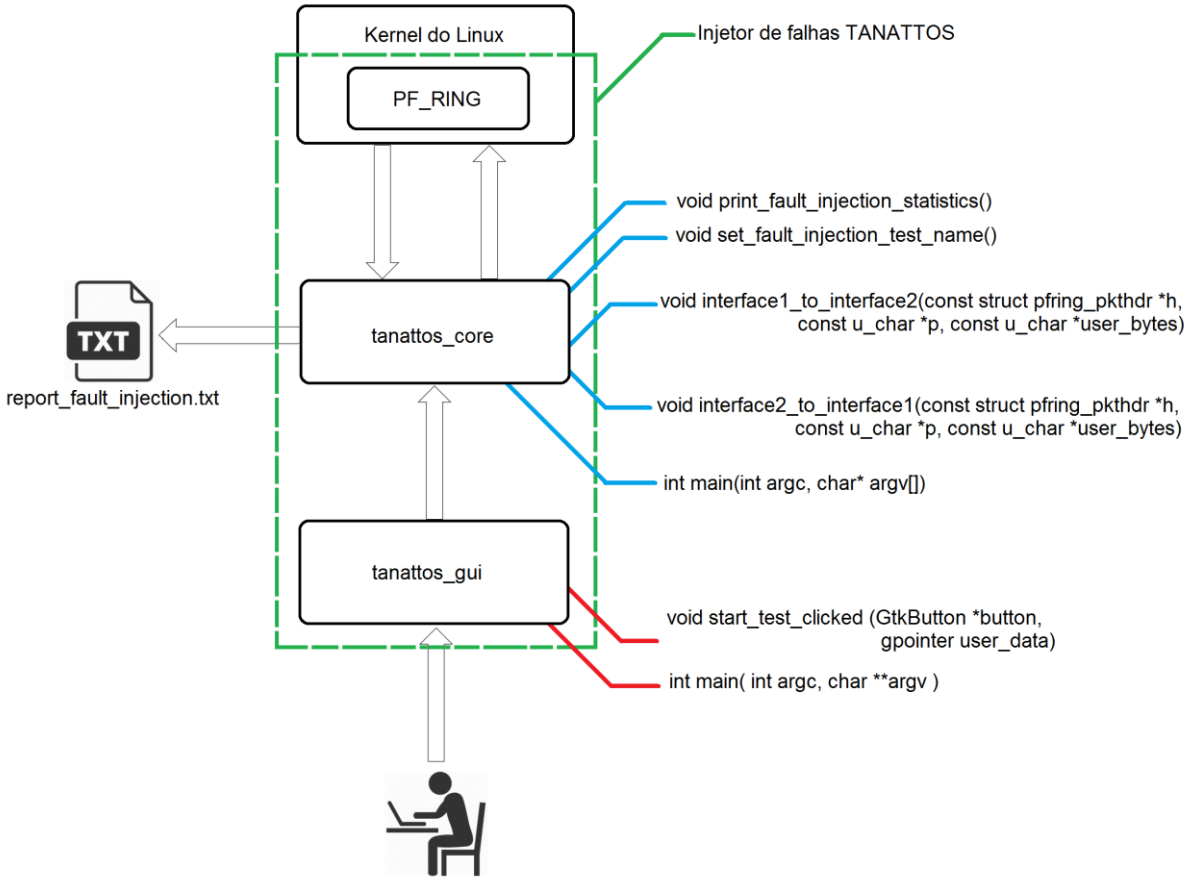
Ziade, H. et al. (2004). “A survey on fault injection techniques”, *International Arab Journal of Information Technology*, v. 1, n. 2, p. 171–186.

ANEXO B – ARQUITETURA INTERNA DO INJETOR DE FALHAS TANATTOS

A seguir é apresentada uma arquitetura completa do injetor de falhas TANATTOS, com as interações entre seus componentes, bem como destacando quais funções devem ser alteradas em cada um destes componentes para adição de novas funções de injeção de falhas.

A Figura B.1 mostra que o injetor TANATTOS é composto de 3 partes separadas: a biblioteca PF_RING carregada no *kernel* do *Linux* para a captura e retransmissão dos pacotes sem intervenção do *kernel*, o módulo *tanattos_core* que concentra as funções de injeção de falhas e a função de geração do *log* pós testes e o módulo *tanattos_gui* que é a interface gráfica.

Figura B.1 – Arquitetura completa do injetor de falhas TANATTOS



Fonte: Próprio autor

Neste cenário, o usuário informa os parâmetros necessários para os testes de injeção de falhas através da interface gráfica que por sua vez inicia envia estes parâmetros para o módulo *tanattos_core*. A partir daí a comunicação se dá apenas entre o PF_RING e o módulo *tanattos_core*, para respectivamente capturar os pacotes e injetar a falha sobre eles. Ao término

dos testes, *tanattos_core* preenche o arquivo de *log "report_fault_injection.txt"* com os dados obtidos durante o teste.

Para adição de novas funções de injeção de falhas, tanto os módulos *tanattos_core* e *tanattos_gui* devem ser modificados. Em *tanattos_core* a função *void print_fault_injection_statistics()* deve ser alterada para que o arquivo de *log* contenha o nome da nova função bem como seus dados mais relevantes levantados durante a execução do teste. A função *void set_fault_injection_test_name()* é modificada para que o programa conheça o nome da nova função e insira-o no arquivo de *log*. As funções *void interface1_to_interface2(...)* e *void interface2_to_interface1(...)* iniciam o processo de injeção de falhas nos pacotes que chegam às interfaces de rede e devem ser alteradas. É nestas funções que se encontra o filtro responsável por separar os pacotes (pelo seu *EtherType* ou outro fator a escolha) e de onde são chamadas as funções de injeção de falhas escolhidas pelo usuário. Na função *int main (int argc, char* argv[])* os dados enviados pela interface gráfica são recebidos tratados e ficam disponíveis para as funções de injeção de falhas.

Já na interface gráfica *tanattos_gui*, a função *void start_test_clicked (GtkButton *button, gpointer user_data)* exibe as janelas correspondentes às funções de injeção de falhas, recebe os parâmetros escolhidos pelo usuário, realiza a consistência destes parâmetros e inicia o módulo *tanattos_core* enviando tais parâmetros. Logo, para uma nova função, é nesta função que os dados recebidos pela sua respectiva janela são recebidos. Na função *int main (int argc, char **argv)* são declaradas todas as janelas gráficas utilizadas pelo programa, sejam janelas das funções de injeção de falhas ou janelas *pop-up* de aviso ao usuário. Então aqui deve ser declarada a janela correspondente à nova função desenvolvida. Além disto, neste módulo é necessária a inserção de uma função própria para abrir a janela gráfica da nova função criada.