Trabalho de Conclusão de Curso

Melhorando a qualidade de código para soluções em Sistemas Embarcados com o Zeta Middleware

Lucas Peixoto de Almeida Cavalcante
lpdac@ic.ufal.br

Orientador:

Me. Rodrigo José Sarmento Peixoto

Maceió, Abril de 2021

Lucas Peixoto de Almeida Cavalcante

Melhorando a qualidade de código para soluções em Sistemas Embarcados com o Zeta Middleware

Monografia apresentada como requisito parcial para obtenção do grau de Bacharel em Engenharia de Computação do Instituto de Computação da Universidade Federal de Alagoas.

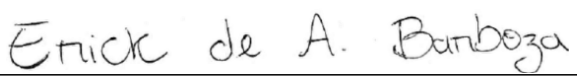
Orientador:

Me. Rodrigo José Sarmento Peixoto

Maceió, Abril de 2021

Monografia apresentada como requisito parcial para obtenção do grau de Bacharel em Engenharia de Computação do Instituto de Computação da Universidade Federal de Alagoas, aprovada pela comissão examinadora que abaixo assina.

Me. Rodrigo José Sarmento Peixoto - Orientador
Instituto de Computação
Universidade Federal de Alagoas



Dr. Erick de Andrade Barboza - Examinador
Instituto de Computação
Universidade Federal de Alagoas

Dr. Baldoino Fonseca dos Santos Neto - Examinador
Instituto de Computação
Universidade Federal de Alagoas

Maceió, Abril de 2021

Agradecimentos

Primeiramente gostaria de agradecer a minha família, meu pai, Rolembergue Mineiro Cavalcante, minha mãe, Maria José Peixoto de Almeida e meus irmãos, por sempre terem me dado forças e estrutura familiar para me desenvolver como pessoa e profissional. Agradeço também a minha namorada, Isabella Padilha, que esteve comigo durante toda a minha graduação me apoiando em todas as minhas decisões.

Agradeço aos meus amigos da faculdade, Alfredo Lima, Igor Theotônio e Marcos Vinicius Costa, que estiveram comigo desde o primeiro período da faculdade. Sem eles tudo teria sido mais difícil e muito menos divertido. Partilhar com eles os momentos dessa fase da minha vida foi essencial para que eu chegasse até onde cheguei, carrego características positivas de cada um deles comigo.

Por fim, agradeço em especial a alguns professores que marcaram minha graduação, que moldaram muitas das minhas decisões e me ajudaram a ser o profissional que sou. São eles: Maria Andrade, minha professora de geometria analítica, Márcio Ribeiro, meu professor de Estrutura de Dados, Erick Barboza, meu professor de Arquitetura de Computadores e Rodrigo Peixoto, meu professor de Sistemas Embarcados, colega de trabalho e amigo.

Lucas Peixoto de Almeida Cavalcante

"You have to believe, if you don't believe you probably will not win."

– Gabriel Toledo

Resumo

Contexto: Sistemas Embarcados (SE) são sistemas computacionais desenvolvidos para propósitos específicos e com limitações de memória e processamento. A Internet das Coisas, por exemplo, vem ajudando a disseminar de forma massiva a quantidade de dispositivos ao redor do mundo. *Firmware* de SE estão crescendo em complexidade e tamanho à medida que a demanda aumenta. Por conta do aumento de complexidade, da minoração do *time-to-market* e por problemas na aplicação de conceitos de engenharia de software, uma grande quantidade de dispositivos do mercado apresentam um custo alto de manutenção de código ao longo do seu ciclo de vida.
Objetivo: A partir de pesquisas do estado da arte, observamos que projetos de SE com diferentes níveis de maturidade apresentam dívida técnica relacionada à arquitetura, código e testes. Portanto, o objetivo é desenvolver uma ferramenta que auxilie novos projetos de SE, a fim de facilitar a implementação e manutenção do firmware, provendo uma arquitetura que facilite testes e reduza a complexidade de código.
Método: Esse trabalho apresenta um *middleware* chamado Zeta que é capaz de fornecer uma arquitetura de software para Sistemas Embarcados, a qual provê em sua essência desacoplamento de tempo, espaço e sincronia. O Zeta conta com uma *Command Line Interface* que tem a função de gerar o código base do Zeta, através da técnica de *Template-Based Code Generation*, e de auxiliar o usuário nas suas customizações.
Resultados: Os resultados sugerem que o Zeta traz um impacto positivo no desenvolvimento, pois, através de um comparativo entre implementações tradicionalmente utilizadas no mercado, a solução desenvolvida com o Zeta teve um desempenho melhor ou igual em todas as métricas que dizem respeito a qualidade de código.
Conclusão: Diante do trabalho desenvolvido e dos resultados obtidos, acreditamos que o Zeta Middleware é capaz de ser utilizado a fim de possibilitar o desenvolvimento de um código com uma qualidade de software maior, o que implica numa velocidade de desenvolvimento e num custo de manutenibilidade menor.

Palavras-chave: IPC, Zeta, Middleware, Desacoplamento, Zephyr, Sistemas Embarcados, Arquitetura, Internet Das Coisas, Publish, Subscriber

Abstract

Context: Embedded Systems (ES) are computational systems designed for specific problems using devices with memory and processing limitations. The Internet of Things (IoT) is helping to spread massively the number of devices around the world. So that, ES firmware is growing in complexity and size as demand increases. Due to the increase in complexity, reduction of time-to-market, and problems related to the application of software engineer concepts, a large number of devices in the market have a high maintenance cost throughout their life cycle.
Objective: From state-of-art research, we observed that ES projects with different maturity levels present technical debt related to architecture, code, and tests. Thus, the objective of this work is to design a tool to assist new ES projects in order to facilitate the firmware implementation and maintenance, providing an architecture that reduces the code complexity and more suitable for tests.
Method: This work presents a middleware named Zeta that provides a software architecture for ES in order to provide time-decoupling, space-decoupling, and synchronization-decoupling. The Zeta has a Command Line Interface responsible to generate the base code using the Template-Based Code Generation technique and assists the user in customizations for your application.
Results: The results suggest that Zeta has a positive impact on the development because, in a comparison between traditional implementations used in the market, the solution developed with Zeta had a performance better or equal in all of the code quality metrics used.
Conclusion: In face of the work developed and their results, we believe that the Zeta Middleware is capable to be used in order to provide a code development with a better software quality, which implies in the speed of development and in a lower maintenance cost.

Keywords: IPC, Zeta, Middleware, Decoupling, Zephyr, Embedded Systems, Architecture, Internet of Things, Publish, Subscriber

Conteúdo

Lista de Figuras

Lista de Códigos

1

Introdução e motivação

Sistemas Embarcados(SE) são sistemas computacionais desenvolvidos para propósitos específicos e com limitações de memória e processamento. Podemos pensar como exemplos desses sistemas no nosso cotidiano as câmeras digitais, celulares, relógios inteligentes, calculadoras digitais, dentre outros. A Internet das Coisas (do inglês, *Internet Of Things* - IoT) [16], *Wireless Sensor Network* [1] e *Cyber-Physical Systems* [24] vem ajudando a disseminar de forma massiva a quantidade de dispositivos ao redor do mundo. A partir desses e de outros estudos, SEs estão crescendo em complexidade e tamanho à medida que mais softwares vem precisando ser desenvolvidos. Para minimizar esses problemas, profissionais e pesquisadores vem se esforçando em lançar diferentes soluções para a etapa de desenvolvimento, como por exemplo novos sistemas operacionais de tempo real (RTOS), *Software Development Kits* (SDK), bibliotecas e plataformas.

Apesar de todos os recursos disponíveis, no contexto de SE ainda existe uma lacuna grande a ser trabalhada no que diz respeito à engenharia de software. Ampatzoglou et al. [2] estabeleceu que projetos de SE com diferentes níveis de maturidade apresentam dívida técnica relacionada à arquitetura, código e testes. O curto período de tempo para o desenvolvimento de projetos, devido ao *time-to-market* que deve ser atendido, é uma das causas para o problema de dívida técnica. Dessa forma, escolhas equivocadas com relação à arquitetura podem afetar o custo de manutenção do projeto ao longo do seu ciclo de vida.

Schoettler [19], definiu um *framework* baseado no modelo *Publish/Subscribe* para SE, com o foco de facilitar o processo de desenvolvimento. Contudo, o usuário deve, manualmente, configurá-lo para ter o *Broker* funcionando corretamente. Essa etapa manual do usuário pode acarretar em erros e problemas que são difíceis de identificar e corrigir. Outro problema nessa solução é a falta de flexibilidade, visto que a implementação é fortemente dependente do uso do FreeRTOS[1]. Outros trabalhos fornecem soluções similares mas com foco em diferentes classes de dispositivos e sistemas operacionais [18, 14]. Marzi et al. [15],

[1] https://www.freertos.org/

descreve um mecanismo de *Inter-Process Communication* (IPC) com aplicabilidade semelhante. Contudo, ele é baseado em mensagens, e esse tipo de estratégia não fornece um desacoplamento de tempo, espaço e sincronia da comunicação entre os processos [8].

Nesse trabalho de conclusão de curso, a proposta foi o desenvolvimento de um middleware chamado Zeta, que fornece uma arquitetura de software de SE baseado em serviços. Essa arquitetura é desenvolvida através de um mecanismo baseado no IPC, chamado de *Inter-Service Communication* (ISC), onde ao invés de processos teremos serviços e esse mecanismo utiliza a técnica de memória compartilhada para trocar informação entre serviços. O mecanismo de ISC implementa um padrão *Publish/Subscribe* que permite um desacoplamento de tempo, espaço e sincronia dos serviços em comunicação. Ao utilizar o Zeta, os desenvolvedores podem gerar toda a arquitetura e código base do ISC baseado num arquivo de entrada que irá descrever como o código deverá ser gerado pela ferramenta. Esse arquivo de entrada permite ao desenvolvedor customizar o código a fim de ter somente as funcionalidades que a aplicação necessita.

Para avaliar o Zeta, especificamos uma aplicação a ser desenvolvida e realizamos uma comparação entre três implementações diferentes, sendo uma delas utilizando o Zeta. Através desses códigos desenvolvidos, comparamos as implementações com base em algumas métricas como complexidade de código, através do NPATH [17], *Cyclomatic* [10], e *Source Lines Of Code* (SLOC). Calculamos também a coesão de código através da contagem do número de funcionalidades da aplicação que são distribuídas ao longo de cada módulo do código. Além disso, também avaliamos o tamanho do footprint, tanto de memória RAM como FLASH, em cada uma das implementações, tendo como base uma placa ARM.

Diante disso, os resultados são encorajadores. Visto que a implementação baseada no uso do Zeta apresentou o menor tamanho de código (menor SLOC) e menor complexidade (menor NPATH e *Complexity*) dentre as três implementações, além de ter uma maior coesão. Entretanto, ele apresentou um tamanho de memória ligeiramente maior(3.33%) comparado às outras implementações. Os resultados indicam que o Zeta fornece uma alternativa de escolha de arquitetura de código, visto que ele possibilita desacoplamento entre serviços, além de uma ferramenta de geração de código do middleware customizada baseado num arquivo de entrada de configuração.

Os capítulos e seções seguintes tem por objetivo explicar em detalhes o desenvolvimento do middleware. Antes de nos aprofundarmos, teremos no capítulo 2 a fundamentação teórica que nos dará a base para o entendimento do Zeta. Em seguida teremos no capítulo os 3 trabalhos correlatos que nos ajudaram a visualizar a lacuna na literatura que faltava ser desenvolvida e implementada. No capítulo 4 apresentaremos o Zeta enquanto middleware, todos os seus componentes de um ponto de vista teórico, como foi realizada a sua implementação, o que o usuário deve fazer para utilizar o Zeta e, por fim, apresentaremos como arquitetamos nossa avaliação experimental. Os resultados e discussões da avaliação experimental serão mostrados no capítulo 5. Por fim, o capítulo 6 apresentamos

tudo que podemos concluir do impacto do Zeta Middleware no contexto de SE e falaremos um pouco sobre trabalhos futuros para melhorar ainda mais a ferramenta.

2

Fundamentação Teórica

Esse capítulo será dividido em seções, onde cada seção irá fundamentar conceitos necessários ao pleno entendimento do trabalho desenvolvido.

2.1 RTOS

De acordo com Skøien [21], os sistemas operacionais de tempo real ou RTOS, sigla em inglês para *Real Time Operating System*, são uma porção de código responsável por gerenciar de forma eficiente a unidade central de processamento (CPU) de um dispositivo. Quando, no contexto de SE, utilizamos o termo "tempo real", não estamos nos referindo a algo ser necessariamente rápido, estamos nos referindo a algo ser preciso, ou seja, as ações devem ser tomadas no momento correto, sem atrasos. Isso é uma característica relevante em SE pois, comumente, os sistemas tem o fator tempo como uma característica crítica.

Uma dúvida comum que ocorre é se perguntar porque não utilizar os sistemas operacionais normais ou também chamados de *General Purpose Operating System* (GPOS) ao invés dos RTOS. A resposta passa por duas questões, onde a primeira é relacionado ao propósito dos GPOS e a segunda tem relação com a pequena capacidade dos dispositivos utilizados em SE, visto que GPOS foram feitos para serem executados em CPUs e em SE utilizamos MCUs. Sobre o proposito dos GPOS, temos que os GPOS são utilizados para sistemas e aplicações que não possuem o tempo como uma característica crítica. Por exemplo, a operação de sacar dinheiro num caixa eletrônico é uma operação crítica com relação ao tempo, visto que o normal é o usuário esperar poucos segundos para ter acesso ao seu dinheiro, não sendo concebível o usuário esperar períodos na ordem de dezenas de minutos ou até horas. Em cenários desse tipo os GPOS não são adequados. Já em situações de por exemplo salvar um documento de texto num computador, essa é uma situação onde o tempo não é crítico e a depender das tarefas que estão sendo executadas em paralelo, essa operação

pode demorar. Logo, como em SE o fator tempo é uma característica crítica, os GPOS não são adequados para esse cenário.

2.2 Zephyr RTOS

Bhartiya [5] enuncia o Zephyr RTOS como um sistema operacional de tempo real escalável, que suporta múltiplas arquiteturas de hardware, otimizado para funcionar em dispositivos com recursos limitados e desenvolvido com preocupação em segurança. O Zephyr é uma iniciativa da *The Linux Foundation* e um dos seus pontos positivos é que ele possui o código aberto e tem uma comunidade bastante ativa, fazendo com que desenvolvedores possam trocar experiências.

Em comparação com outros sistemas operacionais de tempo real do mercado, tal como o FreeRTOS[1], o Zephyr provê mais do que somente funcionalidades e objetos do *kernel*. A ideia por trás do Zephyr é prover toda uma infraestrutura para utilização de periféricos e interfaces de comunicação. Por exemplo, suponha que você possua tipos diferentes de sensores, o Zephyr faz uso de uma biblioteca de sensores onde através da implementação do usuário é possível acessar diferentes sensores através de uma mesma API, tornando o código mais legível e organizado, facilitando na manutenção.

Uma das características que chama a atenção no uso do Zephyr é que o seu código é desenvolvido de tal forma que sirva para qualquer plataforma. Todas as estruturas do *kernel* são disponibilizadas através de uma API, que a depender do hardware que se esteja desenvolvendo, será compilado um código diferente para executar naquele dispositivo. Isso dá a possibilidade de um mesmo código conseguir executar em plataformas diferentes, sendo necessário fazer apenas pequenas alterações.

2.3 Dívida Técnica

O termo dívida técnica foi estabelecido pelo desenvolvedor de software Ward Cunningham [3]. O termo descreve a situação em que o desenvolvedor ou o time de desenvolvimento se vê com a necessidade de desenvolver determinada funcionalidade e entrega-la para o cliente. Contudo, essa mesma funcionalidade precisará ser refatorada num momento futuro.

Toda vez que uma implementação de código gera um potencial trabalho relacionado a melhorias, modificações ou até correções de *bugs*, temos que a dívida técnica do projeto está aumentando. As consequências de se ter uma dívida técnica grande implica em custos que poderiam ser evitados se o desenvolvimento fosse realizado com um pouco mais de planejamento.

[1] https://www.freertos.org/

As causas para esse tipo de problema são diversas, algumas delas são: atender o *time-to-market*, entendimento insuficiente do problema nas etapas iniciais do projeto, falta de processos bem definidos de desenvolvimento, falta de testes que detectem problemas com antecedência, dentre outras causas.

2.4 Inter-Process Communication

Com relação aos tipos de processos existentes, temos dois tipos: independentes e cooperativos. Os processos independentes são auto suficientes e não precisam se comunicar com outros processos para executar sua função, bem como não são afetados pela presença de outros. Já os cooperativos, com o objetivo de atingir determinado fim, normalmente estão presentes dentro de um contexto de comunicação com outros processos. Os processos cooperativos são aconselháveis em alguns cenários, pois, são capazes de modularizar e aumentar a velocidade de computação de uma determinada tarefa. Para que essa comunicação possa ocorrer, existem dois métodos distintos para a troca de informações: memória compartilhada e passagem de mensagem.

O método de memória compartilhada é utilizado quando uma região de memória do dispositivo será reservada para que dois ou mais processos distintos possam utilizá-la para trocar informações. No caso de um *producer* e um *consumer*, geralmente o *producer* irá escrever mensagens até o momento em que o tamanho da região tenha sido atingido. A partir desse momento, ele irá esperar até que o *consumer* leia as mensagens pendentes. Já o consumer irá sempre identificar se existem novas mensagens na região e, em caso contrário, irá começar a fazer o processo de leitura. Esse método começa a ficar mais complexo quando temos mais de um producer para a mesma região. Nesse caso é necessário o tratamento do problema de concorrência de dados.

O segundo método é o de passagem de mensagem. Nesse método temos uma estrutura de dados ou um protocolo sendo utilizado para que uma mensagem seja transmitida de um *consumer* para um *producer*. Um sistema bastante utilizado nesse cenário é o método de *publish/subscribe*. Nesse tipo de situação temos sempre um *publisher* publicando uma informação em algum tipo de estrutura de dados e temos os *subscribers* se registrando em quais informações eles tem o interesse de obter. Dessa forma, toda vez que um *publisher* publica uma mensagem, o *subscriber* recebe um evento de que existe uma nova mensagem e automaticamente está apto para realizar a leitura.

Portanto, o *Inter-Process Communication* (IPC) é um mecanismo implementado pelos sistemas para permitir que processos possam se comunicar, sendo capaz de prover todo um ecossistema que propicia essa interação. Fica a critério dos desenvolvedores quais serão os métodos utilizados de comunicação e quais outros elementos existirão para auxiliar nessa comunicação.

2.5 Desacoplamento de tempo, espaço e sincronia

Acoplamento é um termo utilizado na computação para se referir a, pelo menos, duas partes diferentes do código que tenham algum tipo de relacionamento e, portanto, estão acopladas entre si. Quando mais de uma parte do código tem o conhecimento de outra parte, mais acoplado o código está, visto que um código está sendo implementado de acordo com características de uma outra parte do código. Imagine o cenário onde uma entidade de código A realiza processamentos a partir de métodos (de forma explícita) de uma entidade B. Uma possível alteração na entidade B comprometerá o funcionamento da entidade A. Nesse caso, a ideia por trás do desacoplamento tem como objetivo manter o acoplamento entre duas entidades do código o mínimo possível, de preferência tendo um código onde entidades não se preocupem com a existência de outras entidades.

Para o entendimento dos nossos experimentos e resultados, vamos falar sobre três tipos de desacoplamento, sendo eles: tempo, espaço e sincronia. Por exemplo, quando temos um código orientado a objetos e esses objetos possuem atributos e métodos que utilizam ou precisam diretamente de outros objetos, costumamos dizer que essa implementação está altamente acoplada por espaço, visto que a implementação de um objeto está diretamente relacionada com outras partes do código. Essa situação é a mesma apresentada no parágrafo anterior. Portanto, desacoplamento de espaço tem o objetivo de resolver esse problema. Por exemplo, caso uma entidade X seja desenvolvida de tal modo que para se comunicar com uma entidade Y ela faça uso de uma determinada interface, temos que o funcionamento da entidade X não está diretamente ligado com a implementação da entidade Y, o que faz com que uma entidade X consiga trocar informações com uma entidade Y sem saber necessariamente o funcionamento da entidade Y. A única regra é que as duas entidades tenham sua implementação baseada no uso da interface.

Já para entender o desacoplamento de tempo e sincronia, imagine uma fila e nessa fila há um publicador e um consumidor. Nesse cenário de troca de mensagens através de uma fila, pode existir um acoplamento de tempo, que significa que as duas partes precisam estar disponíveis simultaneamente para se comunicarem, ou seja, precisam ter uma comunicação síncrona. Dessa forma, desacoplamento de tempo significa que é possível ocorrer uma comunicação assíncrona, sem que seja necessário as duas partes estarem disponíveis. Já com relação ao acoplamento de sincronia, existem entidades que podem ficar bloqueadas na espera da troca de mensagens, por exemplo um publicador que ficará esperando, bloqueado, enquanto o consumidor não ler a mensagem. Desacoplamento de sincronia significa que ambas as partes, publicador e consumidor, podem realizar outras atividades enquanto ou não tem nada para publicar (no caso do publicador) ou não tem nada para consumir (no caso do consumidor).

Portanto, o entendimento dos tipos de acoplamento são importantes pois, no con-

texto em que o Zeta Middleware foi projetado, um dos principais problemas foi estudar e avaliar na literatura quais seriam as melhores formas de trocar dados entre entidades diferentes. Em nossos estudos esses três tipos de acoplamento de código foram os mais relevantes e parte da comunicação interna do Zeta foi pensada para resolvê-los.

3

Trabalhos Correlatos

Nesse capítulo vamos ver um pouco dos trabalhos correlatos ao Zeta Middleware. O primeiro deles será o D-BUS, seguido pelo *Embedded System IPC* e o *Automatic Middleware Generation*. Na última seção teremos um resumo a respeito dos trabalhos estudados e como eles motivaram a implementação do Zeta.

3.1 D-BUS

O D-BUS, descrito por Love [14], como um *middleware* que fornece um *Inter-Process-Communication* que visa resolver as necessidades do linux. Esse sistema de barramento substituiu o CORBA pelo sistema de objeto remoto do GNOME. A tecnologia desenvolvida pelo D-BUS não se encaixa num contexto de sistemas operacionais de tempo real devido a sua complexidade e dependência com o Linux. O Zeta possui uma proposta similar ao D-BUS, mas em um contexto de aplicação diferente e menos complexo, bem como é executado num sistema operacional diferente do Linux.

3.2 Embedded Systems IPC

Marzi et al. [15] descreve uma nova abordagem para comunicação entre processos baseado num mecanismo de troca de mensagens adaptado para ser executado num chip ou plataforma específica. Contudo, esse tipo de mecanismo pode levar ao código ter acoplamento de tempo e espaço.

Rajkumar et al. [18] descreve um modelo de *Inter-Process Communication*, para sistemas distribuídos de tempo real que utilizam C++ para habilitar, em tempo de execução, registro de publicações e inscrições. A implementação possui 7000 linhas de código e é executado no LynxOS, baseado em pacotes UDP/IP em um dispositivo com 32MB de memória

RAM, o que se trata de uma abordagem mais alto nível do que pretendemos. O foco do Zeta Middleware são dispositivos com uma quantidade restrita de memória RAM(no máximo 4MB). O código base do Zeta tem cerca de 450 linhas de código em C. Além disso, geralmente, sistemas operacionais de tempo real são desenvolvidos com foco em dispositivos de baixo custo que não suportam o uso de C++ para a implementação de um IPC.

3.3 Automatic Middleware Generation

Zalila et al. [25] investigou a utilização de Geração de Código(GC) para realizar ajustes finos e configurar um middleware com exatamente o código necessário para a execução. Além disso, Gorappa et al. [12], realizou um trabalho similar com foco na redução da memória do footprint do middleware e na melhora da performance ao utilizar GC. Embora com o desenvolvimento do Zeta tenha se levado em conta todos esses mesmos objetivos, o Zeta cobre lacunas que os estudos de Zalila et al. [25], Gorappa et al. [12] não cobrem.

Já Syriani et al. [22] conduziu um estudo de mapeamento sistemático do *Template-Based Code Generation* (TBCG), a fim de dar aos profissionais e pesquisadores um melhor entendimento do TBCG e suas tendências de pesquisa. Baseado nisso, nós podemos estabelecer que essa foi a técnica utilizada para geração de código do Zeta.

Eugster et al. [8] investigou a interação das variantes do paradigma *Publish/Subscribe*, bem como suas semelhanças e divergências. Desse estudo, podemos analisar como desenvolver o Zeta Middleware com base no modelo *Publish/Subscribe*, com o objetivo de fornecer um Inter-Service Communication com desacoplamento de tempo, espaço e sincronia na comunicação entre os serviços.

Schoettler [19] definiu um *framework* utilizando o modelo *Publish/Subscribe* para SE com o objetivo de simplificar o processo de desenvolvimento. Esse trabalho possui similaridades com o nosso trabalho, contudo há significantes distinções. Por exemplo, no trabalho dele existe um *Broker* com um código genérico, o que implica numa perda de flexibilidade e portabilidade. Ele é focado no FreeRTOS e não existem mecanismos ou interesses futuros de portar o trabalho para outros sistemas operacionais de tempo real. Além disso, o usuário deve manualmente criar os seus canais e todos os registros de publicação e inscrição, a fim de ter o *Broker* funcionando corretamente. Devido a esse cenário é possível existir vários erros de código difíceis de identificar e corrigir. Utilizando a técnica de TBCG para gerar o código, é fornecido todo o código relacionado ao Zeta Middleware e o usuário tem o papel de se preocupar somente com o código da aplicação. Além disso, o Zeta Middleware fornece os templates necessários para a plataforma alvo, de modo que desenvolvedores podem portar o Zeta para outros RTOS que não sejam o Zephyr, bastando desenvolver os templates necessários. Por fim, o Zeta fornece uma ferramenta de geração do código do middleware, ganhando flexibilidade suficiente para evitar desperdício de memória e para adicionar novas

features.

Ampatzoglou et al. [2] descreveu um estudo de caso industrial relacionado a percepção da dívida técnica no contexto de sistemas embarcados. O estudo foi conduzido a múltiplos estudos de caso, especificamente para projetos de 5 países diferentes, com períodos de duração de curtos para longos, bem como com empresas pequenas até grandes. Os resultados indicaram que uma das maiores causas do problema de dívida técnica estão relacionados a arquitetura do sistema, aspecto o qual o Zeta Middleware é responsável e tem como foco. Ampatzoglou et al. [2], também estabeleceu que uma das principais razões para a dívida técnica é o tempo curto de desenvolvimento dos projetos devido a necessidade de levar uma solução o mais rápido possível para o mercado. Pensando nisso, o Zeta fez a escolha de fornecer uma arquitetura que ajuda no desenvolvimento do código, ajudando no tempo de desenvolvimento e diminuindo o custo relacionado aos custos de manutenibilidade.

Taivalsaari and Mikkonen [23], descreveu uma taxonomia de arquiteturas de cliente IoT. São arquiteturas no-OS, RTOS, *language-runtime*, *full-OS*, *app-OS*, *server-OS* e *container-OS*. Cada um desses trabalhos funcionam para uma variedade de aplicações. O Zeta é um middleware aplicável para o contexto de arquiteturas com RTOS, onde um dispositivo tem dezenas de kilobytes de memória RAM, tais como sensores, sistemas de controle e *gateways*.

3.4 Resumo dos trabalhos correlatos

Note que nenhum dos trabalhos estudados resolvem o problema em sua plenitude. Contudo, eles servem para de uma forma conjunta indicar qual seria a solução mais próxima do ideal a ser implementada.

Na seção 3.1, temos um exemplo de um ISC e conseguimos identificar as vantagens de utilizar uma comunicação entre entidades internas do código. Já na seção 3.2 temos novamente um exemplo parecido com o D-BUS, mas dessa vez num contexto mais próximo ao de SE, mas ainda diferente, ainda utilizando plataformas mais robustas dos que as MCUs comumente utilizadas nos problemas do mercado. Tendo isso em vista, identificamos que possivelmente a implementação de um tipo de ISC seria uma boa alternativa, mas só encontramos uma pista de como desenvolver isso através do trabalho da seção 3.3. Nos trabalhos apresentados nessa seção conseguimos identificar o modelo de geração de código TBCG como promissor, devido a possibilidade de ter o Zeta implementado em diferentes RTOS. Além disso, o modelo *Publish/Subscriber* nos pareceu uma interessante alternativa de comunicação entre serviços dentro do sistema. Mas, note que, dos trabalhos estudados se encaixava perfeitamente no nosso cenário de aplicação e cada um dos trabalhos existiam lacunas a serem resolvidas ou corrigidas.

Diante disso, foi identificado a necessidade de desenvolver um middleware, com geração de código automático e com uma arquitetura bem definida que permitisse a implementação de soluções com desacoplamento de código, tempo e sincronia e que fosse versátil a ponto de ser utilizada em diferentes plataformas e em diferentes RTOS.

4

Zeta Middleware

Esse capítulo tem como objetivo detalhar o desenvolvimento do Zeta. O projeto Zeta possui duas partes: a primeira é o firmware implementado(Zeta) para funcionar em harmonia com um RTOS, a segunda é uma CLI(Command Line Interface), meio o qual será possível o usuário utilizar o middleware dentro do seu projeto. A CLI desenvolvida possui o nome de ZetaCLI.

4.1 Zeta

Zeta fornece infraestrutura de código e ferramentas que ajudam no desenvolvimento de firmware para SE. O objetivo do Zeta é proporcionar a possibilidade de desenvolver uma solução sob uma arquitetura bem definida que visa uma melhor qualidade de código, de acordo com a análise de algumas métricas que falaremos melhor na seção 4.5. O Zeta possui duas partes centrais: Serviços e Canais. Através deles ocorre a obtenção, armazenamento e troca de informações ao longo da aplicação e veremos melhor como isso ocorre na seção 4.1.1.

Para alcançar o nosso objetivo, o Zeta fornece um mecanismo IPC [13], que segue o padrão publish/subscribe [8], para a comunicação entre os processos dentro de um RTOS. Note que quando falamos a respeito de um firmware implementado utilizando um RTOS, geralmente temos que a solução está dividida em diferentes tarefas que se comunicam entre si.

Está ilustrado na figura 4.1 uma visão geral do Zeta. Na figura 4.1(a) observa-se a camada de Hardware, que consiste na MCU e seus periféricos como RAM, Flash, entre outros. Já na figura 4.1(b) temos a camada destinada ao RTOS e aos Drivers. Decidimos dividir o RTOS e os Drivers, visto que mesmo que alguns RTOS tenham os Drivers incorporados em seu código, não existe uma regra quanto a isso, logo é possível encontrar um RTOS que use

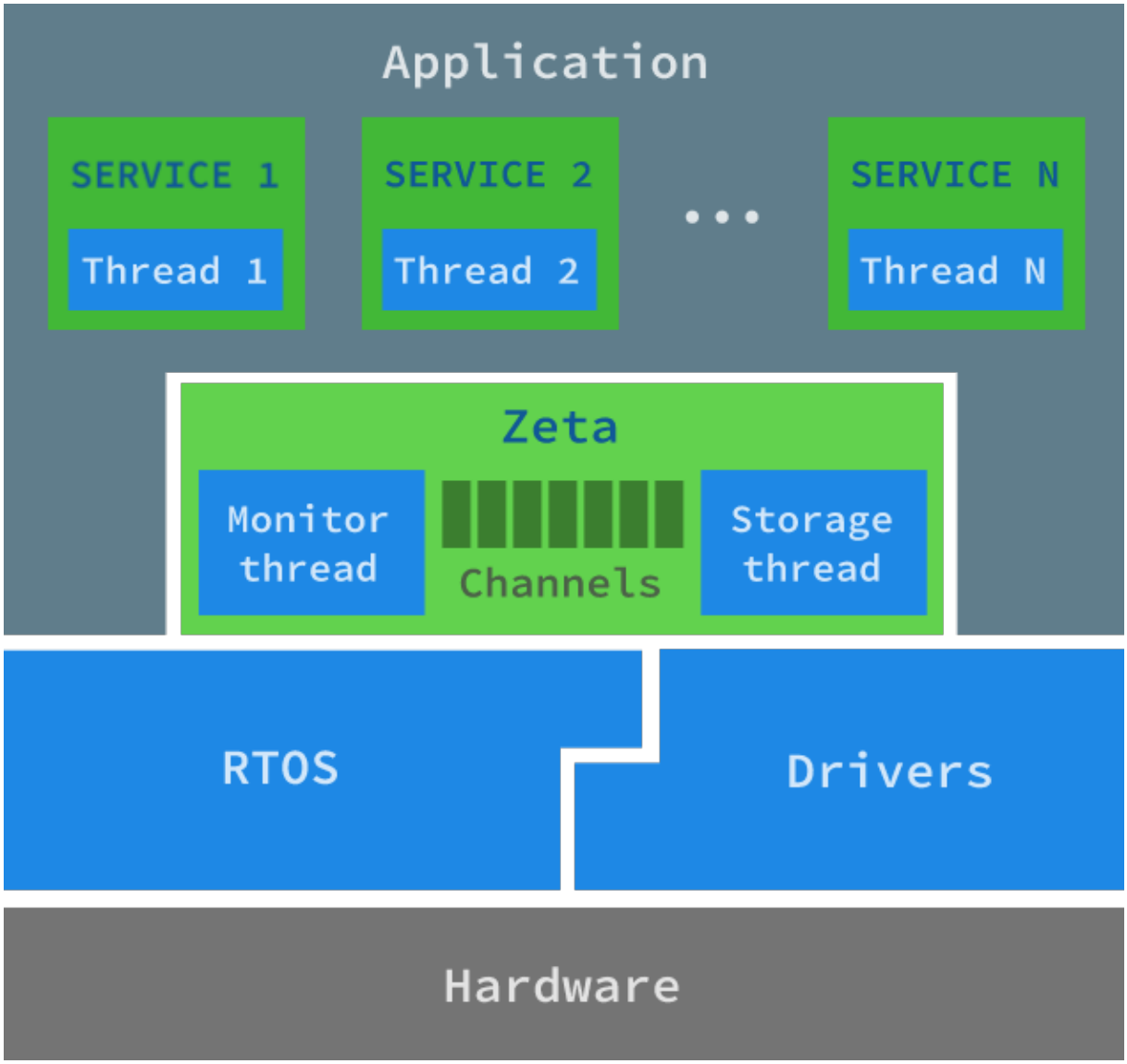


Figura 4.1: Visão geral do Zeta.

Drivers em códigos externos. A camada mais alto nível pode ser visualizada na figura 4.1(c) e se trata da camada de aplicação. Vamos detalhar melhor essa última camada na seção 4.1.1.

4.1.1 Camada de aplicação

Na ilustração da figura 4.1 podemos ver dois blocos: Zeta e Service. No bloco do Zeta temos alguns elementos como `Monitor Thread`, `Storage Thread` e Channel. Já no bloco de Service temos apenas o elemento Thread.

4.1.1.1 Service

O Service no Zeta é um encapsulamento das threads da aplicação. Dessa forma, as threads do RTOS são gerenciadas através do Zeta devido a esse encapsulamento, o que

permite que o Zeta gerencie o seu funcionamento e a sua comunicação com outras partes da aplicação. Dentro do Zeta visualizamos o Service como uma porção de código isolada e independente, que tem como objetivo executar alguma tarefa que corresponde a uma funcionalidade do sistema final a ser implementado.

Em termos de código e programação, denotamos o serviço como uma porção de código que executa sua thread de forma isolada de outras threads que estão executando dentro do RTOS e que possui uma função de callback utilizada pelo middleware.

Note que, é o usuário quem definirá qual a porção de código que será executada dentro de um serviço do Zeta. Sendo assim, é importante, como boas práticas, que o código executado em um serviço não utilize nenhum código externo ao serviço(de outras threads ou serviços) além do código do próprio RTOS ou do Zeta. Em caso contrário, o isolamento do código fornecido pelo Zeta não poderá ser garantido.

Os serviços do Zeta tem as seguintes características: nome, prioridade, tamanho da pilha, função de callback, lista de canais publicados, lista de canais inscritos, função que representa a thread e um comando de inicialização do serviço. O nome é utilizado pelo desenvolvedor durante a implementação, inicialização e uso do serviço. A prioridade indica a prioridade com que o RTOS deve atribuir a thread pertencente ao serviço. O tamanho da pilha indica o tamanho da pilha reservada pelo RTOS para a execução da thread pertencente ao serviço. A função de callback do middleware é a função que deve ser chamada sempre que algum canal em que o serviço está inscrito tenha alguma modificação. A lista de canais publicados, que são os canais com que o serviço possui permissão de escrita, são todos os canais que o serviço poderá alterar o seu valor. A lista de canais inscritos são todos os canais que o serviço deve ser notificado quando o canal sofrer modificações. O Zeta não traz consigo uma limitação no número de serviços que podem existir numa aplicação, essa limitação é baseada no número máximo de threads que o RTOS em questão pode ter e o espaço de memória disponível. A figura 4.2 mostra a execução de threads do RTOS, inicializadas como serviços no Zeta, durante a execução.

4.1.1.2 Channel

Como vimos na seção 4.1.1.1, cada Service possui duas listas de canais, onde através dessas listas os Services serão capazes de escrever algum dado ou serão capazes de receber uma notificação assim que um novo valor for atribuído ao Channel. Entendendo melhor o Channel, temos que ele é uma área de memória compartilhada utilizada para a comunicação entre serviços. Serviços trocam informações entre si através da escrita e leitura de um canal via o método publish/subscribe, fornecendo um ISC(Inter-Service Communication).

Os canais dentro do Zeta tem as seguintes características: ID, região de dados, valor inicial para essa região de dados, flags de leitura, indicador de persistência e um tipo de reação a escritas. O ID tem como objetivo identificar cada canal de forma única. A região

de dados é a região que possui um tamanho fixo, especificado pelo usuário, e que será utilizada pelos serviços para adicionar informações. Todo canal possui flags de leitura que dão informações sobre algumas características do canal, como por exemplo se ele é um canal somente de leitura, ou seja, que não permite operações de publish. O indicador de persistência é utilizado pela `Storage Thread` para salvar ou não as informações de um canal em memória não volátil. O tipo de reação a escritas pode ser com base em atualizações ou em mudanças. Esses dois tipos vão indicar para a `Monitor Thread` como ela deve informar aos Services sobre as alterações em um canal.

4.1.1.3 Monitor Thread

O `Monitor Thread` é uma thread especial do Zeta que executa concorrentemente com a aplicação do usuário implementada nos Services. O objetivo da `Monitor Thread` é gerenciar a comunicação entre Service e Channel.

Para que um serviço reconheça a mudança em algum canal, a `Monitor Thread` do middleware precisa chamar a função de callback de um determinado serviço. Contudo, para que isso ocorra, se faz necessário que esse canal indique para a `Monitor Thread` quando que uma operação de publish pode ser interpretada como uma mudança no canal ou não. Diante disso, cada canal pode reagir com base em mudanças ou atualizações. Quem dirá qual das duas formas será a utilizada é o tipo de reação a escritas presente nas configurações do canal. Quando um canal reage com base em atualizações, o `Monitor Thread` deve chamar o callback de um serviço inscrito no canal independente do valor utilizado na operação de publish. Contudo, quando um canal reage com base em mudanças, o `Monitor Thread` só deve chamar o callback do serviço inscrito se o valor que foi utilizado na operação de publish é um valor diferente do que já havia na região de dados do canal, ou seja, o callback só é chamado quando a região de dados do canal for alterada.

4.1.1.4 Storage Thread

Assim como a `Monitor Thread`, a `Storage Thread` também é uma thread especial do Zeta que executa concorrentemente com a aplicação do usuário implementada nos Services.

Os canais dentro do Zeta podem ser configurados como persistentes, ou seja, devem ter seu conteúdo salvo em memória não volátil com alguma periodicidade configurável. Dessa forma, essa é a thread que é responsável por gerenciar esse armazenamento e efetuá-lo quando necessário

Para ilustrar a interação entre serviços e canais, vamos considerar a configuração entre serviços e canais da figura 4.3. Nessa figura temos dois serviços(1 e 2) e três canais(A, B e C). O serviço 1 está inscrito no canal B e é um publicador do canal A. Já o serviço 2 é um publicador do canal B e está inscrito no canal A. Além disso, temos que o canal B é um canal

persistente, seu conteúdo deve ser salvo em flash periodicamente, os outros dois canais não tem nenhuma particularidade. Diante desse contexto, a figura 4.4 nos fornece um fluxo de operações que pode ocorrer durante o uso do Zeta.

4.1.2 Ilustrando o funcionamento do Zeta

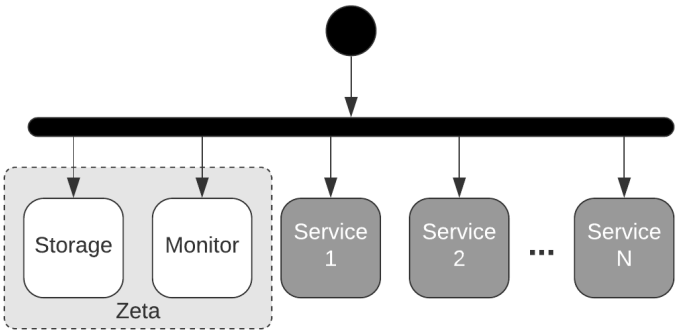


Figura 4.2: Zeta executando em paralelo com threads do RTOS.

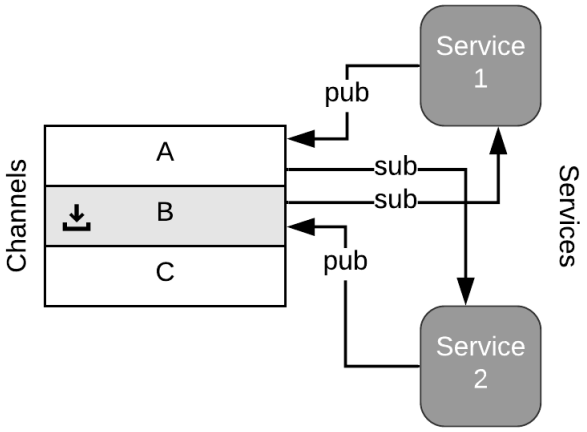


Figura 4.3: Configuração entre serviços e canais.

As threads do Zeta tem um nível de prioridade alta dentro do RTOS, de modo que no início da execução do programa a primeira tarefa a ser executada é a de carregar os dados salvos em flash dos canais persistentes(passo 1) e direcionar esses valores para a `Monitor Thread` atualizar os valores dos canais. Após algum tempo de execução do programa, temos a primeira operação de publish. No passo 3 temos o serviço 1 publicando o valor *val* no canal A. Como o serviço 2 está inscrito no canal A, temos no passo 4 a `Monitor Thread` acionando o callback do serviço 2, que fará com que esse serviço tome conhecimento da mudança no canal A e possa executar alguma rotina de código. Em seguida, temos mais uma operação de publish, mas dessa vez temos uma operação sob um canal persistente. Dessa forma, os passos 5 e 6 são análogos aos passos 3 e 4, mas dessa vez o serviço 2 está publicando no canal B, e a `Monitor Thread` está chamando o callback do serviço 1. A diferença vem

após a operação de publish, periodicamente a `Storage Thread` deve identificar quais os canais persistentes que sofreram alterações em seu valor e salvar essas operações em flash. A verificação dos canais persistentes ocorre no passo 7, no passo 8 temos a `Monitor Thread` indicando o canal que sofreu alteração e o passo final é a `Storage Thread` confirmando que o dado do canal B foi salvo com sucesso. Agora temos uma operação de leitura, note que para um serviço receber a informação em tempo real de que um canal teve alteração ele deve estar inscrito nesse canal. Contudo, para um serviço ler a informação de um canal ele não precisa estar inscrito, a leitura de canais pode ser feita por qualquer serviço do Zeta. No passo 10 temos o serviço 2 lento o dado do canal C e o passo 11 é o retorno desse valor. A última etapa desse fluxo de operações é a tentativa de um serviço escrever num canal que não possui permissão, ou seja, não está publicado para escrever nesse canal. No passo 12 temos o serviço 1 tentando escrever no canal C e recebendo um erro da `Monitor Thread` no passo 13, indicando a falta de permissão. Perceba que toda comunicação de escrita e leitura nunca é feita diretamente entre o serviço e o canal, sempre ocorre via intermédio do `Monitor Thread` e os serviços do sistema executam sem conhecimento dos outros serviços que estão sendo executados paralelamente.

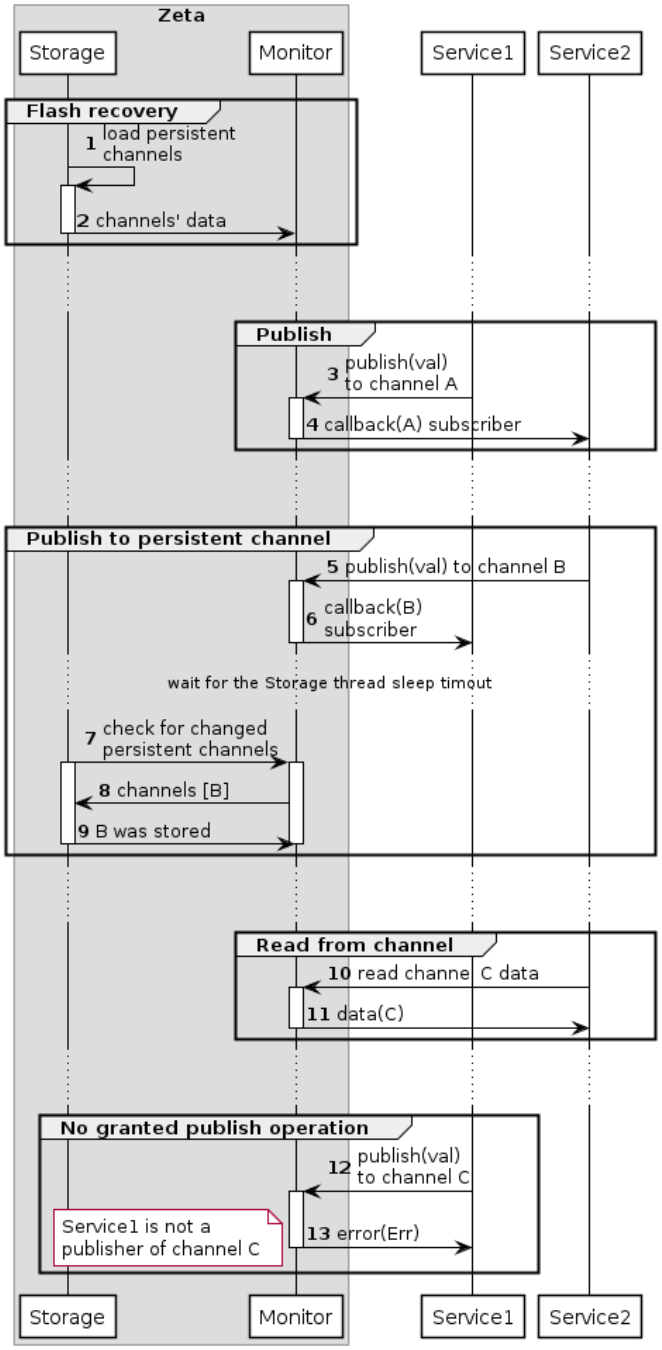


Figura 4.4: Fluxo de operações baseado na configuração entre serviços e canais da figura 4.3

4.2 Zeta Command Line Interface

A fim de fazer com que o Zeta seja utilizado de uma forma mais simples, de modo que ele fique mais flexível, temos o Zeta *Command Line Interface*(ZetaCLI) que dentre suas funções, tem a função principal de gerar o código base do Zeta que irá executar num RTOS de acordo com as customizações de serviços e canais do usuário. Todo código base do Zeta gerado pelo ZetaCLI tem o objetivo de ter o menor footprint possível.

ZetaCLI é um programa desenvolvido em Python que utiliza a técnica Template-Based Code Generation(TBCG) [22]. Portanto, temos uma implementação base e genérica do código Zeta implementado num RTOS específico e através dessa implementação base temos a geração de código com as customizações do usuário. A implementação base do Zeta foi desenvolvida com base no Zephyr RTOS, chamada de Zeta Zephyr Template Database(ZZTD).

O ZetaCLI utiliza um arquivo de entrada no formato YAML, o qual contém informações sobre os serviços, canais e do próprio RTOS que o usuário deseja utilizar em seu projeto. Com base nesse arquivo YAML é gerado todo o código final do middleware através dos templates do ZZTD. Além disso, o ZetaCLI também traz outras funcionalidades, abaixo temos todos os comandos existentes no programa:

- **init**: gera os arquivos de configuração necessários para o Zeta funcionar no RTOS;
- **services**: gera arquivos fonte com uma inicialização básica dos serviços descritos no arquivo YAML do usuário, e altera os arquivos de configuração do RTOS para incluir esses arquivos gerados no processo de compilação;
- **gen**: gera o código do Zeta em si, customizado para conter as informações que o usuário definiu no arquivo YAML;
- **check**: verifica se o Zeta está configurado da forma correta, a fim do processo de compilação não ter erro.

4.3 Utilizando o Zeta no Zephyr

O objetivo dessa seção é mostrar o fluxo do uso do Zeta com o Zephyr RTOS [9]. Iremos identificar os passos necessários para adicionar o Zeta num projeto, realizar as configurações para o middleware funcionar corretamente, desenvolver código, compilar e executar o binário final.

A figura 4.5 ilustra esse fluxo e contém um Template Database(g), artefatos(h-m) e seis passos(a-f) para descrever esse processo. Cada uma das subseções a seguir descreverão cada um dos passos da figura.

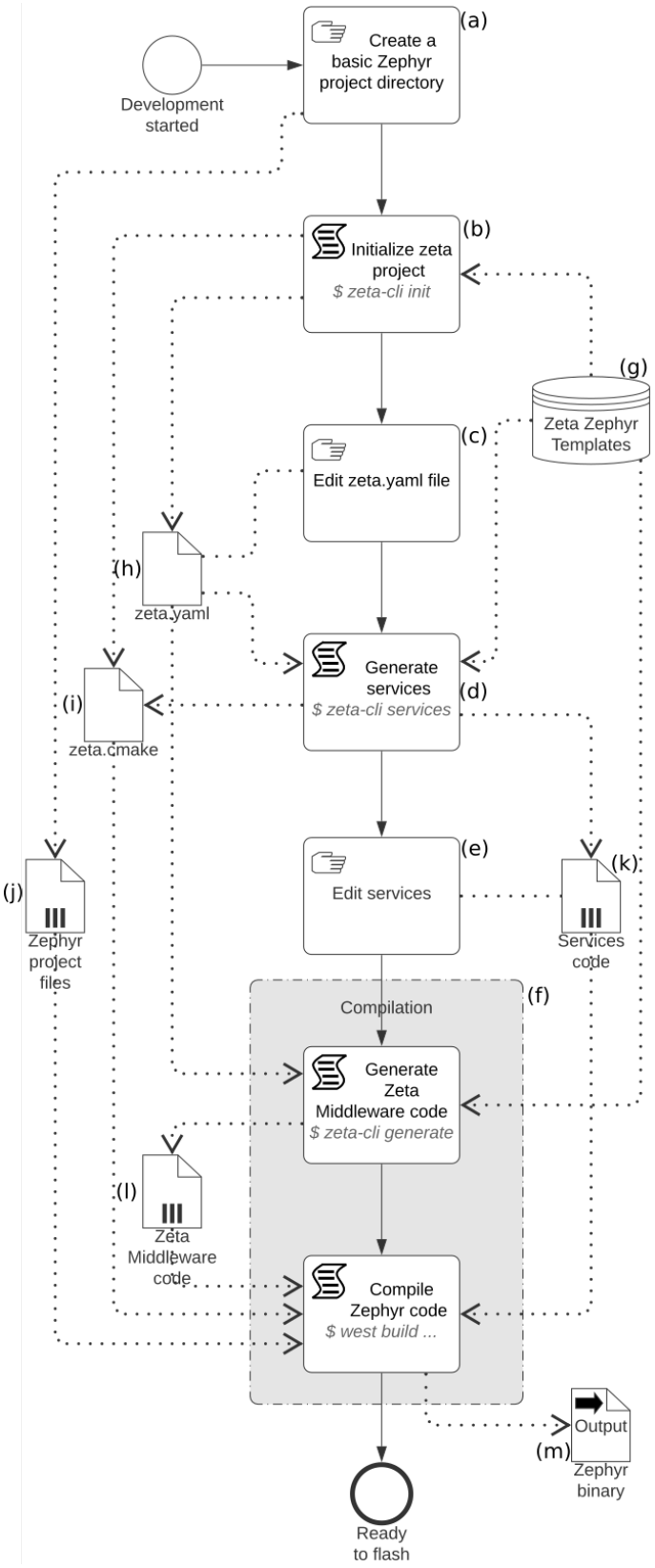


Figura 4.5: Etapas do desenvolvimento de firmware utilizando o Zeta.

4.3.1 Criar o diretório para o Projeto Zephyr (a)

Esse é o primeiro passo a ser feito e se trata de iniciar o diretório do projeto. Dentro do repositório do Zeta Middleware temos uma pasta chamada `samples`, dentro dessa pasta temos alguns exemplos de código de alguns projetos. Um deles é um diretório com um projeto vazio do zephyr, apenas com arquivos de configuração, nesse caso um `CMakeLists.txt` usado pelo CMake para compilar o projeto. Nesse arquivo, a única linha que o desenvolvedor deve alterar é linha referente ao nome do projeto. Esses arquivos de configuração do Zephyr são representados pelo artefato (j) na figura.

4.3.2 Inicialização do Zeta (b)

Como já temos os arquivos de configuração do Zephyr, precisamos adicionar os arquivos de configuração relacionados ao Zeta. Para isso, se faz necessário que dentro do diretório usado seja executado o comando `zeta init`. Esse comando irá gerar os arquivos *zeta.cmake*(i)e o *zeta.yaml*(h). O *zeta.cmake* é utilizado pelo Zeta para inicializar o Zeta como um módulo do Zephyr durante o processo de compilação. Além disso, esse arquivo é responsável para que antes do processo de compilação seja executado o comando `zeta gen`. O arquivo *zeta.yaml* contém todas as definições e configurações dos serviços e canais, bem como a relação entre eles.

4.3.3 Editar o arquivo zeta.yaml (c)

O arquivo *zeta.yaml* tem todas os detalhes de customização do Zeta. O papel do desenvolvedor é ajustar esse arquivo para suas necessidades. Existem três chaves principais nesse arquivo: "Config", "Services"e "Channels". A chave Config contem configurações do Zeta, relacionadas ao funcionamento bem como ao ambiente em que o Zeta será executado. Por exemplo, configurações do funcionamento do uso da flash do Hardware ou a periodicidade que a `Storage Thread` deve salvar os dados em flash. A chave Services contém todos os serviços da aplicação, como por exemplo, suas configurações de tamanho, lista de canais inscritos e publicados. Por fim temos a chave de Channels, onde terá a definição de todos os canais que serão utilizados para estabelecer uma comunicação e transferência de dados entre os serviços.

4.3.4 Gerando os serviços (d)

Com o arquivo *zeta.yaml* definido, devemos agora gerar os arquivos fonte que servirão de base para a implementação dos serviços, ou seja, definir uma função base de loop da thread, de callback e um comando de inicialização do serviço. Para que isso seja feito, nesse passo é chamado o comando `zeta services`, que escolherá os templates necessários

no ZZTD para cumprir sua função. O código relacionado aos serviços pode ser visualizado no artefato (k). Esses arquivos são adicionados numa pasta chamada *src* dentro do projeto. Além disso, outra função desse comando é adicionar no arquivo *zeta.cmake* os arquivos gerados no processo de compilação.

4.3.5 Editando os serviços (e)

Análogo ao passo descrito na subseção 4.3.3, quando os arquivos relacionados ao serviço são gerados, eles não tem nenhum código no corpo de suas funções. Aqui é necessário que o desenvolvedor implemente o corpo das funções de modo que a aplicação atenda as suas necessidades.

4.3.6 Compilação (f)

O último passo é o processo de compilação do projeto Zephyr. Nesse passo temos dois sub-passos, um relacionado a geração de código do Zeta e outro relacionado a compilação do projeto como um todo. Note que, até aqui, tudo que temos relacionado ao Zeta são arquivos de configuração e a definição de qual código deve ser executado nas threads dos serviços e nas suas respectivas funções de callback. Todo o código relacionado ao funcionamento do middleware só é gerado e só aparece no projeto no processo de compilação e é adicionado dentro da pasta *build* que será gerada ao fim do processo de compilação. Sendo assim, o primeiro sub-passo tem a função de gerar esse código do middleware com base nos arquivos de configurações que já foram definidos e implementados nos passos anteriores. Note que, esse sub-passo é executado de forma automática quando o usuário deseja compilar o projeto. Por fim, temos a compilação do projeto em si, o que nos dá como resultado um binário do projeto pronto para execução, que pode ser visualizado no artefato (m).

4.4 Implementando um exemplo com o Zeta

O objetivo desta seção é exemplificar, através dos passos descritos na seção 4.3, como se utiliza na prática o Zeta junto com o Zephyr RTOS. A configuração e relação de canais e serviços que vamos utilizar no exemplo pode ser visualizado na figura 4.6.

Muitos dispositivos utilizados em sistemas embarcados estão associados a sensores que captam alguma informação externa e, através dessa informação, tem a capacidade de se comunicar com algum atuador para efetuar alguma tarefa. Diante desse contexto, pensamos em três serviços e três canais. O serviço `Sensor` possui uma thread que tem como papel coletar os dados do sensor e escrever esses dados no canal `SENSOR_DATA`. Esse canal, por sua vez, é um canal cujo serviço `Core` está inscrito e, portanto, irá ter conhecimento sempre que houver alterações nele. A função do serviço `Core` é avaliar os valores do sensor e, sempre que

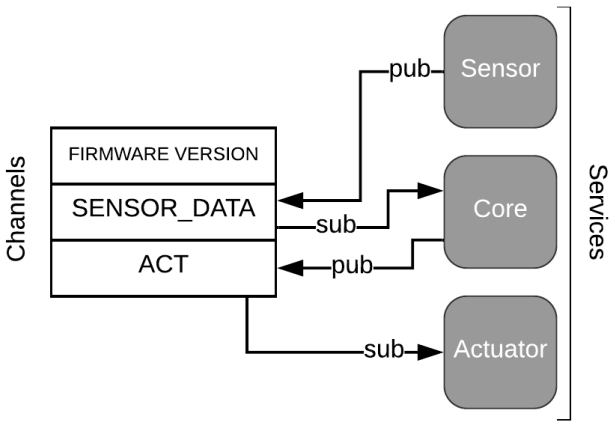


Figura 4.6: Relação e configuração entre canais e serviços usada no exemplo implementado.

uma condição específica for atendida, indicar isso através do canal `ACT`. Por fim, o serviço `Actuator` é o responsável por ligar ou não o atuador, essa decisão fica a cargo dos valores do canal `ACT`, canal cujo serviço está inscrito.

De acordo com o que já foi explicado e está ilustrado na figura 4.5, primeiramente precisamos copiar o template de um projeto do Zephyr, contido dentro do repositório do Zeta, e criarmos assim o diretório do nosso projeto. Esse diretório possui fundamentalmente dois arquivos *prj.conf* e *CMakeLists.txt*. O primeiro é um arquivo onde é possível editar configurações do RTOS e o segundo é utilizado pelo CMake no processo de compilação. O conteúdo dos dois arquivos podem ser vistos nos códigos 4.1 e 4.2. No início de cada projeto o desenvolvedor pode alterar o nome do projeto no *CMakeLists.txt*. Essa alteração pode ser vista no código 4.1, na linha 5.

```
1  cmake_minimum_required(VERSION 3.13.1)
2
3  include(zeta.cmake NO_POLICY_SCOPE)
4  include($ENV{ZEPHYR_BASE}/cmake/app/boilerplate.cmake NO_POLICY_SCOPE)
5  project(RunningExample)
6
7  set(CMAKE_EXPORT_COMPILE_COMMANDS ON)
8
9  include_directories(${HEADERS})
10 target_sources(app PRIVATE ${SOURCES})
```

Código 4.1: CMakeLists.txt utilizado no exemplo

```
1  CONFIG_LOG=y
2  CONFIG_ZETA_LOG_LEVEL_DBG=y
```

Código 4.2: prj.conf utilizado no exemplo

No arquivo 4.2 temos que a linha 1 habilita o sistema de Logging dentro do Zephyr, utilizado no processo de debug do código. Na linha 2 estamos ajustando o nível de verbosidade do Log do Zeta para `DEBUG`, que por sua vez é o nível maior de verbosidade dentro do Zephyr.

Após a inicialização do projeto, precisamos adicionar os arquivos de configuração do zeta. Para isso, é necessário dentro do diretório criado executar o comando `zeta init`. A partir desse comando, teremos como saída os arquivos *zeta.cmake* e *zeta.yaml*. Esse passo é previsto e explicado na seção 4.3.2.

Temos que o arquivo *zeta.yaml* é o arquivo de entrada que o Zeta utiliza no processo de geração de código do middleware, é através dele que serão gerados os canais e que os serviços estarão devidamente associados. Dessa forma, inicialmente, quando executamos o comando `zeta init` temos um arquivo padrão gerado, que pode ser visualizado no código 4.3, e precisamos fazer alterações nesse arquivo para termos uma configuração tal qual a figura 4.6.

```
1  Channels:
2    - FIRMWARE_VERSION:
3        size: 4
4        read_only: true
5        initial_value: [0xF1, 0xF2, 0xF3, 0xF4]
6
7  Services:
8    - CORE:
9        priority: 5
10       stack_size: 512
11   - BOARD:
12       priority: 4
13       stack_size: 512
```

Código 4.3: Arquivo zeta.yaml padrão gerado pelo comando zeta init.

Podemos visualizar no código 4.4 as alterações que foram feitas para atingir o nosso objetivo. Temos da linha 1 até a 5 a definição do canal *FIRMWARE_VERSION*, canal com dois bytes (linha 3), somente de leitura(linha 4) e que será utilizado para armazenar a versão do firmware, definido como 0x0005 (linha 5). Em seguida, temos a criação dos nossos dois canais que serão utilizados pelos nossos serviços de forma mais intensa, o canal `SENSOR_DATA` e o `ACT`, definidos nas linhas 6 a 7 e 8 a 9, respectivamente. Note que no canal `SENSOR_DATA` temos dados de 8 bytes (linha 7) sendo trafegados. Já no canal `ACT` temos somente um byte (linha 9). Com relação aos serviços, `SENSOR`, `CORE` e `ACTUATOR`, eles são definidos da linha 11 até a 28 e possuem, respectivamente, as prioridades 1 (linha 13), 4 (linha 18) e 3 (linha 25). Note que em todos os serviços temos o mesmo tamanho da stack fornecida para suas threads funcionarem, sendo de 512 bytes seu tamanho (linhas 14, 19 e 26). Os campos de `pub_channels` e `sub_channels` são utilizados para definir quais canais o serviço estará publicado e inscrito. O prefixo `!ref` é uma extensão do YAML que implementamos, que tem

como objetivo fazer referência a alguma chave já criada dentro do arquivo. Portanto, temos o serviço SENSOR publicando no canal SENSOR_DATA (linha 16), o serviço CORE publicando no canal ACT e inscrito no canal SENSOR_DATA e, por fim, o serviço ACTUATOR inscrito no canal ACT. Essa etapa de alterações do arquivo YAML para atender as necessidades do desenvolvedor e do seu projeto é comentada na seção 4.3.3.

```
1  Channels:
2    - FIRMWARE_VERSION:
3        size: 2
4        read_only: true
5        initial_value: [5, 0]
6    - SENSOR_DATA:
7        size: 8
8    - ACT:
9        size: 1
10
11 Services:
12   - SENSOR:
13       priority: 1
14       stack_size: 512
15       pub_channels:
16           - !ref SENSOR_DATA
17   - CORE:
18       priority: 4
19       stack_size: 512
20       pub_channels:
21           - !ref ACT
22       sub_channels:
23           - !ref SENSOR_DATA
24   - ACTUATOR:
25       priority: 3
26       stack_size: 512
27       sub_channels:
28           - !ref ACT
```

Código 4.4: Arquivo zeta.yaml alterado para ser utilizado no exemplo.

A partir da estrutura de serviços e canais definida no arquivo YAML, chegamos na etapa de criar os arquivos fonte dos serviços (passo descrito na seção 4.3.4). Para que isso seja feito, devemos executar o comando `zeta services -g`. Como saída desse comando teremos a criação, dentro do diretório raiz do projeto, de um diretório chamado *src* e dentro dele teremos arquivos gerados com os nomes dos serviços, no nosso caso teremos os arquivos *sensor.c*, *core.c* e *actuator.c*. Além disso, esse comando irá fazer alterações no arquivo *zeta.cmake*, a fim de adicionar esses arquivos fonte gerados na lista de arquivos que deverão ser compilados durante o processo de compilação do projeto.

Podemos ver no código 4.5 qual o conteúdo padrão dos arquivos fonte que são gerados. Os três arquivos tem o mesmo conteúdo, mudando somente a referência ao nome do serviço. Da linha 1 até a 3 temos os includes padrões necessários para que o código funcione. Na linha 5 temos a declaração de que as mensagens de LOG eventualmente utilizadas dentro desse arquivo estarão relacionadas com o modulo Zeta e possuirá um nível de verbosidade igual ao definido na linha 2 do código 4.2. Temos na linha 7 a definição de um semáforo binário, que será utilizado pela função de callback do serviço para indicar quando uma alteração em um canal inscrito ocorrer. Podemos ver a utilização desse semáforo na função de callback do serviço, na linha 17 até a 20. Note que, a utilização desse semáforo não é obrigatória, esse código é somente um ponto de partida que decidimos implementar no Zeta. O desenvolvedor é livre para utilizar outras estratégias dentro da função de callback do seu serviço. Temos também da linha 26 até a 32 a implementação inicial da thread do serviço. Por fim, temos na última linha do arquivo (linha 34), a chamada de uma macro do Zeta para inicializar o serviço, precisando ser passado o nome do serviço, o ponteiro para sua função da thread e o ponteiro para a sua função de callback. Note que, é obrigatório que o nome do serviço esteja de acordo com as possibilidades de serviços definidas no arquivo YAML, em caso contrário teremos um erro de compilação. Também teremos um erro de compilação se houver algum serviço definido no arquivo YAML e ele não seja inicializado através da macro `ZT_SERVICE_INIT` dentro dos arquivos fonte compilados.

```
1  #include <logging/log.h>
2  #include <zephyr.h>
3  #include <zeta.h>
4
5  LOG_MODULE_DECLARE(zeta, CONFIG_ZETA_LOG_LEVEL);
6
7  K_SEM_DEFINE(CORE_callback_sem, 0, 1);
8
9
10 /**
11  * @brief This is the function used by Zeta to tell the CORE that one(s) of
        the
12  * channels which it is subscribed has changed. This callback will be
       called passing the
13  * channel's id in it.
14  *
15  * @param id
16  */
17 void CORE_service_callback(zt_channel_e id)
18 {
19     k_sem_give(&CORE_callback_sem);
20 }
21
```

```
22 /**
23  * @brief This is the task loop responsible to run the CORE thread
24  * functionality.
25  */
26 void CORE_task()
27 {
28     LOG_DBG("CORE Service has started...[OK]");
29     while (1) {
30         k_sem_take(&CORE_callback_sem, K_FOREVER);
31     }
32 }
33
34 ZT_SERVICE_INIT(CORE, CORE_task, CORE_service_callback);
```

Código 4.5: Arquivo core.c padrão gerado pelo comando zeta services -g.

Já com relação as mudanças no arquivo *zeta.cmake*, podemos visualizar o conteúdo do arquivo no código 4.6. Temos na linha 1 uma mensagem que será escrita durante o processo de compilação, para que o desenvolvedor visualize as etapas sendo executadas. Da linha 3 até a 8 temos a chamada da execução de um processo que ocorre imediatamente antes ao processo de compilação, que tem o objetivo de gerar os arquivos finais do zeta. Por sua vez, da linha 10 até a 19 temos uma sequência de comandos de controle para definir qual será o arquivo *.conf* que será utilizado pelo Zephyr. A partir daqui é onde ocorrem as mudanças feitas pelo comando `zeta services -g`. Inicialmente (após o comando `zeta init`) tínhamos somente as linhas 21 e 22, que incluem o arquivo de configuração do Zeta e adiciona o diretório que contém os arquivos de cabeçalho do Zeta para o processo de compilação. A linha 23 foi adicionada somente pelo comando `zeta services -g` e tem como objetivo adicionar os arquivos fonte dos serviços que serão utilizados. Por fim, temos na linha 25 (gerada pelo comando `zeta init`) a definição do Zeta como um módulo do Zephyr.

```
1  message("[ZETA]: Running zeta.cmake")
2
3  execute_process(COMMAND zeta gen -b "${CMAKE_CURRENT_LIST_DIR}/build"
4                          ${CMAKE_CURRENT_LIST_DIR}/zeta.yaml RESULT_VARIABLE
       ztcli_gen_exit_code)
5
6  if(ztcli_gen_exit_code GREATER 0)
7    message( FATAL_ERROR "ZetaCli generation failed with exit code: ${
       ztcli_gen_exit_code}")
8  endif()
9
10 if(CONF_FILE)
11 elseif(DEFINED ENV{CONF_FILE})
12   set(CONF_FILE $ENV{CONF_FILE})
13 elseif(EXISTS   ${CMAKE_CURRENT_SOURCE_DIR}/prj_${BOARD}.conf)
```

```
14   set(CONF_FILE ${CMAKE_CURRENT_SOURCE_DIR}/prj_${BOARD}.conf)
15 elseif(EXISTS   ${CMAKE_CURRENT_SOURCE_DIR}/boards/${BOARD}.conf)
16   set(CONF_FILE ${CMAKE_CURRENT_SOURCE_DIR}/prj.conf ${
       CMAKE_CURRENT_SOURCE_DIR}/boards/${BOARD}.conf)
17 elseif(EXISTS   ${CMAKE_CURRENT_SOURCE_DIR}/prj.conf)
18   set(CONF_FILE ${CMAKE_CURRENT_SOURCE_DIR}/prj.conf)
19 endif()
20
21 list(APPEND CONF_FILE "${CMAKE_CURRENT_LIST_DIR}/build/zeta/zeta.conf")
22 list(APPEND HEADERS "${CMAKE_CURRENT_LIST_DIR}/build/zeta/include/")
23 list(APPEND SOURCES "${CMAKE_CURRENT_LIST_DIR}/src/sensor.c" "${
       CMAKE_CURRENT_LIST_DIR}/src/core.c" "${CMAKE_CURRENT_LIST_DIR}/src/
       actuator.c")
24
25 set(ZEPHYR_EXTRA_MODULES "${CMAKE_CURRENT_LIST_DIR}/build/zeta")
```

Código 4.6: Arquivo zeta.cmake atualizado depois do comando zeta services -g.

Diante desse estágio atual, o próximo passo é fazer as modificações nos arquivos fonte gerados para que o código se comporte como o desejado. Esse passo é identificado na seção 4.3.5. Os códigos 4.7, 4.8 e 4.9 mostram a implementação do exemplo. Note que aqui omitiremos parte do código para melhorar a compreensão da parte que importa. O código que foi omitido é similar ao gerado automaticamente pelos comandos do Zeta.

No código 4.7 temos das linhas 1 até a 4 a função responsável por coletar os dados do sensor. No nosso caso, como não fizemos o teste com um sensor de verdade, estamos interpretando o tempo em mili segundos desde que o código iniciou como o dado proveniente de um sensor. Da linha 6 até a linha 17 temos a versão final do código da nossa thread. Temos que a sua tarefa é executada a cada 2 segundos (linha 10) e que seu papel é publicar no canal `SENSOR_DATA` o valor da função `sensor_data_generate` (linha 15).

```
1  u64_t sensor_data_generate()
2  {
3      return (u64_t) k_uptime_get();
4  }
5
6  void SENSOR_task()
7  {
8      LOG_DBG("SENSOR Service has started...[OK]");
9      while (1) {
10         k_sleep(K_SECONDS(2));
11         LOG_DBG("Publishing data to SENSOR_DATA");
12
13         u32_t cycles = k_cycle_get_32();
14         LOG_DBG("Publishing at %u cycles.", cycles);
15         zt_chan_pub(ZT_SENSOR_DATA_CHANNEL, ZT_DATA_U64(
       sensor_data_generate()));
```

```
16      }
17  }
```

Código 4.7: Fragmento do arquivo sensor.c na sua versão final.

Já no código 4.8, temos da linha 1 até a 21 a versão final da implementação da thread do serviço `CORE`. Aqui, temos inicialmente uma leitura (linha 8) sendo feito do canal `FIRMWARE_VERSION` e um *print* (linha 9) do seu valor. Em seguida vamos para o loop da thread (linha 13 até a 20). Nesse loop temos que, toda vez que houver uma mudança nos canais em que o serviço CORE é inscrito (linha 14), no caso somente o serviço `SENSOR_DATA`, iremos fazer uma leitura no canal `SENSOR_DATA` (linha 15) e checar se o valor do sensor bate com uma condição (linha 16). O resultado dessa verificação será publicado no canal `ACT` (linha 19).

```
1  void CORE_task()
2  {
3      zt_data_t *sensor_data = ZT_DATA_BYTES(8, 0);
4      LOG_DBG("CORE Service has started...[OK]");
5
6      /* version's data scope. Remove it after using it */ {
7          zt_data_t *version = ZT_DATA_BYTES(2, 0);
8          zt_chan_read(ZT_FIRMWARE_VERSION_CHANNEL, version);
9          LOG_DBG("Firmware version v%u.%u", version->bytes.value[1],
10                 version->bytes.value[0]);
11     }
12     u8_t condition = 0;
13     while (1) {
14         k_sem_take(&CORE_callback_sem, K_FOREVER);
15         zt_chan_read(ZT_SENSOR_DATA_CHANNEL, sensor_data);
16         condition = (sensor_data->u64.value & 0x10) == 0x10;
17         LOG_DBG("Checking condition 0x%llx & 0x10 = %d", sensor_data->u64.
    value,
18                 condition);
19         zt_chan_pub(ZT_ACT_CHANNEL, ZT_DATA_U8(condition));
20     }
21 }
```

Código 4.8: Fragmento do arquivo core.c na sua versão final.

Por fim temos o código 4.9, onde da linha 1 até a 14 vemos a versão final da implementação da thread do serviço `ACTUATOR`. Em seu loop, temos, assim como na thread do serviço `CORE`, um semáforo (linha 6), que indica que o loop irá ser executado toda vez que um canal, cujo o serviço `ACTUATOR` for inscrito, for modificado. Nesse caso, toda vez que o canal `ACT` for modificado, a thread irá fazer uma leitura do seu valor e a depender do valor teremos o atuador sendo ligado ou desligado (linha 9 ou 11).

```
void ACTUATOR_task()
{
    LOG_DBG("ACTUATOR Service has started...[OK]");
    zt_data_t *act_cmd = ZT_DATA_U8(0);
    while (1) {
        k_sem_take(&ACTUATOR_callback_sem, K_FOREVER);
        zt_chan_read(ZT_ACT_CHANNEL, act_cmd);
        if (act_cmd->u8.value == 0) {
            LOG_WRN("Actuator turning OFF");
        } else {
            LOG_WRN("Actuator turning ON");
        }
    }
}
```

Código 4.9: Fragmento do arquivo actuator.c na sua versão final.

Desta forma, chegamos assim no último passo do processo de utilização do Zeta no Zephyr RTOS, comentado na seção 4.3.6. Perceba que o exemplo implementado não tem uma dependência de hardware, fizemos isso para que o exemplo fosse fácil de ser reproduzível. Para isso, basta escolher a opção *native_posix* ao compilar esse código com o Zephyr. Basicamente, para compilar esse projeto o desenvolvedor deve executar o seguinte comando na raiz do diretório do projeto `rm -rf build && west build -b native_posix`. O primeiro comando irá apagar um diretório *build* que possa existir devido a compilações anteriores e o segundo comando irá compilar de fato o projeto através da ferramenta *west* fornecida pelo Zephyr. A partir do código compilado, como estamos compilando o código para ser executado no próprio computador, não se faz necessário uma etapa de salvar o binário num dispositivo de hardware. Pelo contrário, o binário gerado já pode ser executado na própria máquina através do comando `west build -t run`.

4.4.1 Exemplo da apresentação do trabalho

Durante a apresentação deste trabalho mostramos um exemplo de um robô num galpão industrial que tem a função de mover prateleiras de um ponto inicial para um ponto final e cujo os requisitos são:

- **Requisito 01**: Possui duas rodas, com um motor individual em cada que precisa ser gerenciado.
- **Requisito 02**: Possui um sensor que detecta obstáculos.
- **Requisito 03**: Recebe da rede ordens para mover uma estante de uma posição inicial para uma final.

- **Requisito 04**: Quando a bateria estiver baixa, enviar para a rede um alerta para recarregamento.

O desenvolvimento desse robô imaginário utilizando o Zeta pode ser acessado através do repositório https://github.com/lucaspeixotot/my-zeta-tcc, no diretório samples/tcc-robot.

4.5 Avaliação experimental

De modo a analisar os benefícios de se utilizar o nosso middleware fizemos alguns experimentos com códigos utilizando ou não o Zeta. Portanto, essa seção tem como objetivo definir a avaliação experimental que foi desenvolvida para avaliar o desempenho de um código com o Zeta perante códigos que implementam os mesmos requisitos mas que não utilizam o Zeta.

Analisaremos a complexidade, coesão e tamanho do footprint do código em três implementações distintas. A primeira implementação que utilizaremos é baseada num padrão de código semelhante ao que utiliza-se em código no Arduino. Ou seja, teremos todo o código numa única thread com duas funções principais: setup e loop. Na segunda implementação teremos um código com o uso do Zephyr RTOS, consequentemente utilizando mais de uma thread. A ideia desse código é realizar a implementação da forma mais correta possível em relação as métricas estabelecidas. Por fim, temos a implementação do código com o nosso middleware.

4.5.1 Descrição do experimento

Para tentar demonstrar os benefícios do Zeta escolhemos um experimento que reflete um cenário comum de aplicações em sistemas embarcados relacionadas a sensores e a requisições externas aos dados desses sensores. Dentre as características desse experimento temos comunicações entre threads, armazenamento volátil e não volátil, concorrência de dados e requisições remotas. Diante disso, tentamos implementar um exemplo o mais próximo da realidade possível. Os requisitos do firmware implementado são os seguintes:

- **Requisito 01**: existem três sensores (A, B e C) diferentes e é necessário que sejam salvas amostras desses três sensores a cada três segundos. Os sensores A e B possuem dados com o tamanho de 1 byte e o sensor C possui 4 bytes;
- **Requisito 02**: deve estar salvo, em RAM, apenas as últimas 10 amostragens de cada sensor;
- **Requisito 03**: é possível receber de forma remota seis tipos diferentes de requisições (cada requisição possui 1 byte), sendo elas:

  - **0xA0**: último dado do sensor A;
  - **0xA1**: último dado do sensor B;
  - **0xA2**: último dado do sensor C;
  - **0xA3**: média dos últimos 10 dados do sensor A;
  - **0xA4**: média dos últimos 10 dados do sensor B;
  - **0xA5**: média dos últimos 10 dados do sensor C.

- **Requisito 04**: é necessário salvar, em flash, o último dado do sensor A e a última resposta dada a uma requisição remota. A periodicidade desse armazenamento deve ser de 30 segundos e quando o firmware inicializar deve ser restaurado o último valor desses dados da flash;

- **Requisito 05**: o firmware deve gerar uma requisição remota aleatória a cada 30 segundos.

4.5.2 Características das implementações

Cada uma das três implementações tem características próprias e nessa seção vamos descrever um pouco melhor as características de cada uma das alternativas.

Implementação Single-Thread : Tem como objetivo desenvolver um código seguindo a abordagem utilizada no Arduino, baseado numa única thread com todas as responsabilidades. A modularização típica desse tipo de código é separar em uma função de setup e uma de loop. Onde a função de setup será responsável por realizar as configurações necessárias. Por fim, a função de loop será encarregada de executar a lógica da aplicação.

Implementação Multi-Thread : Tem como objetivo desenvolver um código com práticas de programação aceitáveis utilizadas no mercado. Essa implementação separa o código em quatro módulos. Sendo o primeiro o BOARD, que é responsável por obter as amostras do sensor. O CORE, como módulo central, com a responsabilidade de armazenar o dado volátil, calcular a média e responder as requisições remotas. O módulo STORAGE tem como objetivo armazenar o dado não volátil e restaurá-lo quando a placa é energizada, ou seja, quando o código da aplicação inicia. O último módulo, NET, é responsável por receber as requisições remotas e enviar as eventuais respostas. As estruturas de Kernel utilizadas foram somente semáforos e filas, a fim de deixar o código o mais simples possível.

Implementação baseada no Zeta Middleware : Em termos de separação de código em módulos a implementação com o Zeta se apresenta de forma similar a implementação Multi-Thread. Contudo, toda a comunicação entre os módulos, no zeta chamados de serviços, é feita pelo Zeta, bem como o armazenamento dos canais não voláteis. Consequentemente, temos somente os módulos BOARD, CORE e NET, uma vez que o Zeta é o responsável pela implementação de um módulo para realizar o papel do módulo STORAGE.

4.5.3 Métricas

Para conseguirmos realizar uma avaliação e comparação das três implementações precisamos nos basear em algumas métricas. Desse modo, utilizamos métricas determinísticas, baseadas no código e em seu tamanho para fazer esse estudo e evitar problemas devido a interpretações e subjetividade.

4.5.3.1 Complexidade de código

Primeiramente pensamos em analisar a complexidade do código gerado pelas diferentes implementações. A complexidade do código acaba tendo uma influência grande no que diz respeito a qualidade do código e a sua manutenibilidade ao longo do tempo, segundo Antinyan et al. [4]. Para avaliar a complexidade do código pensamos em três métricas, sendo elas:

- **NPATH Complexity**: Nejmeh [17], mede o quão difícil é cobrir todas as funcionalidades do sistema com testes. Para avaliar essa métrica e obter seus dados utilizamos a ferramenta OClint;
- **Cyclomatic Complexity**: similar a métrica anterior, permite identificar se o código é difícil de testar ou de manter [20]. Também utilizamos a ferramenta OCLint para essa métrica;
- **SLOC**: nos dá o tamanho do código e juntamente com as métricas de complexidade conseguimos ter uma ideia da densidade da complexidade do código [11](embora não a tenhamos calculado). Para essa métrica utilizamos a ferramenta cloc.

4.5.3.2 Coesão de código

Analisar a coesão de código parte do princípio de responsabilidade única dos módulos. Ou seja, cada módulo do sistema deve ter o menor número possível de requisitos do sistema sendo resolvidos pelo módulo. Dessa forma, cada módulo consegue resolver atividades específicas da solução final. Portanto, essa métrica mede a distribuição dos requisitos

do sistema ao longo dos módulos do código [7]. Um alto número de requisitos sendo implementado num único módulo implica numa baixa coesão de código. Para o cálculo da coesão do código nós subtraímos a mediana da distribuição dos requisitos pelos módulos pelo total de requisitos. Por exemplo, se tivermos 5 requisitos do sistema e todos eles tiverem sido implementados num único módulo, temos que a mediana é 5, logo a coesão é de $C = 5-5 = 0$. A fim de categorizar os resultados da coesão de código em coesão baixa, média e alta, dividimos os resultados entre [0,1], [2,3] e 4, respectivamente.

4.5.3.3 Footprint da memória

Nessa métrica analisamos o tamanho do binário do código compilado sendo utilizado para o mesmo hardware, levando em consideração o tamanho de sua RAM e da flash. É importante lembrar que em sistemas embarcados o uso de memória afeta diretamente nos custos de hardware. Um código com um baixo uso de memória RAM ou flash pode talvez ser executado num hardware com um custo menor. O oposto é verdade também, caso se tenha um código que um hardware barato possa executar, mas que o tamanho que o código demanda o hardware não suporte, será necessário utilizar um hardware mais robusto apenas por conta do tamanho de memória utilizado.

4.5.3.4 Latência

Como desenvolvemos em nosso middleware um mecanismo de comunicação entre serviços (ISC), foi necessário adicionar uma porção de código responsável pela troca de informações entre serviços. Essa troca de informações é baseada numa região de memória compartilhada e existem cópias dos dados existentes nessas regiões. Devido a isso, é possível que existe uma perda de desempenho no código desenvolvido com base no Zeta. Dessa forma, essa métrica visa medir a latência associada ao uso do Zeta, do momento em que as publicações são feitas até o momento em que os serviços recebem essa informação. O desempenho foi observado com base em duas situações: comunicação entre serviços um para um e um para *N*.

4.5.4 Ambiente de desenvolvimento

A reprodução do experimento definido e comentado ao longo de toda essa seção é possível ser realizada através do código disponível no repositório X. As ferramentas e suas respectivas versões utilizadas foram as seguintes:

- Zeta-CLI 0.2.0[1];
- Zephyr 2.3.0rc2[2];

[1] https://github.com/zeta-middleware/zeta
[2] https://github.com/zephyrproject-rtos/zephyr/tree/v2.3-branch

- Zephyr SDK 0.11.2[3];
- Python 3.6.9;
- West 0.7.2[4];

Para a medição do tamanho do footprint e da performance dos códigos utilizamos a placa *nRF52840 Dongle*. Utilizamos essa placa para conseguirmos ter uma medição real em dispositivos ARM, visto que o código gerado para Native Posix tem uma montagem e tamanho do binário diferentes em comparação com a arquitetura ARM.

[3]https://docs.zephyrproject.org/2.3.0/getting$_s tarted/index.html install - a - toolchain$

[4]https://docs.zephyrproject.org/2.3.0/getting$_s tarted/index.html get - zephyr - and - install - python - dependencies$

5

Resultados e Discussões

5.1 Resultados

Nessa seção vamos expor os resultados do experimento definido na seção 4.5. Podemos visualizar nas tabelas 5.1, 5.2 e 5.3 a distribuição dos requisitos da aplicação dispostos sob os módulos de cada implementação.

Tabela 5.1: Distribuição dos requisitos na implementação Single-threaded.

Módulo	Requisitos	Total
Setup/Loop	R1, R2, R3, R4 and R5	5

Tabela 5.2: Distribuição dos requisitos na implementação Multi-threaded.

Módulo	Requisitos	Total
CORE	R2 and R3	2
BOARD	R1	1
NET	R5	1
STORAGE	R4	1

Tabela 5.3: Distribuição dos requisitos na implementação Zeta.

Módulo	Requisitos	Total
CORE	R2 and R3	2
BOARD	R1	1
NET	R5	1

Com relação a coesão de código, temos que na tabela 5.1 podemos ver uma baixa coesão de código, visto que a mediana é 5 e temos $C = 5 - 5 = 0$. Na tabela 5.2 temos uma

Tabela 5.4: Resultados da avaliação experimental.

Métrica	Single-threaded	Multi-threaded	Zeta
NPATH Complexity	1606	63	**53**
Cyclomatic Complexity	66	63	**53**
SLOC	251	350	**235**
Code cohesion	*Low*	***High***	***High***
Memory footprint (RAM in Bytes)	**21568**	24654	25550
Memory footprint (Flash in Bytes)	**48148**	49864	51580

alta coesão de código, visto que a mediana é $1(1+1/2=1)$, note que a mediana entre dois elementos é igual a média, e temos $C = 5-1 = 4$. Bem com na tabela 5.3, onde temos o mesmo desempenho com relação a coesão de código. Na tabela 5.4 podemos ver números e comparações entre as três implementações diante das diferentes métricas que definimos na seção 4.5.3. Destacamos em negrito os valores que em cada métrica implicam no melhor resultado.

A implementação *Single-threaded* apresenta o menor tamanho de footprint(RAM e Flash). Contudo, observe que ele apresenta os maiores valores nas métricas relacionadas a complexidade do código, que são as métricas NPATH e *cyclomatic complexity* (3000% e 19.69% mais do que a implementação do Zeta, respectivamente). A causa dessa diferença é pela necessidade de utilização de muitos *timers*, *loops*, instruções de decisão, e *polling* para atingir todos os requisitos do sistema. Outra questão significante que já indicamos no parágrafo anterior é a baixa coesão do código [7].

Na implementação *Multi-threaded*, podemos observar um resultado similar, porém inferior, com relação a implementação do Zeta. A complexidade de código é maior, cerca de 15.87% devido a necessidade de implementar toda a arquitetura e comunicação entre os módulos, bem como o requisito de armazenamento em flash. O footprint da memória é o segundo melhor, mas apenas 3.33% menos do que a implementação do Zeta.

Note que, as três métricas relacionadas a complexidade do código mostram que a implementação com o Zeta teve um código menor e mais simples do que as outras implementações. A única deficiência observada no Zeta nesse experimento foi com relação ao tamanho do footprint da memória, que foi cerca de 6.65% e 3.33% maior do que a implementação *single-threaded* e *multi-threaded*. Esse problema é devido aos metadados que precisam existir para que a comunicação entre canais e serviços funcione corretamente.

Outra característica importante de ser analisada é a latência da troca de mensagens entre canais e serviços dentro do Zeta. Como o Zeta traz uma arquitetura por trás dessa comunicação, é importante analisar quanto isso está influenciando na latência das mensagens. A figura 5.1 mostra a latência da comunicação entre serviços num cenário um para um, variando o tamanho do canal. Já a figura 5.2 mostra os resultados da latência entre serviços num cenário um para N, mantendo o tamanho do canal igual e variando somente o número

de serviços inscritos no canal.

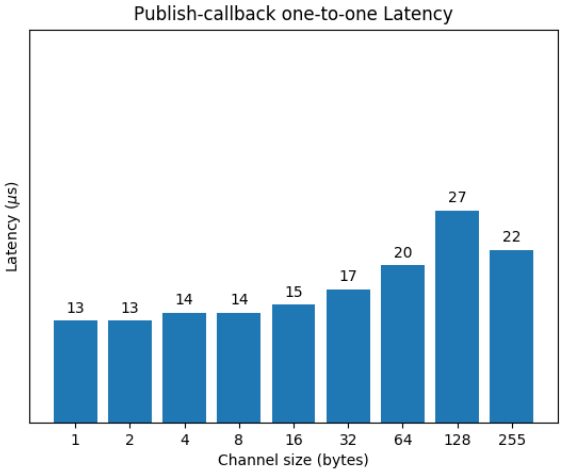


Figura 5.1: Latência entre um publish e a ativação do callback no cenário um para um.

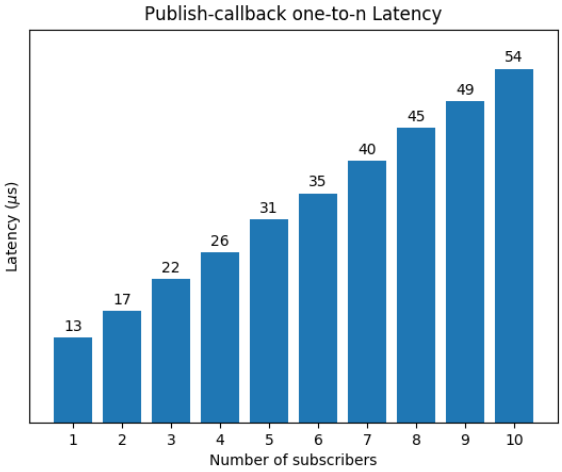


Figura 5.2: Latência entre um publish e a ativação do callback no cenário um para N.

Perceba que, no cenário um para um, a latência cresceu a medida que o tamanho do canal também cresceu. Percebemos que a latência incorporada na utilização do zeta não passa de 30 micro segundos, levando em consideração um tamanho máximo de canal de 255 bytes. Por outro lado, no cenário um para N, a latência cresceu cerca de 5 micro segundos a cada novo serviço inscrito no canal.

5.2 Discussão

Nessa seção vamos discutir a respeito dos resultados obtidos na seção anterior e entender como o Zeta pode ajudar no desenvolvimento de novos projetos, de acordo com alguns pontos que iremos destacar nas próximas sub seções.

5.2.1 Armazenamento e comunicação

O Zeta fornece um mecanismo de armazenamento em flash e uma comunicação entre serviços. Diante disso, temos alguns benefícios associados a essas duas funcionalidades:

1. O desenvolvedor pode focar na implementação das funcionalidades do sistema, ao invés de ter que se preocupar também com a infraestrutura e a rotina de armazenamento em flash;

2. Uma redução no número de módulos quando comparado com a implementação em multi-thread. Enquanto no código com multi-thread foi necessário a implementação dos módulos core, board, net e storage, como descrito na tabela 5.2, a implementação do código com o Zeta não necessita do módulo storage, visto que se trata de uma funcionalidade do próprio middleware;

3. Redução da complexidade do código visto que no código multi-thread é necessário implementar a comunicação entre os serviços(threads) do sistema.

5.2.2 Desvantagens devido a abstração

O Zeta adiciona uma camada de abstração acima do RTOS. Devido a isso, essa camada de abstração pode afetar no consumo de memória. Contudo, o consumo de memória é somente ligeiramente maior do que as outras implementações. No Zeta, cada canal possui 28 bytes de informação em metadados. Com relação ao tamanho de footprint base do Zeta, ou seja, o tamanho do footprint relacionado ao código interno do Zeta(`Storage Thread`, `Monitor Thread`, estruturas e funções internas), temos 26kB de memória flash e 2kB de memória RAM. Caso a aplicação não deseje utilizar canais persistentes, no processo de compilação o ZetaCLI não incluirá o código relacionado ao uso da flash, o que irá salvar 6kB de memória flash.

5.2.3 Latência

Com relação a latência do Zeta podemos pensar no cenário onde temos vários serviços querendo escrever em um único canal ao mesmo tempo. Esse cenário lembra o problema de concorrência por recurso, onde cada canal seria a porção de memória compartilhada, que por sua vez contém um semáforo para indicar quem tem a permissão de escrever. Portanto, quando um serviço está publicando em um canal, outros serviços que queiram escrever irão esperar até que uma escrita anterior seja finalizada. Se esse cenário ocorrer a latência irá piorar na mesma proporção que mais serviços estejam esperando para realizar a escrita. Outro aspecto importante relacionado a latência é com relação ao processo de armazenamento em flash. A thread responsável por essa rotina salva em flash o dado dos canais

persistentes a cada 30 segundos. A `Monitor Thread` tem mais prioridade do que a `Storage Thread`. Assim, a `Storage Thread` não afeta na latência da `Monitor Thread`. Como ler e escrever em flash são operações lentas em comparação com o mesmo processo em RAM, o Zeta mantém uma cópia dos dados dos canais persistentes em RAM, para manter o funcionamento tão rápido quanto possível.

5.2.4 Dívida técnica

No contexto do uso do Zeta, sua arquitetura foi desenvolvida pensando em tornar a manutenibilidade o mais fácil possível para um software de sistemas embarcados, onde o desenvolvedor pode se preocupar somente com as funcionalidades do seu sistema. Na prática, profissionais e pesquisadores podem se beneficiar do uso do Zeta para melhorar o processo de desenvolvimento pelo middleware fornecer uma arquitetura de software desacoplada, focando nas funcionalidades e redução da dívida técnica em projetos longos.

6

Conclusão

A grande difusão de dispositivos embarcados, o time-to-market encurtado e aumento de complexidade de software embarcado são os fatores que nos incentivaram a desenvolver uma ferramente que auxilie o desenvolvimento de projetos de SE, facilitando a implementação e manutenção do firmware. Os resultados obtidos das comparações com outras soluções indicam que o Zeta Middleware é capaz de ser utilizado para desenvolver software para SE. Como consequência do uso da arquitetura provida pelo Zeta temos a criação de um código desacoplado com qualidade superior o que acarreta uma diminuição no tempo de desenvolvimento e num menor custo de manutenção.

As métricas que definimos na seção 4.5.3 podem indicar quão organizado um software pode ter sido implementado. Um dos indicativos que podemos ter a partir dessa análise é saber o nível de dívida técnica que esse software tem. Diante disso, esse trabalho mostra o desenvolvimento e os resultados do Zeta Middleware, que fornece um desacoplamento da comunicação entre serviços, o que ajuda aos desenvolvedores e pesquisadores a desenvolver um código melhor diante de uma arquitetura que fornece desacoplamento. Quando falamos sobre um código melhor neste trabalho, estamos nos referindo, por exemplo, a um software menos complexo, mais coeso e mais desacoplado. Com relação a essas características e outras analisadas, temos na seção 5.1 os resultados experimentais baseados na análise de diferentes alternativas de desenvolvimento de código utilizadas no mercado. Esses resultados indicam a efetividade do uso do Zeta à complexidade de código e coesão, que são características que podem levar a uma menor dívida técnica. Tendo uma menor dívida técnica sabemos que o custo financeiro e técnico de corrigir, adaptar e evoluir um determinado código será menor. Portanto, diante da lacuna de uma ferramenta em SE capaz de prover os benefícios de qualidade de código vistos neste trabalho, visualizamos o Zeta como uma alternativa para o desenvolvimento de soluções para a área.

Como trabalhos futuros um dos objetivos é o desenvolvimento de uma *Interface Definition Language* (IDL), tal como o CDDL [6], para as mensagens trafegadas pelos canais.

Essa mudança melhorará a implementação e confiabilidade da troca de dados entre os serviços. Também temos como objetivo adicionar funcionalidades relacionadas a testes dentro do ZetaCLI, tal como uma infraestrutura dedicada a testes para o middleware, que pode por sua vez acelerar a forma com que os desenvolvedores testam os seus códigos.

Referências bibliográficas

[1] I.F. Akyildiz, W. Su, Y. Sankarasubramaniam, and E. Cayirci. 2002. Wireless sensor networks: a survey. *Computer Networks* 38, 4 (2002), 393 – 422. https://doi.org/10.1016/S1389-1286(01)00302-4

[2] A. Ampatzoglou, A. Ampatzoglou, A. Chatzigeorgiou, P. Avgeriou, P. Abrahamsson, A. Martini, U. Zdun, and K. Systa. 2016. The Perception of Technical Debt in the Embedded Systems Domain: An Industrial Case Study. In *2016 IEEE 8th International Workshop on Managing Technical Debt (MTD)*. 9–16.

[3] Anonymous. 2020. *Technical Debt.* Product Plan. Retrieved Out 07, 2020 from https://www.productplan.com/glossary/technical-debt/

[4] Vard Antinyan, Miroslaw Staron, and Anna Sandberg. 2017. Evaluating code complexity triggers, use of complexity measures and the influence of code complexity on maintenance time. *Empirical Software Engineering* (03 2017). https://doi.org/10.1007/s10664-017-9508-2

[5] Swapnil Bhartiya. 2018. *The Power of Zephyr RTOS.* The Linux Foundation. Retrieved Out 07, 2020 from https://www.linuxfoundation.org/blog/2018/12/the-power-of-zephyr-rtos/#:~:text=The%20Zephyr%20Project%20is%20a,specifically%20with%20security%20in%20mind.

[6] H. Birkholz, C. Vigano, and C. Bormann. 2019. *Concise Data Definition Language (CDDL): A Notational Convention to Express Concise Binary Object Representation (CBOR) and JSON Data Structures.* RFC 8610. RFC Editor.

[7] S. R. Chidamber and C. F. Kemerer. 1994. A metrics suite for object oriented design. *IEEE Transactions on Software Engineering* 20, 6 (June 1994), 476–493.

[8] Patrick Th. Eugster, Pascal A. Felber, Rachid Guerraoui, and Anne-Marie Kermarrec. 2003. The Many Faces of Publish/Subscribe. *ACM Comput. Surv.* 35, 2 (June 2003), 114–131. https://doi.org/10.1145/857076.857078

[9] Linux Foundation. 2020. *Zephyr Project*. Linux Foundation. Retrieved May 13, 2020 from https://www.zephyrproject.org/

[10] G. K. Gill and C. F. Kemerer. 1991. Cyclomatic complexity density and software maintenance productivity. *IEEE Transactions on Software Engineering* 17, 12 (1991), 1284–1288.

[11] G. K. Gill and C. F. Kemerer. 1991. Cyclomatic complexity density and software maintenance productivity. *IEEE Transactions on Software Engineering* 17, 12 (1991), 1284–1288.

[12] S. Gorappa, J. A. Colmenares, H. Jafarpour, and R. Klefstad. 2005. Tool-based configuration of real-time CORBA middleware for embedded systems. In *Eighth IEEE International Symposium on Object-Oriented Real-Time Distributed Computing (ISORC'05)*. 342–349.

[13] Leslie Lamport. 1985. On Interprocess Communication-Part I: Basic Formalism, Part II: Algorithms. *Distributed Computing. Also appeared as SRC Research Report 8.* (December 1985), 77–101. https://www.microsoft.com/en-us/research/publication/interprocess-communication-part-basic-formalism-part-ii-algorithms/

[14] Robert Love. 2005. Get on the D-BUS. *Linux J.* 2005, 130 (Feb. 2005), 3.

[15] H. Marzi, L. Hughes, and Yanting Lin. 2009. Embedded systems with improved Interprocess Communication design. In *2009 7th IEEE International Conference on Industrial Informatics*. 200–203.

[16] Daniele Miorandi, Sabrina Sicari, Francesco [De Pellegrini], and Imrich Chlamtac. 2012. Internet of things: Vision, applications and research challenges. *Ad Hoc Networks* 10, 7 (2012), 1497 – 1516. https://doi.org/10.1016/j.adhoc.2012.02.016

[17] Brian A. Nejmeh. 1988. NPATH: A Measure of Execution Path Complexity and Its Applications. *Commun. ACM* 31, 2 (Feb. 1988), 188–200. https://doi.org/10.1145/42372.42379

[18] Ragunathan Rajkumar, Michael Gagliardi, and Lui Sha. 1995. The Real-Time Publisher/Subscriber Inter-Process Communication Model for Distributed Real-Time Systems: Design and Implementation. 66–75. https://doi.org/10.1109/RTTAS.1995.516203

[19] Michael R. Schoettler. 2017. *A publish-subscribe framework for embedded systems: simplifying the development process*. Master's thesis. University of California, Irvine.

[20] M. Shepperd. 1988. A critique of cyclomatic complexity as a software metric. *Software Engineering Journal* 3, 2 (1988), 30–36.

[21] Kristoffer Rist Skøien. 2019. *RTOS: Real-Time Operating Systems for Embedded Developers*. Nordic. Retrieved Out 07, 2020 from https://blog.nordicsemi.com/getconnected/what-is-rtos-real-time-operating-systems-for-embedded-developers

[22] Eugene Syriani, Lechanceux Luhunu, and Houari Sahraoui. 2018. Systematic mapping study of template-based code generation. *Computer Languages, Systems & Structures* 52 (2018), 43 – 62. https://doi.org/10.1016/j.cl.2017.11.003

[23] A. Taivalsaari and T. Mikkonen. 2018. A Taxonomy of IoT Client Architectures. *IEEE Software* 35, 3 (2018), 83–88.

[24] Jiafu Wan, Hehua Yan, Hui Suo, and Fang Li. 2011. Advances in Cyber-Physical Systems Research. *TIIS* 5 (01 2011), 1891–1908. https://doi.org/10.3837/tiis.2011.11.001

[25] B. Zalila, L. Pautet, and J. Hugues. 2008. Towards Automatic Middleware Generation. In *2008 11th IEEE International Symposium on Object and Component-Oriented Real-Time Distributed Computing (ISORC)*. 221–228.