Universidade Federal de São Carlos– UFSCar
Centro de Ciências Exatas e de Tecnologia– CCET
Departamento de Computação– DC
Programa de Pós-Graduação em Ciência da Computação– PPGCC

Bento Rafael Siqueira

Microcontroladores: definição e implementação de controladores estruturalmente flexíveis para sistemas adaptativos

São Carlos
2023

Bento Rafael Siqueira

Microcontroladores: definição e implementação de controladores estruturalmente flexíveis para sistemas adaptativos

Tese apresentada ao Programa de Pós-Graduação em Ciência da Computação do Centro de Ciências Exatas e de Tecnologia da Universidade Federal de São Carlos, como parte dos requisitos para a obtenção do título de Doutor em Ciência da Computação.

Área de concentração: Metodologias e Técnicas de Computação

Orientador: Fabiano Cutigi Ferrari
Coorientador: Rogério de Lemos

São Carlos
2023

UNIVERSIDADE FEDERAL DE SÃO CARLOS

Centro de Ciências Exatas e de Tecnologia
Programa de Pós-Graduação em Ciência da Computação

Folha de Aprovação

Defesa de Tese de Doutorado do candidato Bento Rafael Siqueira, realizada em 14/10/2022.

Comissão Julgadora:

Prof. Dr. Fabiano Cutigi Ferrari (UFSCar)

Prof. Dr. Daniel Lucrédio (UFSCar)

Prof. Dr. Auri Marcelo Rizzo Vincenzi (UFSCar)

Prof. Dr. Nabor das Chagas Mendonca (UNIFOR)

Prof. Dr. Juliano Zanuzzio Blanco (IFSP)

O Relatório de Defesa assinado pelos membros da Comissão Julgadora encontra-se arquivado junto ao Programa de Pós-Graduação em Ciência da Computação.

Resumo

Contexto: Este trabalho parte da hipótese de que controladores para sistemas adaptativos que são particionados em elementos, neste trabalho chamados *microcontroladores*, que sejam dedicados a cada serviço envolvido nos estágios de uma malha de controle têm a vantagem da flexibilidade estrutural sem comprometer a reconfiguração e o desempenho do sistema alvo. **Objetivo:** Este trabalho tem por objetivo propor e avaliar uma abordagem para projetar controladores flexíveis estruturalmente (isto é, baseado em microcontroladores), investigando ganhos da abordagem ao comparar diferentes configurações de controladores. **Metodologia:** Para se atingir o objetivo definido para este trabalho, a seguinte série de atividades foi realizada: (1) definição de controladores flexíveis como microcontroladores independentes, implementados como microsserviços em malhas de controle; (2) condução de um estudo exploratório, realizando análise qualitativa quanto à reconfiguração do controlador; (3) realização de estudos comparativos para avaliar a abordagem proposta neste trabalho frente às abordagens da literatura; e (4) especificação de uma implementação utilizando a abordagem proposta neste trabalho, com o intuito de demonstrar a flexibilidade da mesma. **Resultados:** Os resultados e contribuições deste trabalho são: uma abordagem para a definição de controladores flexíveis estruturalmente; um estudo exploratório com análise qualitativa sobre reconfiguração em tempo de projeto e execução; um conjunto de estudos experimentais realizando comparações de abordagens; e uma proposta de evolução de um controlador em multicamadas como forma de demonstrar a efetividade da abordagem proposta. **Conclusão:** Conclui-se que o projeto de um controlador em multicamadas, baseado em microcontroladores, provê a base para definir controladores flexíveis estruturalmente em tempo de execução e pode promover reúso no tempo de projeto. Além disso, mesmo demandando mais recursos computacionais, a abordagem não compromete o desempenho e a reconfiguração do sistema alvo.

Abstract

Context: This work starts from the hypothesis that controllers for adaptive systems that are partitioned into elements, in this work called *microcontrollers*, which are dedicated to each service involved in the stages of a control loop have the advantage of structural flexibility without compromising the reconfiguration and performance of the target system. **Objective:** This work aims to propose and evaluate an approach to develop controllers that are structurally flexible (*i.e.* based on microcontrollers), investigating advantages of the approach by comparing with different configurations of controllers. **Methodology:** In order to achieve the objective, the following activities were performed: (1) definition of flexible controllers by promoting them as a set of independent microcontrollers, demonstrating them as microservices implementations of the control loops; (2) conduction of an exploratory study and associated qualitative analysis regarding the reconfigurations of the controller; (3) execution of comparative studies to evaluate the approach that is being proposed against other approaches found in the literature; and (4) specification of an implementation by using the microcontroller-based approach, to demonstrate the flexibility of the approach. **Results:** the results and contributions are: an approach to define controllers that are structurally flexible; an exploratory study involving a qualitative analyse about reconfiguration in design time and runtime; a set of comparative studies evaluating different approaches from the literature; a proposal involving the evolution of the multi-layered controller to demonstrate the feasibility of the approach of this work. **Conclusion:** As conclusion, a multi-layered controller design, based on micro-controllers, provides the basis for defining structurally flexible controllers at operational-time, and may promote reuse at development-time. In addition, even demanding more computational resources, the approach does not impact on the performance and on the reconfiguration of the target system.

Agradecimentos

"O presente trabalho foi realizado com apoio da Coordenação de Aperfeiçoamento de Pessoal de Nível Superior – Brasil (CAPES) – Código de Financiamento 001"

Listagens de Códigos

Lista de Figuras

Lista de Tabelas

Lista de Abreviaturas e Siglas

A-FSM	*Adaptation Finite-State Machine*
ACME	*Architectural Description of Component-Based Systems*
API	*Application Programming Interface*
Des-KZ	Configuração decentralizada utilizando o Kubow
EUREMA	*ExecUtable RuntimE MegAmodels*
GPS	*Global Positioning System*
HTTP	*Hypertext Transfer Protocol*
Kube-ZNN	Kubernetes + ZNN.com
Kubow	Kubernetes + Rainbow
MAPE-K	*Monitor, Analyser, Planner, Executor and Knowledge*
Meta-KZ	Configuração decentralizada com um metacontrolador utilizando o Kubow
Mon-KZ	Configuração monolítica utilizando o Kubow
QP	Questão de Pesquisa
RSL	Revisão Sistemática da Literatura (*Systematic Review*)
REMaP	*RuntimE Microservices Placement*
SA	Sistema Adaptativo (*Adaptive System*)
SADT	*Structured Analysis for Requirements Definition*
SLO	*Service Level Objectives*
TCP-IP	*Transmission Control Protocol - Internet Protocol*

Sumário

Capítulo

1

Introdução

1.1 Contexto

Com a demanda de sistemas que necessitam lidar com variáveis de ambiente e, ainda, adaptarem-se de acordo com tais variáveis, surge a necessidade em desenvolver Sistemas Adaptativos (SAs) (Lalanda et al., 2013). SAs possuem duas partes conceitualmente distintas: o subsistema gerenciado – isto é, o sistema alvo (do inglês, *target system*) – e o subsistema gerenciador – isto é, o controlador. O *sistema alvo* é o sistema base, responsável por fornecer os requisitos esperados pelo usuário. O controlador é o que faz o sistema ser considerado adaptativo, pois ele tem a responsabilidade de monitorar o sistema base e disparar as adaptações. Além disso, o controlador é composto por malhas de controle (do inglês, *feedback control loops*), que é um conceito oriundo da teoria do controle (Garlan et al., 2004; Weyns et al., 2013). A malha de controle é um termo para abstrair as atividades realizadas pelo controlador de: monitorar dados do sistema alvo; analisar os dados monitorados; planejar ações a serem realizadas no sistema alvo e ambiente; e executar as ações planejadas.

No que diz respeito a controladores, *Rainbow* (Garlan et al., 2004) é um dos poucos exemplos de *frameworks* para ser desenvolver controladores. Pode-se destacar também a característica genérica do *framework* Rainbow, uma vez que pode ser utilizado em diversos domínios de aplicação (Schmerl et al., 2014) e por diferentes pesquisadores (Cámara et

al., 2013) (mais detalhes no Capítulo 2). Em geral, um fator que restringe o reúso de controladores é que estes são intrinsecamente dependentes do sistema alvo. Para mitigar tal dificuldade, *Rainbow* utiliza a abstração de um sistema alvo no nível de arquitetura; assim, pode-se reduzir o acoplamento entre o controlador e sistema alvo (Garlan et al., 2004). No entanto, o amplo contexto de um sistema alvo pode demandar uma ampla gama de serviços e funcionalidades que devem ser esperadas para os diferentes estágios de um controlador. Portanto, utilizar uma abordagem com o *Rainbow* para desenvolver um controlador que seja genérico e que este promova uma ampla gama de serviços é desafiador. Isso se deve à característica de tal controlador em ser monolítico e que é somente parametricamente configurável – que também é o caso da maioria dos controladores existentes (Krupitzer et al., 2016).

Um controlador cujos serviços são compostos de acordo com as necessidades atuais do sistema alvo pode demandar futura flexibilidade e variabilidade. Essas capacidades são difíceis de serem implementadas em um controlador monolítico, já que não somente adaptações paramétricas, mas também adaptações estruturais podem ser necessárias (McKinley et al., 2004). Neste caso, adaptações paramétricas e estruturais estão no nível do controlador, ou seja, o controlador deve estar apto para se reconfigurar parametricamente e/ou estruturalmente frente às necessidades do sistema alvo.

Na literatura, os trabalhos não enfatizam questões envolvendo níveis de flexibilidade (Banijamali et al., 2020; Baylov e Dimov, 2017; Florio e Di Nitto, 2016; Hassan e Bahsoon, 2016; Pereira et al., 2020; Sampaio Jr. et al., 2019). Um pouco disso remete-se ao emprego de abordagens com uma visão *top-down* da estrutura do controlador, dificultando-se a definição específica dos componentes do mesmo (Banijamali et al., 2020; Gerostathopoulos et al., 2019; Iftikhar e Weyns, 2014; Pereira et al., 2020). No que diz respeito à descentralização de controladores, os trabalhos dão ênfase à definição baseada no MAPE-K e de especificamente serem projetados para domínios particulares (por exemplo, sistemas que são baseados em microsserviços e sistemas baseado em serviços web) (Florio e Di Nitto, 2016; Nallur e Bahsoon, 2013; Sampaio Jr. et al., 2019). O reúso em si é abordado utilizando-se controladores monolíticos, o que inclui o uso de modelos que restringem a estrutura/comportamento do controlador e uso de modelos de componentes em execução.

Algumas iniciativas de controladores descentralizados utilizam o modelo MAPE-K como base (Weyns et al., 2013), que é um modelo de referência arquitetural para implementar malhas de controles de um controlador (IBM, 2005). Nessa linha de investigação, no grão mais fino da descentralização tem-se cada um dos estágios do MAPE-K. Porém, cada estágio do MAPE-K pode ser composto por várias atividades. Dependendo da diversidade nas atividades presentes em cada estágio do MAPE-K – ao utilizá-lo como

modelo de referência – necessita-se personalizar controladores. Então, com a aplicação de uma separação de interesses mais refinada, pode-se diminuir a complexidade do controlador; em um único estágio pode-se demandar diferentes atividades dependendo de *o quê* e *como* as mesmas devem ser realizadas para o estágio em particular (Weyns et al., 2013).

Ao adotar uma abordagem modular e flexível estruturalmente para construir um controlador genérico, os serviços individuais de um controlador devem se tornar estruturalmente independentes. Neste trabalho, demonstra-se a flexibilidade estrutural que se pode obter ao utilizar controladores descentralizados com apoio de um metacontrolador. Evidências já obtidas (Siqueira et al., 2020b), e estudos subsequentes, indicam que mesmo com o aumento da quantidade de recursos, obtém-se ganhos provenientes de se modularizar um controlador por meio dos aqui chamados *microcontroladores*. Dentre os ganhos, podem-se destacar (1) a possibilidade de criar microcontroladores específicos e enxutos que podem eventualmente ser reutilizados em diferentes controladores; e (2) uma vez que sejam microcontroladores enxutos e específicos, os mesmos podem utilizar somente respectivos recursos necessários – quando tais microcontroladores estiverem habilitados.

1.2 Hipótese e Questões de Pesquisa

Nesta tese, tem-se como hipótese que *um controlador que é baseado em microcontroladores tem a vantagem da flexibilidade estrutural sem comprometer a reconfiguração e o desempenho do sistema alvo.* Alinhando-se a essa hipótese, tem-se duas questões de pesquisa como se seguem.

- **QP1:** Um *controlador descentralizado* é vantajoso com relação a adaptações estruturais do sistema alvo quando comparado com um *controlador monolítico*?
- **QP2:** Um *controlador descentralizado* é vantajoso com relação a adaptações paramétricas do sistema alvo quando comparado com um *controlador monolítico*?

Ressalta-se que alocação de recursos e reconfiguração do sistema alvo estão, respectivamente, relacionadas a adaptações estruturais e paramétricas.

A fim de avaliar a hipótese definida e em busca de derivar respostas às questões de pesquisa, neste trabalho, utilizou-se inicialmente um controlador chamado *PhoneAdapter* (Sama et al., 2008) e a plataforma Android – isto é, uma plataforma que é instalada como um sistema operacional em *smartphones* repleta por recursos gerenciados (por exemplo, sensores tais como GPS e atuadores tais como modificadores de áudio do ambiente), os quais são manipulados pelo *framework* Android e pelo *PhoneAdapter*. Ressalta-se, assim,

que a plataforma Android foi utilizada como sistema alvo em um estudo piloto (Siqueira et al., 2020b). Em um estudo subsequente e mais robusto, utilizou-se um controlador implementado a partir do *framework Kubow* (Aderaldo et al., 2019) e o sistema alvo Kube-ZNN (Aderaldo e Mendonça, 2022).

Expectativa: Uma vez que controladores sejam compostos por microcontroladores, espera-se que tais microcontroladores possam ser reutilizáveis em diferentes controladores. Além disso, a expectativa é que mesmo com o aumento do uso de recursos demandados pelos microcontroladores, as recomposições gerenciadas pelo metacontrolador resultem em controladores compostos por microcontroladores específicos, de modo que tais microcontroladores possam lidar de forma mais eficaz com as suas respectivas funções específicas quando comparados com um controlador monolítico. Por outro lado, um controlador com vários microcontroladores e com uma camada adicional do metacontrolador – demandando mais recursos – pode eventualmente lidar com problemas de alocação de recursos e a reconfiguração do sistema alvo. Além disso, uma vez que a definição de microcontroladores demanda eventualmente propriedades similares entre tais microcontroladores, espera-se um grau de flexibilidade que permita a reutilização de código entre a definição de microcontroladores.

1.3 Objetivos

O objetivo deste trabalho é introduzir e avaliar uma abordagem para projetar controladores que consistem de uma coleção independente de microcontroladores. Com isso, os objetivos específicos são:

1. A proposição de uma abordagem para projetar controladores que podem ser sintetizados a partir de conjuntos independentes de microcontroladores.

2. A condução de um estudo exploratório utilizando microcontroladores e um metacontrolador para ilustrar a manipulação de mudanças em *tempo de execução* e em *tempo de projeto*.

3. A condução de estudos exploratórios para comparar diferentes configurações de controladores, a fim de avaliar o desempenho da abordagem deste trabalho frente ao estado da arte.

4. A especificação de um microcontrolador de teste de regressão, a fim de demonstrar a flexibilidade da abordagem em incluir diferentes estágios de uma malha de controle que podem incluir/personalizar funcionalidades a um controlador, bem como definir

microcontroladores que podem ser reutilizáveis em diferentes controladores e sistemas alvo.

1.4 Justificativa

Em relação ao objetivo 1, o qual envolve a a proposta de controladores flexíveis, propõem-se a criação de controladores utilizando-se microcontroladores baseados em microsserviços como forma de se beneficiar das características advindas de microsserviços – tais como auxílio no reuso e especificidade de funcionalidades. Assim, viabiliza-se a recomposição do controlador em tempo de execução.

Em relação ao objetivo 2, o estudo exploratório forneceu subsídios iniciais de que a abordagem de recompor controladores em tempo de execução funciona. Além disso, tal estudo demonstrou até eventuais dificuldades que seriam encontradas na abordagem – como a de lidar com inconsistências (por exemplo, geradas no compartilhamento de recursos) geradas na recomposição do controlador.

O conjunto de estudos experimentais relacionados ao objetivo 3 forneceu comparações de diferentes abordagens de controladores. Assim, pôde-se observar que mesmo que a abordagem de microcontroladores eventualmente utilizasse mais recursos que uma abordagem monolítica, isso não impactou o desempenho e funcionalidades do sistema alvo. Além disso, as diferentes configurações implementadas (ou seja, monolítica e descentralizada) puderam demonstrar como que o controlador funciona quando não há um metacontrolador. Em especial, com relação à versão descentralizada, observa-se que ela poderia ter um desempenho melhor com a presença de um metacontrolador. Vale ressaltar que as implementações que não têm um metacontrolador contém código-fonte com funcionalidades "misturadas", como é o caso da abordagem descentralizada em que um componente é responsável tanto por sua principal funcionalidade (por exemplo, mudar a qualidade de mídia do ZNN.com) quanto por ativar/desativar componentes do controlador. Quando essa ativação/desativação fica a cargo do metacontrolador (como demonstrado na terceira implementação do estudo), o código-fonte dos microcontroladores torna-se mais enxuto.

Por fim, com respeito ao objetivo 4, buscou-se explorar a perspectiva de se incluir novas funcionalidades, por meio de microcontroladores, para fazerem parte da recomposição de um controlador em tempo de execução. Para tal, foi elaborado um microcontrolador de teste de regressão que poderia fazer parte do controlador. Assim, uma Revisão Sistemática foi conduzida a fim de identificar abordagens de teste que são relacionadas ao teste de regressão – uma vez que qualquer adaptação em tempo de execução gera demandas de teste de regressão.

1.5 Metodologia

Objetivo específico (1) - para atingir esse objetivo, teve-se como proposta a definição de uma abordagem na qual se tem uma hierarquia multicamadas em que o controlador também é alvo de adaptação, não somente o sistema alvo. Sendo assim, surge a definição de um controlador composto por uma série de microcontroladores– baseados em atividades e estágios específicos de uma malha de controle – que são gerenciados por um nível superior (o chamado *metacontrolador*). Para isso, neste trabalho, adota-se o uso de microsserviços para definição de atividades que fazem parte de estágios de uma malha de controle (por exemplo, como o modelo MAPE-K). Assim, utilizam-se tecnologias que auxiliam a definição de microcontroladores e um metacontrolador como microsserviços que possam ser configurados para atuar em ambientes de execução com o sistema alvo.

Atividades metodológicas
Nesta etapa investigaram-se tecnologias e conceitos para tornar viável a implementação de microcontroladores. Assim, concluiu-se que o uso de microsserviços era o mais adequado. Isso, devido às características inerentes a eles (isto é, reuso e especificidade) serem uma opção coerente para se trabalhar com a proposta de microcontroladores.

Objetivo específico (2) - para atingir esse objetivo, conduziu-se um estudo exploratório utilizando-se o sistema *PhoneAdapter* e a plataforma Android (isto é, o sistema operacional baseado no linux para dispositivos móveis da Google (Android, 2022)). No estudo, foram realizadas implementações com a inclusão de um metacontrolador e de um conjunto de microcontroladores. Com isso, investigou-se a composição de diferentes controladores personalizados utilizado o conjunto de microcontroladores. Tais composições foram investigadas e levantaram-se questões a respeito da composição de controladores em tempo de projeto e em tempo de execução.

Atividades metodológicas
Nesta etapa escolheu-se a plataforma Android para o desenvolvimento da abordagem de microcontroladores. Com a plataforma, utilizaram-se serviços do Android para realizar a definição de microcontroladores e de um metacontrolador. O serviço metacontrolador tinha como objetivo utilizar mensagens do Android, contendo dados de sensores, para realizar as recomposições de diferentes controladores em tempo de execução.

Objetivo específico (3) - para atingir esse objetivo, conduziram-se três estudos exploratórios utilizando-se o sistema alvo Kube-ZNN, o *framework Kubow*, e a plataforma Kubernetes. O primeiro estudo envolveu implementações e adequações de um controlador monolítico. O segundo estudo envolveu implementações de um controlador descentralizado. Por fim, o terceiro estudo envolveu implementações de um controlador descentralizado com um metacontrolador. Nos três estudos realizaram-se testes de carga a fim de gerar dados quantitativos para a comparação de desempenho entre as configurações de controladores com seus respectivos sistemas alvo.

Atividades metodológicas
Nesta etapa, à luz da plataforma Kubernetes e de microsserviços, utilizou-se o arcabouço Kubow (mais detalhes podem ser encontrados no Capítulo 2) para a definição de loops de controle pertencentes a cada microcontrolador e metacontrolador. Depois, utilizou-se o sistema alvo ZNN.com (também descrito no Capítulo 2) como base para as diferentes configurações de controladores. Por fim, implementaram-se funcionalidades de gerenciamento de falhas em cada controlador a fim de realizar comparações em experimentos relacionados à geração de falhas em tempo de execução.

Objetivo específico (4) - para atingir esse objetivo, conduziu-se uma revisão sistemática da literatura em teste para sistemas adaptativos. Desta revisão, selecionaram-se os estudos especificamente sobre o teste de regressão, no contexto de *tempo de execução*. Por fim, especificou-se um conjunto de modelos a fim de serem utilizados para a construção de um microcontrolador de teste de regressão que pode ser incluído em diferentes controladores.

Atividades metodológicas
Nesta etapa, utilizaram-se ideias, tecnologias e conceitos advindos dos experimentos realizados com a plataforma Kubernetes, as abordagens empregadas nos experimentos e do arcabouço *Kubow*. Para demonstrar como a abordagem permite a inclusão de novos microcontroladores, propô-se a definição de um microcontrolador de teste de regressão para ser acoplado e desacoplado no tempo de execução. Assim – para caracterizar as características do microcontrolador – realizou-se uma revisão sistemática da literatura, envolvendo as atividades: 1) identificação de estudos pilotos para identificar a precisão das *strings* de busca por trabalhos; 2) realização da seleção inicial de estudos, envolvendo a análise de títulos e resumos de trabalhos; 3) realização da seleção de estudos, envolvendo a análise dos trabalhos na íntegra e, também, a 4) extração de dados para ajudar na análise dados, caracterização do estado da arte e respostas às questões de pesquisa.

1.6 Organização do Trabalho

Esta tese está organizada em oito capítulos. O próximo é o Capítulo 2, contendo os principais conceitos envolvidos nos estudos realizados. O Capítulo 3 contém os principais trabalhos encontrados no estado da arte sobre controladores para sistemas adaptativos. O Capítulo 4 contém o detalhamento da abordagem proposta neste trabalho. O Capítulo 5 contém descrições referentes aos estudos realizados com o sistema *PhoneAdapter* e a plataforma Android. O Capítulo 6 contém as descrições envolvendo os estudos conduzidos que utilizaram o sistema alvo Kube-ZNN, o *framework Kubow* e a plataforma Kubernetes. O Capítulo 7 contém um extrato de resultados e análise da revisão sistemática sobre teste de sistemas adaptativos, assim como uma especificação de um microcontrolador de teste de regressão para sistemas adaptativos. Por fim, o Capítulo 8 conclui este trabalho, também destacando as principais lições aprendidas, limitações e possibilidades de trabalhos futuros.

Capítulo

2

Fundamentos

2.1 Considerações Iniciais

Para a compreensão da abordagem definida neste trabalho e dos experimentos conduzidos, fazem-se necessárias a definição de conceitos e as descrições de tecnologias e ambientes de execução que auxiliem em fundamentar os estudos que são relatados. Neste capítulo, na Seção 2.2, são introduzidos os principais fundamentos acerca de sistemas adaptativos, para contextualizar controladores, sistemas alvo e seus tipos de comunicação. Na sequência, na Seção 2.3 detalham-se algumas implementações de controladores e *frameworks* para viabilizar as implementações de controladores. Na Seção 2.4 descrevem-se sistemas alvos utilizados nos experimentos conduzidos neste trabalho. Por fim, as considerações finais são apresentadas na Seção 2.5. Ressalta-se que os conceitos de embasamento associados a cada um dos tópicos abordados são também apresentados nas respectivas seções.

2.2 Sistemas Adaptativos

Na literatura, podem-se identificar descrições de sistemas que estão aptos a avaliarem o próprio comportamento e mudá-lo (isto é, adaptá-lo), quando a avaliação indica que o software não está executando de acordo com o esperado (ou mesmo uma melhor

funcionalidade/desempenho foi identificada). Esses sistema que estão aptos a modificar o próprio comportamento e estrutura são os chamados sistemas adaptativos (SAs).

SAs podem expor uma ou mais das seguintes propriedades: autoconfiguração, auto-cura, auto-otimização e autoproteção (Kephart e Chess, 2003) (isto é, *Self-configuring*, *Self-healing*, *Self-optimising* e *Self-protecting*). Em outras palavras, um SA pode estar apto a modificar não somente seu comportamento, mas também a modificar sua própria estrutura a fim de obter um melhor funcionamento do sistema, ou mesmo melhorando o nível de qualidade de serviço do sistema (Salehie e Tahvildari, 2009).

Para que diferentes tipos de adaptações sejam realizadas em sistemas, pode-se utilizar uma estrutura baseada em controladores específicos (De Lemos et al., 2017). Neste sentido, na sequência apresentam-se os conceitos envolvendo controladores de SAs, com foco em controladores que são abordados neste trabalho.

2.2.1 Controladores de Sistemas Adaptativos

Um SA é tipicamente composto por dois sistemas, o controlador e o sistema alvo. Para realizar uma adaptação, o controlador realiza os estágios de monitoramento, análise, planejamento e execução de mudanças no sistema alvo. Para se implementar tais estágios, tem-se utilizado bastante o modelo de referência MAPE-K para a estruturação de um controlador de malha fechada (IBM, 2005) O MAPE-K (*Monitor, Analyzer, Planner, Executor - Knowledge-based*) é um modelo conceitual de *malha* de controle e serve como base para a estruturação de SAs especificando seus elementos conceituais. Em geral, o controlador utiliza sensores pare obter dados do sistema alvo e ambiente, e atuadores para afetar o sistema alvo.

Na Figura 2.1, o controlador implementa os estágios do MAPE-K a fim de monitorar (via sensores) e adaptar (via atuadores) um sistema alvo. O estágio *Monitor* permite a obtenção do estado do sistema alvo e do ambiente. O estágio *Analisador* analisa o estado do sistema alvo e o ambiente, a fim de (i) decidir se uma adaptação deve ser realizada; e (ii) identificar ações apropriadas se uma adaptação for requerida. O estágio *Planejador* é responsável por (i) selecionar as melhores ações alternativas para a dada adaptação; e (ii) gerar o plano que realizará as respectivas ações (isto é, respectivamente, *Tomador de Decisão* e *Síntese do Planejamento*). O estágio *Executor* executará os planos que implementam as ações de adaptação do sistema alvo. Por fim, o *Conhecimento* (*Knowledge*) representa o estado do sistema alvo e do ambiente.

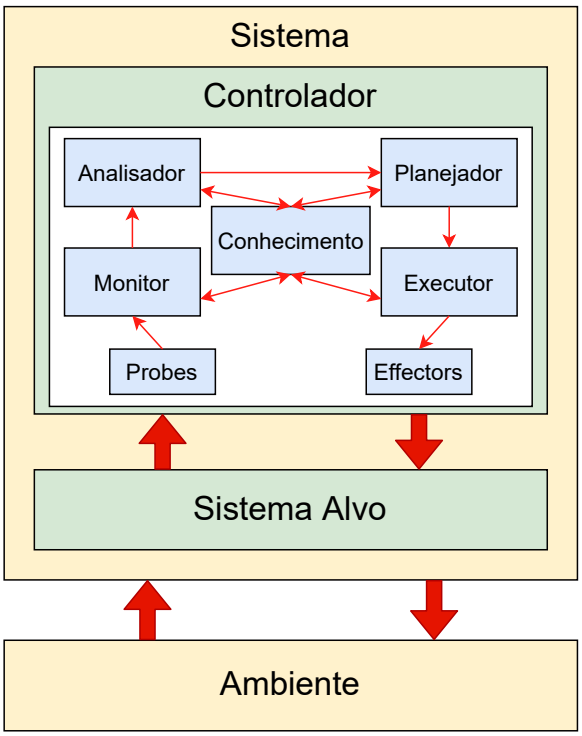

Figura 2.1: Modelo de referência MAPE-K – adaptado do trabalho de IBM (2005)

2.2.2 Tipos de Interação nos Controladores

Os componentes e estágios presentes nos sistemas controlador e sistema alvo podem utilizar abordagens baseadas em tempo ou eventos como maneiras de efetuar interações (notificações) entre os mesmos (de la Iglesia e Weyns, 2015; Weyns et al., 2013). Na Figura 2.2 ilustra-se um exemplo desses tipos de interações realizadas.

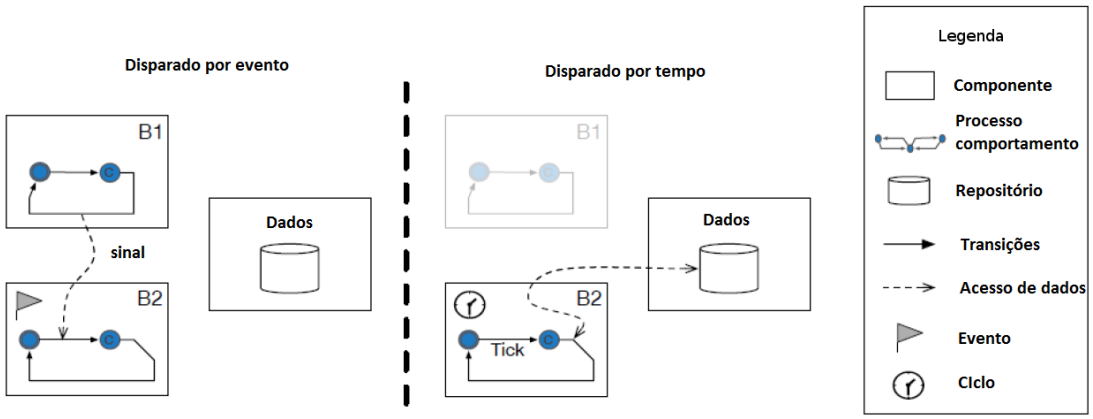

Figura 2.2: Interação entre componentes de um sistema por meio de evento e ciclo de tempo – adaptada do trabalho de de la Iglesia e Weyns (2015).

Na Figura 2.2 são ilustradas duas abordagens. Na primeira, em que se utiliza disparo por evento (lado esquerdo da figura), B1 envia um sinal para B2 que, por sua vez, procede de acordo com a chamada de B1, podendo iniciar um respectivo estágio MAPE-K. Na segunda abordagem, representada no lado direito da figura, utiliza-se disparo por tempo. Nessa abordagem, existe um temporizador responsável por realizar chamadas após um determinado tempo ter decorrido. Ao lidar com um sistema que é composto por um sistema alvo e um controlador as iterações baseadas em tempo e eventos são intrínsecas (de la Iglesia e Weyns, 2015; Weyns et al., 2013).

2.3 Exemplos de Implementações e Frameworks para Controladores

Uma vez que se tem o MAPE-K como modelo teórico para implementação de controladores, diversas são as implementações possíveis baseadas nesse modelo. Alguns exemplos são as implementações *PhoneAdapter* (Liu, 2013), *Rainbow* (Garlan et al., 2004), e *Kubow* (Aderaldo et al., 2019), as quais são utilizadas nos experimentos conduzidos no escopo deste trabalho de doutoramento.

Para se implementar um SA, os tipos de interação baseada em tempo e evento são utilizados em controladores. Tanto no *PhoneAdapter* quanto no *Rainbow* e no *Kubow*, as interações ocorrem via troca de mensagens entre diferentes serviços. Nestes sistemas, a comunicação é realizada essencialmente com base em tempo. No caso dos gerenciadores de contexto e adaptação do *PhoneAdapter*, os mesmos contém temporizadores que periodicamente realizam leitura de dados de sensores e realizam adaptações. No caso dos *frameworks Rainbow* e *Kubow*, também utilizando uma comunicação baseada em tempo, as mensagens são obtidas por meio de protocolos como HTTP e TCP-IP e as adaptações são baseadas em arquitetura (utilizando-se as linguagens Stich e ACME). Mais especificamente, nos gauges são especificados períodos de monitoramento e, posteriormente, a comunicação baseada em eventos é realizada com base em estratégias de adaptação definidas. Mais detalhes de cada implementação são apresentados não seções seguintes. Em particular, na Seção 2.3.1 descreve-se o *PhoneAdapter* e as características de sua implementação. Na Seção 2.3.2 descreve-se o *Rainbow* e como é realizado o uso de sua arquitetura. Por fim, na Seção 2.3.3 descrevem-se tecnologias e ferramentas que foram incorporadas no *framework Kubow* como forma de se estruturar e implementar com microsserviços controladores baseados no Rainbow.

2.3.1 PhoneAdapter

O *PhoneAdapter* é uma implementação de um controlador que foi proposto pela comunidade de Aplicações Adaptativas Sensíveis ao Contexto para exemplificar a definição de regras de adaptação (Sama et al., 2008). Trata-se de aplicação móvel para dispositivos Android que permite que usuários configurem perfis e regras. Os perfis referem-se a diferentes contextos e as regras referem-se a regras de adaptação que adaptam recursos do Android. Para realizar adaptações, o *PhoneAdapter* contém dois principais componentes: um gerenciador de contexto (ou seja, *Context Manager*) que coleta e mantém dados de contextos; e um gerenciador de adaptação (ou seja, *Adaptation Manager*) que adapta o sistema alvo de acordo com o dado de contexto e um conjunto de regras de adaptação. Ressalta-se que uma mensagem que é trocada entre os diferentes serviços no Android é nomeada de *Intent*, sendo ela uma forma já padronizada de realizar comunicação entre diferentes aplicação na plataforma Android (Android, 2022).

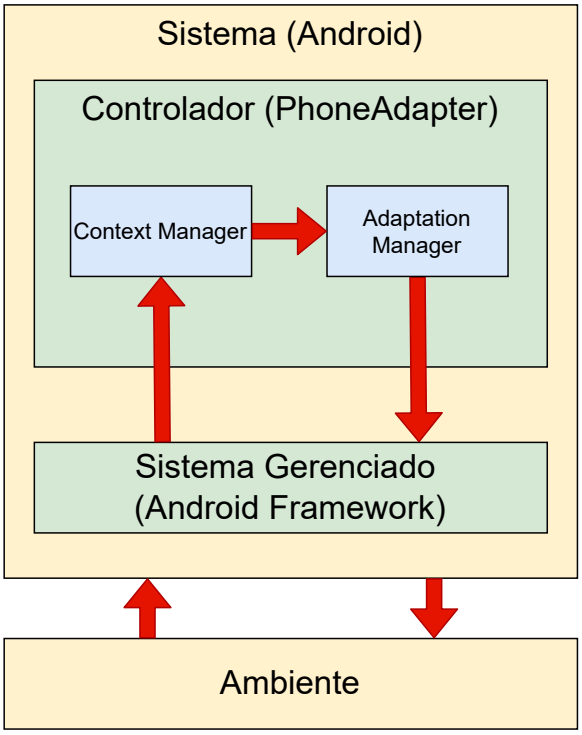

Figura 2.3: Versão original do *PhoneAdapter* ilustrada em um trabalho anterior (Siqueira et al., 2020b).

Na Figura 2.3 ilustra-se uma visão geral da versão original do *PhoneAdapter* (Sama et al., 2008). O elemento PhoneAdapter é o controlador, ao passo que o Android Framework é o sistema alvo. Portanto, o controlador contém dois componentes: o ContextManager

e o AdaptationManager. O componente ContextManager é responsável por identificar o contexto do dispositivo móvel dependendo dos valores de sensores fornecidos pelo Android Framework. Por outro lado, o componente AdaptationManager é responsável por controlar os recursos do AndroidFramework (por exemplo, volume do áudio) com uso de atuadores dependendo do contexto identificado.

No estudo de caso original (Sama et al., 2008) assume-se que não existem interações entre o usuário e o PhoneAdapter em tempo de execução. Assim, não existe interação direta entre o ambiente do dispositivo móvel e o PhoneAdapter.

2.3.2 Rainbow

Rainbow provê uma infraestrutura reutilizável – ou seja, consiste em um *framework* – e um conjunto de ferramentas para apoiar a auto-adaptação, assim requerendo pouco esforço de personalização para diferentes sistema alvo (Garlan et al., 2004). Rainbow contém vários componentes para armazenar o modelo arquitetural do sistema alvo, avaliando a necessidade para uma adaptação, selecionando a adaptação apropriada e executando a respectiva adaptação. Na Figura 2.4 ilustram-se as principais camadas e principais componentes presentes no controlador (isto é, o componente identificado como Controller na figura). Rainbow conta com um modelo de arquitetura do sistema alvo, e uma coleção de estratégias de adaptação. Para definir os modelos de arquitetura utilizados pelo Rainbow, utiliza-se a linguagem ACME (Garlan et al., 2000) que é responsável por descrever a arquitetura do sistema alvo que é manipulado pelo controlador. Já para a definição de estratégias de adaptação, utiliza-se a linguagem Stich que é responsável por efetivamente realizar as adaptações necessárias no sistema alvo (Cheng e Garlan, 2012). Outros componentes também importantes no Rainbow são as *probes* (gauges) e os *effectors* (atuadores), sendo que as *probes* periodicamente atualizam o modelo arquitetural em ACME. Já os *effectors*, de acordo com as estratégias e táticas escalonadas, efetivam as adaptações demandadas no sistema alvo.

Apesar do Rainbow apoiar o desenvolvimento de controladores genéricos, controladores baseados em Rainbow são tipicamente monolíticos. Isto se deve ao fato de que os componentes são utilizadas como um todo – tais componentes podem ser configurados, mas não podem ser modificados (por exemplo, para incorporar novos serviços).

O *Rainbow*, proposto por Garlan et al. (2004), surgiu como uma infraestrutura para desenvolver sistemas adaptativos para diferentes domínios. Alinha-se a isso a possibilidade de redução de custos relacionados ao desenvolvimento de um controle externo ao sistema

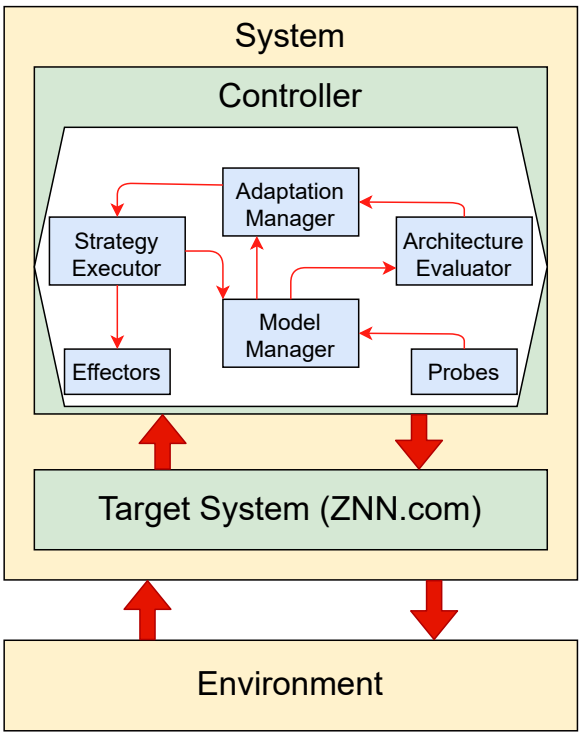

Figura 2.4: Componentes do *framework* Rainbow - inspirada do trabalho de Garlan et al. (2004).

alvo, diminuindo a criação de diferentes monitoramentos, modelagens e mecanismos de detecção de problemas para cada novo sistema envolvido.

Em linhas gerais, o *Rainbow* adota uma abordagem baseada em arquitetura que provê uma infraestrutura reutilizável, junto com mecanismos para especializar a infraestrutura com base nas especificidades do sistema. Os mecanismos para serem especializados auxiliam o desenvolvedor a escolher características de adaptação tais como: (i) quais são os modelos e monitoramentos o sistema terá; (ii) quais são as condições utilizadas para o sistema realizar adaptações; e (iii) como o sistema realizará adaptações.

Visando a facilitar o uso do *framework* Rainbow, Aderaldo et al. (2019) propuseram o *Kubow* como alternativa, o qual é apresentado na próxima seção.

2.3.3 Kubow

No trabalho de Aderaldo et al. (2019) foi proposto o *framework Kubow*[1] que é a adaptação do *Rainbow* utilizando o Kubernetes[2]. Dessa forma, uma instância *Kubow* (ou seja, um

[1] `https://github.com/ppgia-unifor/kubow` – acessado em setembro, 2022
[2] `https://kubernetes.io/` – acessado em setembro, 2022.

controlador que é baseado em *Kubow*) é também implementado como uma infraestrutura de microsserviços.

Em geral, o *Kubow* foi implementado ao personalizar e estender o *Rainbow* com suporte a contêineres Docker[3] e Kubernetes (Aderaldo et al., 2019). Esta personalização e extensão envolvem implementações, tais como probes e efetores do Rainbow, usando-se as APIs Kubernetes. Estas APIs permitem acessar diferentes tipos de recursos gerenciados pelo Kubernetes (por exemplo, *pods*). Consequentemente, todos os componentes *Rainbow* são instâncias *Kubow* dentro de um *Cluster* Kubernetes. Além disso, o *Kubow* provê ferramentas que permitem a integração de outros tipos de aplicações Kubernetes. Por exemplo, podem-se acessar métricas coletadas pelos próprios serviços de monitoramento do Kubernetes (por exemplo, Kubelet e cAdvisor), bem como ao utilizar outras ferramentas de monitoramento externas (por exemplo, Prometheus).

Os principais recursos do *Kubow* são (Aderaldo et al., 2019):

- personalizações do *Rainbow* para viabilizar sua implantação em um *cluster* Kubernetes;
- personalizações do *Rainbow* para lidar com APIs do Kubernetes; e
- configuração de ferramentas de monitoramento no *cluster* Kubernetes, sendo elas:
  - Prometheus,[4]
  - Metrics server[5]; e
  - Grafana.[6]

O *Kubow* pode ser implantado em um *cluster* Kubernetes. Isso permite o desenvolvimento e integração de outros controladores e aplicações, uma vez que o Kubernetes fornece uma rede que apoia tecnologias tais como (Kubernetes, 2022): (i) comunicação de contêineres dentro de um *pod* via rede; (ii) *pods* se comunicando uns com os outros; (iii) *pods* sendo acessados de fora do *cluster* por aplicações terceiras; e (iv) opções de serviço somente para consumo dentro do próprio *cluster*.

[3] `https://www.docker.com/` - acessado em setembro, 2022.
[4] `https://prometheus.io/` – acessado em setembro, 2022
[5] `https://github.com/kubernetes-sigs/metrics-server` – acessado, setembro, 2022
[6] `https://grafana.com/` – acessado em setembro, 2022

2.4 Android e Kube-ZNN: Sistemas Alvos Utilizados neste Trabalho

Nesta seção apresentam-se alguns dos principais conceitos utilizados nos sistemas alvos utilizados neste trabalho. Na Seção 2.4.1 abordam-se os recursos gerenciados utilizados, bem como estão classificados na plataforma Android. Na Seção 2.4.2 abordam-se os principais atributos relacionados ao ZNN.com – que consiste em uma aplicação Web adaptativa – e a sua versão que foi portada para o Kubernetes (Aderaldo et al., 2019).

2.4.1 Android

Uma vez que se pode desenvolver novas aplicações e serviços na chamada pilha de software da plataforma *Android*, o *PhoneAdapter* (Sama et al., 2010) foi desenvolvido a fim de manipular camadas mais baixas da plataforma. Como ilustrado na Figura 2.5, cada camada pode acessar a abstração que se encontra abaixo dela e assim por diante.

Apesar de ser apenas um sistema prova de conceito, que pode ser instalado em qualquer dispositivo *Android* a partir da versão 2.3, o *PhoneAdapter* ilustra bem a definição de um controlador de SA. Este sistema obtém as variáveis de contexto do ambiente Android utilizando o respectivo *framework* (por exemplo dados de GPS e de *bluetooth* – isto é, por meio da camada *Hardware abstraction Layer* como ilustrado na Figura 2.5), cria instâncias de contexto para tais variáveis, analisa as instâncias de contexto de acordo com as regras, e executa as adaptações no *smartphone* em que o *PhoneAdapter* está instalado.

O sistema alvo, neste caso, é o próprio Android e os respectivos recursos do *smartphone*, sendo esses últimos denominados recursos gerenciados (Lalanda et al., 2013). Exemplos de recursos gerenciados são o GPS, o calendário o *Bluetooth* (sensores), Tipo de Toque, Volume, Vibrador, e Modo Avião (atuadores), que estão acessíveis por meio da camada *Hardware Abstraction Layer*, como ilustrado na Figura 2.5. Os sensores são responsáveis por obter dados do ambiente e os atuadores são responsáveis por disparar adaptações.

2.4.2 Kube-ZNN

Para ilustrar o desenvolvimento de controladores de multicamadas que são baseados em microcontroladores, neste trabalho utilizou-se o sistema alvo Kube-ZNN (Aderaldo e Mendonça, 2022) que é baseado no ZNN.com[7] (Cheng et al., 2009). O sistema ZNN.com,

[7] `https://www.hpi.uni-potsdam.de/giese/public/selfadapt/exemplars/model-problem-znn-com/` – acessado em setembro, 2022.

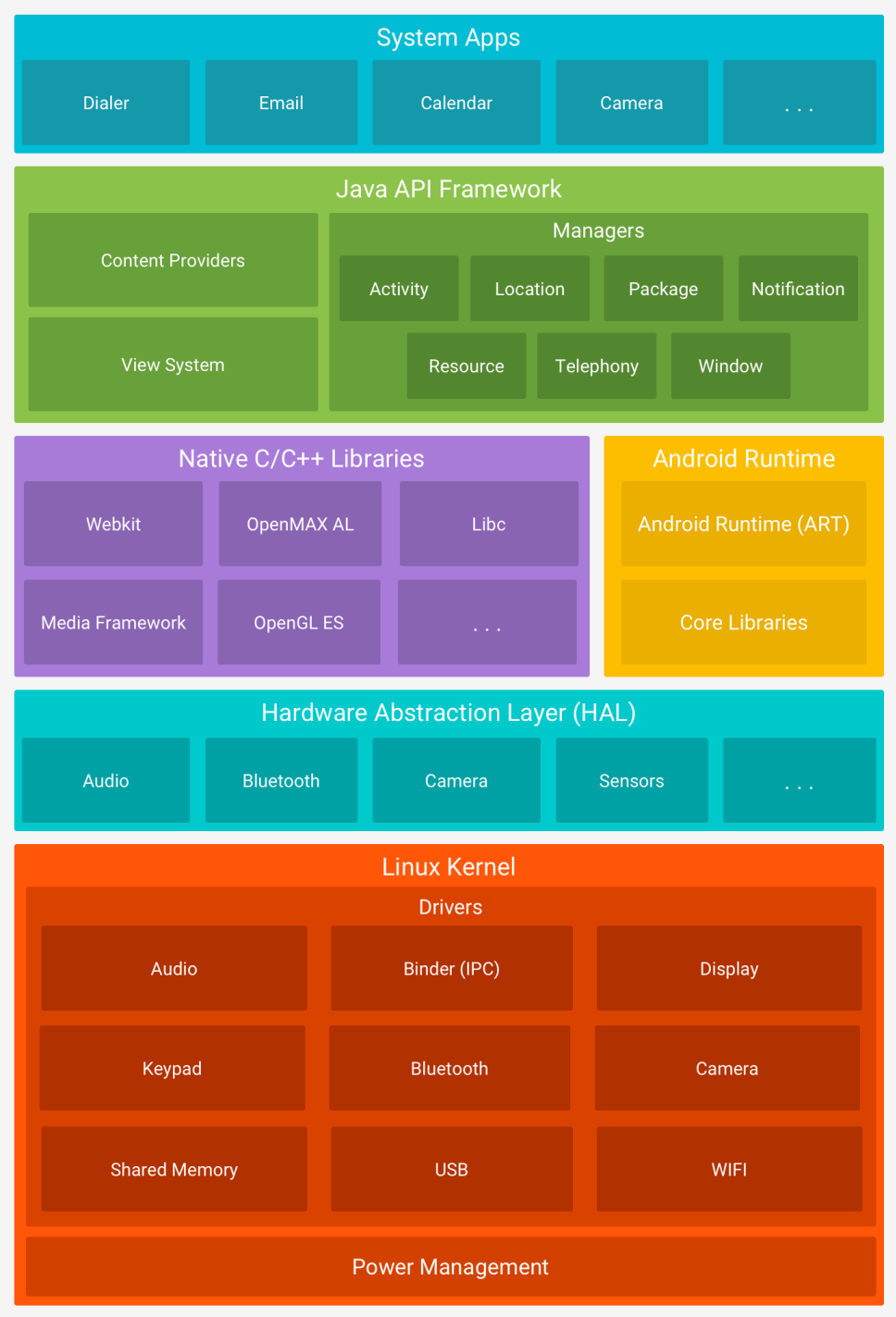

Figura 2.5: Arquitetura da Plataforma Android (Android, 2022)

.

ilustrado na Figura 2.6, está disponível para reproduzir uma típica infraestrutura para um website de notícias.

O ZNN.com contém um arquitetura de três camadas consistindo de servidores que são provedores de novos conteúdos para usuários. O objetivo do ZNN.com é prover conteúdo

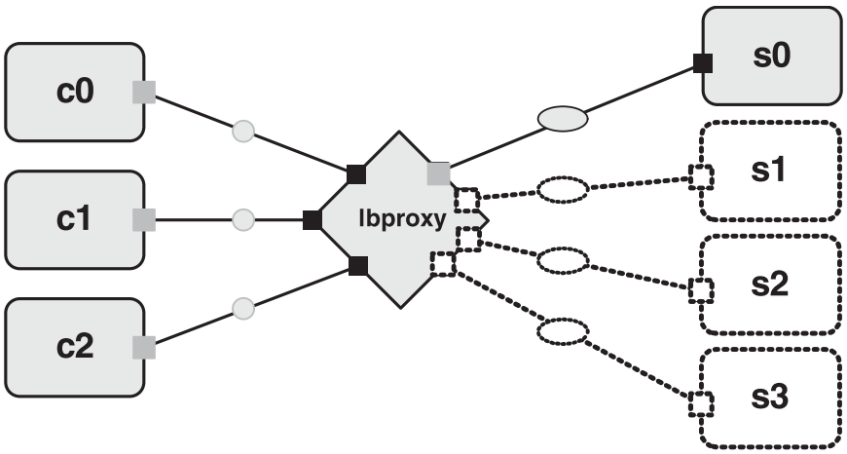

Figura 2.6: Arquitetura do sistema ZNN.com (Cámara et al., 2014).

aos usuários dentro de um tempo de resposta razoável, mantendo o custo do serviço dentro de um orçamento possível para uso. O ZNN.com tem uma arquitetura cliente e servidor que é baseada em web. Ele usa um balanceador de carga para lidar com requisições por meio de um conjunto de servidores replicados. O número de servidores pode ser adaptado de acordo com a demanda de uso dos serviços.

Atributos: Existem dois principais atributos associados ao ZNN.com:

- **Escalabilidade:** No caso de um servidor ser sobrecarregado, novas réplicas (isto é, cópias) do servidor são criadas. Por outro lado, caso uma réplica de servidor não seja mais necessária quando a demanda diminuir, ela pode ser removida. Este atributo adiciona elasticidade para o sistema.

- **Fidelidade de mídia:** Dependendo do custo associado à demanda de serviço, diferentes níveis de fidelidade são providos para melhorar a experiência do usuário (por exemplo, uso de diferentes recursos tais como *texto*, *imagens*, ou *vídeos* implicam em diferentes custos). Por exemplo, se o limite máximo do número de servidores é atingido, os servidores podem começar a prover imagens ao invés de vídeos. Assim, a demanda de serviço continuará apropriada, sem perda de desempenho. Similarmente à elasticidade mencionada na escalabilidade, em fidelidade uma vez que a demanda pela mídia diminua, o serviço fornecido pelo sistema alvo ZNN.com aumenta sua qualidade. Por outra lado, quando a demanda pela mídia aumenta, o serviço fornecido tem sua qualidade diminuída.

2.5 Considerações Finais

Neste capítulo foram introduzidos os principais fundamentos acerca do domínio de sistemas adaptativos, *frameworks* e ferramentas utilizados, bem como os sistemas alvo utilizados neste trabalho. A fim de elucidar o estado da arte para a criação de controladores de SAs, no próximo capítulo será abordada uma revisão da literatura envolvendo trabalhos de desenvolvimento de (i) microsserviços para estes sistemas; (ii) controladores flexíveis; (iii) controladores descentralizados; e (iv) reúso para estes sistemas.

Capítulo

3

Trabalhos Relacionados

3.1 Considerações Iniciais

Uma vez identificadas as características monolíticas que *frameworks* para desenvolver controladores, na sequência descrevem-se as categorias que foram utilizadas para buscas informais por trabalhos que propuseram o desenvolvimento de controladores com malhas de controle que pudessem ser flexíveis estruturalmente.

- Microsserviços: a categoria de microsserviços foi utilizada devido a suas respectivas propriedades em permitir o desenvolvimento modular de software.
- Controladores Flexíveis: abordagens para controladores concebidos a partir da ideia de se reconfigurarem podem ser trabalhos promissores para a definição de um controlador flexível estruturalmente.
- Controladores Descentralizados: trabalhos neste contexto podem fornecer uma visão geral de como componentes distribuídos de um controlador poderiam se comunicar e se gerenciar.
- Reúso: esta categoria também pode fornecer diretrizes, ou mesmo abordagens, de reúso para contribuir com a definição de componentes que poderiam ser genéricos e flexíveis para serem utilizados em diferentes controladores.

Para essas categorias, neste capítulo, apresentam-se os trabalhos da literatura encontrados. Primeiro, na Seção 3.2, descrevem-se trabalhos especificamente no contexto de microsserviços devido a características tais como responsabilidades bem definidas e específicas. Segundo, na Seção 3.3, descrevem-se trabalhos que abordaram a definição e uso de *frameworks*, ferramentas e abordagens para a definição de controladores flexíveis. Em seguida, na Seção 3.4, descrevem-se trabalhos no contexto de descentralização de controladores a fim de identificar que abordagens têm sido utilizadas para tais características descentralizadas. Finalmente, na Seção 3.5, descrevem-se trabalhos da literatura que abordaram o reúso de software para controladores de sistemas adaptativos a fim de identificar técnicas e abordagens de reúso.

3.2 Microsserviços para Sistemas Adaptativos

Nesta seção são descritos os trabalhos relacionados ao tópico de microsserviços para SAs. A seção se inicia com a descrição do trabalho de Baylov e Dimov (2017), no qual se apresenta um modelo de referência baseado no MAPE-K. Em seguida, apresenta-se o trabalho de Hassan e Bahsoon (2016), no qual se introduz uma abordagem para realizar a migração de um sistema não adaptativo para um adaptativo. Por fim, apresenta-se o trabalho de Sampaio Jr. et al. (2019), que propõem uma ferramenta para auxiliar na adaptação de sistemas baseados em afinidade de microsserviços.

O Trabalho de Baylov e Dimov (2017)

Baylov e Dimov (2017) propuseram uma arquitetura de referência para sistemas adaptativos baseados em microsserviços que podem conter malhas de controle estruturadas de acordo com o modelo MAPE-K. A arquitetura de referência dos autores contem cinco principais componentes:

1. Service Consumer: São os atores que consomem funcionalidades expostas pelos serviços.

2. Service Registry: É um catalogo que provê informação sobre os serviços, tais como o provedor ou o próprio serviço. Service Consumers buscam os Service Registry a fim de encontrar serviços requeridos e determinam a melhor maneira de invocá-los.

3. Service Provider: São responsáveis por fazer os serviços serem acessíveis na rede (internamente e externamente).

4. Service Instance: Juntamente com os Adaptation Registry, Service Instances são os componentes núcleo na arquitetura de referência. Eles proveem funcionalidades de negócio necessárias pelos Service Consumers com auxílio de malhas de controle.

5. Adaptation Registry: É um repositório de mecanismos de adaptação. Ele mantém alternativas de serviços contendo as mesmas funcionalidades.

Uma vantagem na abordagem dos autores é a de poder definir a estrutura e mecanismos dos microsserviços a partir de uma abordagem *bottom-up*. Esta características pode auxiliar em um futuro reúso dos microsserviços definidos. Porém, nenhuma avaliação da abordagem é apresentada pelos autores; apenas um exemplo ilustrativo é descrito no artigo.

O Trabalho de Hassan e Bahsoon (2016)

No contexto de migrar uma aplicação para ser baseada em microsserviços, Hassan e Bahsoon (2016) abordaram dois principais *trade-offs*: (1) balanço entre o tamanho do microsserviço e o número de microsserviços em uma arquitetura; e (2) balanço entre satisfação de níveis de requisitos não funcionais e a satisfação do sistema como um todo. Um dos principais problemas, nas palavras dos próprios autores, está em "*finalizar o nível de granularidade tão cedo, pode dificultar o projeto*". Em outras palavras, o problema reside em definir partes concretas quando se instancia uma arquitetura abstrata do sistema. Os autores lidam com alguns dilemas, tais como: (1) quando realizar a decomposição de funcionalidades em microsserviços menores?; (2) quando mesclar micros-serviços menores para unidades maiores; e (3) quando manter o nível atual da granularidade de microsserviços sem decompor ou sem mesclá-los.

No trabalho, propõe-se uma abordagem baseada em auto-adaptatividade (isto é, *self-adaptivity*) que auxilia em lidar com ambos os *trade-offs*. Para isso, os autores explanam, a partir dos estágios do MAPE-K, que estudos na literatura contribuem para cada estágio.

Uma possível ponto forte da abordagem é o direcionamento que se pode ter em uma solução ao adotar microsserviços. Um ponto fraco é a ênfase a partir dos principais estágios MAPE-K, tudo em alto nível de abstração. Uma limitação nítida do trabalho é o seu teor puramente teórico, tendo-se em vista que os autores utilizam somente um exemplo ilustrativo para apresentar a abordagem.

O Trabalho de Sampaio Jr. et al. (2019)

Sampaio Jr. et al. (2019) investigaram um processo de reconfiguração em *tempo de execução* de aplicações que são baseadas em microsserviços. Nesse sentido, os autores abordam alguns desafios do processo de reconfiguração, destacando dificuldades encontradas quando se compõem diversos microsserviços. Com isso, os autores propuseram um mecanismo de adaptação (nomeado REMaP) que é apoiado por uma ferramenta desenvolvida pelos autores.

Durante o aumento de demanda por recursos em determinados servidores com microsserviços implantados, novas configurações em *tempo de execução* de microsserviços são necessárias. Nesse aspecto, os autores enfatizam que as ferramentas de gerenciamento de recursos e de implantação de microsserviços são limitadas. Essa limitação decorre de quais características se devem levar em consideração quando realizar reconfigurações. Por exemplo, o Kubernetes leva em consideração informação como a disponibilidade alta e baixa de recursos. Já informações envolvendo a relação entre os microsserviços não é realizada pelo Kubernetes de forma nativa.

A abordagem dos autores, nomeada de REMaP, contém um gerenciador de adaptação baseado no MAPE-K para realizar a reconfiguração dos microsserviços em *tempo de execução*, isto é, reconfiguração baseada em implantação de microsserviços.

A REMaP utiliza conceitos de Modelsrun.time e abstraem a diversidade das ferramentas de monitoramento e gerenciamento. Tais ferramentas contém uma visão unificada com relação ao gerenciador de adaptação que, por sua vez, aplica reconfigurações aos microsserviços. A abordagem utiliza a comunicação entre os microsserviços, obtendo o histórico da comunicação entre eles para categorizar a *afinidade* entre os mesmos.

Uma das vantagens da abordagem é poder lidar com adaptações que levam em consideração o sistema em seu *tempo de execução*. Como os autores destacam, a abordagem está em contraste com a literatura que, em geral, propõem abordagem estáticas que lidam somente com informações de *tempo de projeto*. Como uma desvantagem, além da solução dar ênfase somente nos estágios do MAPE-K, propõe-se uma abordagem para realizar reconfigurações somente base na comunicação entre os microsserviços. Em outras palavras, utiliza-se uma abordagem de controle descentralizado, focando-se na reconfiguração baseada na coreografia dos microsserviços.

Observações Finais sobre os Trabalhos que Abordam Microsserviços sobre SAs

A arquitetura de referência de Baylov e Dimov (2017) apoia o desenvolvimento de microsserviços adaptativos. Entretanto, para a composição de controladores, tal arquitetura de referência implica em uma estrutura específica que não provê o nível de flexibilidade.

Hassan e Bahsoon (2016) propuseram a criação de um controlador adaptativo baseado em microsserviços. Tal controlador é similar à proposta de Sampaio Jr. et al. (2019), que apoia a reconfiguração de sistemas baseados em microsserviços de acordo com afinidades e histórico do uso de recursos.

Em contraste com abordagens de Hassan e Bahsoon (2016) e de Sampaio Jr. et al. (2019), a abordagem do trabalho desta tese promove o uso de microcontroladores (implementados como microsserviços) para arquitetar controladores flexíveis. Por outro lado, e alinhado à sugestão de Mendonça et al. (2019), um dos desafios em se construir sistemas de software adaptativos baseados em microsserviços reside em como implantar tais controladores. Com relação à granularidade do controlador, duas possibilidades são discutidas: o controlador sendo um único serviço monolítico, ou sendo decomposto em uma coleção de microsserviços gerenciados e desenvolvidos independentemente.

3.3 Controladores Flexíveis e o Uso de Metacontroladores

Nesta seção são descritos os trabalhos relacionados ao tópico de controladores flexíveis para SAs. Primeiramente, apresenta-se o trabalho de Banijamali et al. (2020), o qual propôs um *framework* baseado no MAPE-K com o objetivo de otimizar o ciclo de vida do SA. Depois, apresenta-se o trabalho de Gerostathopoulos et al. (2019), o qual aborda a estruturação de SAs em diferentes camadas. Por fim, apresenta-se o trabalho de Pereira et al. (2020), no qual os autores propuseram uma plataforma para auxiliar o uso de atributos de confiabilidade em SAs.

O Trabalho de Banijamali et al. (2020)

Banijamali et al. (2020) investigaram como microsserviços podem ser utilizados no projeto de sistemas adaptativos no domínio automotivo. Além disso, os autores medem quais *trade-offs* de qualidade ocorrem em *tempo de execução* ao utilizar microsserviços adaptativos no respectivo domínio. O *framework* Kuksa* foi proposto, sendo esse baseado

na malha de controle MAPE-K com o objetivo de melhorar o ciclo de vida do sistema com acúmulo de dados via componente *knowledge*. Nomeado Eclipse Kuksa, tal framework fornece uma plataforma baseada em nuvem para conectar veículos e realizar diferentes interconexões entre objetos e a internet. A plataforma é toda baseada na infraestrutura de microsserviços e do Docker e do Kubernetes. Assim, utilizam-se recursos de implantação, modificação e balanceamento de carga de um *cluster* Kubernetes.

Em seu trabalho, Banijamali et al. (2020) concluem que as adaptações que são realizadas em *tempo de execução* utilizam: 1) listas de serviços em execução; 2) níveis de satisfação de requisitos de qualidade; 3) acessibilidade de dados; 4) adaptabilidade de recursos; e 5) contextos de funcionamentos do sistema. Isso implica que quanto mais microsserviços são incluídos para a malha de controle gerenciar, maior será a complexidade dos sistemas como um todo.

Como uma vantagem, apesar dos autores utilizarem o domínio específico de sistemas automotivos, conjectura-se que o uso de microsserviços e do MAPE-K é possível de ser implementada em outros domínios. Como a definição da malha de controle é baseada em microsserviços, a adaptabilidade do controlador também é

Uma desvantagem refere-se ao fato de que o o desempenho do sistema depende do desempenho de cada serviço. Uma outra desvantagem é que as adaptações foram providas apenas a um conjunto de microsserviços, para evitar aumento da complexidade do sistema como um todo.

O Trabalho de Gerostathopoulos et al. (2019)

No trabalho Gerostathopoulos et al. (2019), contextualiza-se que falhas podem acontecer e propõe-se incluir uma camada para manter o sistema em estado operacional mesmo com incerteza em *tempo de execução*. Tal camada contempla a introdução de mudanças em *tempo de execução* para estratégias de adaptação baseadas em arquitetura de acordo com estímulos do ambiente. Tal abordagem é apoiada por meio de quatro mecanismos:

1. Sensoriamento colaborativo: este mecanismo é responsável por fornecer dados aproximados de sensores, em caso de mal funcionamento dos mesmos. Ele auxilia a malha de controle, que é baseado no MAPE-K, a lidar com a falta de dados vinda dos sensores. Os dados aproximados são coletados por meio de simulações ou execuções anteriores do sistema.

2. Isolamento de componente com defeito a partir de adaptação: este mecanismo visa a isolar o mal funcionamento de um componente, trocando-o por outro componente

similar. Ele faz com que, mesmo quando parte do sistema para de funcionar, aconteça uma reconfiguração dos componentes, evitando-se assim estados inconsistentes generalizados. Tais situações de reconfiguração são baseadas em modelos de defeitos e/ou conhecimento empírico.

3. Mudança de modo de aprimoramento: este mecanismo visa a verificar se a evolução do sistema está comprometida ou se está restrita. Ele faz com que caso sejam encontradas restrições em adaptar, as mesmas sejam mitigadas para que o sistema continue em contante aprimoramento. Tal mecanismo é baseado em máquina de estados, de modo que modelos dessas máquinas são modificados e aprimorados.

4. Ajustes de guardas nos modos de mudanças: este mecanismo visa a constantemente verificar se os predicados atuais do sistema precisam de modificação/evolução. Uma vez que tais predicados gerem as adaptações do sistema, tem-se a premissa de que com o tempo, tais predicados podem/devem ser ajustados a fim do sistema continuar em constante evolução e aprimoramento. Este mecanismo conta com auxílio de algoritmos de otimização.

No trabalho, desenvolveu-se uma camada de homeostases que está acima do controlador, como na camada do metacontrolador deste trabalho, a qual é responsável por modificar estratégias de adaptação nas camadas mais baixas em tempo de execução. No respectivo trabalho, os autores deram ênfase em desenvolver mecanismos para esta camada de homeostases.

Um ponto forte do trabalho diz respeito a avaliações. Realizam-se avaliações envolvendo dois estudos de caso para os quatro mecanismos descritos. Outro ponto forte diz respeito ao possível uso da abordagem em diversos domínio de aplicação. Uma desvantagem está relacionada à não clareza da camada e que quais possíveis níveis de atuação a mesma pode estar relacionada, apesar de se argumentar que a mesma possa atualizar acima de uma camada do controlador.

O Trabalho de Pereira et al. (2020)

Pereira et al. (2020) abordaram propriedades de qualidade de computação em nuvem, destacando-se atributos como confiabilidade, disponibilidade, segurança e privacidade. Tais atributos são abordados como modelos de qualidade que são aplicados via planos de adaptação, por meio de regras de avaliação. Neste contexto, os autores propuseram uma plataforma que implementa uma malha de controle baseado no MAPE-K, contendo mecanismos de monitoramento baseado em *probes* e adaptações baseadas em *atuadores*.

A plataforma proposta auxilia desenvolvedores a criarem software com atributos de confiabilidade. Cada componente baseado no MAPE-K é projetado como um microsserviço para ser implantado como contêiner Docker, facilitando também a integração do mesmo em ambientes como, por exemplo, o Kubernetes. Mais especificamente, a plataforma dos autores é composta por:

- Uma definição de propriedades e métricas utilizadas para caracterizar a confiabilidade;
- Um ciclo de vida de confiabilidade inspirado no ciclo da malha de controle MAPE-K, cobrindo as fases de *tempo de projeto* e *tempo de execução*;
- Instrumentos de medidas que avaliam a informação utilizada;
- Modelos de qualidade que definem como as medidas serão utilizadas para calcular pontuações;
- Atuadores que implementam lógica de adaptação em sistemas alvos e permitem o aprimoramento da confiabilidade do sistema.

Como uma vantagem, o fato da malha de controle ser baseado no MAPE-K e em microsserviços contribui para uma flexibilidade no uso. Entretanto, para o uso da abordagem de forma mais generalizada, devem-se realizar extensões e trabalhos futuros da abordagem, envolvendo diferentes possíveis atributos a serem utilizados em uma arquitetura para SAs. Nesse sentido, nota-se que o trabalho demonstrou a aplicação da abordagem utilizando-se um único domínio de aplicação.

Observações Finais sobre os Trabalhos que Abordam Metacontroladores

Banijamali et al. (2020) enfatizaram o crescente interesse por microsserviços para SAs, em particular no domínio automotivo. Na abordagem proposta pelos autores, que é similar à proposta neste trabalho, considera-se um controlador formado por microsserviços. Entretanto, os autores não abordam o papel de uma entidade de coordenação para gerenciar microsserviços. Isto foi abordado por Gerostathopoulos et al. (2019), os quais incorporaram uma camada chamada de homeostases Shaw (2002) em um controle adicional. Tal camada é responsável por modificar estratégias de adaptação nas camadas mais baixas em tempo de execução.

Pereira et al. (2020), por sua vez, propuseram o desenvolvimento de controladores flexíveis que são baseados no MAPE-K, nos quais cada estágio do MAPE-K é implementado

como um único microsserviço. No caso da abordagem proposta nesta tese, microcontroladores não são limitados aos componentes MAPE-K. Além disso, na abordagem proposta nesta tese, pode-se destacar a inclusão de um metacontrolador. como uma camada adicional que está apta a reconfigurar o controlador.

3.4 Controladores Descentralizados

Na sequência são descritos os trabalhos relacionados ao tópico de controladores descentralizados para SAs. Primeiro, apresenta-se o trabalho de Florio e Di Nitto (2016), no qual foram propostas malhas de controle baseados no MAPE-K organizados em componentes separados. Por fim, apresenta-se o trabalho de Nallur e Bahsoon (2013), no qual os autores introduziram uma uma solução que enfatiza em adaptações baseadas em análise de demanda do sistema alvo.

O Trabalho de Florio e Di Nitto (2016)

Florio e Di Nitto (2016) propouseram incluir atributos autonômicos em microsserviços, sem alterar a maneira como são implementados. Para isso, os autores propuseram malhas de controle baseados no MAPE-K que são descentralizados, intercambiando mensagens entre eles. Na abordagem, considera-se o uso de um componente intermediário (nomeado *autonomic enabler*. O componente intermediário em si não tem características de adaptação; na verdade, ele tem por objetivo facilitar a ação de um controlador em um sistema alvo.

A abordagem descentralizada em questão parte do princípio que as malhas de controle são inteiramente descentralizados. Porém, eles se inter-comunicam para obter informações do sistema como um todo. Em outras palavras, as malhas de controle contém lógica implementada para interagirem entre si.

A abordagem faz uso de um repositório que armazenam os chamados *service-descriptors* e a configuração das malhas de controle (nomeados *Gru-Agents*). Tais configurações contém parâmetros, tais como tempo de intervalo de cada malha, relacionados à execução das malhas de controle. Além disso, outras informações referentes às malhas também são armazenadas, a fim de disponibilizar as mesmas para usos futuros para gerenciamento.

Os *service-descriptors* são abstrações que modelam os microsserviços presentes em todo o sistema. Tais abstrações incluem as características necessárias para o funcionamento do respectivo microsserviço, incluindo: restrições de qualidade, tempo máximo de resposta e recursos necessários para o serviço funcionar.

Os nomeados *Gru-Agents* são malhas de controle do sistema e atuam por meio de contêineres Docker implantados (executando como um microsserviço). Cada malha compartilha informação sobre seus dados internos com os demais malhas envolvidas. Para que tal descentralização não gere um *overhead* para todo o sistema, os *Gru-Agentes* são organizados em subconjuntos para que os dados sejam compartilhados.

Ao passo que a abordagem dos autores é descentralizada, contribui-se para que definições de novos *Gru-Agents* independentes e especializados. Entretanto, tal descentralização pode também gerar estados inconsistentes e conflituosos, além do *overhead* presente na própria arquitetura descentralizada.

O Trabalho de Nallur e Bahsoon (2013)

O contexto do trabalho está em propor uma solução para adaptar uma aplicação de acordo com os recursos disponíveis e demanda da mesma. Caracteriza-se, por exemplo, um cenário que pode ser utilizado em ambientes em nuvem, os quais demandam alto investimento nos provedores de serviço e alocação de recursos caso os recursos não sejam gerenciados adequadamente.

Os autores propuseram um *framework* para auxiliar a criação de sistemas baseados em *web services* com atributos que auxiliem a lidar com níveis de qualidade. Estes níveis de qualidade fornecem subsídios para a adaptação do sistema.

A abordagem dos autores utiliza o controle baseado em mercado, que envolve modelar o sistema como uma loja de serviços (isto é, um *marketplace*). Com isso, a recomposição de serviços necessários ocorre de acordo com uma estratégia de alocação e economia de recursos, utilizando-se somente recursos necessários para um determinado momento.

Uma das vantagens que se destacam na abordagem dos autores é a escalabilidade. Isso deve-se ao mecanismos de recomposição baseado em alocação de recursos. Por outro lado, uma desvantagem diz respeito ao uso dos níveis de qualidade atrelados à abordagem descentralizada proposta. Isto deve-se ao aumento da complexidade que a descentralização produz.

Observações Finais sobre os Trabalhos que Abordam Controladores Decentralizados

No trabalho de Florio e Di Nitto (2016) deu-se ênfase em adicionar capacidades para sistemas baseados em microsserviços conteinerizados que não foram originalmente projetados para serem adaptativos. A abordagem dos autores consiste em um controlador descentralizado que é implementado como um sistema multi-agentes. Cada agente tem uma malha de controle que é responsável por gerenciar um subconjunto de microsserviços.

Nallur e Bahsoon (2013) também desenvolveram uma abordagem de autoadaptação baseada em multi-agentes. Neste caso, tal abordagem é descentralizada e baseada em serviços web que são implantados na nuvem.

Ao considerar ambos os trabalhos (Florio e Di Nitto, 2016; Nallur e Bahsoon, 2013) e o trabalho descrito nesta tese, as principais diferenças são:

- Ambos os trabalhos Florio e Di Nitto (2016); Nallur e Bahsoon (2013) foram especificamente projetados para serem aplicados em domínios particulares (ou seja, sistemas que são baseados em microsserviços e sistemas baseado em serviços web, respectivamente).
- Os controladores baseados em sistemas multi-agentes não são reconfiguráveis.

3.5 Reúso para Sistemas Adaptativos

Nesta seção descrevem-se trabalhos relacionados ao tópico de reúso de controladores para SAs. Inicialmente, apresenta-se o trabalho de Ramirez e Cheng (2010), que propuseram 12 padrões de projetos para o desenvolvimento de SAs. Depois, apresenta-se o trabalho de Vogel e Giese, o qual aborda soluções de desenvolvimento dirigido por modelos de *tempo de execução*. Por fim, apresentam-se o trabalho de Weyns et al., que propuseram o uso de diversas malhas de controle MAPE-K que são baseados em modelos hierárquicos.

O trabalho de Ramirez e Cheng (2010)

Os autores propuseram um conjunto de 12 padrões de projeto para o desenvolvimento de SAs. Foram analisados projetos *open source* e trabalhos da literatura, totalizando 30 fontes ao todo, e os autores derivaram 12 padrões a partir desses.

Os autores separaram os padrões em lógica adaptativa e lógica funcional à luz de recorrentes desafios envolvendo monitoramento, tomada de decisão e atividades de reconfiguração. Mais especificamente, os autores criaram as seguintes três categorias de padrões para organizar os 12 padrões caracterizados.

- Primeira categoria - *Monitoring Patterns*: objetiva a habilitar um SA para lidar com condições ambientais e do sistema que devem levar a reconfigurações. São três os padrões de monitoramento.

    1. Sensor Factory: este padrão gerencia sensores distribuídos, coletando dados do mesmo independentemente de onde tais sensores estejam.

2. Reflective Monitoring: este padrão utiliza conceitos de reflexão para obter dados de sensores sem a necessidade de um alto acoplamento entre componentes.
3. Content-based Routing: este padrão utiliza conceitos de produtor/consumidor para gerenciar filas de mensagens para comunicação entre componentes, a fim de diminuir *overhead* de um acesso múltiplo a sensores.

- Segunda categoria - *Decision-Making Patterns*: objetiva determinar quando e como reconfigurar um SA em resposta ao monitoramento obtido. São cinco os padrões de apoio à decisão.

1. Adaptation Detector: este padrão define limiares de dados que são obtidos por meio do monitoramento. Tais limiares auxiliam na definição de regras de adaptação e serão gatilhos para disparo de modificações.
2. Case-based Reasoning: este padrão facilita a definição da tomada de decisão sendo definida e expressada como sentenças if-then-else. Tais expressões auxiliam em diminuir a complexidade de SAs.
3. Divide and Conquer: este padrão diminui a complexidade de regras de adaptação que são uma dependente da outra. Além de otimizar a execução das adaptações, este padrão também auxilia em evitar regras redundantes e definição de regras complementares.
4. Architecture-based: este padrão auxilia em lidar com as tomadas de decisão em nível de componentes e relações entre componentes. Tal padrão auxilia em lidar com o nível de abstração na perspectiva arquitetural, facilitando a representação do sistema em *tempo de execução*.
5. TradeOff-Based: este padrão auxilia na definição de reconfigurações que são baseadas em custo-benefício do sistema. Em tal padrão, podem-se definir funções de utilidade que ajudam a definir configurações alternativas a fim de otimizar as reconfigurações do sistema.

- Terceira categoria - *Reconfiguration Patterns*: objetiva realizar mudanças comportamentais e estruturais dinamicamente, sem deixar o sistema em estado inconsistentes. São quatro os padrões de reconfiguração.

1. Component Insertion: este padrão visa à definição de estados pré-determinados para serem inicializados durante a inclusão de novos componentes. Estes estados podem ser novos ou estados que foram preservados de reconfigurações anteriores.

2. Component Removal: este padrão visa à complementariedade do padrão *Component Insertion*, ou seja, durante a remoção de componentes, os estados devem ser preservados a fim de permitir a posterior execução e reconfiguração do sistema.
3. Server Reconfiguration: este padrão visa a gerenciar o uso do sistema durante as reconfigurações. Esta garantia está no contexto de gerenciar conexões ativas tanto de outros componentes e aplicações quanto de outros usuários. O intuito é gerenciar o estado do sistema até que qualquer reconfiguração ocorra.
4. Decentralized Reconfiguration: este padrão visa a gerenciar o estado de diferentes componentes distribuídos, a fim de deixá-los consistentes após as reconfigurações. Assim, questões envolvendo inicialização de estados, transações entre componentes e estabelecimento de comunicação distribuída são abordados.

Os padrões de projetos foram propostos por meio da análise de diversos outros projetos, o que contribui ainda mais para a generalização e convergências dos mesmos. Entretanto, assim como os próprios autores reforçam, mais estudos precisam ser conduzidos para que os padrões sejam avaliados tanto em domínios específicos quanto em qualquer domínio.

Os trabalhos de Vogel e Giese (2010, 2012, 2014)

No trabalho de Vogel e Giese (2010) foi proposto um conjunto de modelos arquiteturais de *tempo de execução* em diferentes níveis de abstração, baseando-se na adaptação envolvida. A proposta de uma abordagem dirigida por modelos de *tempo de execução*, chamados de *megamodelos*, visa a ter modelos que são independentes de implementação e não unicamente conceituais.

Em um trabalho subsequente (Vogel e Giese, 2012), os autores também propuseram uma linguagem de modelagem. Esta linguagem visa a expressar e lidar com megamodelos de *tempo de execução*, facilitando o desenvolvimento da lógica de adaptação ao fornecer uma abordagem de modelagem específica de domínio. Além disso, também apresentaram um interpretador em *tempo de execução* para partes do sistema adaptativo. A linguagem como um todo facilita o desenvolvimento de múltiplas e explícitas malhas de controle em alto nível de abstração. Ademais, ela fornece suporte para o desenvolvimento de sistemas complexos com essas várias malhas de controle que interagem entre si. Evoluções das ferramentas também suportam motores de adaptação, as quais estão incorporadas na contribuição mais recente intitulada ExecUtable RuntimE MegAmodels (EUREMA) (Vogel e Giese, 2014).

Como vantagem, a abordagem proposta por Vogel e Giese propicia uma maior aderência em termos de reusabilidade e extensão de controladores. Desvantagens sobre os trabalho de Vogel e Giese relacionam-se com a curva de aprendizagem e o trabalho oneroso e detalhista do uso de aplicações baseadas exclusivamente em modelos.

Os Trabalhos de Weyns et al.

Weyns et al. (2013) abordaram o uso de múltiplos MAPE para auxiliar na adaptação do sistema. Além disso, abordaram como a coordenação de múltiplos MAPE pode ser realizada. Os autores propuseram uma notação para descrever interações entre malhas MAPE que auxiliam a alcançar uma abordagem sistemática de controle descentralizado. A notação captura as iniciais M-A-P-E do padrão MAPE de malha de controle para abordar os componentes distribuídos e suas interações. Um controlador eventualmente pode ser representado por um conjunto M-A-P-E, cujos elementos estejam distribuídos em diferentes nós. Os autores enfatizam que podem existir diversos M-A-P-E separadamente, e mais de um M ou A ou P ou E. Com exemplo, um controlador pode ser representado por n M e n E, juntamente com únicos A e P. Diferentes níveis hierárquicos também podem ser previstos nesta notação, de modo que um conjunto de elementos M-A-P-E podem coordenador outros elementos M-A-P-E (ou combinações desses).

Os autores também sumarizam padrões para a definição de controladores descentralizados. Inicialmente, abordaram-se dois padrões com abordagens inteiramente descentralizadas (*Coordinated Control* e *Information sharing*), sendo antítese para uma malha de controle centralizado. Ambas as abordagem são baseadas em um modelo distribuído, no qual existem malhas de controle MAPE operando em paralelo para gerenciar a auto-adaptação do sistema. Depois, abordaram-se três padrões baseados em um modelo de distribuição hierárquico (isto é, *Master/Slave*, *Regional planning* e *Hierarchical control*), no qual o nível mais alto é uma malha de controle MAPE que controla outras malhas de controle MAPE subordinados. Estes três padrões hierárquicos podem ser considerados intermediários entre controles inteiramente descentralizados e centralizados. Além disso, a hierarquia em si constitui em um ponto central.

Em outro trabalho do mesmo grupo de pesquisam Iftikhar e Weyns (2014) propuseram uma abordagem – de nome ActivFORMS – que auxilia a definição das malhas de controle em *tempo de projeto* e sua respectiva execução em *tempo de execução*. A abordagem ActivFORMS é inteiramente formal, baseada em modelos, de modo que se definem modelos representando o sistema alvo e também a malha de controle. Utiliza-se de auxílio de uma máquina virtual para realizar adaptações.

Abordagens que são inteiramente formais e baseadas em modelos têm a vantagens de auxiliar na corretude do sistema. Entretanto, tanto o sistema alvo quanto o controlador devem ser também inteiramente especificados. Com aumento do sistema, as complexidades de desenvolvimento e manutenção também aumentam.

Observações Finais sobre os Trabalhos que Abordam Reúso para SAs

Mendonça et al. (2018) discutiram dificuldades para reutilizar serviços de adaptação e arcabouços em diversos sistemas adaptativos. A partir de perspectivas de generalidade e reusabilidade, eles argumentam que existe uma incompatibilidade entre necessidades de adaptação de sistemas modernos e as soluções atuais.

Alguns trabalhos apoiam o reúso conceitual de controladores ao utilizar o modelo de referência MAKE-K (Kephart e Chess, 2003); padrões de projeto (Ramirez e Cheng, 2010), e padrões de controle (Weyns et al., 2013).

Outros trabalhos apoiam o reúso técnico do controlador. No Rainbow (Garlan et al., 2004), o nível de abstração é elevado à arquitetura de software para que um controlador execute uma adaptação arquitetural de forma genérica. Neste sentido, são feitas inclusões, remoções e reconfiguração de componentes em um sistema alvo. Para isso, Rainbow tem que ser utilizado e adaptado sob-medida com *probes* e *atuadores* específicos para o sistema alvo.

Outras abordagens habilitam o reúso de componentes de execução para controladores especificados por modelos (Iftikhar e Weyns, 2014; Vogel e Giese, 2012). Os modelos resultantes são específicos para cada sistema alvo, mas os componentes de execução são genéricos e reutilizáveis. Este princípio tem sido estendido para estágios individuais do controlador, que permite o reúso de componentes em um nível de granularidade mais fina (Vogel e Giese, 2014). Ao passo que isso facilita o desenvolvimento de controladores, os modelos têm que ser criados específicamente para o sistema alvo.

Com relação às abordagens de reúso, os controladores resultantes destas abordagens não abordam um vasto número de necessidades do sistema alvo. Nota-se que a unidade de reúso é um controlador monolítico, ou templates que restringem a estrutura ou o comportamento de controladores, ou ainda modelos de componentes de execução. Em contraste, nesta tese tem-se por ênfase a proposta de uma abordagem que não seja monolítica e que o reúso seja uma consequência das características descentralizadas da mesma.

3.6 Considerações Finais

Com relação ao uso de microsserviços para sistemas adaptativos, os trabalhos encontrados provêm arquiteturas de referência que não enfatizam questões envolvendo níveis de flexibilidade. No contexto de controladores flexíveis, foram identificadas abordagens com uma visão *top-down* da estrutura do controlador, dificultando-se a definição específica dos componentes do mesmo.

No que diz respeito à descentralização de controladores, os trabalhos têm ênfase na definição baseada no MAPE-K e, especificamente, são projetados para serem aplicados em domínios particulares (por exemplo, sistemas que são baseados em microsserviços e sistemas baseado em serviços web).

Finalmente, com relação ao reúso, as abordagens em geral fazem uso de controladores monolíticos. Isto inclui o uso de modelos que restringem a estrutura/comportamento do controlador, e uso de modelos de componentes em execução.

Nesta tese, com a proposição de uso de microcontroladores, pode-se abordar um abrangente número de necessidades que podem ser aplicadas aos sistemas alvos por meio da síntese de controladores baseada em coleções de microcontroladores genéricos e específicos. Na Tabela 3.1, sumarizam-se os estudos relacionados a este trabalho com o desenvolvimento de controladores, com relação a: (i) microsserviços para SAs; (ii) controladores flexíveis; (iii) controladores decentralizados; e (iv) reúso para SAs.

Tabela 3.1: Estudos e respectivas abordagens para o projeto de controladores na literatura.

	Abordagem para o controlador referente a:			
Estudo	**Microsserviços**	**Flexibilidade**	**Decentralização**	**Reúso**
(Baylov e Dimov, 2017)	x			
(Hassan e Bahsoon, 2016)	x			
(Sampaio Jr. et al., 2019)	x		x	
(Banijamali et al., 2020)	x	x		
(Gerostathopoulos et al., 2019)		x		
(Pereira et al., 2020)	x	x		
(Florio e Di Nitto, 2016)	x		x	
(Nallur e Bahsoon, 2013)			x	
(Ramirez e Cheng, 2010)				x
(Vogel e Giese, 2010, 2012, 2014)				x
(Iftikhar e Weyns, 2014; Weyns et al., 2013)		x		x
Este trabalho	x	x	x	x

Como ilustrado na Tabela 3.1, os estudos realizados até o momento não abordam os quatro tópicos de forma conjunta para a definição de controladores flexíveis. Ao se abordar esses de forma conjunta, contribui-se para o desenvolvimento de controladores configuráveis, fornecendo subsídio para a definição de partes genéricas (chamadas aqui

de microcontroladores) para serem reutilizáveis em diferentes controladores. No próximo capítulo apresenta-se a abordagem proposta nesta tese.

Capítulo

4

Controladores Baseados em Microcontroladores

4.1 Considerações Iniciais

Um controlador que é composto por funcionalidades definidas de acordo com o sistema alvo pode alcançar uma maior flexibilidade e variabilidade, comparado a controladores monolíticos (McKinley et al., 2004). Para alcançar tal abordagem flexível estruturalmente, componentes do controladores devem ser implementados de forma estruturalmente independentes uns dos outros. Assim, neste trabalho propõe-se para SAs uma abordagem para projetar controladores que consistem em uma coleção independente de microcontroladores.

Nas próximas seções apresenta-se a descrição da abordagem, iniciando-se pela Secão 4.2, que contextualiza sobre definições arquiteturais de SAs. Dessa forma, discorre-se a respeito de adaptações paramétricas e estruturais envolvendo mudanças de um controlador. Depois, na Seção 4.3, descrevem-se os microcontroladores, destacando-se como eles podem ser definidos, bem como a ilustração de um controlador baseado no modelo MAPE-K. Finalmente, na Seção 4.4, descreve-se a definição do metacontrolador, ressaltando-se a sua importância, bem como artefatos que podem ser utilizados para a construção do mesmo (por exemplo, Rainbow/*Kubow*).

4.2 Descrição da Abordagem

A abordagem proposta neste trabalho considera controladores de múltiplas camadas como forma de promover reúso no tempo de desenvolvimento, e flexibilidade estrutural no tempo de execução. Para promover reúso de controladores, a abordagem combina controladores genéricos, que podem ser facilmente configurados, cujo papel é gerenciar uma coleção de controladores específicos. A flexibilidade estrutural é alcançada ao empregar controladores simples e específicos que podem ser facilmente trocados e orquestrados por um controlador genérico. A ideia em se terem controladores de múltiplas camadas não é nova (Braberman et al., 2015; IBM, 2005), além de se adequar bem em seu uso em um padrão de controle hierárquico (Weyns et al., 2013).

A motivação em promover controladores de múltiplas camadas é para desacoplar o controlador do sistema alvo, uma vez que tal acoplamento pode trazer intrinsecamente complexidade ao software. O forte acoplamento entre o sistema alvo e o controlador pode dificultar o reúso de controladores em diferentes aplicações e domínios, pois os sistemas precisam fornecer serviços diferentes com diferentes níveis de qualidade.

Um SA é composto por um sistema alvo e um controlador, e esse sistema deve ser considerado em seus contextos e seu ambiente. As mudanças de contexto são relacionadas ao sistema alvo e ao respectivo ambiente (isto é, fenômeno externo relacionado a outros sistemas ou seres humanos). Assim, as adaptações são geridas pelo controlador que empiricamente observa mudanças para construir modelos que o permitam controlar o sistema alvo. Adaptações podem ser paramétricas ou estruturais Andersson et al. (2009); McKinley et al. (2004). Adaptação paramétrica modifica variáveis de programa que determinam o comportamento do sistema. Um exemplo é o protocolo de controle de transmissão da Internet (isto é, TCP – *Transmission Control Protocol*), que ajusta seu comportamento ao alterar valores que controlam o gerenciamento e retransmissão de pacotes em resposta ao congestionamento da rede. Uma inerente fraqueza quanto à adaptação paramétrica é a não possibilidade de se incluir novos algorítimos e componentes durante o tempo de execução do sistema. Em contraste, a adaptação estrutural está apta a modificar algoritmos e a estrutura do sistema, trocando componentes e melhorando a estrutura atual do sistema mesmo durante o tempo de execução. Este tipo de adaptação habilita a chamada *recomposição dinâmica* do sistema. Como exemplo, considera-se o caso em que se adapta a estrutura do sistema de acordo com a limitação de recursos de memória ou processamento.

A solução que está sendo proposta neste trabalho, para desacoplar o controlador do sistema alvo, consistem na introdução de um controlador de segunda camada, como é

ilustrado na Figura 4.1. A camada *Controller* é um controlador estruturalmente flexível que deve estar apto a se ajustar facilmente de acordo com as necessidades do sistema alvo (isto é, *target system*). A camada *Metacontroller* é um controlador sofisticado, como o Rainbow (Garlan et al., 2004), que deve estar apto a controlar a camada *Controller* ao adaptar sua estrutura e seus parâmetros. Embora adaptações paramétricas de controladores monolíticos sejam mais comuns (Krupitzer et al., 2016), adaptações estruturais também são possíveis. Porém, tais adaptações podem requerer re-avaliação e re-implantação do controlador. Neste contexto, a principal ideia que está sendo promovida neste trabalho é que controladores não precisam ser estruturalmente estáticos no tempo de execução (isto é, uma vez implantados, não esperados para serem alterados); pelo contrário, controladores podem ser tanto responsáveis por adaptações quanto também podem receber adaptações.

Na sequência, apresenta-se a ideia principal de controladores de duas camadas que tende a melhorar a flexibilidade estrutural e facilitar o reúso de controladores. Primeiramente, será introduzida a ideia de microcontroladores e então a ideia de um metacontrolador para controlar configurações de microcontroladores.

4.3 Microcontroladores

A principal ideia que está sendo proposta é a troca de um controlador monolítico, tal como uma implementação da malha de controle MAPE-K (Garlan et al., 2004; Kephart e Chess, 2003), pela composição de conjuntos de microcontroladores. Tais microcontroladores não seriam restritos a implementações de serviços providos pelos distintos estágios de uma malha de controle MAPE-K (Pereira et al., 2020). Pelo contrário, os microcontroladores seriam associados com serviços específicos que compõem estágios individuais de um controlador (de Lemos e Potena, 2017). Por exemplo, ao invés de se ter um microcontrolador implementando o estágio de análise da malha de controle MAPE-K, um microcontrolador implementaria serviços associados com teste de integração (Da Silva e De Lemos, 2011), ou checagem de modelos (Sharifloo e Metzger, 2013).

O número de microcontroladores para implementar um controlador dependeria das necessidades do sistema alvo, e da granularidade dos microcontroladores em termos de serviços que eles proveem. Portanto, na abordagem aqui proposta, um controlador para um SA seria implementado como um conjunto de microcontroladores que executam serviços específicos. Tais microcontroladores podem tanto ser da malha de controle aberta quanto de malha de controle fechado. Um microcontrolador da malha de controle aberto requer conhecimento predeterminado sobre o sistema alvo específico. Em contraste, um microcontrolador da malha de controle fechado dinamicamente leva em conta as

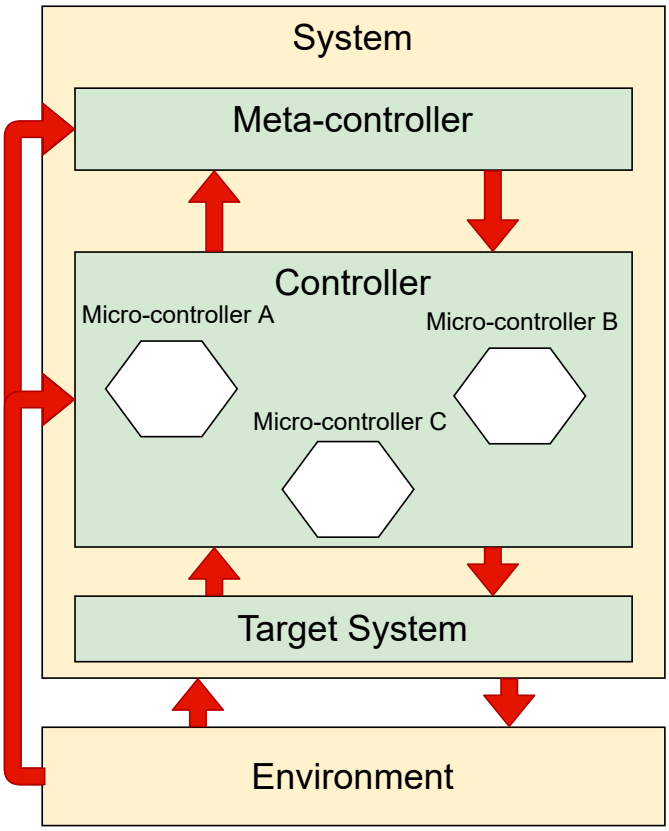

Figura 4.1: Controlador de múltiplas camadas.

necessidades do sistema alvo – por exemplo, um microcontrolador da malha de controle fechado para teste online pode refinar a respectiva estratégia de teste caso a cobertura alcançada não seja suficiente.

Dependendo da diversidade nas atividades presentes em cada estágio do MAPE-K – ao utilizá-lo como modelo de referência – necessita-se da inclusão de vários microcontroladores. Como um exemplo, na Figura 4.2 apresenta-se uma extensão da representação do MAPE-K realizada por de Lemos e Potena (2017). Com a separação de interesses ilustrada no diagrama da figura, pode-se diminuir a complexidade do controlador, pois em um único estágio pode-se demandar diferentes atividades dependendo de *o quê* e *como* as atividades devem ser realizadas para o estágio em particular (Weyns et al., 2013). Retomando-se o exemplos mostrado na Figura 4.2, para o estágio *Monitor*, diferentes técnicas de monitoramento podem ser utilizadas (por exemplo, tempo e/ou evento), de acordo com as características do sistema alvo e do ambiente (de la Iglesia e Weyns, 2015). Neste mesmo contexto, o estágio *Analisador* (*Domínio de Solução*) pode ser associado a diferentes atividades de teste de software (por exemplo, teste de integração e de regressão). Os microcontroladores podem gerar evidência para o respectivo estágio MAPE-K ao qual

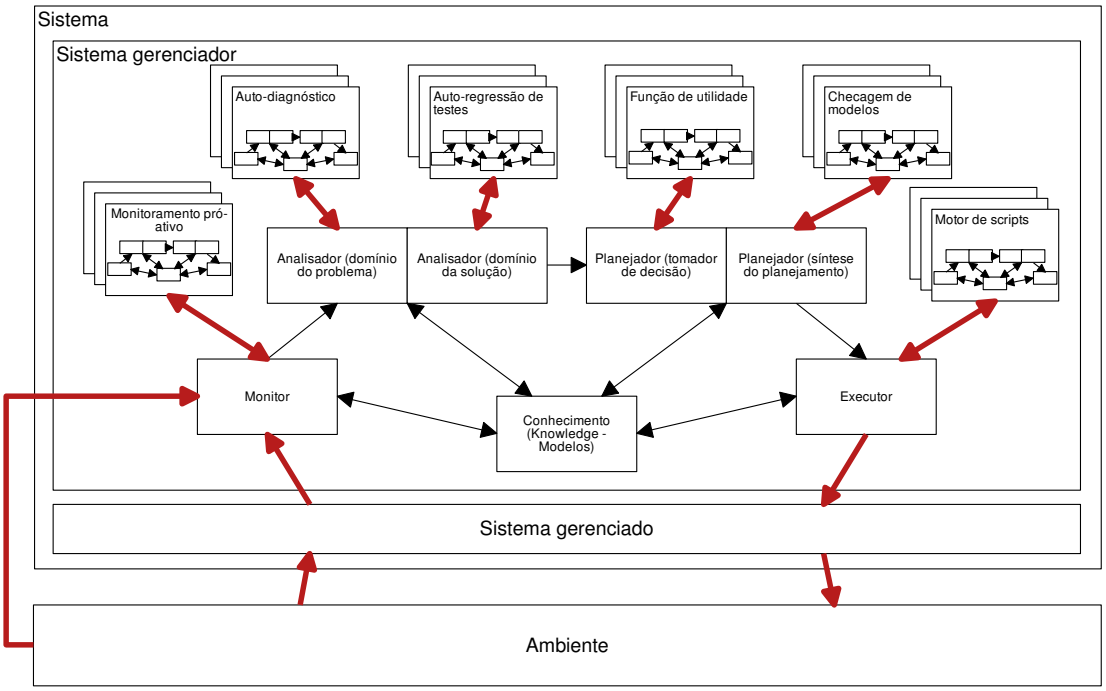

Figura 4.2: Modelo de referência MAPE-K extendido para uso de Controlador Principal e Controladores Específicos – adaptado do trabalho de de Lemos e Potena (2017).

mesmos estão relacionados. Assim, tal evidência de forma coletiva contribui para o bom funcionamento do controlador e do sistema alvo.

de Lemos e Potena (2017) detalharam um componente (neste trabalho de doutorado caracterizado como um microcontrolador) associado ao estágio *Analisador (Domínio do Problema)*. Tal microcontrolador poderia ser responsável pelo autodiagnóstico do sistema alvo (esquerda superior na Figura 4.2), tendo como objetivo a localização de possíveis defeitos relacionados à adaptação a ser executada no sistema alvo. Neste microcontrolador de autodiagnóstico, seu estágio *Monitor* coletaria dados relacionados à configuração arquitetural e o estado operacional dos elementos arquiteturais. Seu estágio *Analisador (Domínio do Problema)* localizaria os componentes defeituosos, gerando evidência para o controlador principal (por exemplo, avaliando níveis de confiabilidade e precisão nos resultados). Assim, o estágio de *Analisador (Domínio da Solução)* seria responsável por técnicas alternativas de diagnóstico. Depois, o estágio *Planejador (Tomador de decisão)* selecionaria a melhor configuração com respeito a técnicas a serem aplicadas. O estágio *Planejador (Síntese de Planejamento)* geraria o plano para implantação do

novo diagnóstico. Por fim, o estágio *Executor* do microcontrolador de autodiagnóstico notificaria o estágio *Analisador (Domínio do Problema)* do controlador sobre o diagnóstico realizado. O autodiagnóstico é um exemplo dos vários microcontroladores que poderiam ser sintetizados com base na Figura 4.2.

Microcontroladores podem ser projetados como microsserviços. Cada microcontrolador pode ser desenvolvido como um processo separado para maximizar a independência de implementação. A comunicação entre microcontroladores pode ser realizada utilizando interfaces e protocolos tais como REST. Alguns dos benefícios de implementar controladores como uma coleção de microcontroladores incluem projeto e implementação independentes, e suporte operacional dinâmico, proporcionando assim atividades como versionamento e escala de serviços.

4.4 Metacontrolador

Se o controlador, caracterizado como uma coleção de microcontroladores, também pode receber adaptações, tem-se então a necessidade de um controlador adicional numa camada superior. Assim, possibilitaria-se o gerenciamento de mudanças ocorrendo no controlador, com eventuais adaptações realizadas no conjunto de microcontroladores. Neste trabalho, nomeou-se tal controlador de nível superior como metacontrolador, como ilustrado na Figura 4.1. Um exemplo de tal controlador poderia ser o Rainbow (Garlan et al., 2004), ou variações do mesmo (Aderaldo et al., 2019). Na Figura 4.1, em amarelo representam-se o sistema e o ambiente; a forma em verde do topo refere-se ao metacontrolador; a forma em verde do meio refere-se ao controlador; e a forma em verde de baixo refere-se ao sistema gerenciado. No controlador, cada hexágono em branco refere-se a um microcontrolador implementado como um microsserviço.

No contexto de SAs, a adaptação que é realizada por um metacontrolador para diferentes conjuntos de microcontroladores seria mais simples porque o "sistema-alvo" do metacontrolador seria apenas uma coleção de microcontroladores, e não uma ampla gama de tipos de componentes oriundos do sistema alvo. O papel do metacontrolador seria orquestrar os serviços providos pelos microcontroladores que implementam o controlador. Ele deve manter uma visão consistente que os microcontroladores têm a respeito do sistema alvo e do ambiente Nii (1986). Isto também inclui o estado dos microcontroladores, assim permitindo microcontroladores serem sem estado (isto é, stateless). Embora microcontroladores devessem ser independentes, a coordenação entre tais devem ser seguir um fluxo de controle similar à malha MAPE-K (Kephart e Chess, 2003). A existência de vários microcontroladores não impede que todos eles sejam capazes de acessar o sistema alvo.

Conflitos potenciais que podem acontecer entre microcontroladores devem ser manipulados pelo metacontrolador.

Em resumo, a abordagem proposta, a de sintetizar controladores a partir de microcontroladores, é benéfica porque as diferentes necessidades de um sistema alvo são atendidas por diferentes conjuntos de microcontroladores específicos. Isso é alcançado devido à flexibilidade estrutural dos conjuntos de microcontroladores, que é gerenciada por um metacontrolador. Assim, a camada do controlador que consiste em uma coleção de microcontroladores reflete nas mudanças que afetam o sistema alvo e o ambiente, enquanto a camada do metacontrolador reflete nas mudanças que afetam o controlador e o ambiente do sistema.

Embora neste trabalho tenha-se descrito um controlador consistindo de duas camadas, conceitualmente, um controlador poderia ter múltiplas camadas (Weyns et al., 2013). Por exemplo, pode-se imaginar uma arquitetura para o controlador consistindo em mais de um metacontrolador. Se uma metacamada adicional seria (ou não) necessária para coordenar vários metacontroladores é especulação para pesquisas futuras.

4.5 Considerações Finais

Neste capítulo descreveu-se a abordagem, a definição de microcontroladores e a definição de metacontroladores. No contexto de adaptações estruturais – que fornecem habilidades para a reconfiguração dinâmica da estrutura de um sistema – ao definir um controlador apto a ser reestruturado em tempo de execução (isto é, baseado em microcontroladores), pode-se aumentar a flexibilidade estrutural do controlador de um SA. Tal controlador não somente realizará adaptações a um sistema alvo, mas também receberá adaptações de uma camada de nível superior (isto é, de um metacontrolador). A fim de demonstrar a viabilidade da abordagem, nos próximos capítulos serão apresentados dois estudos de SAs que fazem uso desta abordagem. O primeiro estudo utiliza o *framework* Android, e o segundo estudo utiliza o *framework Kubow* para realização de experimentos.

Capítulo

5

Estudo Exploratório para Investigar Ganhos com a Abordagem

5.1 Considerações Iniciais

Controladores flexíveis são promissores em suas características de viabilizar a reconfiguração do controlador de acordo com a necessidade do sistema alvo. Para explorar o uso da abordagem, inicialmente optou-se pelo uso do *framework* Android, devido à facilidade de manuseio e implementação. Neste capítulo, descreve-se um estudo piloto utilizando o sistema *PhoneAdapter* a fim de evidenciar características presentes em tempo de projeto, e em tempo de execução.

Em detalhes, neste capítulo, na Seção 5.2 revisita-se brevemente a descrição do sistema *PhoneAdapter* (que já foi introduzido no Capítulo 2), fornecendo-se características sobre o sistema alvo e seu ambiente. Em seguida, na Seção 5.3 descreve-se o experimento, demonstrando quais foram as modificações pontuais no *PhoneAdapter* para que o mesmo fosse utilizável para demonstrar microcontroladores e o metacontrolador. Na Seção 5.4 apresenta-se a avaliação do estudo, ilustrando tanto a manipulação de mudanças em tempo de execução quanto mudanças em tempo de projeto. e como ambos os tipos de mudança afetam o sistema. Na Seção 5.5 discutem-se ameaças à validade do estudo realizado, e, por

fim, nas seções 5.6 e 5.7 apresentam-se as lições aprendidas em conjunto com conclusões do estudo realizado, e as considerações finais este capítulo, respectivamente.

5.2 PhoneAdapter e Android

Neste trabalho, o *PhoneAdapter* foi estendido para ilustrar a abordagem de controladores baseados em microcontroladores. A motivação de escolher esse sistema foi a sua simplicidade, permitindo mostrar como microcontroladores podem ser utilizados para definir controladores flexíveis seguindo um padrão de controle hierárquico Weyns et al. (2013).

Neste estudo, assumiu-se que sensores e atuadores individuais podem falhar arbitrariamente devido ao número de possíveis configurações do sistema alvo ser combinatorial. Ao considerar n sensores e m atuadores, cada um pode estar em operação ou em falha, então 2^{n+m} diferentes configuração poderiam existir para o sistema alvo. Para lidar com a explosão combinatorial, considerou-se que dados de ContextManagers e AdaptationManagers podem ser gerados dependendo da disponibilidade de sensores e atuadores, respectivamente.

O *PhoneAdapter* contém dois sensores, GPS e *Bluetooth*, e uma sonda (do inglês, *probe*) para a aplicação de calendário. Estes sensores e a *probe* podem ser acessados por meio do Android Framework e eles permitem que o ContextManager identifique os diferentes contextos do dispositivo móvel. Uma representação esquemática da arquitetura composta pelo controlador, sistema alvo e ambiente foi apresentada na Figura 2.3.

A máquina de estado — originalmente chamada *Máquina de estados finita de Adaptação*, (ou seja, *Adaptation Finite-State Machine (A-FSM)* Sama et al. (2008)) — ilustrada na Figura 5.1 contém nove possíveis estados contextuais dependendo dos valores dos sensores e *probe*, com o estado General sendo o estado inicial. Essa máquina de estados é chamada ContextManagerAllSensors uma vez que se assume que todos os sensores estão disponíveis.

As transições entre os estados são definidos em termos de regras contextuais que o ContextManager utiliza para identificar as trocas de contextos baseadas em valores dos sensores e probe que são constantemente atualizados (Sama et al., 2008). Um subconjunto das regras contextuais da Figura 5.1 é apresentado na Tabela 5.1. Os diferentes contextos podem ser identificados baseando-se nos sensores e na *probe*. Entretanto, sensores podem falhar, o que podem resultar em um menor número de contextos identificados. Assim, têm-se diferentes espaços contextuais, dependendo do que e como os sensores podem falhar. Por exemplo, se o sensor de GPS falha, o ContextManager não está apto a identificar os estados contextuais Jogging, Outdoor, e DrivingFast, como ilustrados na Figura 5.2 (ContextManagerNoGPS). Variações da máquina de estados que consideram falhas do sensor

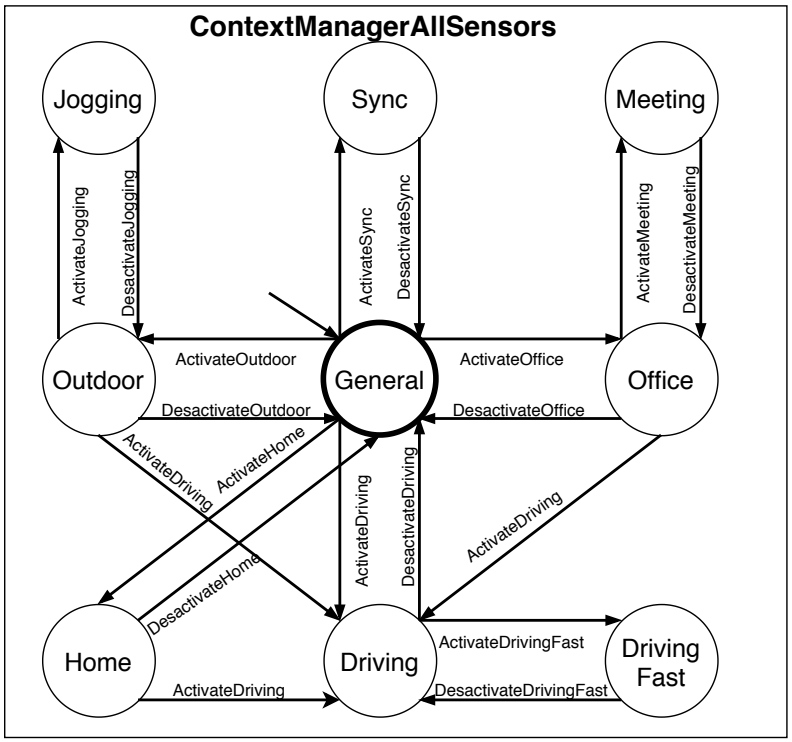

Figura 5.1: A-FSM original (Sama et al., 2008).

de Bluetooth, e falhas em ambos os sensores (GPS e Bluetooth) são apresentadas nas figuras 5.3 e 5.4, respectivamente.

Tabela 5.1: Um subconjunto de regras contextuais do ContextManagerAllSensors e as saídas de adaptações correspondentes.

Id	Rule Name	Current State	New State	Full Predicate	Adaptation Output	
					Volume	Vibration
a	ActivateOutdoor	General	Outdoor	GPS.isValid() && !GPS.location()=home && !GPS.location=office	100	OFF
b	DeactivateOutdoor	Outdoor	General	!ActivateOutdoor	50	OFF
c	ActivateJogging	Outdoor	Jogging	GPS.isValid() && GPS.speed() >5	25	OFF
d	DeactivateJogging	Jogging	Outdoor	!ActivateJogging	100	OFF
m	ActivateMeeting	Office	Meeting	Time >= meeting_start && BT.count() >= 3	0	OFF

Estratégias de Adaptação: Com relação aos atuadores, estes referem-se a diferentes recursos de controles do PhoneAdapter sobre o Android Framework, tais como volume de toque, vibração, intensidade de iluminação, volume do áudio e chamadas do telefone. Como um exemplo, no contexto de Meeting, o AdaptationManager deve silenciar o áudio de toque e desabilitar a vibração (conforme a regra *m* na Tabela 5.1). Como outro exemplo, no contexto Jogging, o AdaptationManager deve diminuir o volume do áudio de toque e desabilitar a vibração (conforme a regra *c* na Tabela 5.1).

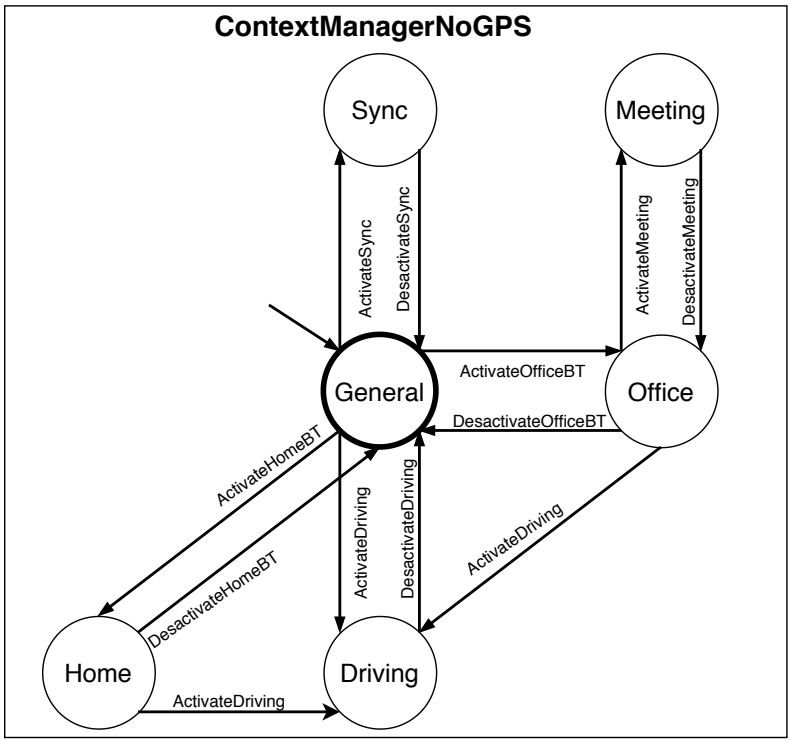

Figura 5.2: A-FSM para falhas do sensor de GPS.

Uma vez que atuadores podem falhar, as regras de adaptação tem que ser modificadas de acordo com a disponibilidade dos atuadores. Similarmente à máquina de estados para os diferentes contextos processados pelo ContextManager, diferentes máquinas de estados definindo o comportamento adaptativo (regras) do AdaptationManager foram especificadas de acordo com a disponibilidade de atuadores. Estas máquinas de estados definem como o AdaptationManager reage com relação à troca de contexto ao utilizar regras de adaptação que usam ou não atuadores com falhas.

O controlador do *PhoneAdapter* consiste em dois componentes que incorporam as funcionalidades básicas de uma malha de controle de controle (Figura 2.3).

5.3 Experimento

Neste estudo utilizaram-se diferente espaços de adaptação e contexto como uma justificativa para ter múltiplos ContextManagers e AdaptationManagers, respectivamente, para ilustrar como controladores podem ser adaptados de acordo com as necessidades atuais do sistema alvo (por exemplo, disponibilidade de sensores e atuadores).

O resultado da refatoração do controlador para uma solução que é baseada em micro-controladores, nomeando-o como $\mu PhoneAdapter$, é ilustrado na Figura 5.5. Detalhes são apresentados a seguir.

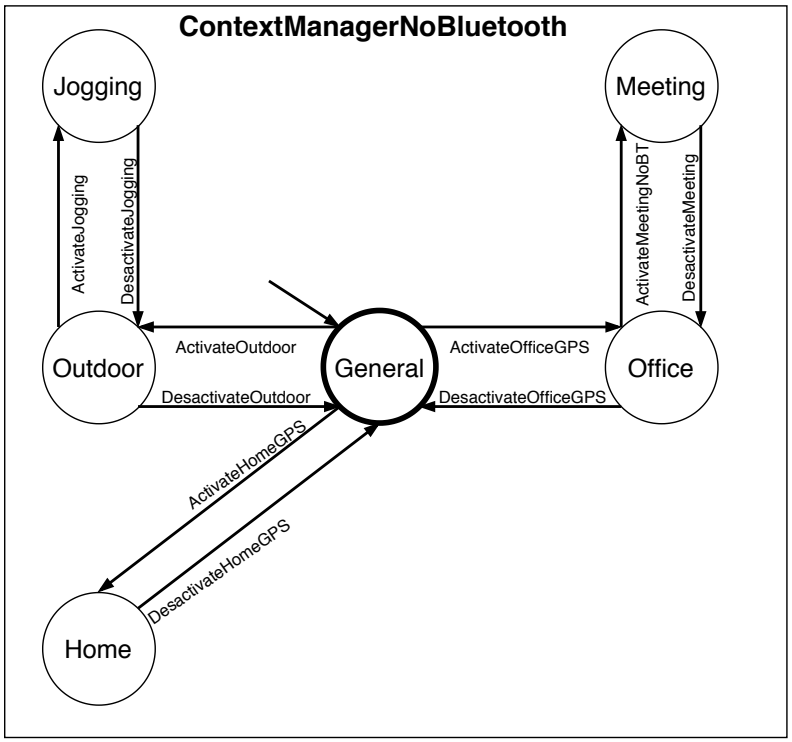

Figura 5.3: A-FSM para falhas do sensor de Bluetooth.

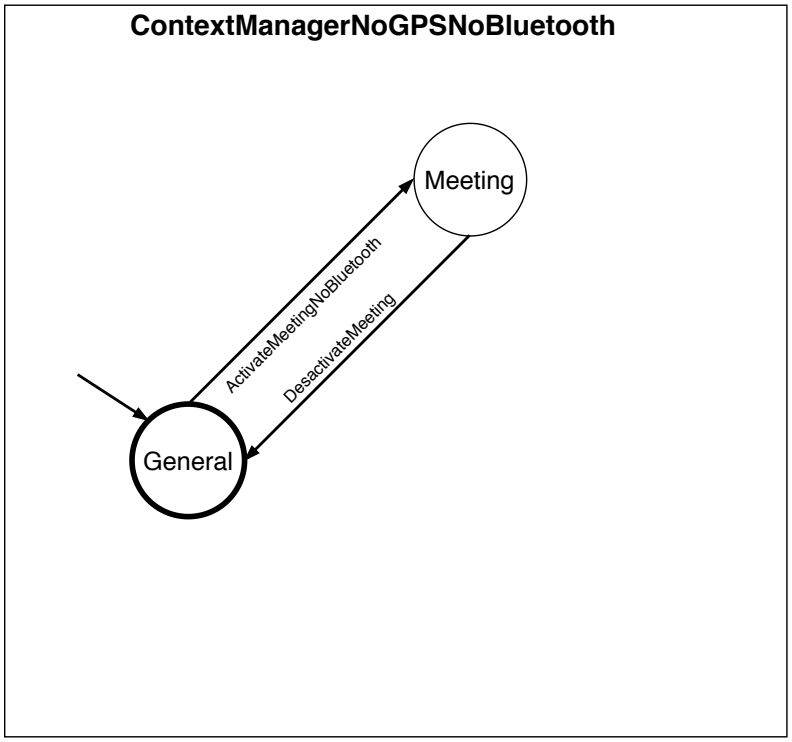

Figura 5.4: A-FSM para falhas do sensor de GPS e Bluetooth.

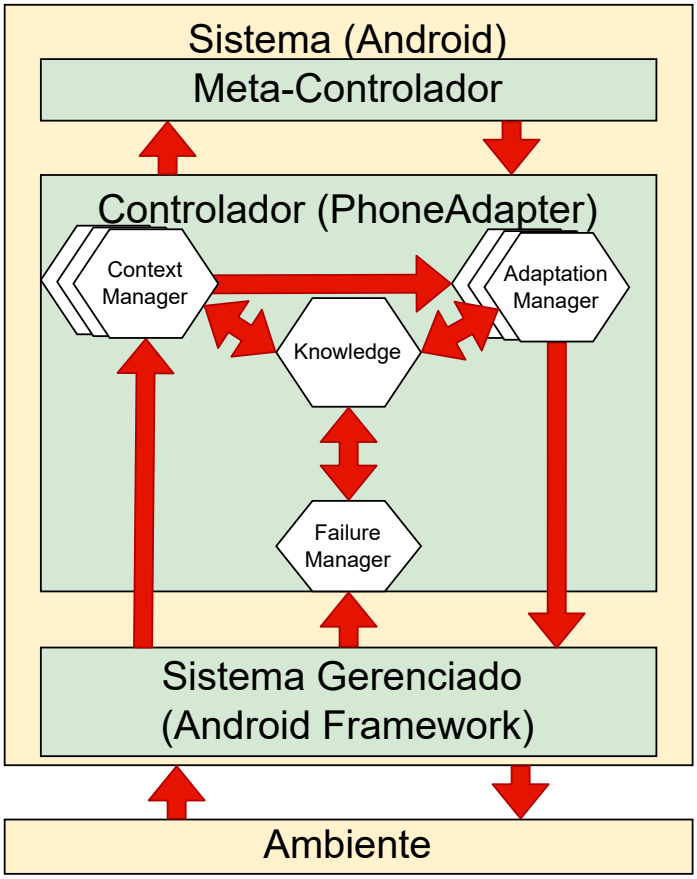

Figura 5.5: *PhoneAdapter* implementado com microcontroladores.

Microcontroladores

Ao invés de um único par de um ContextManager e AdaptationManager, o projeto do *µPhoneAdapter* consiste na combinação de vários microcontroladores que podem capturar variações dos ContextManager e AdaptationManager originais. Estas variações dependem de falhas nos sensores e atuadores.

A motivação de ter variações de ContextManager e AdaptationManager é economizar bateria do dispositivo. Com isto, evita-se a invocação desnecessária de sensores que tenham falhado. Além disso, evita-se a adaptação do sistema alvo sem sucesso devido a atuadores terem falhado.

Além de uma combinação de variações de ContextManager e AdaptationManager, o controlador do *µPhoneAdapter* inclui os microcontroladoresFailureManager e Knowledge. O primeiro é utilizado para monitorar a situação operacional dos sensores e atuadores. Assim, com base em tal monitoramento, definem-se modelos de mudanças do sistema alvo

e ambiente (ou seja, modelos de consciência (Giese et al., 2017)). Com isso, o sistema pode se auto-adaptar e ser sensível ao contexto. Ao externalizar o Knowledge para um microcontrolador dedicado, consideraram-se os outros microcontroladores para serem componentes sem um estado armazenado (do inglês, *stateless*).

Metacontrolador

O papel do metacontrolador é gerenciar o controlador em termos de reconfiguração dos microcontroladores. Isso será realizado a depender do estado operacional dos sensores e atuadores. Assim, o Kownledge permite que a configuração atual dos microcontroladores seja avaliada, auxiliando a decisão relacionada à necessidade de adaptações no controlador. Portanto, o metacontrolador terá subsídios para selecionar os microcontroladores apropriadamente, sem afetar a operação do sistema alvo.

Variantes de Microcontroladores e suas Implantações

As variações dos microcontroladores Context-Manager e AdaptationManager do *μPhoneAdapter* foram desenvolvidas utilizando-se serviços criados sob medida – similares aos apresentados no trecho de código da Figura 5.6.

Na Tabela 5.2 apresenta-se um subconjunto desses microcontroladores com as respectivas operações, entradas e saídas. As operações definidas por cada microcontrolador podem ser mapeadas para elementos de código interno. Por exemplo, ao considerar o microcontrolador AdaptationManagerAllEffectors (trecho de código ilustrado na Figura 5.6), as operações `/processNewContext` e `/processRuleChange` são mapeadas para o método `onReceive` que representa um evento no Android Framework. Dado que um único método (`onReceive`) é mapeado para duas operações de microcontroladores, decisões internas (baseadas nos valores de ações embarcadas no objeto `Intent`) definem a lógica correta que está associada a uma operação particular a ser processada.

A implantação de um microcontrolador é realizada por duas operações: `unregisterReceiver(...)`, que desabilita o microcontrolador atual (por exemplo, ContextManagerAllSensors), e `registerReceiver(...)`, que habilita um microcontrolador alternativo que está em conformidade com o contexto atual (por exemplo, ContextManagerNoGPS, caso o sensor de GPS falhe).

Como já mencionado, ao assumir que os sensores e atuadores podem falhar, definiram-se ContextManagers e AdaptationManager alternativos para compor o controlador. Na parte central da Figura 5.6 ilustram-se os microcontroladores do *μPhoneAdapter*, e os respectivos fluxos de dados (as flechas seguem o mesmo fluxo como na Figura 5.5). Se falhas acontecerem, por exemplo, no sensor de *GPS* e no atuador de *Ringtone*, o con-

trolador será reconfigurado com dois microcontroladores alternativos (especificamente, ContextManagerNoGPS e AdaptationManagerNoRingtone). A troca do ContextManager ocorre quando o FailureManager detecta e armazena uma falha de sensor no Knowledge (ao utilizar a operação `/verifySensors` listada na Tabela 5.2). Um cenário similar ocorre quando o FailureManager detecta e armazena alguma falha de *effectors* (ao utilizar a operação `/verifyEffectors` listada na Tabela 5.2). Como um exemplo, em caso do *effector* Ringtone estiver falhando, realiza-se uma troca do AdaptationManagerAllEffectors pelo AdaptationManagerNoRingtone.

Tabela 5.2: Um subconjunto das operações dos microcontroladores no *µPhoneAdapter*.

Micro-controller	Operation	Input	Output
ContextManagerAllSensors	/generateContext	-	An intent message with: GPS, Bluetooth, Calendar data
	/sensingBluetooth	-	A bluetooth collection
	/locationListener	-	Latitude, Longitude data
AdaptationManagerAllEffectors	/processNewContext	newContext: GPS, Bluetooth and Calendar data	New state in volume; vibration; airplane
	/processRuleChange	ruleChange	Back to general state
Knowledge	/NewSensorContext	SensorContext: GPS status and Bluetooth status	An intent message with a new generated sensors context
	/NewEffectorData	EffectorData: Ringtone status, vibration status, volume number	An intent message with a new generated effectors data
FailureManager	/verifySensors	GPS, Bluetooth status	An intent message localising a sensor failure
	/verifyEffectors	Audio, Vibration	An intent message localising an effector failure

Similarmente aos microcontroladores descritos na seção anterior, o MetaControlador na aplicação *µPhoneAdapter* é caracterizado como um microsserviço do framework Android, como ilustrado na Tabela 5.3. A ativação e desativação dos microcontroladores é gerenciada pelo serviço Metacontrolador com suporte do serviço FailureManager, levando em conta se sensores e atuadores individuais estão funcionando ou não.

Tabela 5.3: Um subconjunto de operações do Meta-Controller no *µPhoneAdapter*.

Meta-controller	Operation	Input	Output
Meta-Controller	/sensorsFailure	failure message regarding GPS or Bluetooth status	A new ContextManager
	/effectorsFailure	failure message regarding Audio or Vibration	A new AdaptationManager

5.4 Avaliação

A demonstração utilizando o *PhoneAdapter* é discutida na sequência, enfatizando-se a flexibilidade estrutural do controlador e como ele apoia mudanças em (i) tempo de execução e (ii) tempo de projeto.

```
1   private class AdaptationManagerAllEffectors
2                       extends BroadcastReceiver{
3
4       @Override
5       public void onReceive(Context c, Intent i) {
6           String action=i.getAction();
7           if(action.equals("newContext")){
8               ...
9           }else if(action.equals("ruleChange")){
10              ...
11          }else{
12              ...
13          }
14      }
15  }
```

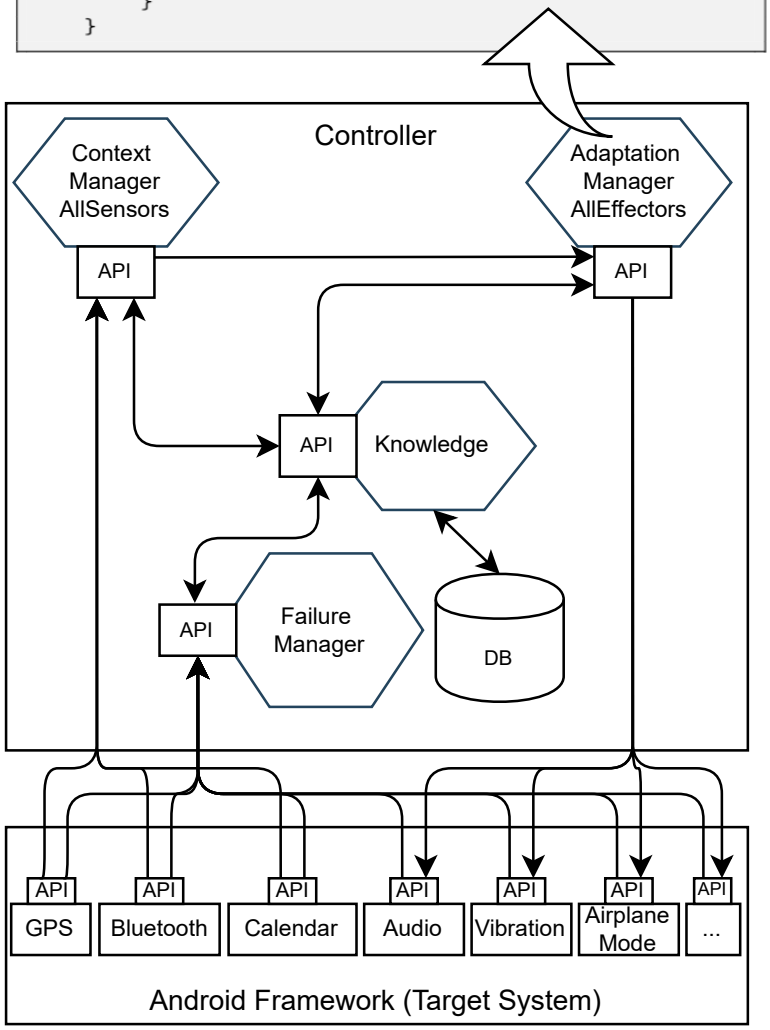

Figura 5.6: Uma configuração possível do *µPhoneAdapter* de um trabalho anterior (Siqueira et al., 2020b).

Manipulando mudanças em tempo de execução: Um controlador é estruturalmente flexível em tempo de execução se ele pode adaptar sua estrutura a fim de manipular o contexto

em cenários de falhas, tais como as ilustradas nas Figuras 5.1 e 5.2. Nessa linha, alguns destaques em relação à implementações original e reestruturada do *PhoneAdapter* são mostrados a seguir.

- ***PhoneAdapter* Original:** Se o sensor de GPS falhar, o *PhoneAdapter* gera estados inconsistentes, por exemplo, um Lat. de 0.0; e um Long. de 0.0. Esta falha impacta o monitoramento dos dados de GPS realizado pelo ContextManager. Em geral, consequências podem ser: geração de dados de sensor inconsistentes (por exemplo, localização incorreta); processamento e, consequentemente, uso de bateria desnecessários (por exemplo, ao tentar acessar o sensor de GPS); e ativação de estados inconsistentes (por exemplo, estado "Driving" está habilitado mesmo se "Driving Fast" está acontecendo, ou seja, velocidade foi incorretamente calculada).

- ***μPhoneAdapter*:** Se a mesma falha do sensor de GPS ocorrer, ela é identificada pelo microcontrolador FailureManager. Assim, o MetaControlador mitiga as respectivas consequências. Depois, o FailureManager classifica a falha para o serviço Knowledge, e o MetaControlador detecta as mudanças e realiza, por exemplo, a troca entre os ContextManagerAllSensors e o ContextManagerNoGPS. Este cenário é ilustrado nas Figuras 5.1 e 5.2. Nota-se que o número de transições na máquina de estado do ContextManagerNoGPS é menor do que para o ContextManagerAllSensors.

Manipulando mudanças em tempo de projeto: Um controlador é estruturalmente flexível em design time, se o controlador é facilmente trocado por um alternativo. Se um controlador é complexo de forma a incorporar mais comportamentos das mudanças de contextos e falhas, isto pode dificultar a substituição. Entretanto, se microcontroladores expressam seus comportamentos de modo simplificado, eles podem ser facilmente substituídos. Nessa linha, alguns destaques em relação à implementações original e reestruturada do *PhoneAdapter* são apresentados a seguir.

- ***PhoneAdapter* Original:** Para lidar com falhas em sensores e atuadores, ou para diferentemente processar valores de contextos ou regras, na versão original é necessário refatorar o ContextManager e/ou o AdaptationManager para estas mudanças ou para novos requisitos. Isto faz com que tais componentes sejam mais complexos, dificultando a respectiva substituição.

- ***μPhoneAdapter*:** Se o comportamento do controlador precisa ser modificado, necessita-se remover ou incluir microcontroladores adicionais ou variações (por exemplo,

implementações personalizadas adicionais de um ContextManager). Estes são independentemente desenvolvidos em tempo de projeto, e eles são coordenados em tempo de execução pelo MetaControlador.

Em linhas gerais, na comparação de ambas versões do *PhoneAdapter*, identificam-se benefícios que a abordagem de microcontroladores provê em termos de flexibilidade estrutural do projeto de controladores. A abordagem habilita a definição de novos microcontroladores para compor controladores de acordo com novos requisitos, sem requerer substanciais mudanças de código. *Microcontroladores* e suas variações podem ser ativadas ou desativadas em tempo de execução. Isto habilita o controlador a operar apropriadamente sobre o sistema alvo.

5.5 Ameaças à Validade

A viabilidade da abordagem com ênfase na flexibilidade de controladores foi demonstrada com um estudo de caso em aplicações Android. Entretanto, existem alguns destaques que afetam a validade dos resultados. Uma ameaça à validade interna refere-se ao fato de que o Android não é o ambiente mais apropriado para implementar fielmente microcontroladores como microsserviços. Utilizaram-se os recursos disponíveis no Android para incorporar princípios básicos associados a microsserviços.

Outra ameaça à validade interna é relacionada ao reúso. Evidência para apoiar a reusabilidade seria obtida ao utilizar um microcontrolador em diferentes configurações arquiteturais, além disso também utilizando diferentes sistemas alvo e diferentes domínios de aplicação. Entretanto, neste trabalho, demonstrou-se que microcontroladores podem ser utilizados em diferentes configurações arquiteturais de um controlador para um único sistema alvo.

Outra ameaça está relacionada à flexibilidade em sintetizar controladores utilizando-se um conjunto de microcontroladores. Demonstrou-se isso no contexto de um pequeno número de microcontroladores e um metacontrolador simples e sob medida. Entretanto, para contornar a limitação de um metacontrolador sob medida, o Rainbow (Garlan et al., 2004) poderia ser utilizado. De fato, o Rainbow foi utilizado como base para a definição de microcontroladores e do metacontrolador em um estudo que é apresentado no próximo capítulo desta dissertação,

Com relação a ameaças externas, uma questão chave que limita a síntese de controladores baseados em microcontroladores, bem como a reusabilidade de tais componentes, diz respeito à falta de repositórios com microcontroladores existentes. Uma avaliação minuciosa

da reutilização somente poderia ser realizada pela comunidade, como testemunhado por Rainbow, cuja avaliação inicial da reutilização foi apenas preliminar (Garlan et al., 2004). Porém, outras avaliações com uso do Rainbow foram realizadas em outros trabalhos (Cámara et al., 2013; Schmerl et al., 2014; Yuan et al., 2013), o que também auxiliou na motivação de conduzir o estudo descrito no próximo capítulo.

5.6 Lições Aprendidas

Um controlador estruturalmente flexível, no contexto de implantação contínua (do inglês, CD - *continuous deployment*), promoveria uma responsividade para mudanças em tempo de projeto e de execução, além da reusabilidade de microcontroladores por meio de várias aplicações. Além disso, no contexto de CD, o projeto do controlador precisa ser responsivo para mudanças que afetam o software de um sistema alvo. A viabilidade de toda a abordagem foi demonstrada no contexto de um estudo de caso utilizando o *framework* Android.

Nota-se que um controlador de duas camadas tende a aumentar a complexidade desses sistemas, e isso pode restringir a aplicabilidade da solução. Por exemplo, uma solução baseada em microcontroladores não seria apropriada para aplicações que tenham quantidade limitada de recursos de processamento. Entretanto, outros sistemas se beneficiariam da abordagem, como aqueles em que o controlador precisa ser dinamicamente reconfigurável em resposta a mudanças, ou até mesmo sistemas que contenham uma quantidade de mudanças que podem afetar a qualidade de serviço do sistema como um todo.

5.7 Considerações Finais

No estudo apresentado neste capítulo, teve-se como proposta a sintetização de controladores a partir de uma coleção de microcontroladores, tais como sendo microsserviços específicos. O gerenciamento desses microcontroladores é feito por um metacontrolador, que poderia ser implementado utilizando um controlador monolítico como o Rainbow. Além disso, ao realizar experimentos utilizando o ambiente Android, não é possível utilizar *frameworks* tais como Rainbow para realizar comparações com soluções já propostas pela comunidade. Isto decorre do fato de que os recursos desenvolvidos não podem ser utilizados em diferentes sistemas alvo. Portanto, desenvolveram-se outros estudos utilizando o *framework* Rainbow em tecnologias heterogêneas, conforme descrito no próximo capítulo.

Capítulo

6

Estudos exploratórios para avaliar diferentes configurações de controladores

6.1 Considerações Iniciais

No estudo do *PhoneAdapter*, notaram-se limitações quanto a representar microsserviços e na comparação com diferentes estilos arquiteturais para sistemas adaptativos presentes na literatura. Devido a tais limitações, optou-se pelo uso de outro conjunto de ferramentas para utilizar o *framework* Rainbow, assim conduzindo-se outro conjunto de estudos com o mesmo. A fim de investigar e avaliar o desempenho do uso de microcontroladores e um metacontrolador, neste capítulo descrevem-se experimentos realizados utilizando o sistema Kube-ZNN. Esse novo experimento, que deste ponto em diante será chamado de *experimento Kube-ZNN,* teve o objetivo de comparar o uso de controladores baseados em microcontroladores e outras abordagens. Com o sistema alvo Kube-ZNN, foi possível utilizar o *framework* Kubow que é baseada no Rainbow. Com isso, possibilitou-se o uso de configurações arquiteturais já disponíveis na literatura.

Em detalhes, neste capítulo, na Seção 6.2 revisita-se brevemente a descrição do sistema Kube-ZNN, descrito no Capítulo 2, fornecendo-se características sobre o sistema alvo e seu ambiente. Na Seção 6.3 descreve-se o experimento Kube-ZNN, demonstrando qual foi a principal definição acerca das diferentes configurações arquiteturais utilizadas no estudo. Na Seção 6.4 apresenta-se a avaliação do estudo, destacando-se os principais resultados com relação à escalabilidade e à fidelidade de conteúdo, comparando o desempenho das três diferentes configurações arquiteturais. Nas Seção 6.5 apresentam-se as ameaças à validade do experimento. Por fim, nas Seção 6.6 apresentam-se as conclusões tiradas com a execução do experimento e também as considerações finais deste capítulo.

6.2 Kube-ZNN e Kubow

Neste trabalho, adotou-se o Kube-ZNN (Aderaldo e Mendonça, 2022), que é uma nova versão de implantação do ZNN.com baseado em aplicações conteinerizadas usando Docker e Kubernetes. Cada servidor em Kube-ZNN (isto é, uma máquina física ou uma máquina virtual) é um objeto pod em Kubernetes. Para gerenciar o Kube-ZNN, precisa-se realizar uma implementação do *framework* Kubow para lidar com estratégias de adaptação que gerenciem o ZNN.com. As requisições do usuário são manipuladas por um objeto service – que trabalha de forma similar ao *lbproxy* do ZNN.com original, como ilustrado na Figura 2.6 – que é responsável por distribuir e entregar em tempo de execução as requisições de acordo com a carga do ambiente.

Os artefatos utilizados neste trabalho estão disponível em `https://github.com/californi/ExperimentsForMicrocontrollers/`

Estratégias de Adaptação: Existem quatro estratégias de adaptação associadas ao experimento Kube-ZNN:

- IncreaseServers: Esta estratégia aumenta o número de servidores disponíveis, e utiliza a tática addReplica. Uma vez que existem mais recursos, a latência do Kube-ZNN é diminuída.

- DecreaseServers: Diferentemente da estratégia anterior, esta é responsável por diminuir o número de servidores disponíveis, utilizar a tática removeReplica. Assim, a latência do Kube-ZNN tende a aumentar quando número de requisições de usuário é aumentado.

- IncreaseMediaSize: Esta estratégia aplica melhoria na qualidade de mídia provida pelo sistema alvo. Nota-se que a qualidade de mídia entregue pelo Kube-ZNN é por padrão alta, mas quando o número de requisições aumenta, a latência também aumenta.

- DecreaseMediaSize: Em contraste com a estratégia anterior, esta estratégia diminui a qualidade da mídia. Consequentemente, esta estratégia garante que a latência é mantida em um nível aceitável, reduzindo a qualidade da mídia, isso quando o número de requisições de usuário começa a aumentar a latência do Kube-ZNN.

6.3 Experimento

Atributos adicionais: Além dos atributos de escalabilidade e de fidelidade da versão original do ZNN.com, neste trabalho definiu-se um terceiro atributo:

- **Número de falhas**: Dependendo do número de falhas que afetam as réplicas de servidores do Kube-ZNN, diferentes estratégias podem ser selecionadas para aplicar melhorias entre a escalabilidade e a fidelidade com relação à experiência do usuário.

Recursos tais como CPU, memória e armazenamento são associados aos *Clusters Kubernetes*. Quando recursos não estão disponíveis, uma falha é disparada pelo Kubernetes. Neste trabalho, utilizaram-se somente falhas de CPU, que são manipuladas utilizando-se o *Kubernetes API*.

Estratégias de adaptação adicionais: Baseado no atributo *número de falhas*, três estratégias adicionais foram definidas:

- ActivateNoFailureRate: Se não há falhas no servidor, Kube-ZNN provê alta escalabilidade e alta fidelidade.
- ActivateLowFailureRate: Se a taxa de falhas nos servidores é baixa, a escalabilidade é adaptada para baixa, mas a fidelidade continuará alta.
- ActivateHighFailureRate: Se a taxa de falhas nos servidores é alta, Kube-ZNN provê baixa escalabilidade e baixa fidelidade.

Na Tabela 6.1 sumarizam-se as estratégias de adaptação, e suas respectivas táticas que foram definidas na versão do Kube-ZNN deste estudo. Para a execução de uma estratégia, o predicado associado a tal estratégia deve ser satisfeito. Por exemplo, para a execução de IncreaseServers e da tática addReplica, ambas as variáveis lowSLO e canAddReplica do predicado associado devem ser verdadeiras. A primeira variável mostra se a taxa de perda das requisições de usuários está alta. A segunda variável mostra se o limite de servidores foi atingido.

Os predicados que levam em conta o número de falhas de CPU são definidos como na sequência:

- noFailureRate: É *verdadeiro* quando não há falha de CPU por 30 segundos.
- lowFailureRate && isScalabilityA: É *verdadeiro* quando a taxa de falhas está 0.5 por 30 segundos, e o microcontrolador ScalabilityA está ativo.
- highFailureRate && isScalabilityA: É *verdadeiro* quando a taxa de falha está 1.0 por 30 segundos, e o microcontrolador ScalabilityA está ativo.

Tabela 6.1: Regras nas estratégias de adaptação.

Stitch				Arquitetura	
Atributo	**Estratégia**	**Tática**	**Predicado**	**Configuração**	**Componente**
Scalability	IncreaseServers	addReplica	lowSLO && canAddReplica	Mon-KZ	Controller
				Des-KZ	ScalabilityA, ScalabilityB
				Meta-KZ	ScalabilityA, ScalabilityB
Scalability	DecreaseServers	removeReplica	highSLO && canRemoveReplica	Mon-KZ	Controller
				Des-KZ	ScalabilityA, ScalabilityB
				Meta-KZ	ScalabilityA, ScalabilityB
Fidelity	IncreaseMediaSize	raiseFidelity	highSLO && lowFidelity	Mon-KZ	Controller
				Des-KZ	FidelityA, FidelityB
				Meta-KZ	FidelityA, FidelityB
Fidelity	DecreaseMediaSize	lowerFidelity	lowSLO && cannotAddReplica	Mon-KZ	Controller
				Des-KZ	FidelityA, FidelityB
				Meta-KZ	FidelityA, FidelityB
Failures	ActivateNoFailureRate	addHighScalabilityHighQuality	noFailureRate	Mon-KZ	Controller
				Meta-KZ	Meta-controller
Failures	ActivateLowFailureRate	activateLowScalabilityHighQuality	lowFailureRate && isScalabilityA	Mon-KZ	Controller
				Meta-KZ	Meta-controller
Failures	ActivateHighFailureRate	addLowScalabilityLowQuality	highFailureRate && isScalabilityA	Mon-KZ	Controller
				Meta-KZ	Meta-controller
Failures	ActivateFidelityB	addLowQuality	highFailureRate	Des-KZ	FidelityA
Failures	DeactivateFidelityB	removeLowQuality	noFailureRate || lowFailureRate	Des-KZ	FidelityA
Failures	ActivateFidelityA	addHighQuality	noFailureRate || lowFailureRate	Des-KZ	FidelityB
Failures	DeactivateFidelityA	removeHighQuality	highFailureRate	Des-KZ	FidelityB
Failures	ActivateScalabilityB	addLowScalability	lowFailureRate || highFailureRate	Des-KZ	ScalabilityA
Failures	DeactivateScalabilityB	removeLowScalability	noFailureRate && (!canAddReplica || isScalabilityB)	Des-KZ	ScalabilityA
Failures	ActivateScalabilityA	addHighScalability	noFailureRate	Des-KZ	ScalabilityB
Failures	DeactivateScalabilityA	removeHighScalability	lowFailureRate || highFailureRate	Des-KZ	ScalabilityB

Ressalta-se que para a configuração descentralizada (ou seja, Des-KZ, a qual é descrita na próxima seção), derivaram-se oito estratégias específicas que são incorporadas nos microcontroladores da respectiva configuração. Destaca-se também que tais estratégias específicas precisamente refletem o raciocínio incorporado nas estratégias ActivateNoFailureRate, ActivateLowFailureRate, e ActivateHighFailureRate; Tais estratégias foram implementadas a fim de habilitar os microcontroladores a se adaptarem por eles mesmos, dependendo da taxa de falhas atual afetando as réplicas de servidores. Essas estratégias que são específicas configuração Des-KZ são listadas nas últimas oito linhas da Tabela 6.1.

Para demonstrar e avaliar a abordagem deste trabalho, definiram-se três configurações de controladores que são representativas de soluções existentes: um controlador monolítico (Mon-KZ); um controlador decentralizado (Des-KZ); e um controlador decentralizado com um metacontrolador (Meta-KZ). Nas figuras 2.4, 6.1 e 6.2 ilustram-se as estruturas e o relacionamento entre os principais componentes das três configurações arquiteturais. Mais detalhes são apresentados na sequência.

Mon-KZ: esta configuração é identificada como um controlador monolítico uma vez que um único controlador implementa todas as estratégias e táticas de adaptação. Esta é uma evoluação da implementação original de Aderaldo et al. (2019) a fim de incluir um novo conjunto de estratégias e táticas para lidar com o número de falhas nas réplicas de servidores. Como ilustrado na Figura 2.4, no Capítulo 2, Mon-KZ inclui uma única instância *Kubow* (nomeado como *Controller*).

As estratégias levam em conta as variações na taxa de falhas objetivando adaptar o comportamento do controlador para lidar com um número maior ou menor de servidores Kube-ZNN ativos no sistema alvo (isto é, as adaptações com respeito à propriedade de escalabilidade), e a qualidade de mídia mais alta ou mais baixa que é distribuída aos servidores Kube-ZNN para os usuários (isto é, as adaptações com respeito à propriedade de fidelidade de conteúdo).

Des-KZ: esta é uma implementação descentralizada da configuração Mon-KZ, como ilustrado na Figura 6.1. Ela consiste de cinco microcontroladores, dos quais quatro são implementados como instâncias *Kubow* (nomeadas ScalabililyA, ScalabililyB, FidelityA e FidelityB), e um implementado sob medida (FailureManager) a fim de demonstrar a possibilidade de integrar projetos de controladores heterogêneos. Analogamente, para os comportamentos implementados no controlador Mon-KZ, os microcontroladores do Des-KZ que lidam com escalabilidade e fidelidade têm duas variações cada. Ao passo que ScalabililyA e FidelityA, respectivamente, lidam com alto número de servidores ativos e alta

qualidade de mídia, suas variações ScalabililyB e FidelityB manipulam um menor número de servidores e uma qualidade de mídia reduzida, respectivamente.

De acordo com as estratégias descritas anteriormente, somente uma variação de cada microcontrolador está ativa em um momento em particular, dependendo da taxa de falhas gerada pelo FailureManager.

Ressalta-se, novamente, que as falhas consideradas nestas implementações dizem respeito a falhas no Kube-ZNN relacionadas à falta de recursos de CPU. A despeito disso, nota-se que o FailureManager pode ser facilmente adaptado para manipular outros tipos de falhas.

O processo de decompor o controlador monolítico em microcontroladores descentralizados levou em consideração a extração dos microcontroladores a partir das estratégias Stich utilizadas no Rainbow para o Kube-ZNN.

Meta-KZ: Esta configuração inclui um metacontrolador para controlar configuração de microcontroladores, como ilustrado na Figura 6.2. Diferentemente da Des-KZ, em que a coreografia de microcontroladores é responsável pela adaptação, na configuração Meta-KZ o metacontrolador orquestra as adaptações. Dependendo do estado gerado pelo microcontrolador FailureManager (isto é, a taxa de falha), o metacontrolador – que é também implementado como um controlador *Kubow* – seleciona as variações de microcontrolador Scalability e Fidelity que devem ser habilitadas. As estratégias para adaptar o controlador são descritas na Tabela 6.1.

6.3.1 Recursos para cada Configuração

Nas duas últimas colunas da Tabela 6.1 sumarizam-se, para cada configuração arquitetural, qual é a responsabilidade dos componentes do controlador, a fim de implementar as estratégias de adaptação (como ilustrado nas figuras 2.4, 6.1 e 6.2).

Uma vez que a descentralização tende a aumentar o número de *pods*, a Tabela 6.2 captura o número de *pods* associados a cada uma das três configurações. Durante a avaliação e comparação de diferentes configurações, esta informação é importante para entender o impacto da decentralização.

Em particular, com respeito ao sistema alvo Kube-ZNN, o número de *pods* pode variar de 1 a N; nos experimentos definiram-se dois limites: 4 e 10 (mais detalhes a respeito são abordados na Seção 6.4). Na Tabela 6.2 sumariza-se os demais componentes que requerem um *pod* cada, com exceção do FailureManager que requer dois *pods*. Consequentemente, o intervalo do total de *pods* Kubernetes para cada configuração em particular é descrito na última linha da tabela.

Com relação a quanto recurso é demandado por cada *pod*, as primeiras duas colunas da Tabela 6.2 sumarizam tal demanda em termos de CPU e memória. As unidades utilizadas para representar demanda de CPU e memória são, respectivamente, milliCPUs (m) e Gigabytes (Gi). Na coluna CPU, 1000m representa uma alocação completa de uma CPU e 250m representa 1/4 de uma CPU, ao passo que na coluna Memória 100mi representa 100 megabytes.

Tabela 6.2: Configurações arquiteturais e respectivos recursos.

Recurso por componente			**Número de pods Kubernetes**		
CPU	**Memória**	**Componente**	**Mon-KZ**	**Des-KZ**	**Meta-KZ**
250m	100mi	Target System (Kube-ZNN)	[1 ... 4/10]	[1 ... 4/10]	[1 ... 4/10]
1000m	1Gi	Controller	1		
1000m	1Gi	ScalabilityA		1	1
1000m	1Gi	ScalabilityB		1	1
1000m	1Gi	FidelityA		1	1
1000m	1Gi	FidelityB		1	1
200m	64mi	FailureManager		2	2
1000m	1Gi	Meta-controller			1
Intervalo do total de pods Kubernetes			[2 ... 5/11]	[7 ... 10/16]	[8 ... 11/17]

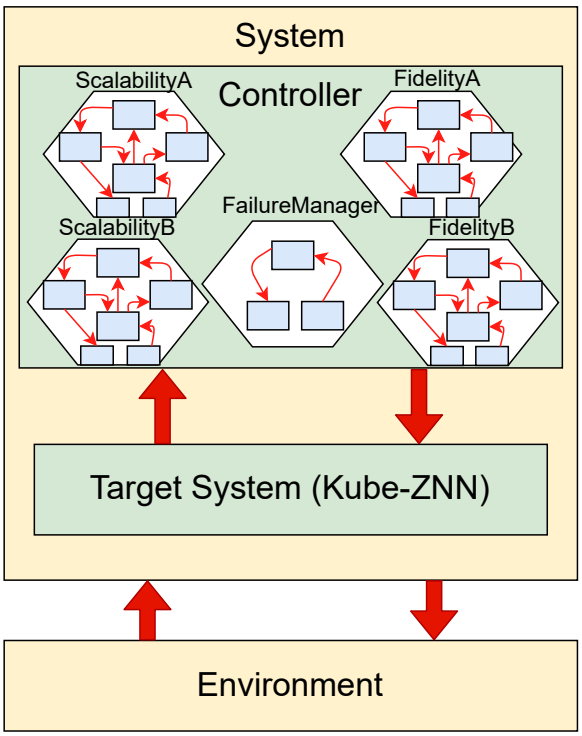

Figura 6.1: Configuração Des-KZ.

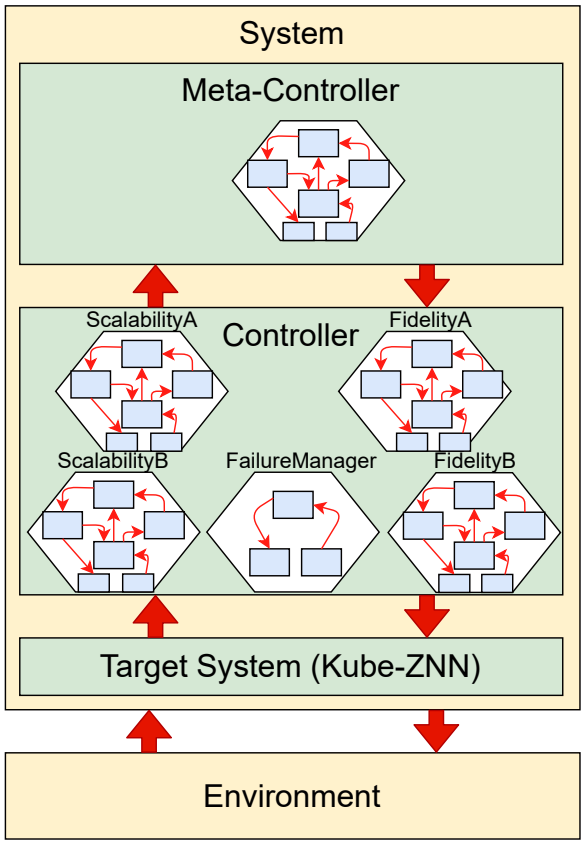

Figura 6.2: Configuração Meta-KZ.

6.3.2 Infraestrutura para a Execução do Experimento

A infraestrutura de software e hardware empregada foi a mesma para a demonstração, avaliação e comparação das três configurações arquiteturais. Os experimentos foram conduzidos sobre uma única máquina física equipada com dois processadores AMD Ryzen 5 de 3.6 GHz, 16GB de memória RAM e 250GB de armazenamento em SSD. O sistema operacional MS Windows 10 foi utilizado e apoiado pelas ferramentas: Hyper-V[1]. para criar uma nova instância local do cluster Kubernetes; Minikube[2] versão v1.23.1 para configurar localmente um único nó do Kubernetes cluster; e Kubernetes[3] CLI versão v1.22.2 com ferramentas de linha de comando para manipular os clusters Kubernetes. Para simular um ambiente em nuvem típico e também para dinamicamente mudar a alocação de recursos, cada aplicação foi implantada com todas as respectivas camadas,

[1] `https://learn.microsoft.com/pt-br/virtualization/hyper-v-on-windows/about/` – acessado em 05 de outubro de 2022

[2] `https://minikube.sigs.k8s.io/docs/start/` – acessado em 05 de outubro de 2022

[3] `https://kubernetes.io/pt-br/` – acessado em 05 de outubro de 2022

dentro da própria máquina virtual, por exemplo, utilizando um Minikube como um cluster Kubernetes.

6.3.3 Procedimentos a Execução do Experimento

Para conduzir os experimentos, os seguintes passos foram seguidos:

1. Configuração da infraestrutura computacional, incluindo o computador, máquinas virtuais, Kubernentes CLI e Minikube;

2. Implantação de ferramentas de monitoramento, envolvendo as ferramentas para apoiar o armazenamento de dados e o monitoramento do cluster Kubernetes (isto é, metrics-server, prometheus e kube-state-metric);

3. Implantação de objetos do sistema alvo, envolvendo a implantação de objetos responsáveis para gerenciar o ponto de entrada do sistema alvo (isto é, kube-znn-svc e ngnix) em tempo de execução. Além disso, armazenar a mídia atual que é acessada pelos usuários (isto é, db-svc);

4. Seleção da estrutura do controlador, isto é, selecionar qual controlador é utilizado no experimento (Mon-KZ, Des-KZ ou Meta-KZ);

5. Implantação do simulador de sobrecarga de trabalho, envolvendo a implementação de ferramentas responsáveis por simular a carga de requisições de usuários (isto é, a ferramenta k6); e

6. Coleta de dados de execução, envolvendo a implantação de ferramentas desenvolvidas sob-medida (por exemplo, Python scripts and ShellScrips) responsáveis pela coleta de dados do cluster Kubernetes.

6.4 Avaliação

Avaliou-se a hipótese de que um controlador que é baseado em microcontroladores tem vantagens de ser estruturalmente flexível sem comprometer o desempenho do sistema alvo. O desempenho do sistema alvo é considerado maior quando o sistema responde a uma determinada demanda ao utilizar um menor número de servidores, ou ao distribuir uma qualidade de mídia mais alta do que ele faria em diferentes configurações de execução. A avaliação compreende duas questões de pesquisa:

- RQ1: Qual é o impacto causado por um controlador descentralizado com relação a adaptações arquiteturais do sistema gerenciado quando comparado com um controlador monolítico?
- RQ2: Qual o impacto causado por um controlador descentralizado com relação a adaptações paramétricas do sistema gerenciado quando comparado com um controlador monolítico?

Para responder essas questões, realizaram-se experimentos com três configurações de sistemas, isto é, Mon-KZ, Des-KZ, e Meta-KZ. A avaliação do estado do sistema alvo, em termos de escalabilidade e de fidelidade em momentos particulares, provê evidências que foram coletadas por meio das seguintes métricas:

M1: O estado do meio com respeito ao número de servidores ativos.

M2: O estado final com respeito ao número de servidores ativos.

M3: O estado do meio com respeito à fidelidade.

M4: O estado final com respeito à fidelidade.

Cenários de execução experimental

Para avaliar a hipótese definida na seção anterior, na sequência apresenta-se a informação com relação aos cenários de execução experimental.

1: Definiram-se os seguintes cenários:

- 4 e 10 servidores: estes valores representam o número máximo de instâncias de servidores para o sistema alvo (isto é, Kube-ZNN).
- Períodos curto (*short*) e longo (*long*) de carga/descarga de trabalho: estes períodos consistem em tempos pré-definidos durante uma carga/descarga de trabalho, em termos de requisições de usuário aos servidores Kube-ZNN, que são realizadas. Assim, pode-se simular como o controlador e o sistema alvo estão funcionando. Para execuções curtas, definiram-se 2 minutos de carga/descarga de trabalho, ao passo que para execuções longas definiram-se 5 minutos.

2: Com relação às métricas M1, M2, M3 e M4, para ambos os estados do meio e final, os dados selecionados para serem analisados foram: (a) o número de servidores ativos; e (b) o nível de fidelidade.

3: Para cada cenário (isto é, máximo de 4 ou 10 servidores, tempo de execução curto ou longo) em cada configuração em particular, realizaram-se 10 execuções.

4: Para cada conjunto de 10 execuções (40 execuções de cada configuração; 120 execuções no total), computaram-se as métricas M1 a M4. As distribuições dos valores referentes às 10 execuções por configuração são representadas por meio de um conjunto de gráficos *boxplot*. Como um exemplo, conforme mostrado na Figura 6.5, para as 10 execuções da configuração Mon-KZ, considerando um máximo possível de 10 servidores ativos e com tempo curto de carga/descarga, pode-se observar que a maior concentração de resultados foi entre o intervalo de 6 a 9 servidores ativos no momento final de descarga, sendo que o menor número foi 4 servidores e o maior resultado foi 9 servidores. A mesma análise pode ser realizada em todos os gráficos (Figuras 6.3 a 6.6). A representação utilizando *boxplot* auxilia na verificação da variação dos resultados nas 10 execuções, bem como a mediana desses e os valores fora do padrão. Assim, possibilitou-se a caracterização da performance de uma dada configuração com respeito às propriedades observadas.

6.4.1 Análise dos Resultados Referentes ao Atributo Escalabilidade

Sumarizam-se as execuções com relação à escalabilidade, para as três configurações, nas figuras 6.3, 6.4, 6.5, e 6.6. Na sequência são descritos os resultados sobre: (1) o pico da demanda de sistema (isto é, no fim do período de carga de trabalho) e (2) o fim do período de descarga de trabalho. Comparam-se também os resultados das três configurações.

Escalabilidade durante o pico de demanda do sistema: Ao considerar os estados do meio, para as execuções envolvendo até 4 servidores (Figuras 6.3 e 6.4), todas as configurações alcançaram o limite de servidores disponíveis. Para execuções envolvendo até 10 servidores (Figuras 6.5 e 6.6), as configurações Mon-KZ and Des-KZ novamente alcançaram o limite, enquanto que a configuração Meta-KZ demandou 8 e 7 servidores para execuções curtas e longas, respectivamente.

Cada configuração implementa o FailureManager ao seu modo, seguindo a respectiva configuração – por exemplo, Mon-KZ contém uma versão monolítica do FailureManager, ao passo que a Meta-KZ contém uma versão decentralizada com controle do metacontrolador. Entretanto, mesmo todas configurações tendo tal FailureManager, a abordagem utilizada em cada um resulta em diferenças no resultados, como observado nas figuras 6.3, 6.4, 6.5, e 6.6.

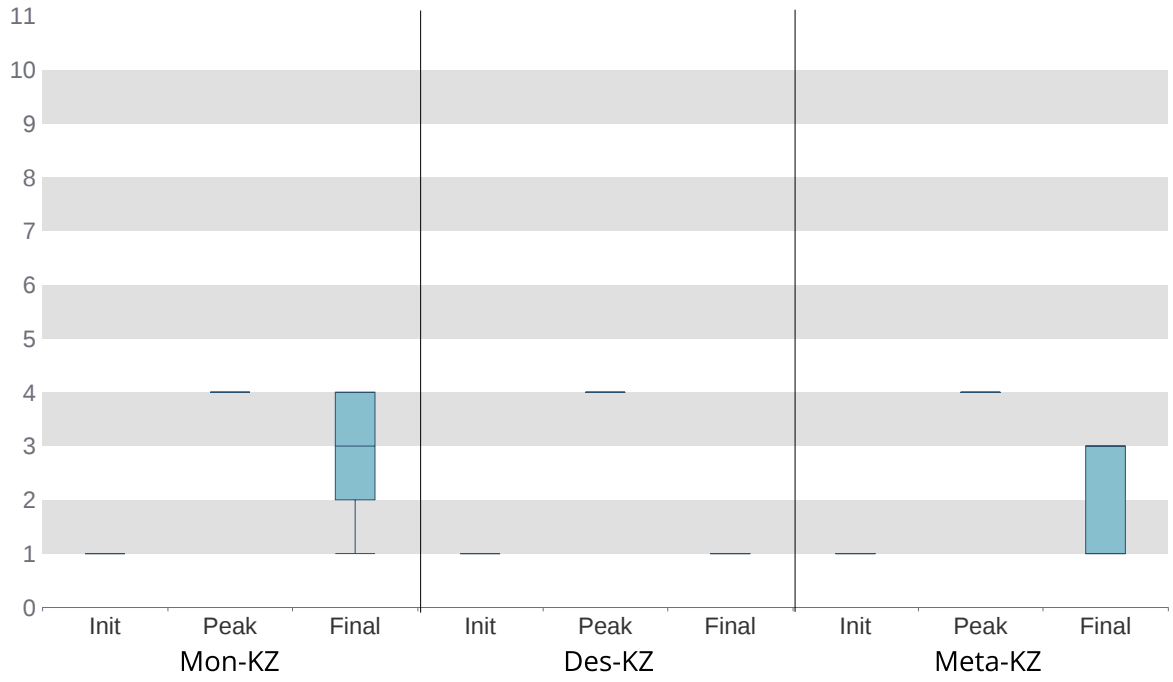

Figura 6.3: Escalabilidade (com até 4 servidores ativos, tempo curto de carga e descarga).

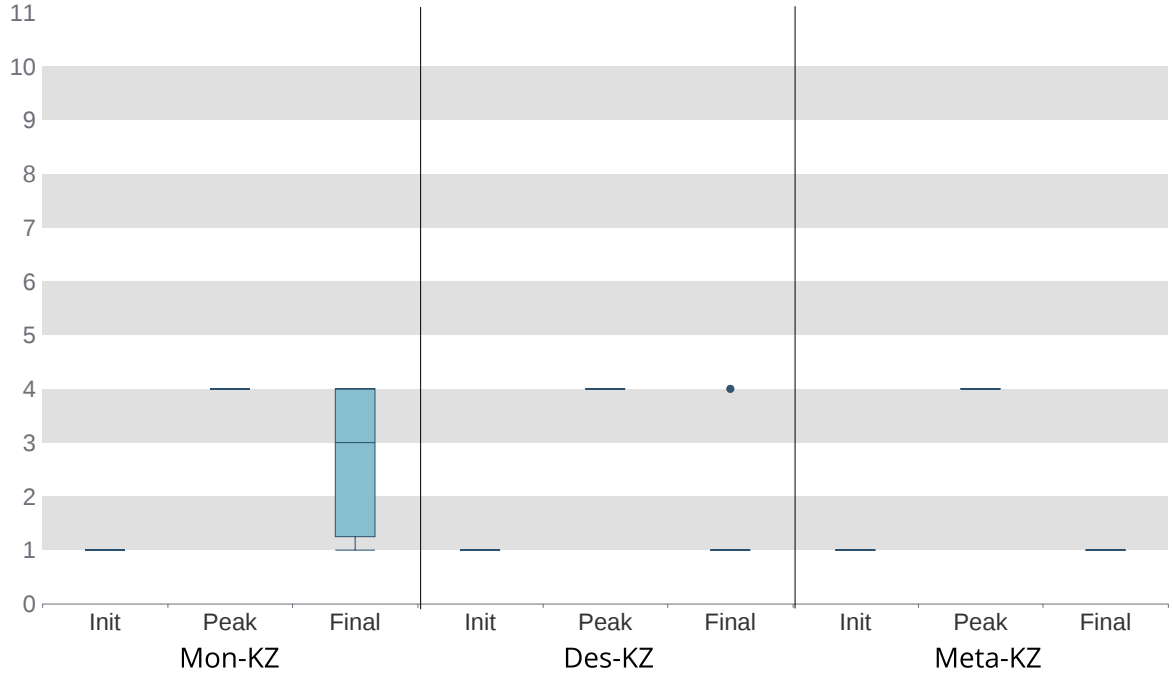

Figura 6.4: Escalabilidade (com até 4 servidores ativos, tempo longo de carga e descarga).

Escalabilidade após o período de descarga de trabalho do sistema: Ao considerar os estados finais, independentemente do tempo de execução, nota-se algumas diferenças nos resultados para execuções envolvendo até 4 servidores (Figuras 6.3 e 6.4).

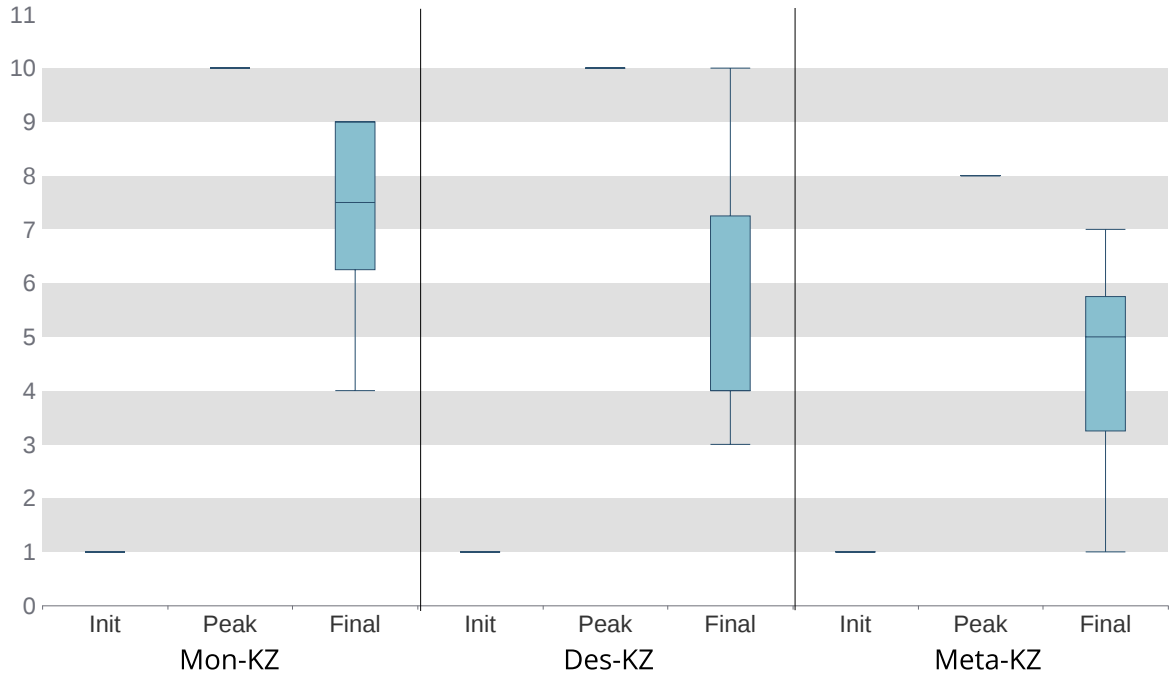

Figura 6.5: Escalabilidade (com até 10 servidores ativos, tempo curto de carga e descarga).

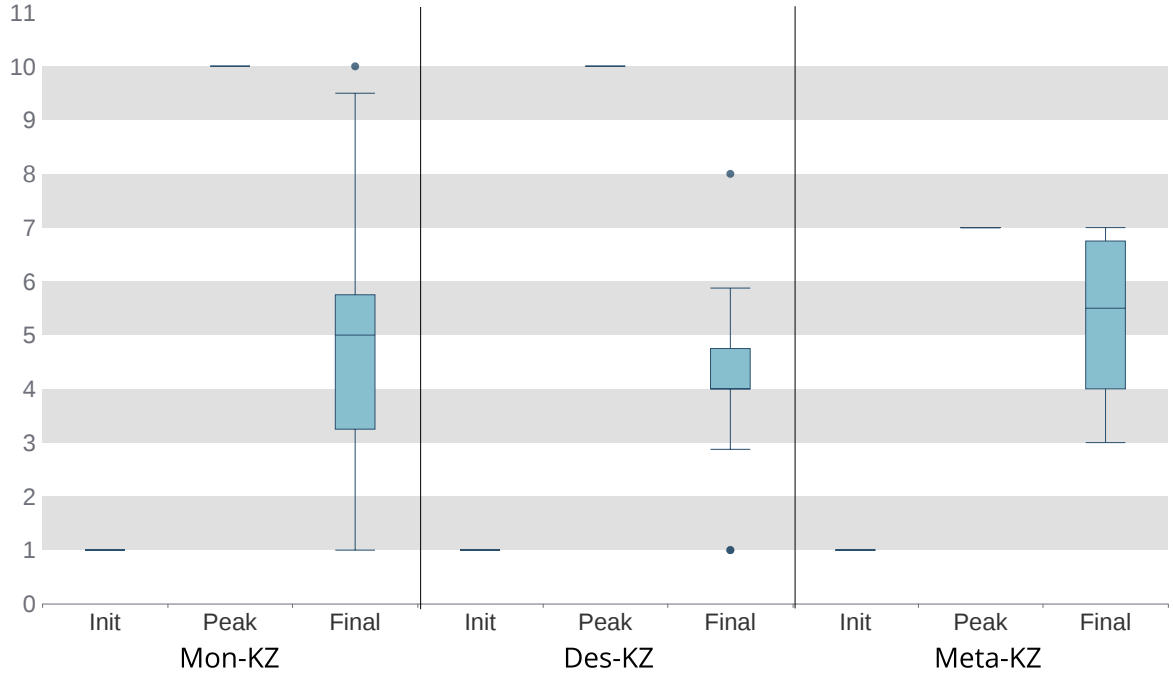

Figura 6.6: Escalabilidade (com até 10 servidores ativos, tempo longo de carga e descarga).

O gráfico boxplot 6.3 ilustra a comparação de resultados de execuções das abordagens Mon-KZ, Des-KZ e Meta-KZ para 4 servidores. Observa-se que para Mon-KZ o estado Final variou entre 1 e 4, concentrando-se entre 2 e 4 servidores do Kube-ZNN. Para a

Des-KZ não houve variação, na qual todos os estados finais terminaram em 1 servidor do Kube-ZNN. Já para a Meta-KZ a variação ocorreu entre 1 e 3 servidores do Kube-ZNN.

No gráfico boxplot 6.4 ilustra-se que houve variação somente na Mon-KZ variando entre 1 e 4 servidores do Kube-ZNN. Tanto a Des-KZ quanto a Meta-KZ não houve variação, finalizando todos em 1 servidor do Kube-ZNN – uma única exceção ocorre para Des-KZ ao qual uma das execuções o estado final resulta em 4 servidores do Kube-ZNN.

O gráfico boxplot 6.5 ilustra uma maior variação no estado Final entre os resultados para todas configurações. A Mon-KZ varia entre 4 e 9 servidores, concentrando entre 6 e 9 servidores do Kube-ZNN. Na Des-KZ o estado Final varia entre 3 e 10 servidores, concentrando entre 4 e 7 servidores do Kube-ZNN. Finalmente, na Meta-KZ estado Final varia entre 1 e 7 servidores, concentrando entre 3 e 6 servidores Kube-ZNN.

No gráfico boxplot 6.6, a Mon-KZ tem uma variação entre 1 e 10 servidores, concentrando entre 3 e 6 servidores do Kube-ZNN. A Des-KZ tem uma variação entre 3 e 6, concentrando entre 4 e 5 servidores do Kube-ZNN. Finalmente, Meta-KZ tem uma variação entre 3 e 7 servidores, concentrando entre 4 e 7 servidores do Kube-ZNN.

Em particular, nas execuções que envolveram as configurações Des-KZ e Meta-KZ, obteve-se uma redução para apenas um único servidor ativo. Entretanto, nas execuções que envolveram a configuração Mon-KZ, a redução foi de 4 para 3 servidores ativos. Diferenças foram também notadas para execuções envolvendo até 10 servidores (Figuras 6.5 e 6.6). Nestes casos, em execuções curtas e longas, respectivamente, a configuração Mon-KZ teve o respectivo estado final com 7 e 5 servidores. Tais números para a configuração Des-KZ são ambos 4 para execuções curtas e longas, ao passo que para a configuração Meta-KZ o número final de servidores são 5 para ambas.

Comparação: Destaca-se que nos gráficos 6.5 e 6.6, devido ao lidar com 10 servidores máximos do Kube-ZNN, observa-se uma maior variação no estado Final das execuções. E a partir da comparação entre as três configurações, nota-se que a configuração Meta-KZ é a que apresenta menor variação, além de ter menores números no estado *Peak*.

Com relação ao atributo escalabilidade, ao considerar execuções envolvendo até 4 servidores, as configurações Des-KZ e Meta-KZ desempenharam ligeiramente melhor do que Mon-KZ ao observar os estados finais. Por outro lado, resultados envolvendo execuções com 10 servidores mostraram que a configuração Meta-KZ teve um melhor desempenho do que as configurações Mon-KZ e Des-KZ. Particularmente no pico de demanda (isto é, ao observar estados do meio), como ilustrado nas figuras 6.3, 6.4, 6.5, e 6.6. A configuração Meta-KZ demandou 8 e 7 servidores, ao considerar execuções curtas e longas, respectivamente. Já as configurações Mon-KZ e Des-KZ demandaram o número máximo (isto é, 10)

de servidores em ambos os casos. Ao considerar o estado final, os resultados variam nas configurações e nenhum se destacou para execuções envolvendo 10 servidores.

Pode-se também observar que as configurações Mon-KZ e Des-KZ têm maior variação nos resultados finais, como observados nos gráficos mencionados acima. Nota-se que, no caso da Mon-KZ, para *10 servidores - curta* existe variação entre 4 e 9 servidores; e para *10 servidores - longa* a variação ocorre entre 1 e 10. Tais variações ligeiramente diminuem para a configuração Des-KZ e diminuem consideravelmente para a configuração Meta-KZ como ilustram-se nos gráficos.

Com relação à RQ1 (*Qual é o impacto causado por um controlador descentralizado com relação a adaptações arquiteturais do sistema gerenciado quando comparado com um controlador monolítico?*), em termos de adaptações estruturais do sistema alvo, resultados experimentais mostram que Meta-KZ desempenhou melhor do que as outras configurações.

6.4.2 Análise dos Resultados Referentes ao Atributo Fidelidade

As figuras 6.7, 6.8, 6.9, e 6.10 sumarizam as execuções das configurações Mon-KZ, Des-KZ, e Meta-KZ com relação ao atributo fidelidade. Similarmente a discussão apresentada para escalabilidade, na sequência descreve-se os resultados sobre fidelidade com respeito a: (1) o pico de demanda do sistema (isto é, ao fim do período de carga de trabalho), e (2) ao fim do período de descarga de trabalho. Finalmente, também comparam-se os resultados para as três configurações.

Fidelidade no pico de demanda de sistema: Ao considerar os estados do meio, independemente do tempo de execução e independemente do número máximo de servidores permitidos, as três configurações diminuíram o nível de fidelidade para o mínimo (isto é, 400 KB).

Fidelidade após o período de descarga de trabalho: Ao considerar os estados finais, em todas execuções as configurações Mon-KZ e Meta-KZ tiveram o mesmo comportamento: ambas aumentaram o nível de fidelidade para o máximo (isto é, 800 KB). Com relação a configuração Des-KZ, em execuções envolvendo até 4 servidores, o nível de fidelidade final também foi 800 KB, ao passo que em execuções envolvendo 10 servidores o estado final foi de 600 KB no nível de fidelidade.

Comparação: Com respeito ao atributo fidelidade, o comportamento das três configurações foi similar. A partir dos resultados ilustrados nas figuras 6.7, 6.8, 6.9, e 6.10, somente a

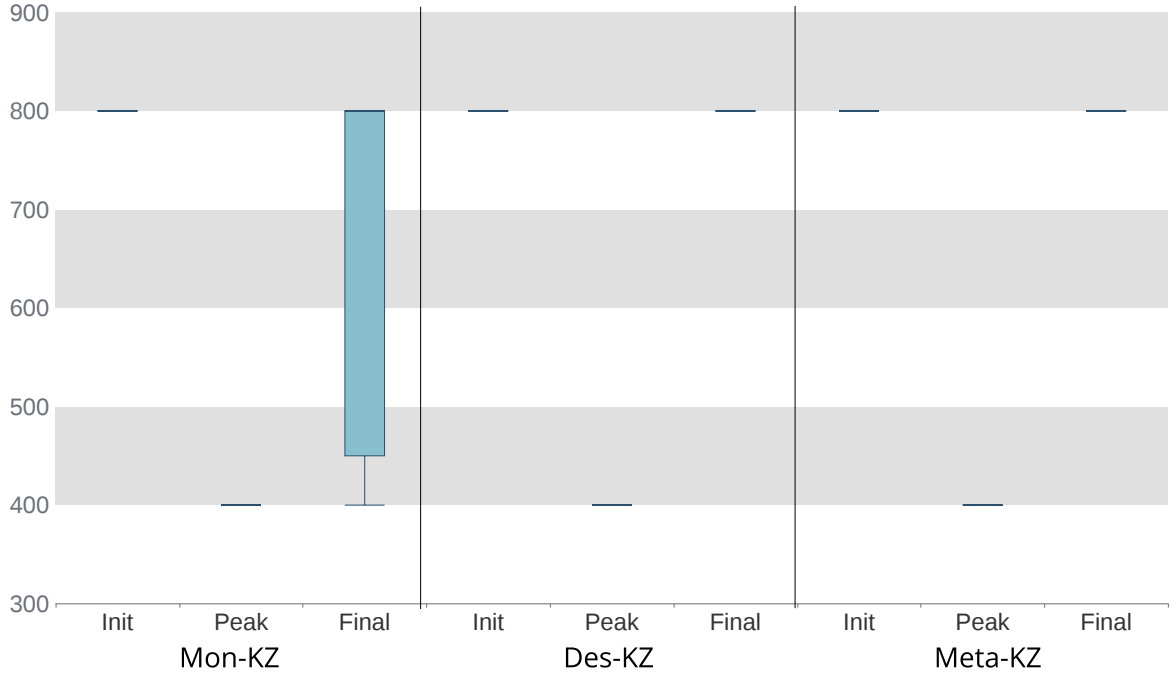

Figura 6.7: Fidelidade (com até 4 servidores ativos, tempo curto de carga e descarga).

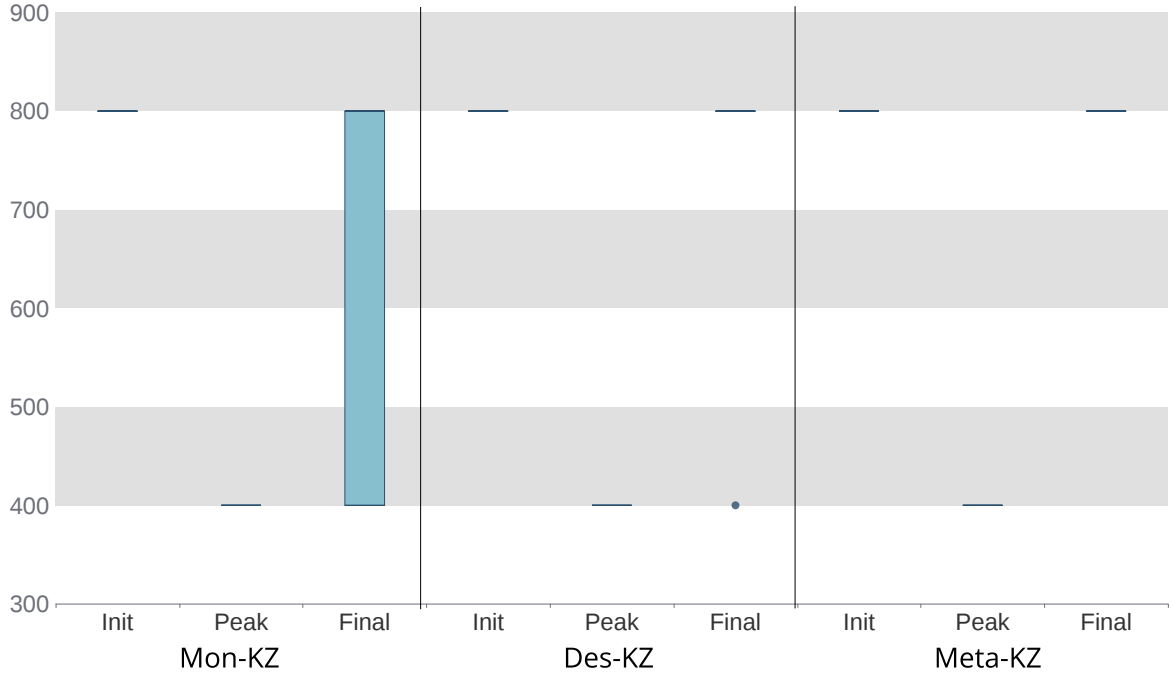

Figura 6.8: Fidelidade (com até 4 servidores ativos, tempo longo de carga e descarga).

configuração Des-KZ não foi restaurada para o nível máximo de fidelidade após o período de descarga de trabalho.

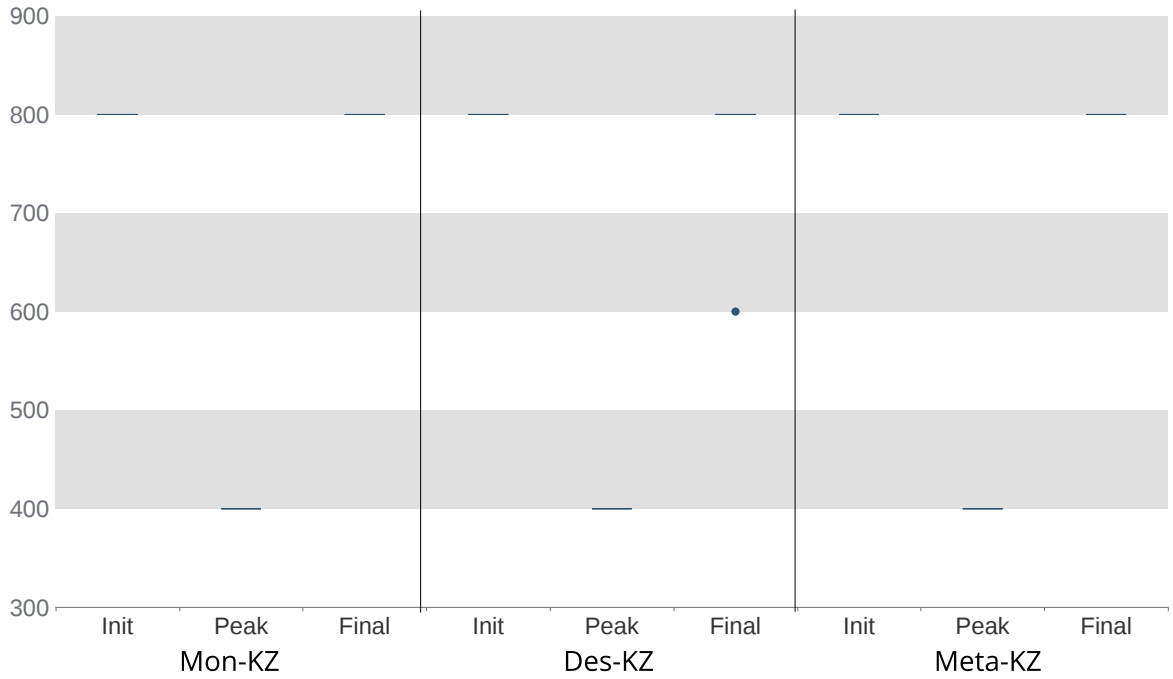

Figura 6.9: Fidelidade (com até 10 servidores ativos, tempo curto de carga e descarga).

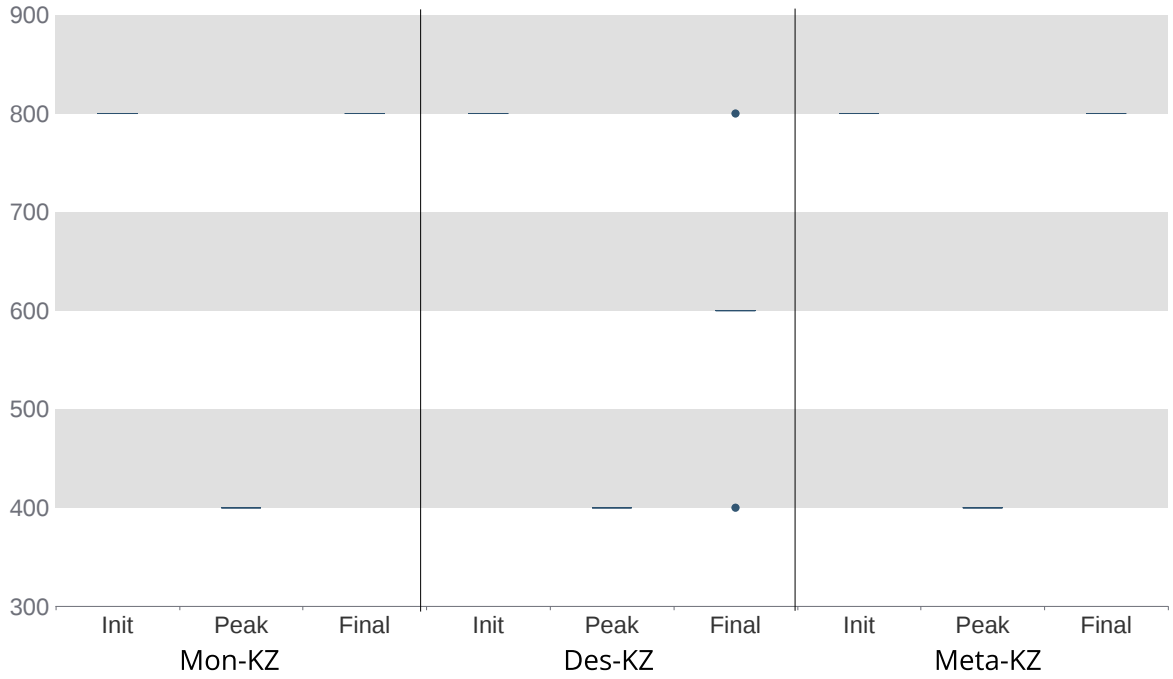

Figura 6.10: Fidelidade (com até 10 servidores ativos, tempo curto de carga e descarga).

Com relação à RQ2 (*Qual o impacto causado por um controlador descentralizado com relação a adaptações paramétricas do sistema gerenciado quando comparado com um controlador monolítico?*), em termos de adaptações paramétricas do sistema alvo, os resultados dos experimentos não apontam para benefícios ou impactos adversos para qualquer configuração.

6.4.3 Discussões Adicionais

Alguns itens de discussão adicionais são apresentados nesta seção, envolvendo a restauração do sistema alvo ao seu estado original, conflitos enfrentados durante a reconfiguração do controlador, e uso de recursos pelas configurações arquiteturais experimentadas.

Restauração: Ao comparar as três configurações, configurar um alto número de servidores do Kube-ZNN pode impactar negativamente no tempo de restauração para as configurações que precisam lidar com números maiores de componentes (por exemplo, Des-KZ e Meta-KZ). No contexto deste trabalho, restauração sobre escalabilidade significa retornar o sistema alvo para um único servidor ativo, ao passo que a restauração sobre fidelidade significa aumentar a qualidade da mídia distribuída para os clientes para o máximo (isto é, 800 KB). Experimentos para abordar o tempo restauração serão conduzidos em trabalho futuro.

Conflitos: Da perspectiva de conflitos e sincronização de estados do sistema alvo, a configuração Mon-KZ contem um único conjunto de estratégias de adaptação. Como tal, o controlador *Kubow* foi responsável por aplicar as estratégias, sem a necessidade de resolver desafios relacionados a conflitos das estratégias, nem mesmo questões relacionadas à sincronização do estado do sistema alvo. Para a configuração Des-KZ, por outro lado, conflitos podem surgir devido a características inerentes dos controladores descentralizados. Como um exemplo, se a taxa de falha muda durante a ativação das variações ScalabilityA ou ScalabilityB, ambas podem adaptar o sistema alvo simultaneamente, assim produzindo um estado inconsistente com relação ao número de servidores Kube-ZNN ativos. No configuração Meta-KZ, em contraste, tais problemas são evitados, dado que o metacontrolador é responsável por manter ativo somente uma variação de um particular microcontrolador.

Uso de recursos: Mesmo com a configuração Meta-KZ tendo demandado mais recursos (pelo menos 3450m de CPU e 3164mi de memória), quando comparada com a configuração Mon-KZ (pelo menos 1250m de CPU e 1100mi de memória), como sumarizado na Tabela 6.2,

os resultados para Meta-KZ não foram comprometidos (conforme pode ser observado nas figuras 6.3 a 6.10).

6.5 Ameças à Validade

Nesta seção são discutidas as ameaças a validade do experimento de acordo com as categorias listadas por Wohlin et al. (2012). O experimento envolveu uma *variável independente* (ou seja, a abordagem do projeto do controlador) em que três *tratamentos* foram aplicados (ou seja, monolítico, descentralizado e descentralizado com um metacontrolador). As *variáveis dependentes* são as variações em termos de escalabilidade e fidelidade observadas no sistema alvo quando os diferentes tratamentos são aplicados. O sistema Kube-ZNN foi o único sistema utilizados nos experimentos.

Validade interna: Neste estudo, o software sob análise apresenta alguns fatores de incerteza. Tais fatores podem levar a variações nos resultados observados para determinados tratamentos. Em particular, a incerteza está presente no controlador (incerteza de processamento relacionada ao monitoramento periódico dos componentes do controlador e do sistema alvo); na comunicação entre componentes (com relação a inconsistências entre os estados observados do sistema alvo para permitir decisões a serem realizadas pelo controlador); e no sistema alvo e ambiente (variações nos recursos disponíveis para executar o sistema alvo dentro de um Cluster Kubernetes implantado em uma única máquina física). Para mitigar esta ameaça, realizaram-se 10 execuções de cada cenário, analisando os valores médios dos dados quantitativos coletados.

Validade de construção: Neste estudo, uma ameaça poderia ser a falta de conhecimento prévio para projetar e lidar com as tecnologias adotadas, que foram Rainbow / *Kubow* e ZNN.com / Kube-ZNN. Para mitigar isso, estabeleceu-se comunicação com os desenvolvedores originais do *Kubow* e Kube-ZNN (Aderaldo et al., 2019), os quais forneceram ajuda e correções de *bugs* da versão original (tais correções foram propagadas para todas as configurações utilizadas neste trabalho). Também foram utilizas documentações disponíveis sobre as tecnologias empregadas neste trabalho.

Uma outra ameaça de construção refere-se a interação de diferentes tratamentos. Neste caso, houve uma evolução natural das configurações arquiteturais utilizadas nos experimentos. Iniciou-se com a versão original das implementações *Kubow* e Kube-ZNN (Aderaldo et al., 2019) e evoluiu-se para criar as outras duas configurações compostas por microcontroladores e metacontrolador. Para mitigar essa ameaça, aplicaram-se boas práticas de projeto,

realizando revisões de código e avaliação das implementações com os desenvolvedores do *Kubow* e Kube-ZNN.

Validade externa: Neste estudo, utilizou-se um único sistema alvo (Kube-ZNN) e um conjunto reduzido de microcontroladores. Assim, os resultados podem não ser generalizáveis para outras configurações não utilizadas neste trabalho, por exemplo, utilizando diferentes sistemas alvo, conjunto de microcontroladores, ou mesmo infraestruturas de execução.

6.6 Conclusões sobre o Experimento e Considerações Finais

Neste estudo, demostrou-se uma abordagem baseada em microsserviços para o projeto e desenvolvimento de controladores de multicamadas para sistemas adaptativos. A viabilidade da abordagem, que promove flexibilidade e reúso, foi demonstrada e avaliada utilizando-se três diferentes configurações arquiteturais (monolítica, descentralizada e descentralizada com um metacontrolador) para o controlador, que foi implantada no estudo de caso Kube-ZNN.

A evidência coletada para a avaliação foi relacionada a dois atributos do Kube-ZNN: flexibilidade estrutural em termos do número de servidores (isto é, escalabilidade de recursos), e adaptação paramétrica relacionada ao conteúdo provido pelos servidores (isto é, fidelidade de conteúdo).

No experimento, identificou-se que mesmo que a configuração descentralizada com um metacontrolador tenha demandado mais recursos computacionais, ela também teve melhor desempenho com relação as outras configurações com respeito ao atributo escalabilidade Além disso, com relação à fidelidade de conteúdo, a configuração arquitetural com um metacontrolador teve resultados similares quando comparada com as outras configurações.

Finalmente, conclui-se que o projeto de controladores baseados em microcontroladores permite a definição de controladores que são flexíveis estruturalmente, Isso sem comprometer o desempenho do sistema como um todo.

Capítulo

7

Microcontrolador de Teste de Regressão

7.1 Considerações Iniciais

Neste capítulo, a fim de demonstrar a flexibilidade da abordagem de projetar controladores com base em conjuntos de microcontroladores, apresenta-se uma proposta de definição e implementação de um microcontrolador de teste de regressão para sistemas adaptativos (SAs). Para embasar o trabalho aqui apresentado, conduziu-se uma Revisão Sistemática da Literatura (RSL) (Siqueira et al., 2021), na qual caracterizou-se o estado da arte em teste de SAs. Dentre o conjunto de mais de 100 estudos analisados na RSL, identificaram-se alguns que lidaram com teste em tempo de execução e teste de regressão de SAs. de modo que ambas as linhas de trabalhos englobam as principais abordagens que embasaram a proposta do microcontrolador de teste de regressão aqui apresentada.

Na sequência deste capítulo, na Seção 7.2 apresenta-se uma visão geral dos trabalhos relacionados a teste em *tempo de execução* de SAs. Na Seção 7.3 sumarizam-se trabalhos que exploram o teste de regressão de SAs. Na Seção 7.4 traz-se uma breve análise crítica sobre os tópicos abordados (isto é, teste em *tempo de execução* e teste de regressão). Por fim, nas seções 7.5 a 7.7 apresentam-se especificações e implementações do microcontrolador RegressionTest utilizando-se o *framework Kubow*.

7.2 Teste em Tempo de Execução em Sistemas Adaptativos

Na literatura analisada na revisão sistemática conduzida no escopo deste trabalho (Siqueira et al., 2021), existem diversas abordagens que foram inspiradas no teste em *tempo de execução* de SAs. Um destaque consiste nas abordagens no contexto de *captura de dados*, que em geral utilizam dados a partir da interação entre componentes e o ambiente em *tempo de execução*. Outro destaque se refere às abordagens de *testes embutidos*, que em geral armazenam conjuntos de teste juntamente com os componentes do sistema. Esses dois destaques são utilizados de diferentes modos e contribuições por diferentes autores, resumindo-se como se segue:

- Alguns estudos exploraram como a composição dinâmica de componentes pode auxiliar na definição de teste embutidos (Eberhardinger et al., 2018; Fredericks, 2018; Fredericks e Cheng, 2015; King et al., 2011a,b, 2007; Merdes et al., 2006; Niebuhr e Rausch, 2007; Niebuhr et al., 2009; Reichstaller e Knapp, 2018).

- Outros autores propuseram a captura e gerenciamento de um histórico de falhas que acontecem em *tempo de execução*, as quais são utilizadas para prever futuro acontecimento de falhas no SA em *tempo de execução* (Fredericks, 2018; Garvin et al., 2013; Horányi et al., 2013; Luo et al., 2017; Qin et al., 2016; Xu et al., 2012).

- Algumas abordagens propuseram a geração de casos de teste com base na captura de dados realizada a priori (Akour et al., 2014; Chan et al., 2006; Fredericks et al., 2014; Jaw et al., 2008; Lindvall et al., 2017; Reichstaller et al., 2018; Vassev et al., 2010; Yu et al., 2016).

- Da Silva e De Lemos (2011) propuseram uma abordagem para gerar planos de teste no nível de integração durante o *tempo de execução*. Conforme os testes são executados em *tempo de execução*, novos planos são gerados se necessário. Os mesmos autores (Da Silva e De Lemos, 2013) propõem a geração e gerenciamento de processos de teste, também no nível de integração.

- Hansel et al. (2015) investigaram como o teste pode ser realizado ao utilizar um modelo de referência MAPE-K. A abordagem é baseada em estados arquiteturais que representam iterações que os estágios do MAPE-K realizam. Tais estágios são análogos a estados de uma máquina de estados finita, de modo que uma iteração pode ser associada a uma transição entre estados.

- Ma et al. (2018) propuseram realizar o teste de um sistema juntamente com um teste de um modelo executável que representa o sistema. A abordagem é baseada em um ambiente controlado no qual o usuário realiza inclusões de mudanças. Além disso, utiliza-se o modelo UML para auxiliar na captura de comportamentos esperados do sistema alvo, gerando-se, assim, dados de teste para o mesmo.

SAs caracterizam-se por poderem incluir e excluir componentes do sistema durante o *tempo de execução* (Niebuhr e Rausch, 2007). Este tipo de característica demanda que a atividade de teste seja realizada durante a execução do sistema (isto é, em *tempo de execução*). Relacionado a isso, o principal desafio de teste está em definir, durante a etapa de projeto (isto é, em *design time*), testes que somente serão executados durante alguma adaptação em *tempo de execução*, incluindo ou excluindo componentes do sistema. Mais precisamente, alguns autores destacam:

- A necessidade de se manipular o espaço de teste. Em outras palavras, existe um limite na execução (por exemplo, possíveis caminhos ou configurações) do sistema que possa ser investigado/previsto na etapa de projeto?

- A dificuldade de detectar e evitar configurações que são incorretas. Ou seja, mesmo que o teste seja realizado no *tempo de execução*, como detectar e evitar quando algum teste falhar?

Portanto, assim como abordado no trabalho anterior (Siqueira et al., 2021), ao lidar com o teste em *tempo de execução* para SAs, estarão presentes desafios como 1) *Como detectar e evitar configurações incorretas definidas em tempo de execução*; o 2) *Como testar um sistema que executa em um ambiente distribuído e heterogêneo*; e o 3) *Como antecipar todas mudanças relevantes e quando podem impactar no comportamento do sistema.* Além disso, com relação a abordagens de teste em *tempo de execução* para SAs, têm-se proposto abordagens relacionadas principalmente à captura de dados e a teste embutidos.

Na sequência apresentam-se as abordagens que propõem contribuições para o teste de regressão em *tempo de execução* para SAs.

7.3 Teste de Regressão em Sistemas Adaptativos

Com relação ao teste de regressão no contexto de SAs, o conjunto de estudos selecionados na revisão sistemática (Siqueira et al., 2021) inclui principalmente iniciativas para a minimização (Akour et al., 2014) e a seleção (Al-Refai et al., 2016a; Lahami et al., 2015)

de casos de teste, as quais são descritas a seguir. Ressalta-se que não foram encontrados estudos que abordaram a priorização de casos de teste.

7.3.1 Abordagem de Akour et al.

Akour et al. (2014) propuseram uma abordagem baseada em modelos que sincroniza dados de teste em *tempo de execução*, após adaptações dinâmicas serem realizadas durante a execução do SA. Os autores utilizam a abordagem baseada em modelos para realizar rastreabilidade entre os artefatos de casos de teste e de componentes do SA. Tal rastreabilidade auxilia em inclusões e exclusões nos conjuntos de teste após adaptações ocorrerem em *tempo de execução*. As adaptações consistem na modificação de estrutura e de comportamento do sistema, em particular inclusão e remoção de componentes, sendo que essas operações são definidas e investigadas por eles, respectivamente, como mudanças aditivas e redutivas. A abordagem engloba um modelo de componentes e um modelo de testes, utilizando-se de tais modelos para realizar uma sincronização em *tempo de execução* entre os mesmos.

Na abordagem de teste de regressão proposta por Akour et al., mudanças aditivas incluem novas interfaces de componentes e novas implementações em *tempo de execução*, as quais levam à geração automática de novos casos de teste funcionais e estruturais. Nota-se que testes estruturais são gerados apenas quando há disponibilidade da implementação do componente que está sendo inserido na arquitetura do sistema.

Mudanças redutivas, por sua vez, removem interfaces de componentes e respectivas implementações em *tempo de execução*. Quando uma remoção de componente ocorre, casos de teste em nível de unidade que estão associados ao componente alvo de remoção podem ser removidos a partir do modelo de teste. Com relação a testes de integração, testes de componentes chamadores do componente removido devem ser atualizados após a adaptação ocorrer, caso contrário farão chamadas a um componente que deixará de existir no sistema dada a nova arquitetura que estará em execução.

7.3.2 Abordagem de Lahami et al.

Lahami et al. (2015) também utilizaram uma abordagem baseada em modelos para identificar testes reutilizáveis e novos casos de teste a partir de um conjunto de testes inicial. Propôs-se uma abordagem dividida em três fases, sendo elas: (i) diferenciação que detecta similaridades e diferenças entre um modelo de comportamento inicial e evoluído; (ii) classificação de teste que identifica testes que podem ser reutilizáveis; e, (iii) geração de casos de testes para cobrir novos comportamentos e adaptar testes anteriores.

Para cada uma das fases supracitadas (i, ii e iii) os autores criaram um módulo ferramental que que lida com os artefatos envolvidos na abordagem.

O módulo que lida com a fase (ii) tem como responsabilidades, além de eliminar os casos de testes desnecessários (obsoletos), a identificação de conjuntos de testes reutilizáveis e retestáveis, e o encaminhamento de novos casos de teste (baseados em novos caminhos alcançáveis) a serem gerados pelo módulo de geração. O módulo referente à fase (iii) recebe um modelo que especifica as diferenças entre o sistema alvo antes e depois da adaptação, assim como os caminhos alcançáveis. Com base nessas informações, o módulo analisa os casos de teste antigos – a fim de identificar adaptações necessárias aos mesmos – e novos casos de teste são criados para cobertura de possíveis caminhos alcançáveis.

Ressalta-se que a abordagem de Lahami et al. (2015) é uma abordagem de teste de regressão utilizando-se técnicas de seleção. Apesar de não se enfatizar o uso de técnicas de minimização, a minimização também é utilizada durante a remoção de casos de teste obsoletos. Além disso, ela é baseada em modelos de máquinas de estados finitas que são utilizados na classificação de casos de teste.

7.3.3 Abordagem de Al-Refai et al.

Al-Refai et al. (2016b) propuseram uma abordagem na qual modelos são gerados a partir de um conjunto inicial de testes. Na abordagem, realiza-se uma rastreabilidade por meio dos modelos gerados e de adaptações sendo realizadas em *tempo de execução* no SA. Tal rastreabilidade apoia a geração de casos de testes que podem testar as mudanças realizadas no SA. A abordagem é apoiada por modelos de classes e de atividades para representar comportamento e casos de teste. Além disso, uma ferramenta de comparação de modelos é utilizada para identificar mudanças durante o processo de adaptação.

Com relação a abordagem proposta pelos autores, a mesma inclui três fases que contam com respectivo apoio computacional. A fase (i) consiste no cálculo de matriz de rastreabilidade, no qual relacionam-se os casos de teste e métodos do sistema por meio de diagramas de atividades. Informação (metadados) sobre a execução dos casos de teste são são coletados (por exemplo, fluxos de dados envolvidos em cada execução) A fase (ii) contempla a identificação de mudanças nos modelos originais. Por exemplo, são detectadas inclusões, remoções e modificações de transições, fluxos de execuções, chamadas, etc.. A fase (iii), por fim, classifica os casos de teste com base na adaptação envolvida nos modelos. Em particular, os casos de teste são classificados em obsoletos, reutilizáveis e retestáveis.

Ressalta-se que Al-Refai et al. (2016b) utilizaram a mesma técnica de seleção indicada por Yoo e Harman (2012) e Rothermel et al. (1999), a qual diz respeito ao uso da

Técnica Segura de Seleção que é baseada no uso de casos de teste classificados como *modification-traversing*.

7.4 Visão Geral da Literatura

Dado o estudo secundário conduzido (Siqueira et al., 2021) e os trabalhos encontrados relacionados ao *tempo de execução* de SAs e ao teste de regressão, na sequência são apresentadas as principais lacunas encontradas para possíveis microcontroladores de teste de regressão para SAs.

Necessidade de Regressão e Negligência da Limitação de Recursos: Uma adaptação pode gerar mudanças não apenas na estrutura e no comportamento do sistema alvo, mas também gerar mudanças em conjuntos de testes. Com isso, existe a necessidade da aplicação do teste de regressão em *tempo de execução* (Akour et al., 2014). Apesar da inerência da aplicação do teste de regressão em SAs, encontram-se poucos trabalhos que contribuem para o tópico em questão. Dos trabalhos de teste de regressão para SAs, o trabalho de Akour et al. (2014) e o trabalho de Al-Refai et al. (2016b) abordam a aplicação do teste em *tempo de execução*. Entretanto, os trabalhos não abordam a limitação de recursos que pode inviabilizar a aplicação do teste em *tempo de execução*.

Arquitetura de SAs e o Teste de Regressão: Os estudos que abordam o teste de regressão não levam em consideração uma arquitetura descentralizada com camadas de microcontroladores e metacontrolador. Além disso, nos trabalhos que levam em consideração a arquitetura baseada em controlador e sistema alvo, o teste de regressão é negligenciado.

Minimização de Conjunto de Teste de Regressão em SAs: Um único estudo (Akour et al., 2014) aborda a minimização no contexto do teste de regressão em *tempo de execução* para SAs. Entretanto, apenas são fornecidas diretrizes de rastreabilidade entre componentes e casos de teste, negligenciando as demais lacunas mencionadas acima.

Dadas as principais lacunas no contexto de teste de regressão para o tempo de execução em SAs, na sequência é apresentada a especificação de um microcontroladorcom essa finalidade.

7.5 Contexto para Especificar um Microcontrolador de Teste de Regressão

Nos estudos exploratórios descritos nos Capítulos 5 e 6 desenvolveram-se microcontroladores para lidar com falhas (isto é, o *FailureManager*). Alinhado a isso, na Figura 4.2 ilustraram-se alguns possíveis microcontroladores para auxiliar na geração de evidência para o controlador. Em tal figura, que foi adaptada do trabalho de De Lemos e Potena (2017), ilustrou-se também o microcontrolador nomeado "Auto-regressão de testes". A investigação, viabilização e implementação deste microcontrolador é o contexto da especificação apresentada neste capítulo. Mais especificamente, a atividade de teste de regressão sendo aplicada em *tempo de execução* em SAs – em particular ao uso de Técnicas de Minimização de Conjuntos de Testes e Seleção de Conjuntos de Testes – é um segundo passo em direção a mais uma demonstração de flexibilidade da abordagem (o primeiro passo foi a demonstração e desenvolvimento do *FailureManager*).

Para realizar a especificação do microcontrolador, utiliza-se a Análise Estruturada para Definição de Requisitos (Structured Analysis for Requirements Definition - SADT) (Ross e Schoman, 1977). Com base na SADT, descrevem-se os estágios da malha de controle MAPE-K nas seguintes propriedades: (i) entrada de dados; (ii) controle de dados; (iii) mecanismo; e (iv) saída de dados.

Para o microcontrolador de teste de regressão, *entrada de dados* refere-se a elementos arquiteturais, conjuntos de teste e nós de hardware; tais dados são monitorados e são oriundos do sistema alvo e do ambiente. *Controle de dados* refere-se à qualidade esperada (por exemplo, níveis de confiança e precisão; estimativa de confiabilidade). *Mecanismo* refere-se a técnicas utilizadas (por exemplo, para estimar a confiabilidade). Por fim, *saída de dados* refere-se ao resultado com respeito ao mecanismo. Com base nessas quatro propriedades, os estágios baseados no MAPE-K são definidos para o microcontrolador de teste de regressão em *tempo de execução*, como se segue:

- **Monitor:** o monitor tem o objetivo de identificar a necessidade de realizar uma adequação nos conjuntos de testes existentes, sendo que isso que pode ocorrer durante a manipulação de uma adaptação realizada pelo controlador, ou até mesmo caso seja identificada a necessidade de melhoria nos conjuntos de testes como, por exemplo, a eliminação de testes redundantes.

- **Analisador:** o analisador identifica o impacto da regressão, isto é, faz uma análise de hierarquias, dependências e mudanças entre casos de teste. Ressalta-se que a

análise de impacto envolve analisar um conjunto de testes inicial e definir possíveis conjuntos de testes finais.

- **Planejador:** o planejador caracteriza a adequação a ser realizada nos testes. Esta caracterização leva em consideração limitações envolvendo tempo e recursos disponíveis (uma vez que o teste de regressão é aplicado em *tempo de execução*). Neste estágio, planos como os presentes na Tabela 7.1 podem ser gerados – levando em consideração tanto métricas de software quanto atributos vindos de propriedades do controlador principal.

- **Executor:** após caracterizada a adequação a ser realizada nos testes, o executor responsabilizar-se-á por realizar tal adequação. Isso envolve preparar passos necessários para a adequação nos testes. Em outras palavras, o executor preparará os *scripts* necessários para as atividades de (i) eliminação de testes obsoletos e redundantes; e (ii) seleção de conjuntos de testes. Ressalta-se que como os microcontroladores têm o objetivo de auxiliar o controlador com a geração de evidências, o executor também pode gerar dados relevantes para o controlador (por exemplo, dados envolvendo as adequações e execuções de testes).

- ***Knowledge*:** os dados de entrada, os resultados de cada estágio e os dados de saída serão armazenados continuamente no estágio *Knowledge*. Este estágio terá por objetivo representar o sistema alvo sob teste, mais especificamente armazenará dados envolvendo componentes e conexões entre componentes do sistema alvo – por exemplo, na Figura 7.1 e nas figuras subsequentes das próximas seções, os quadros (a), (b), (c) e (d) podem ser dados armazenados no estágio *Knowledge* e utilizados por todos os outros estágios.

Dada a visão geral dos estágios baseados no MAPE-K do microcontrolador de teste de regressão pretendido, na sequência apresentam-se as atividades do contexto deste microcontrolador. Tais atividades envolvem a minimização (isto é, eliminação de testes obsoletos e redundantes) e a seleção (isto é, identificação de testes baseados na limitação de ser aplicado em *tempo de execução*) de conjuntos de testes realizadas pelo microcontrolador.

7.5.1 Eliminação de Testes Obsoletos

Na Figura 7.1 ilustram-se componentes passando por uma adaptação de uma versão A1 origem para uma A2 destino (isto é, C2 e C3 tornando-se C2' e C3') e como isso pode impactar o conjunto de testes, mais especificamente relacionado à eliminação de

testes obsoletos realizada pelo estágio Executor. Tal eliminação ocorrerá de acordo com informações de rastreabilidade definidas em estágios anteriores. Como exemplo, considerando-se que na versão do sistema gerenciado A1 exista um conjunto de testes T1 de 27 casos de testes (ilustrado no quadro (a) da Figura 7.1) após a eliminação de testes obsoletos, o conjunto T2 (quadro (b)) será composto por 17 casos de testes. Como ilustrado na Figura 7.1, para cada componente tem-se um conjunto de teste respectivo àquele componente (por exemplo, componente C2 tem 10 casos de teste e C2' teria 5 casos de teste). Assim, neste trabalho a eliminação de testes obsoletos dará enfoque em testes de unidade (isto é, a regressão ocorrida em cada componente em particular). Além disso, ressalta-se também que a eliminação de testes obsoletos, na Figura 7.1, é representada nos quadros pela cor cinza. Finalmente, na Figura 7.1 ilustra-se no quadro (e) a efetivação da eliminação de obsoletos ocorrida no estágio Executor.

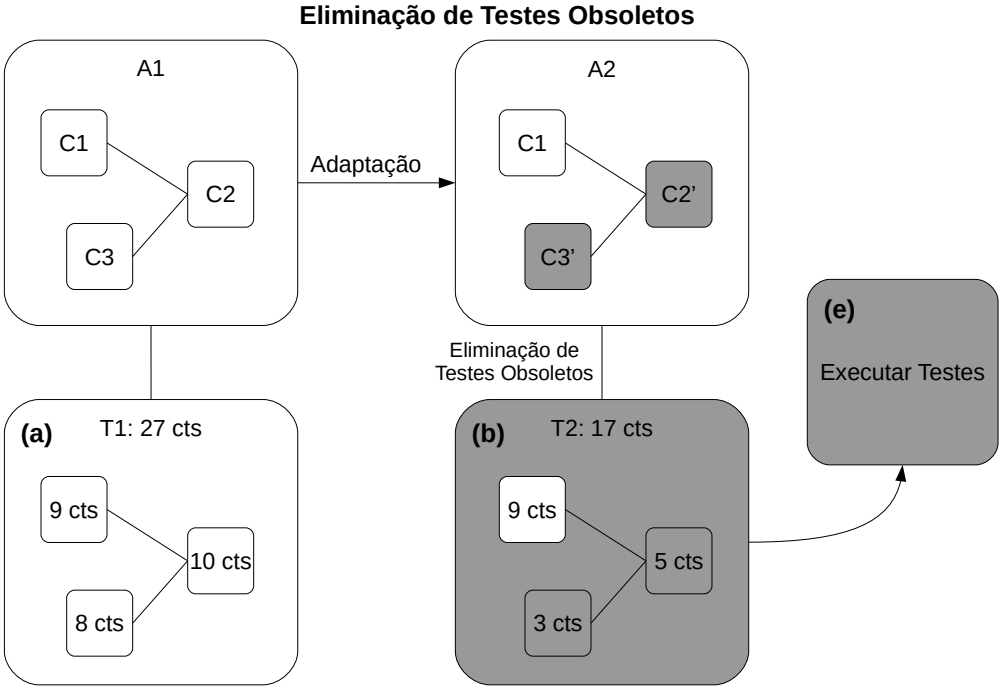

Figura 7.1: Mecanismo do estágio Executor relacionado à Eliminação de Testes Obsoletos.

7.5.2 Eliminação de Testes Redundantes

Na Figura 7.2 apresentam-se, juntamente com a eliminação de testes obsoletos, a eliminação de testes redundantes. Neste caso, observa-se que no quadro (b) o conjunto de testes T2 é resultado da eliminação de obsoletos e no quadro (c) o conjunto de testes T2' é

resultado da eliminação de testes redundantes. Como ilustra-se na Figura 7.2, para os casos de teste origem de cada componente em T2 têm os casos de teste destino em T2' – por exemplo, 9 casos de testes do componente C1 (em branco no quadro (b)) tornando X casos de teste (em branco no quadro (c)). Mais especificamente, um componente será representado como uma unidade de código e esta unidade de código resultará em um grafo de fluxo de controle. Depois, um conjunto de testes mínimo será gerado para cada grafo de fluxo de controle, no qual testes *redundantes* denotam casos de testes que executam um mesmo caminho em particular. Neste processo, poderá ser utilizado um módulo de diferenciação para os componentes baseando-se, por exemplo, numa proposta como a do trabalho de Lahami et al. (2015) (que leva em consideração modelos comportamentais origem e destino, descritos na Seção 7.3.2). Finalmente, na Figura 7.2 ilustra-se no quadro (e) a efetivação da minimização de testes no estágio Executor, composta pelas eliminações de testes obsoletos e redundantes.

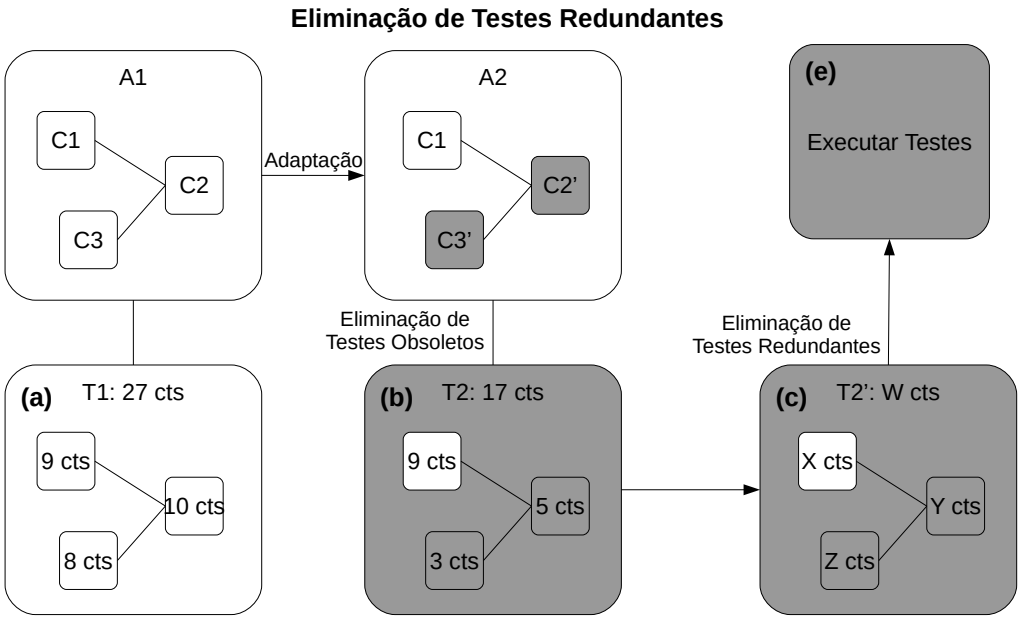

Figura 7.2: Mecanismo do estágio Executor relacionado à Eliminação de Testes Redundantes.

7.5.3 Seleção de Conjuntos de Testes

Na Figura 7.3 apresenta-se por meio do quadro (d) – alternativamente à eliminação de redundantes mostrada no quadro (c) da Figura 7.2 – que se pode utilizar a Seleção de Conjuntos de Testes após a eliminação de testes obsoletos. Especificamente, no estágio Planejador são identificados os recursos estáticos e dinâmicos do sistema, bem como as limitações referentes ao *tempo de execução*. Tais recursos e limitações podem definir a forma como o teste de regressão irá ocorrer. Na Tabela 7.1 ilustram-se alguns exemplos de planos que o estágio Planejador poderá gerar. Assim, a coluna "Plano" refere-se a uma identificação para um plano; a coluna "Tarefa" refere-se às tarefas que podem ser realizadas em um determinado plano; a coluna "Tempo" refere-se a quantos segundos determinada tarefa leva para ser executada; e a coluna "Recursos Utilizados" refere-se aos algoritmos e uso de memória que cada tarefa irá consumir. Neste sentido, o estágio Planejador gerará planos de teste de regressão similares aos da Tabela 7.1, de modo a apoiar na escolha da estratégia adequada, de acordo com os recursos e tempo disponíveis. Por exemplo, se o tempo disponível para a regressão for de 10 segundos, os planos 2 e 4 seriam candidatos a serem executados. Em particular, o plano 2 é composto por 2 tarefas que levam 6 segundos para serem executadas (oriundas da execução de 4 algoritmos e a soma de 2 diferentes usos de memória). O plano 4, por sua vez, é composto por 3 tarefas que levam 10 segundos para serem executadas (oriundas da execução de 6 algoritmos e a soma de 3 diferentes usos de memória).

Dadas a técnica de Minimização (eliminação de testes obsoletos e redundantes) e a técnica de Seleção, o controlador poderá utilizar ambas em conjunto. Neste sentido, na Figura 7.4 ilustra-se que após a Eliminação de Testes Obsoletos (quadro (b)), pode-se utilizar a Eliminação de Testes Redundantes (quadro (c)) e a Seleção de Conjuntos de Testes (quadro (d)) de modo que uma técnica auxilie na outra. Por exemplo, se a limitação de recursos impactar na execução dos testes, a seleção de conjuntos de testes pode ser mais viável de ser aplicada inicialmente, ao passo que a eliminação de redundância poderá ser aplicada em um segundo momento. Ressalta-se também que, neste caso, deve-se avaliar um *trade-off* envolvendo ambas as atividades de Eliminação de Testes de Redundantes e Seleção de Conjuntos de Testes. Este *trade-off* poderia ser incluído também nos planos gerados no estágio Planejador, ilustrados na Tabela 7.1.

Os testes aos quais demandariam de ser executados e armazenados em si estão a cargo dos planos de testes, como apresentados na Tabela 7.1. Então, um plano de teste conteria os respectivos mecanismos que fariam parte do microcontrolador testExecutor.

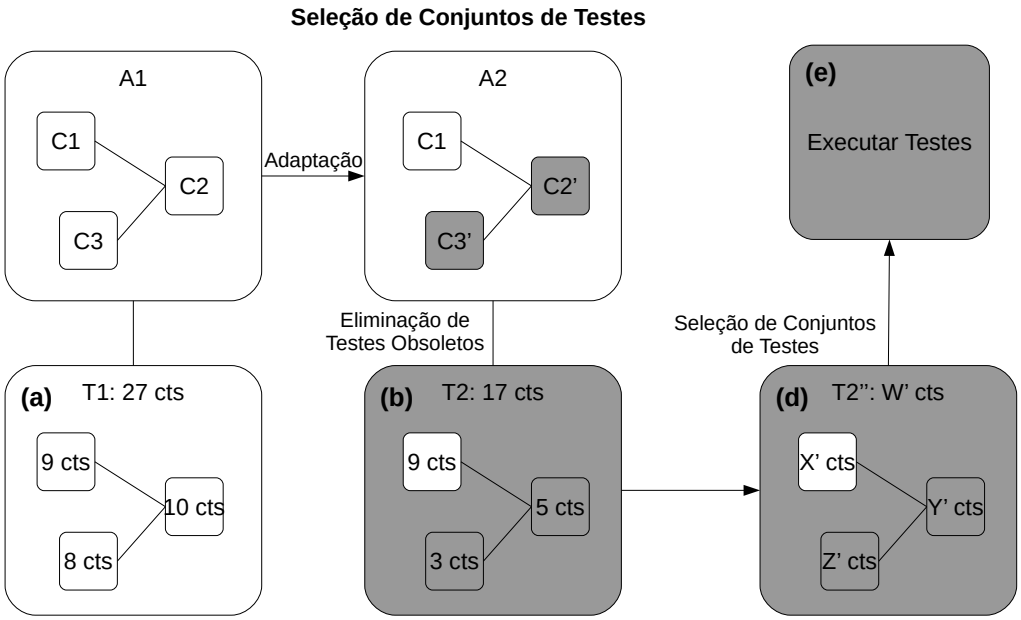

Figura 7.3: Processo do estágio Executor relacionado à Seleção de Conjuntos de Testes.

7.6 Especificação de um Microcontrolador de Teste de Regressão

Nesta seção apresentam-se as especificações a respeito dos componentes necessários para a criação de um microcontrolador de teste de regressão. Neste caso, enfatiza-se um cenário de minimização de casos de testes, em particular, para a eliminação de teste obsoletos e testes redundantes.

A especificação do microcontrolador de teste de regressão utiliza o *framework Kubow* e os artefatos necessários para uso no *Kubow* (isto é, implementações em ACME e Stich). Além disso, também decidiu-se pela proposição de um microcontrolador não baseado no *Kubow* que é responsável por executar os testes. Para isso, utilizou-se a configuração Meta-KZ, descrita no Capítulo 6. Ademais, nota-se que as estratégias Stich a serem implementadas no microcontrolador, que é ilustrado na Figura 7.5, são descritas na Tabela 7.2.

Metacontrolador: O metacontrolador continua tendo o seu principal objetivo, isto é, reconfigurando o controlador (composto por microcontroladores) de acordo com a ne-

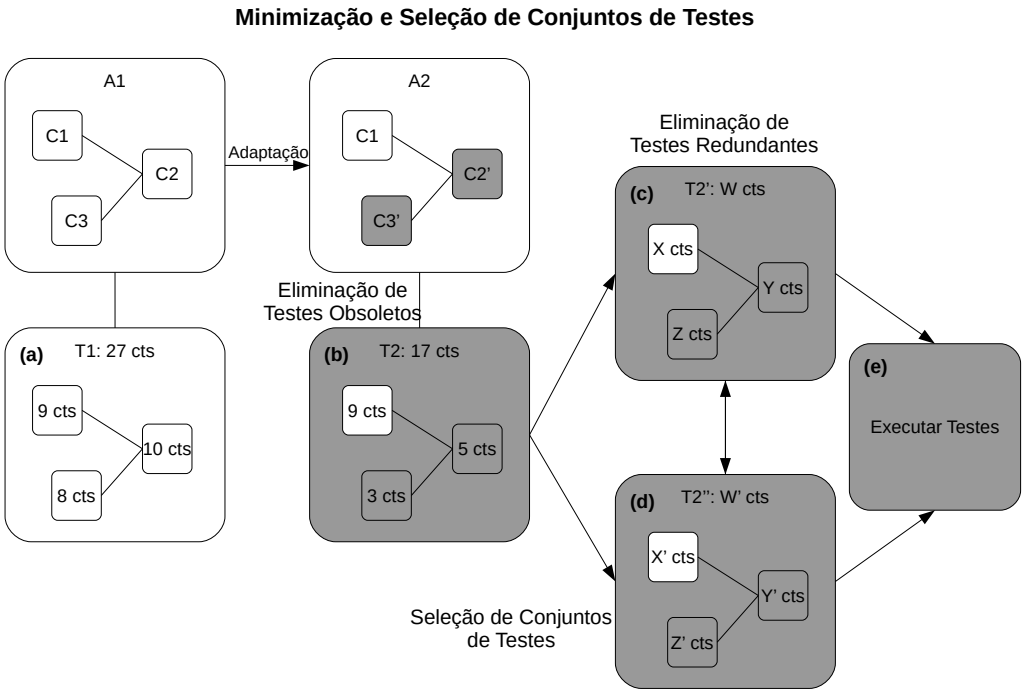

Figura 7.4: Visão geral do processo do estágio Executor referente à Minimização e à Seleção de Conjuntos de Testes.

cessidade. Para o metacontrolador manipular o microcontrolador de teste de regressão, sugeriram-se duas estratégias a serem implementadas no metacontrolador.

- activateRegressionTest: esta estratégia obtém os limiares de falhas gerados pelo microcontrolador FailureManager (descrito no Capítulo 6). Assim, caso o predicado isHighFailureRate seja verdadeiro, a tática removeRegressionTest desabilitará o microcontrolador RegressionTest.

- deactivateRegressionTest: assim como em activateRegressionTest, esta estratégia também obtém os limiares de falhas do FailureManager. No entanto, em caso do predicado isLowFailureRate ser verdadeiro, então a tática addRegressionTest habilitará o microcontrolador RegressionTest.

Microcontrolador RegressionTest: Este microcontrolador tem o objetivo de selecionar atividades de teste para compor planos de execução de teste. As duas atividades de teste sugeridas foram: (1) eliminar testes obsoletos; e (2) eliminar testes redundantes. Para habilitar e desabilitar planos de teste, definem-se duas estratégias de adaptação.

- activatePlanHighSLO: Esta estratégia obtém limiares quanto à carga de execução presente no Kube-ZNN. Em caso do predicado HighSLO && !runningTests ser verdadeiro,

Tabela 7.1: Exemplos de planos de regressão que podem ser gerados e analisados durante o estágio Planejador.

Plano	Tarefa	Mecanismo	Tempo	Recursos Utilizados
1	1	Eliminação de obsoletos	3 segundos	Algoritmo 1; Algoritmo 2; X em uso de Memória
1	2	Eliminação de redundantes	8 segundos	Algoritmo 2; Algoritmo 3; Y em uso de Memória
1	3	Execução de testes	3 segundos	Algoritmo 4; Algoritmo 6; Z em uso de Memória
	Total	3 tarefas	14 segundos	Algoritmos 1, 2, 2, 3, 4, 6; Uso de memória X + Y + Z
2	1	Eliminação de obsoletos	3 segundos	Algoritmo 1; Algoritmo 2; X em uso de Memória
2	2	Execução de testes	3 segundos	Algoritmo 4; Algoritmo 6; Z em uso de Memória
	Total	2 tarefas	6 segundos	Algoritmos 1, 2, 4, 6; Uso de memória X + Z
3	1	Eliminação de obsoletos	3 segundos	Algoritmo 1; Algoritmo 2; X em uso de Memória
3	2	Eliminação de redundantes	8 segundos	Algoritmo 2; Algoritmo 3; Y em uso de Memória
3	3	Novos casos de testes	5 segundos	Algoritmo 7; W em uso de memória
3	4	Execução de testes	3 segundos	Algoritmo 4; Algoritmo 6; Z em uso de Memória
	Total	4 tarefas	19 segundos	Algoritmos 1, 2, 2, 3, 7, 4, 6; Uso de memória X + Y + W + Z
4	1	Eliminação de obsoletos	3 segundos	Algoritmo 1; Algoritmo 2; X em uso de Memória
4	2	Seleção de casos de testes	4 segundos	Algoritmo 8; Algoritmo 9; V em uso de Memória
4	3	Execução de testes	3 segundos	Algoritmo 4; Algoritmo 6; Z em uso de Memória
	Total	3 tarefas	10 segundos	Algoritmos 1, 2, 8, 9, 4, 6; Uso de memória X + V + Z

conclui-se que os servidores do Kube-ZNN não estão sobrecarregados e não existe outro plano de teste em execução. Portanto, a tática addCompleteTestPlan define um conjunto completo de atividades de teste. Em particular, neste caso, tanto a atividade de eliminação de testes obsoletos quanto a eliminação de testes redundantes são escalonadas.

- activatePlanLowSLO: No caso desta estratégia, o limiar considerará situações em que o predicado LowSLO && !runningTests seja verdadeiro. Ou seja, em caso dos servidores do Kube-ZNN estiverem sobrecarregados, a tática addLightTestPlan irá escalonar somente a atividade de eliminação de testes obsoletos.

As táticas, que irão ser executadas para escalonar planos de testes utilizarão o atuador rollOut. Este atuador, por sua vez, irá modificar a imagem do microcontrolador TestExecutor.

Microcontrolador TestExecutor: Este microcontrolador tem o exclusivo objetivo de executar testes escalonados pelo microcontrolador RegressionTest. As principais características são:

- **Estratégia**: a estratégia manageTestPlans será responsável por identificar novos escalonamentos realizados pelo microcontrolador RegressionTest.
- **Tática**: a tática execTestPlan irá executar o respectivo plano escalonado, armazenando resultados de testes para outros microcontroladores, e o próprio metacontrolador, utilizarem posteriormente.
- **Imagem**: a imagem seriam as diferentes implementações para contemplar a execução dos planos de teste. Em outras palavras, por exemplo, a imagem poderia ser uma implementação em Python (similar à implementada no microcontrolador FailureManager). Assim, diferentes imagens podem ser implementadas a fim de compor planos de teste escalonados pelo microcontrolador RegressionTest.

Tabela 7.2: Regras nas estratégias de adaptação com a inclusão do microcontrolador de teste de regressão.

Componente	Estratégia	Tática	Predicado
Metacontrolador	activateRegressionTest	addRegressionTest	isLowFailureRate
Metacontrolador	deactivateRegressionTest	removeRegressionTest	isHighFailureRate
Microcontrolador RegressionTest	activatePlanHighSLO	addCompleteTestPlan	HighSLO && !runningTests
Microcontrolador RegressionTest	activatePlanLowSLO	addLightTestPlan	LowSLO && !runningTests
Microcontrolador TestExecutor	manageTestPlans	execTestPlan	isThereTestsToExecute

7.7 Implementação Sugerida do Microcontrolador de Teste de Regressão

Dada a especificação geral de atividades que um microcontrolador poderia realizar, nesta seção enfatiza como que o mesmo poderia ser incluído numa configuração como a Meta-KZ. Nesta seção apresenta-se uma possível implementação do microcontrolador especificado na seção anterior. Em resumo, a seguir apresentam-se, dado uma configuração como

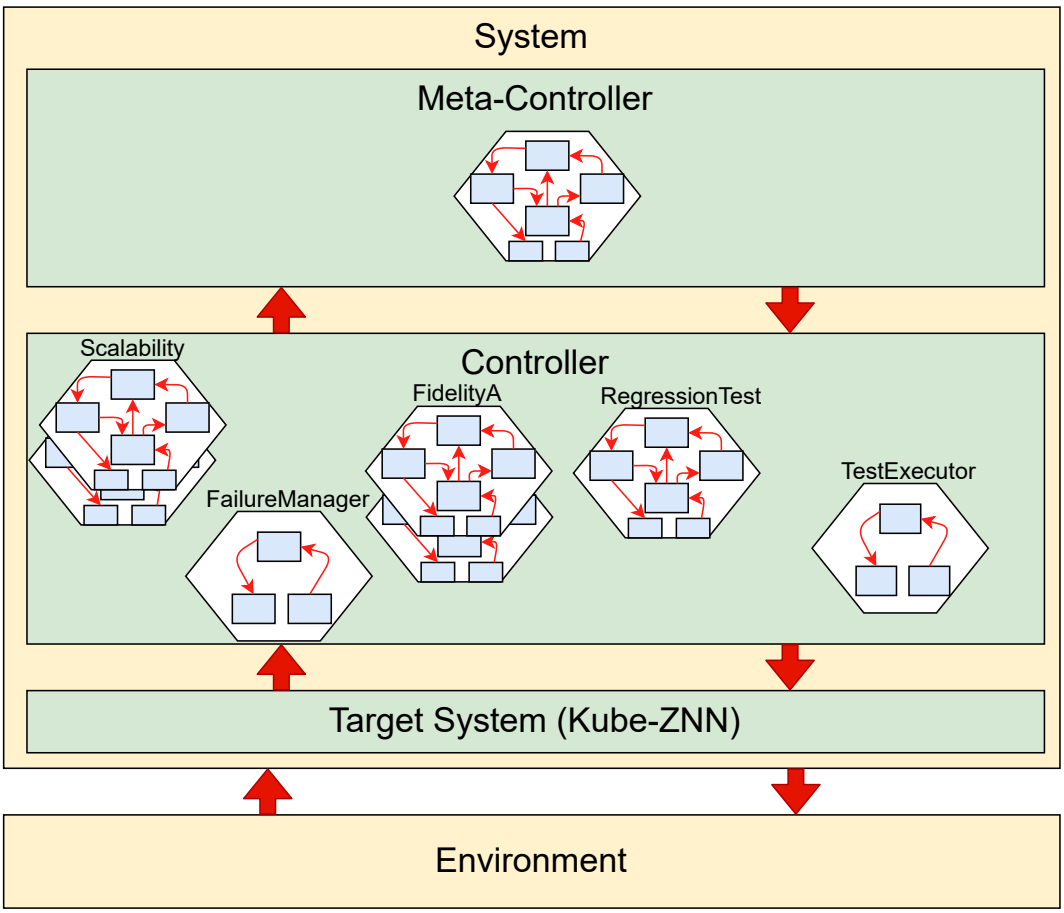

Figura 7.5: A configuração Meta-KZcom a inclusão de um microcontrolador de teste de regressão.

a Meta-KZ, as modificações deveriam ser realizadas no metacontrolador (isto é, para habilitar/desabilitar o microcontrolador) às quais fazem parte a inclusão de trechos de código ACME e estratégias de adaptação Stich. O uso de algumas probes já presentes na versão anterior (por exemplo, probe SLO). E também a implementação do próprio RegressionTest como uma instância Kubow, incluindo modelos ACME que representam o sistema alvo e estratégias stich para agendamento de planos de teste. Finalmente, a implementação de uma aplicação que poderia realizar a interface entre o microcontrolador RegressionTest e as execuções dos planos de testes.

Modificações no metacontrolador: o metacontrolador, que utiliza uma instância *Kubow*, deverá receber modificação no modelo de arquitetura ACME para contemplar o novo microcontrolador RegressionTest (como ilustrado no trecho de código 7.1). Uma vez definido o modelo de arquietura em ACME, o metacontrolador, via *framework Kubow*, irá interagir

com o respectivo microcontrolador utilizando o ambiente Kubernetes – isto é, utilizando *Services* e *Deployments* como destacados no trecho de código 7.1 entre as linhas 7 e 17.

```
1  ...
2    /*
3      Metacontroller RegressionTest architecture
4    */
5
6      //Component RegressionTest Deployment
7      Component RegressionTestD : DeploymentT = new DeploymentT
8        extended with {
9          Port apiPort  = { }
10
11         Property namespace = "default";
12         Property name = "RegressionTest";
13
14         Property minReplicas = 0;
15         Property maxReplicas = 1;
16     }
17
18     Component RegressionTestS : ServiceT =
19         new ServiceT extended with {
20       Port apiPort  = { }
21       Property name = "RegressionTest";
22     }
23
24     Connector RegressionTestServiceConnector  =
25         new LabelSelectorConnectorT extended with {
26       Property selectors =
27       <[name : string = "app";value : string = "RegressionTest";]>;
28     }
29
30     Attachment RegressionTestS.apiPort to
31         RegressionTestServiceConnector.callee;
32     Attachment RegressionTestD.apiPort to
33         RegressionTestServiceConnector.caller;
34
35 ...
```

Listagem de Código 7.1: Exemplo de um trecho de implementação do componente do microcontrolador RegressionTest

Também será necessária a inclusão de estratégias e táticas Stich para gerenciar a ativação e desativação do respectivo microcontrolador. O trecho de código 7.2 demonstra a

inclusão de tais estratégias, também destacadas na Tabela 7.2. Portanto, o metacontrolador torna-se apto a ativar e desativar o microcontrolador RegressionTest (isto é, linhas 4 e 12) de acordo com a taxa de falhas provenientes do microcontrolador FailureManager, que são obtidas do sistema alvo Kube-ZNN (isto é, linhas 3 e 11).

```
1  ...
2      /* Estrategia para adicionar o microcontrolador RegressionTest */
3      strategy activateRegressionTest [ LowFailureRate ] {
4        t0: (isLowFailureRate) -> addRegressionTest() @[20000 /*ms*/] {
5          t0a: (success) -> done;
6        }
7        t1: (default) -> TNULL;
8      }
9
10     /* Estrategia para remover o microcontrolador RegressionTest */
11     strategy deactivateRegressionTest [ HighFailureRate ] {
12       t0: (isHighFailureRate) -> removeRegressionTest() @[20000 /*ms*/] {
13         t0a: (success) -> done;
14       }
15       t1: (default) -> TNULL;
16     }
17 ...
```

Listagem de Código 7.2: Exemplo de um trecho de implementação de estratégias em Stich do Metacontrolador

Instância *Kubow* do microcontrolador RegressionTest: uma vez utilizando-se o *framework Kubow* como base para a definição do microcontrolador RegressionTest, os seguintes componentes são definidos:

- **Probes e Gauges**: os Probes e Gauges utilizados rementem-se aos dados de nível de qualidade devido à carga de acesso no Kube-ZNN (isto é, LowSLO e HighSLO). Assim, para monitorar tais valores, Probes e Gauges serão geradas. Como exemplo, tem-se um trecho da probe no Código 7.3. Entre as linhas 15 e 19 tais níveis são obtidos.

```
1  ...
2      /* Probe para obtencao do nivel de SLO no Kube-ZNN */
3
4    SloProbe:
5      alias: kube-znn.slo.probe
6      location: 127.0.0.1
7      type: java
```

```
8       javaInfo:
9         class: br.unifor.kubow.probes.PrometheusProbe
10        period: 5000
11        args.length: 4
12        args.0: "default"
13        args.1: "kube-znn"
14        args.2: "slo"
15        args.3: "sum(
16          rate(request_duration_seconds_bucket{
17          le=\"00.010\"}[30s]))
18          /
19          sum(rate(request_duration_seconds_count[30s]))"
20  ...
```

Listagem de Código 7.3: Exemplo de um trecho de implementação da SLO do Kube-ZNN

- **Modelo de arquitetura**: uma vez que o microcontrolador RegressionTest tem como objetivo lidar com o sistema alvo e o microcontrolador TestExecutor, o modelo ACME, similar ao ilustrado no trecho de código 7.1, terá a representação tanto do Kube-ZNN quanto do TestExecutor.

- **Estratégias e táticas**: a definição das estratégias e táticas implementadas em Stich habilitam o microcontrolador a manipular tanto o sistema alvo (isto é, Kube-ZNN) quanto o TestExecutor, via modelo de arquitetura ACME. Como ilustrado no trecho de código 7.4, nas linhas 4 e 12 as estratégias respectivas executam as táticas para escalonar planos de teste, de acordo com os níveis de SLO obtidos nas linhas 3 e 11.

```
1   ...
2       /* Estrategia para adicionar o plano de teste completo */
3       strategy activatePlanHighSLO [ HighSLO ] {
4         t0: (!runningTests) -> addCompleteTestPlan() @[20000 /*ms*/] {
5           t0a: (success) -> done;
6         }
7         t1: (default) -> TNULL;
8       }
9
10      /* Estrategia para adicionar o plano de teste reduzido */
11      strategy activatePlanLowSLO [ LowSLO ] {
12        t0: (!runningTests) -> addLightTestPlan() @[20000 /*ms*/] {
13          t0a: (success) -> done;
14        }
15        t1: (default) -> TNULL;
16      }
```

```
17  ...
```

Listagem de Código 7.4: Exemplo de um trecho de implementação de estratégias em Stich do Microcontrolador RegressionTest

Atuadores e acesso ao ambiente Kubernetes: Ainda no microcontrolador RegressionTest, a definição de táticas para habilitar os planos de testes é necessária. Como ilustrado no trecho de código 7.5, as linhas 9 e 12 verificam qual plano de teste deve ser habilitado. Dependendo do plano de teste, nas linhas 10 e 13 define-se qual imagem irá pertencer ao microcontrolador TestExecutor.

```
1  ...
2      /* Estrategia para modificar a imagem do TestExecutor */
3      tactic changeTestPlan() {
4        runningTests = true;
5        condition {
6          isCompletePlan || isLightPlan;
7        }
8        action {
9          if (isCompletePlan) {
10           M.rollOut(M.testExecutor, "testExecutor", CompletePlanImage1);
11         }
12         if (isLightPlan) {
13           M.rollOut(M.testExecutor, "testExecutor", LightPlanImage1);
14         }
15       }
16       effect @[10000] {
17         runningTests = false;
18       }
19     }
20 ...
```

Listagem de Código 7.5: Exemplo de um trecho de implementação de táticas para mudar as imagens do microcontrolador TestExecutor.

Uma imagem, para ser utilizada em um microcontrolador TestExecutor, poderia utilizar a API do Kubernetes (linhas 5 e 6 do trecho de código 7.6) para interagir com o sistema alvo e ambiente. Na linha 9, tem-se uma estrutura de repetição em Python para lidar com os eventos e mensagens que ocorrem no ambiente Kubernetes. Entre as linhas 12 e 16, define-se um objeto a partir dos eventos obtidos do Kubernetes. Depois, entre as linhas 18 e 29, é realizada a interação com o sistema alvo. Em particular nas linhas 27, 28 e 29, um teste de exemplo é realizado. Assim, seguindo a mesma estrutura, com uso da API do

Kubernetes, podem-se gerar diferentes scripts que podem compor as diferentes imagens a serem utilizadas pelo TestExecutor.

```
...

    def main():

    config.load_incluster_config()
    v1 = client.CoreV1Api()
    w = watch.Watch()

    for event in w.stream(v1.list_event_for_all_namespaces):
        if event['object'].type == "Warning":

            message = {"name": event['object'].metadata.name,
                       "type": event['object'].type,
                       "message": event['object'].message,
                       "dateevent": str(event['object']
                    .metadata.creation_timestamp)}

            resultNamePod = re.search(
                'kube-znn', str(message["name"]), re.IGNORECASE)
            resultMessageFound = re.search(
                'Mensagem esperada', str(message["message"]),
                re.IGNORECASE)

            if resultNamePod and resultMessageFound:
                print("menssagem encontrada")
                logging.warning(message)
                response = httpx.post(f"{url_host}/performingTest",
                                      headers=headers,
                                      json=message)

    logging.info("Finished pod stream.")

...
```

Listagem de Código 7.6: Manipulando o ambiente Kubernetes

Em linhas gerais, a implementação proposta neste capítulo, deu-se ênfase na configuração Meta-KZ poder receber mais microcontroladores para comporem o controlador que lida com o sistema alvo (neste caso, o Kube-ZNN). Para inclusão do mesmo, necessitaria de modificações no metacontrolador (para habilitar/desabilitar o microcontrolador); a definição do próprio microcontrolador RegressionTest; e de um microcontrolador responsável

por executar planos de testes (isto é, TestExecutor) agendados pelo RegressionTest. Tal proposta somente contempla o nível de abstração da ativação/desativação do microcontrolador RegressionTest realizada pelo metacontrolador; e o agendamento de planos de teste com base no contexto do sistema alvo. Desdobramentos dessa especificação e implementação seriam as implementações de cada um dos mecanismos descritos da Tabela 7.1 (segunda coluna), bem como novos componentes e microcontroladores para adequar as demais atividades de teste de regressão (por exemplo, minimização, seleção e priorização).

7.8 Considerações Finais

Neste capítulo, apresentaram-se os principais resultados de uma Revisão Sistemática (RS) da Literatura sobre teste de Sistemas Adaptativos; destacaram-se as principais abordagens no contexto do teste em *tempo de execução*; descreveram-se as abordagens de teste de regressão; contextualizou-se o uso de abordagem de teste de regressão para se criar um microcontrolador com o objetivo de atuar na minimização de testes; e, por fim, especificou-se a implementação do microcontrolador RegressionTest utilizando o *framework Kubow*, com auxílio de um microcontrolador sob medida para a execução de testes (isto é, TestExecutor). Com a criação de mais um microcontrolador que pode ser incluído numa configuração como a Meta-KZ, evidencia-se a flexibilidade da abordagem de controladores baseados em microcontroladores.

Capítulo

8

Conclusões

Na literatura, empregam-se abordagens com uma visão *top-down* da estrutura do controlador, dificultando-se a definição específica dos componentes do mesmo. Ao abordar descentralização de controladores, em geral os trabalhos dão ênfase na definição baseada no MAPE-K. Além disso, o reúso em si tem sido abordado utilizando-se controladores monolíticos, o que inclui o uso de modelos que restringem a estrutura/comportamento do controlador e o uso de modelos de componentes em execução. Ainda, cada estágio do MAPE-K pode ser composto por várias atividades. Dependendo da diversidade nas atividades presentes em cada estágio do MAPE-K – ao utilizá-lo como modelo de referência – necessita-se personalizar controladores.

Nesta tese, teve-se como hipótese que *um controlador que é baseado em microcontroladores tem a vantagem da flexibilidade estrutural sem comprometer a reconfiguração e o desempenho do sistema alvo.* Alinhando-se à hipótese, duas questões de pesquisa foram investigadas:

- **QP1:** Um *controlador descentralizado* é vantajoso com relação a adaptações estruturais do sistema alvo quando comparado com um *controlador monolítico*?
- **QP2:** Um *controlador descentralizado* é vantajoso com relação a adaptações paramétricas do sistema alvo quando comparado com um *controlador monolítico*?

Dadas a hipótese a ser avaliada e as questões de pesquisa a ela relacionadas, delineou-se uma expectativa que é apresentada a seguir. Após os estudos realizados, observou-se uma realidade que também é descrita na sequência.

Expectativa: Uma vez que controladores foram compostos por microcontroladores, esperava-se que tais microcontroladores pudessem ser reutilizáveis em diferentes controladores. Além disso, a expectativa era que mesmo com o aumento do uso de recursos demandados pelos microcontroladores, as recomposições gerenciadas pelo metacontrolador resultariam em controladores compostos por microcontroladores específicos, de modo que esses últimos poderiam lidar de forma mais eficaz com as suas respectivas funções quando comparados com um controlador monolítico. Por outro lado, um controlador com vários microcontroladores e com uma camada adicional do metacontrolador – demandando mais recursos – poderia eventualmente lidar com problemas de alocação de recursos e a reconfiguração do sistema alvo. Além disso, uma vez que a definição de microcontroladores demanda eventualmente propriedades similares entre tais microcontroladores, esperava-se um grau de flexibilidade que permitisse reutilização de código entre a definição de microcontroladores.

Realidade: Observado-se principalmente o estudo que utilizou o *PhoneAdapter*, criar microcontroladores específicos e enxutos os torna aptos a eventualmente serem reutilizados em diferentes controladores; e por serem enxutos e específicos, podem-se utilizar somente recursos necessários. Com relação ao observado no estudo que utilizou o *Kubow*, mesmo com uma maior demanda por recursos no uso de microcontroladores específicos, puderam-se otimizar os resultados do sistema alvo. A configuração Meta-KZ demandou uma menor quantidade de servidores ativos no Kube-ZNN. Além disso, os resultados quanto à qualidade de mídia no sistema alvo não foram impactados. A flexibilidade também pôde ser observada, por exemplo, durante a criação dos novos microcontroladores para o controlador baseado no *framework Kubow*. Mais especificamente, fez-se: (1) o desmembramento da configuração Mon-KZ para gerar diferentes microcontroladores; (2) a adição do microcontrolador para gerenciamento de falhas; e (3) a especificação do microcontrolador para teste de regressão.

Com relação a hipótese de que *um controlador que é baseado em microcontroladores tem a vantagem da flexibilidade estrutural sem comprometer a reconfiguração e o desempenho do sistema alvo*, a mesma se confirma, uma vez que a flexibilidade estrutural é constatada tanto no estudo do *PhoneAdapter* quanto no estudo envolvendo o Kubow. Além disso, o

desempenho e a reconfiguração também não são comprometidos ao analisar as comparações realizadas nos estudos envolvendo o *framework* Kubow.

Algo que se deve destacar a respeito do uso de uma abordagem baseada em microcontroladores é a demanda por recursos que a abordagem requer. Como ilustrado na Tabela 6.2, nota-se que uma abordagem descentralizada (isto é, Des-KZ) demanda uma quantidade maior de recursos que uma abordagem monolítica (isto é, Mon-KZ), ao passo que uma abordagem descentralizada com um metacontrolador demanda ainda mais recursos.

8.1 Sumarizações das Contribuições

1) A proposição de uma abordagem para projetar controladores que podem ser sintetizados a partir de conjuntos independentes de microcontroladores.

No Capítulo 4 apresenta-se a abordagem hierárquica de multicamadas em que o controlador é alvo de adaptação. Este alvo de adaptação faz com que uma série de microcontroladores sejam utilizados para compor diferentes controladores de acordo com as necessidades do sistema alvo e ambiente. Cada um dos estágios pertencentes a uma malha de controle, por exemplo, MAPE-K, pode conter variadas atividades que podem ser sintetizadas em diferentes microcontroladores. Um exemplo de microcontrolador foi o FailureManager utilizado tanto no estudo do *PhoneAdapter* quanto no estudo do Kubow. Em particular o microcontrolador RegressionTest foi especificado para o estudos com uso do Kubow. A composição de microcontroladores pode ser realizada com base em diferentes estágios presentes em uma malha de controle MAPE-K, tanto o microcontrolador FailureManager quanto o microcontrolador RegressionTest podem estar presentes em um estágio de Analisador (*Domínio do Problema* e *Domínio da Solução*) como é ilustrado no Capitulo 4 na Figura 4.2.

2) A condução de um estudo exploratório utilizando microcontroladores e um metacontrolador para ilustrar a manipulação de mudanças em tempo de execução e em tempo de projeto

No Capítulo 5 apresentou-se um estudo exploratório a fim de comparar duas versões do *PhoneAdapter*. Identificaram-se benefícios que a abordagem desta tese provê em termos de flexibilidade estrutural na composição de controladores. Os benefícios foram observados ao se analisar ambas as versões monolítica e com microcontroladores. Na versão monolítica, o trato com com novos sensores impactava diferentes trechos de código fonte de um mesmo componente. Até pequenas modificações futuras requereriam refatorações generalizadas.

Ressalta-se que é desnecessário instanciar componentes que contenham todos os sensores disponíveis de um Smartphone em situações que apenas um sensor em particular estaria operacional – por exemplo, somente sensor de *Bluetooth* ligado.

A abordagem com a definição microcontroladores, por outro lado, além de ajudar na separação de interesses, possibilitou a definição de microcontroladores específicos para lidar com sensores específicos. Assim, estando o controlador dividido em vários microcontroladores (cada um responsável por um particular sensor), bastava o metacontrolador reconfigurar somente quais microcontroladores eram demandados pelo sistema alvo em *tempo de execução.*

3) A condução de estudos exploratórios para comparar diferentes configurações de controladores, a fim de avaliar o desempenho da abordagem deste trabalho frente ao estado da arte.

No Capítulo 6, a viabilidade da abordagem, que pôde também prover flexibilidade e reúso, foi demonstrada e avaliada com o uso de três diferentes configurações arquiteturais de controladores (monolítica, descentralizada, e descentralizada com um metacontrolador), e implantada sobre o estudo de caso utilizando o sistema alvo Kube-ZNN. A evidência coletada para a avaliação foi relacionada a dois atributos chaves do Kube-ZNN: flexibilidade estrutural em termos de alocação de recursos em números de servidores (isto é, escalabilidade de recursos), e adaptação paramétrica relacionada à reconfiguração de conteúdos providos pelos servidores (isto é, fidelidade de conteúdo). A partir dos resultados obtidos, concluiu-se que o projeto de controladores baseados em microcontroladores permite a definição de controladores que são estruturalmente flexíveis, sem comprometer o desempenho do sistema como um todo. Isso foi demonstrado ao analisar – por meio de dados quantitativos de simulação de carga de requisições de usuário – as adaptações estruturais e paramétricas que o controlador realizou no sistema alvo.

4) A especificação de um microcontrolador de teste de regressão, a fim de demonstrar a flexibilidade da abordagem em incluir diferentes estágios de uma malha de controle que podem incluir/personalizar funcionalidades a um controlador, bem como definir microcontroladores que podem ser reutilizáveis em diferentes controladores e sistemas alvo.

No Capítulo 7, a flexibilidade da abordagem foi demonstrada com a proposição de mais um microcontrolador; em específico, o microcontrolador de teste de regressão. A proposição incluiu a especificação do microcontrolador e uma possível implementação utilizando o *framework Kubow*.

A flexibilidade foi demonstrada ao adicionar um novo microcontrolador que agrega funcionalidades ao controlador em execução. Bastando o sistema alvo ter *probes* e os microcontroladores compartilharem o ambiente (ou terem protocolos de comunicação bem definidos), podem-se adicionar microcontroladores para suprir futuras demandas do sistema alvo. Além disso, a atuação deste microcontrolador pode ser personalizada pelo metacontrolador. Como no Capítulo 7, o microcontrolador RegressionTest só é habilitado quando a taxa de falhas no sistema alvo está baixa.

8.2 Limitações e Lições Aprendidas

Nesta seção ressaltam-se as principais limitações e lições aprendidas na condução dos experimentos.

Plataforma Android: Apesar da facilidade em desenvolver e depurar possíveis controladores na plataforma Android, esta plataforma não foi utilizada na literatura para a definição de controladores genéricos e flexíveis – como no caso do Rainbow. Assim, para ter abordagens e plataformas para realizar comparações, o uso da plataforma Android neste trabalho (isto é, com *PhoneAdapter*) foi apenas como estudo exploratório.

***Framework* Rainbow**: As linguagens ACME e Stich foram as linguagens utilizadas utilizadas para a definição do modelo arquitetural e a definição das estratétias de adaptação, respectivamente. Apesar de ACME e Stich serem linguagens de fácil aprendizagem, realizar atividades de manutenção e depuração são atividades onerosas e demoradas, especialmente devido a falta de ferramentas para a manipulação das mesmas.

Framework Kubow: *Kubow* foi uma iniciativa desenvolvida por Aderaldo et al. (2019) com o intuito de facilitar a especificação e o desenvolvimento de controladores baseados no Rainbow para SAs. Apesar da curva de aprendizagem envolvendo as ferramentas do *Kubow*, o mesmo foi essencial à especificação e ambientação das abordagens utilizadas neste trabalho. A portabilidade para o Kubernetes realizada pelo autores do *Kubow* forneceu todos os subsídios para a ambientação, desenvolvimento e monitoramento dos experimentos realizados neste trabalho.

Monitoramento e relógios lógicos: A dificuldade de se depurar as linguagens ACME e Stich sem ferramentas foi agravada devido aos componentes do *framework* Rainbow serem

distribuídos e realizarem suas comunicações baseada em eventos e tempo. Apesar da dificuldade, uma forma de mitigar consistiu em realizar experimentos com várias execuções a fim de capturar variações entre os resultados.

Limitação de recursos: no Capítulo 6, descreveu-se onde os experimentos foram realizadas para comparar as configurações Mon-KZ, Des-KZ e Meta-KZ deste trabalho. Por um lado, propositalmente, limitou-se os recursos para analisar como que cada configuração lidaria com uma quantidade reduzida de recursos. Por outro lado, em caso de os experimentos serem realizados sem um limite de recursos, as configurações poderiam desempenhar diferentemente.

Cargas de execução: para simulação de carga de execução, refletida em requisições de usuários, utilizaram-se dois diferentes períodos para comparação: 2 minutos (nomeado de *short*) e 5 minutos (nomeado de *long*). Por um lado, caso sejam utilizados outros períodos, diferentes dados podem ser gerados. Por outro lado, outras formas de analisar os dados gerados seriam as não baseadas em um período predeterminado, mas sim analisar quanto tempo leva para um conjunto de recursos ser alocado e/ou desalocado.

Geração e variação de dados: assim como mencionado no tópico *Monitoramento e relógios lógicos*, devido a características como comunicação distribuída presente no sistema alvo e nos componentes do controlador, variações entre diferentes execuções ocorrem. Para os experimentos, realizaram-se 20 execuções para cada configuração Mon-KZ, Des-KZ e Meta-KZ. Assim, utilizaram-se gráficos *boxplot* para auxiliar na interpretação dos dados e das respectivas variações para cada configuração.

8.3 Publicações no Período

Nesta seção apresentam-se os trabalhos divulgados e conduzidos no contexto da tese em questão. Na sequência, apresentam-se tais trabalho em ordem de relação e contribuição direta para com a tese.

Trabalho Siqueira et al. (2020b)

Título: Micro-controllers: Promoting Structurally Flexible Controllers in Self-Adaptive Software Systems

Descrição: no trabalho Siqueira et al. (2020b) iniciou-se a definição de microcontroladores utilizando a plataforma Android. No respectivo trabalho conduziu-se uma avaliação qualitativa quanto ao uso de controladores flexíveis. Assim, gerou-se a hipótese para realizar comparações entre diferentes configurações foi inicializada.
Contribuição para a tese: Observou-se a relevância do tema para se contribuir para a definição de abordagens envolvendo a flexilidade de controladores para SAs. A condução de tal trabalho também forneceu indícios de possíveis problemas que poderiam ser encarados na geração de resultados quantitativos – como por exemplo, as variações de resultados provenientes da comunicação distribuída.
Autoria e co-autoria: SIQUEIRA, BENTO R.; FERRARI, FABIANO C.; VOGEL, THOMAS; DE LEMOS, ROGÉRIO. As principais contribuições neste trabalho referem-se às implementações e adequações do sistema *PhoneAdapter* para condução de estudos exploratórios na plataforma Android para utilizar microcontroladores distribuídos e coordenados por um metacontrolador.

Trabalho Siqueira et al. (2021)

Título: Testing of adaptive and context-aware systems: approaches and challenges
Descrição: no trabalho Siqueira et al. (2021) conduziu-se uma Revisão Sistemática da Literatura sobre teste de Sistemas Adaptativos.
Contribuição para a tese: Tal trabalho contribuiu a fim de fornecer a literatura sobre teste de sistemas adaptativos. Mais especificamente, utilizaram-se os trabalhos no contexto do teste em *tempo de execução*, em particular aqueles trabalhos no contexto do teste de regressão. Tais trabalhos forneceram subsídios para identificar as características e modelagens para ser criar um microcontrolador de teste de regressão. Além disso, o mesmo também fornece demais trabalhos que podem ser utilizados para desdobramentos de outros microcontroladores envolvendo o teste de Sistemas Adaptativos.
Autoria e co-autoria: SIQUEIRA, BENTO R.; FERRARI, FABIANO C.; SOUZA, KATHIANI E.; CAMARGO, VALTER V. ; DE LEMOS, ROGÉRIO. As principais contribuições neste trabalho referem-se à condução do processo de Revisão Sistemática da Literatura, bem como analise e divulgação das caracterizações do estado da arte.

Trabalho Siqueira et al. (2020a)

Título: Faults Types of Adaptive and Context-Aware Systems and Their Relationship with Fault-based Testing Approaches

Descrição: no trabalho Siqueira et al. (2020a) caracterizou-se um conjunto de defeitos para sistemas adaptativos e relacionou-se a influência de tais defeitos com o *PhoneAdapter* utilizado no trabalho Siqueira et al. (2020b).
Contribuição para a tese: Uma vez que a caracterização de defeitos do trabalho Siqueira et al. (2020a) também utilizou o sistema *PhoneAdapter* como base de ilustração. O respectivo trabalho também auxiliou na investigação e estudos do controlador abordado no Capítulo 5 e no trabalho Siqueira et al. (2020b). Além disso, o desenvolvimento de um microcontrolador FailureManager utilizou como base de estudos os defeitos características no trabalho trabalho (Siqueira et al., 2020a).
Autoria e co-autoria: SIQUEIRA, BENTO R.; FERRARI, FABIANO C.; SOUZA, KATHIANI E.; SANTIBANEZ, DANIEL S. M.; CAMARGO, VALTER V. As principais contribuições neste trabalho referem-se à interpretação dos trabalhos da literatura, caracterização de defeitos e análise dos mesmos a fim de relacionar com o sistema alvo.

Trabalho Siqueira et al. (2018)

Título: Experimenting with a Multi-Approach Testing Strategy for Adaptive Systems
Descrição: no trabalho Siqueira et al. (2018) conduziram-se experimentos envolvendo a combinação de múltiplas abordagens de teste, com diferentes técnicas e critérios complementares.
Contribuição para a tese: Uma vez que no trabalho Siqueira et al. (2018) demonstraram-se resultados positivos quanto à combinação de diferentes abordagens de teste com o intuito de melhorar a qualidade dos testes realizados, no Capítulo 7 parte-se do princípio que um microcontrolador de teste pode ser definido ao se inspirar na cominação de abordagens de teste.
Autoria e co-autoria: SIQUEIRA, BENTO R.; JÚNIOR, MISAEL COSTA; FERRARI, FABIANO C.; SANTIBÁÑEZ, DANIEL S. M.; MENOTTI, RICARDO; CAMARGO, VALTER V. As principais contribuições neste trabalho referem-se às caracterizações do estado da arte em teste de SAs; combinação teórica das abordagens, bem como a condução de estudos experimentais conduzidos.

Trabalho Martín et al. (2020)

Título: Characterizing Architectural Drifts of Adaptive Systems
Descrição: no trabalho Martín et al. (2020) conduziu-se uma análise em diferentes códigos fonte de SAs a fim de caracterizar desvios arquiteturais. Assim, gerou-se um conjunto de

desvios existes para tais sistemas, contribuindo para literatura as características observadas e os respetivos desvios observados.
Contribuição para a tese: No trabalho Martín et al. (2020), as trocas de experiências em se analisar código fonte de SAs, bem como classificar implementações referindo-se a determinados componentes comuns de um modelo arquitetural, tal como o MAPE-K, auxiliaram no acúmulo de experiência para identificação e desenvolvimento de estágios baseados no MAPE-K para os microcontroladores.
Autoria e co-autoria: MARTIN, DANIEL SAN; **SIQUEIRA, BENTO** ; DE CAMARGO, VALTER VIEIRA; FERRARI, FABIANO. As principais contribuições neste trabalho referem-se às discussões, bem como trocas de experiências referente ao estado da arte e análise de códigos fonte de SAs.

Trabalho Serikawa et al. (2016)

Título: Towards the Characterization of Monitor Smells in Adaptive Systems
Descrição: no trabalho Serikawa et al. (2016) caracterizaram-se *Smells* arquiteturais presentes em SAs, particularmente no componente monitor, seguinte o modelo MAPE-K.
Contribuição para a tese: No trabalho Serikawa et al. (2016), o principal sistema utilizado foi o *PhoneAdapter*, que é o mesmo sistema ao qual se conduziu o estudo exploratório do Capitulo 5. Entre outras análises, a experiência em se analisar monitores contribuiu para as definições dos estágios baseados no MAPE-K e desenvolvimento de microcontroladores.
Autoria e co-autoria: SERIKAWA, MARCEL A.; LANDI, ANDRE DE S.; **SIQUEIRA, BENTO R.**; COSTA, RENATO S.; FERRARI, FABIANO C.; MENOTTI, RICARDO; CAMARGO, VALTER V. DE. As principais contribuições neste trabalho referem-se às discussões, bem como trocas de experiências referente ao estado da arte e análise de códigos fonte de SAs.

Trabalho Siqueira et al. (2016)

Título: Characterisation of Challenges for Testing of Adaptive Systems
Descrição: por meio de um Mapeamento Sistemático da literatura no trabalho Siqueira et al. (2016), caracterizaram-se os desafios encontrados na atividade de teste em SAs.
Contribuição para a tese: No trabalho Siqueira et al. (2016), os estudos e análise conduzidas para a caracterização do estado da arte contribuíram para obtenção de experiência para análise da literatura de SAs, destacando características, propriedades e abordagens para com esses sistemas.

Autoria e co-autoria: SIQUEIRA, BENTO RAFAEL; FERRARI, FABIANO CUTIGI; SERIKAWA, MARCEL AKIRA; MENOTTI, RICARDO; DE CAMARGO, VALTER VIEIRA. As principais contribuições neste trabalho referem-se à condução do processo de Revisão Sistemática da Literatura, bem como analise e divulgação das caracterizações do estado da arte.

8.4 Agradecimentos

Explicitam-se aqui os agradecimentos aos autores dos trabalhos que embasaram parcialmente o trabalho aqui relatado; em particular, Aderaldo et al. (2019) e Aderaldo e Mendonça (2022). Os autores contribuíram por meio de vídeo chamadas com explicações, seminários e dúvidas a respeito do *framework Kubow*. Além disso, os autores – Nabor Mendonça e Carlos Aderaldo, contribuíram com as correções de defeitos e sugestões encontradas durante o processo de preparação/desenvolvimento das configurações Mon-KZ, Des-KZ e Meta-KZ.

Referências

Aderaldo, C.; Mendonça, N. C.; Schmerl, B.; Garlan, D. Kubow: An architecture-based self-adaptation service for cloud native applications. In: *Proceedings of the 13th European Conference on Software Architecture (ECSA) – Posters, Tools, Demos Track*, Paris, France: ACM, 2019.

Aderaldo, C. M.; Mendonça, N. C. Kube-znn repository. Online, `https://github.com/ppgia-unifor/kubow/tree/master/samples/samples/kube-znn` - accessed in January, 2022, 2022.

Akour, M.; Akour, I.; King, T. M. Runtime test case synchronization in adaptive software. *International Journal of Business Research and Information Technology (IJBRIT)*, v. 1, n. 1, 2014.

Al-Refai, M.; Cazzola, W.; Ghosh, S. Model-based regression test selection for validating runtime adaptation of software systems. In: *Proceedings of the* 9^{th} *International Conference on Software Testing, Verification and Validation (ICST)*, Chicago, IL, USA: IEEE, 2016a.

Al-Refai, M.; Cazzola, W.; Ghosh, S.; France, R. Using models to validate unanticipated, fine-grained adaptations at runtime. In: *Proceedings of the* 17^{th} *International Symposium on High Assurance Systems Engineering (HASE)*, Orlando, Florida - USA: IEEE, 2016b.

Andersson, J.; de Lemos, R.; Malek, S.; Weyns, D. Reflecting on self-adaptive software systems. In: *Proceedings of the 4th Workshop on Software Engineering for Adaptive and Self-Managing Systems (SEAMS)*, Vancouver, BC, Canada: IEEE, 2009.

Android Arquitetura da plataforma android. Online, https://developer.android.com/guide/platform - acessado em Agosto em de 2022., 2022.

Banijamali, A.; Kuvaja, P.; Oivo, M.; Jamshidi, P. Kuksa*: Self-adaptive microservices in automotive systems. In: Morisio, M.; Torchiano, M.; Jedlitschka, A., eds. *Product-Focused Software Process Improvement*, Cham: Springer International Publishing, 2020.

Baylov, K.; Dimov, A. Reference architecture for self-adaptive microservice systems. In: Ivanović, M.; Bădică, C.; Dix, J.; Jovanović, Z.; Malgeri, M.; Savić, M., eds. *Intelligent Distributed Computing XI*, Belgrade, Serbia: Springer, 2017.

Braberman, V.; D'Ippolito, N.; Kramer, J.; Sykes, D.; Uchitel, S. Morph: A reference architecture for configuration and behaviour self-adaptation. In: *Proceedings of the 1st International Workshop on Control Theory for Software Engineering (CTSE)*, New York, NY, USA: ACM, 2015.

Cámara, J.; De Lemos, R.; Laranjeiro, N.; Ventura, R.; Vieira, M. Testing the Robustness of Controllers for Self-Adaptive Systems. *Journal of the Brazilian Comp. Society*, v. 20, n. 1, 2014.

Chan, W. K.; Chen, T. Y.; Lu, H.; Tse, T. H.; Yau, S. S. Integration testing of context-sensitive middleware-based applications: a metamorphic approach. *International Journal of Software Engineering and Knowledge Engineering*, v. 16, n. 5, 2006.

Cheng, S.; Garlan, D.; Schmerl, B. Evaluating the effectiveness of the rainbow self-adaptive system. In: *Proceedings of the 4th international workshop on software engineering for adaptive and self-managing systems (SEAMS 2009)*, Vancouver, 2009.

Cheng, S.-W.; Garlan, D. Stitch: A language for architecture-based self-adaptation. *Journal of Systems and Software (JSS)*, v. 85, n. 12, p. 2860–2875, 2012. Disponível em https://www.sciencedirect.com/science/article/pii/S0164121212000714

Cámara, J.; Correia, P.; de Lemos, R.; Garlan, D.; Gomes, P.; Schmerl, B.; Ventura, R. Evolving an adaptive industrial software system to use architecture-based self-adaptation. In: *Proceedings of the 8th International Symposium on Software Engineering for Adaptive and Self-Managing Systems (SEAMS)*, San Francisco, CA, USA: IEEE, 2013.

Da Silva, C. E.; De Lemos, R. Dynamic plans for integration testing of self-adaptive software systems. In: *Proceedings of the 6th International Symposium on Software Engineering for Adaptive and Self-Managing Systems (SEAMS)*, Waikiki, Honolulu, HI, USA: ACM Press, 2011.

Da Silva, C. E.; De Lemos, R. Dynamic management of integration testing for self-adaptive systems. In: *Proceedings of the Workshop on Dependable in Adaptive and Self-Managing Systems (WDAS)*, Rio de Janeiro – Brazil: KAR repository, 2013.

de la Iglesia, D. G.; Weyns, D. MAPE-K Formal Templates to Rigorously Design Behaviors for Self-Adaptive Systems. *ACM Transactions on Autonomous and Adaptive Systems (TAAS)*, v. 10, n. 3, 2015.

De Lemos, R.; Garlan, D.; Ghezzi, C.; Giese, H. *Software engineering for self-adaptive systems iii assurances*, v. 9640 LNCS. Dagstuhl Castle – Germany: Springer, 2017.

de Lemos, R.; Potena, P. Identifying and handling uncertainties in the feedback control loop. In: Mistrik, I.; Ali, N.; Kazman, R.; Grundy, J.; Schmerl, B., eds. *Managing Trade-Offs in Adaptable Software Architectures*, 1st ed, Boston: Science Direct, 2017.

De Lemos, R.; Potena, P. Identifying and Handling Uncertainties in the Feedback Control Loop. In: *Managing Trade-Offs in Adaptable Software Architectures*, n. i in 1, Elsevier, 2017.

Eberhardinger, B.; Ponsar, H.; Siegert, G.; Reif, W. Case Study: Adaptive Test Automation for Testing an Adaptive Hadoop Resource Manager. In: *Proceedings of the International Conference on Software Quality, Reliability and Security Companion (QRS-C)*, Lisbon – Portugal: IEEE, 2018.

Florio, L.; Di Nitto, E. Gru: An approach to introduce decentralized autonomic behavior in microservices architectures. In: *Proceedings of the 13th International Conference on Autonomic Computing (ICAC)*, Wurzburg, Germany: IEEE, 2016.

Fredericks, E. M. An Empirical Analysis of the Mutation Operator for Run-Time Adaptive Testing in Self-Adaptive Systems. In: *Proceedings of the 11th International Workshop on Search-Based Software Testing (SBST)*, Gothenburg – Sweden: IEEE, 2018.

Fredericks, E. M.; Cheng, B. H. C. Automated generation of adaptive test plans for self-adaptive systems. In: *Proceedings of the 10th International Symposium on Software Engineering for Adaptive and Self-Managing Systems (SEAMS)*, Florence - Italy: IEEE, 2015.

Fredericks, E. M.; DeVries, B.; Cheng, B. H. C. Towards run-time adaptation of test cases for self-adaptive systems in the face of uncertainty. In: *Proceedings of the* 9^{th} *International Symposium on Software Engineering for Adaptive and Self-Managing Systems (SEAMS)*, Hyderabad - India: IEEE, 2014.

Garlan, D.; Cheng, S.-W.; Huang, A.-C.; Schmerl, B. R.; Steenkiste, P. Rainbow: Architecture-Based Self-Adaptation with Reusable Infrastructure. *Computer*, v. 37, n. 10, 2004.

Garlan, D.; Monroe, R. T.; Wile, D. *Acme: Architectural description of component-based systems* USA: Cambridge University Press, p. 47–67, 2000.

Garvin, B. J.; Cohen, M. B.; Dwyer, M. B. Failure avoidance in configurable systems through feature locality. In: *Assurances for Self-Adaptive Systems*, Springer-Verlag Berlin Heidelberg: Springer, 2013.

Gerostathopoulos, I.; Skoda, D.; Plasil, F.; Bures, T.; Knauss, A. Tuning self-adaptation in cyber-physical systems through architectural homeostasis. *Journal of Systems and Software*, v. 148, 2019.

Giese, H.; et al. Architectural concepts for self-aware computing systems. In: Kounev, S.; Kephart, J. O.; Milenkoski, A.; Zhu, X., eds. *Self-Aware Computing Systems*, Cham: Springer, 2017.

Hansel, J.; Vogel, T.; Giese, H. A testing scheme for self-adaptive software systems with architectural runtime models. In: *Proceedings of the* 9^{th} *International Conference on Self-Adaptive and Self-Organizing Systems Workshops (SASOW)*, Cambridge, MA - USA: IEEE, 2015.

Hassan, S.; Bahsoon, R. Microservices and their design trade-offs: A self-adaptive roadmap. In: *Proceedings of the 13th IEEE International Conference on Services Computing (SCC)*, San Francisco, CA, USA: IEEE, 2016.

Horányi, G.; Micskei, Z.; Majzik, I. Scenario-based automated evaluation of test traces of autonomous systems. In: *Proceedings of the* 32^{nd} *International Conference*

on Computer Safety, Reliability and Security (SAFECOMP), Toulouse – France: Inria, 2013.

IBM *An architectural blueprint for autonomic computing.* Technical Report, IBM, 2005.

Iftikhar, M. U.; Weyns, D. Activforms: Active formal models for self-adaptation. In: *Proceedings of the 9th International Symposium on Software Engineering for Adaptive and Self-Managing Systems (SEAMS)*, Hyderabad, India: ACM Press, 2014.

Jaw, L. C.; Homan, D.; Crum, V.; Chou, W.; Keller, K.; Swearingen, K.; Smith, T. Model-based Approach to Validation and Verification of Flight Critical Software. In: *Proceedings of the* 29^{th} *International IEEE Aerospace Conference (AeroConf)*, Big Sky/Montana - USA: IEEE, 2008.

Kephart, J. O.; Chess, D. M. The vision of autonomic computing. *Computer*, v. 36, n. 1, 2003.

King, T. M.; Allen, A.; Wu, Y.; Clarke, P.; Ramirez, A. E. A comparative case study on the engineering of self-testable autonomic software. In: *Proceedings of the* 8^{th} *International Conference and Workshops on Engineering of Autonomic and Autonomous Systems (EASe)*, Los Alamitos/CA - USA: IEEE, 2011a.

King, T. M.; Allen, A. A.; Cruz, R.; Clarke, P. J. Safe runtime validation of behavioral adaptations in autonomic software. In: *Proceedings of the* 8^{th} *International Conference Autonomic and Trusted Computing (ATC)*, Banff - Canada: Springer Link, 2011b.

King, T. M.; Ramirez, A.; Cruz, R.; Clarke, P. An integrated self-testing framework for autonomic computing systems. *Journal of Computers*, v. 2, n. 9, 2007.

Krupitzer, C.; Roth, F. M.; Pfannemüller, M.; Becker, C. Comparison of approaches for self-improvement in self-adaptive systems. In: *Proceedings of the IEEE International Conference on Autonomic Computing (ICAC)*, Wurzburg, Germany: IEEE, 2016.

Kubernetes Production-grade container orchestration. Online, `https://kubernetes.io/` - accessed in January, 2022., 2022.

Lahami, M.; Krichen, M.; Barhoumi, H.; Jmaiel, M. Selective test generation approach for testing dynamic behavioral adaptations. In: *Proceedings of the* 27^{th}

International Conference on Testing Software and Systems (ICTSS), New York, NY - USA: Springer, 2015.

Lalanda, P.; McCann, J. A.; Diaconescu, A. *Autonomic computing.* 1st ed. Springer-Verlag London: Springer, 2013.

Lindvall, M.; Porter, A.; Magnusson, G.; Schulze, C. Metamorphic Model-Based Testing of Autonomous Systems. In: *Proceedings of the 2nd International Workshop on Metamorphic Testing (MET)*, Buenos Aires – Argentina, 2017.

Liu, Y. Phoneadapter. Online, `http://sccpu2.cse.ust.hk/afchecker/phoneadapter.html` - último acesso em 18/04/2014, 2013.

Luo, C.; Kuutila, M.; Klakegg, S.; Ferreira, D.; Flores, H.; Goncalves, J.; Mäntylä, M.; Kostakos, V. TestAWARE: A Laboratory-Oriented Testing Tool for Mobile Context-Aware Applications. *Proc. ACM Interact. Mob. Wearable Ubiquitous Technol.*, v. 1, n. 3, 2017.

Ma, T.; Ali, S.; Yue, T. Modeling foundations for executable model-based testing of self-healing cyber-physical systems. *Software & Systems Modeling*, v. 1, n. 1, 2018.

Martín, D. S.; Siqueira, B.; de Camargo, V. V.; Ferrari, F. Characterizing architectural drifts of adaptive systems. In: *Proceedings of the 27th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, 2020.

McKinley, P. K.; Sadjadi, S. M.; Kasten, E. P.; Cheng, B. H. C. Composing adaptive software. *Computer*, v. 37, n. 7, 2004.

Mendonça, N. C.; Garlan, D.; Schmerl, B. R.; Cámara, J. Generality vs. reusability in architecture-based self-adaptation: The case for self-adaptive microservices. In: *Proceedings of the 1st International Workshop on Architectural Knowledge for Self-adaptive Systems (AKSAS)*, Madrid, Spain: ACM Press, 2018.

Mendonça, N. C.; Jamshidi, P.; Garlan, D.; Pahl, C. *Developing self-adaptive microservice systems: Challenges and directions.* Technical Report, University of Fortaleza (Brazil); University of South Carolina (USA); Carnegie Mellon University (USA); Free University of Bozen-Bolzano (Italy), 2019.

Merdes, M.; Malaka, R.; Suliman, D.; Paech, B.; Brenner, D.; Atkinson, C. Ubiquitous rats: How resource-aware run-time tests can improve ubiquitous software

system. In: *Proceedings of the 6th International Workshop on Software Engineering and Middleware (SEM)*, New York/NY – USA: ACM, 2006.

Nallur, V.; Bahsoon, R. A decentralized self-adaptation mechanism for service-based applications in the cloud. *IEEE Transactions on Software Engineering*, v. 39, n. 5, 2013.

Niebuhr, D.; Rausch, A. A concept for dynamic wiring of components: Correctness in dynamic adaptive systems. In: *Proceedings of the 6th Workshop on Specification and Verification of Component-Based Systems (SAVCBS) - held in conjunction with ESEC/FSE*, Dubrovnik – Croatia: ACM, 2007.

Niebuhr, D.; Rausch, A.; Klein, C.; Reichmann, J.; Schmid, R. Achieving dependable component bindings in dynamic adaptive systems - a runtime testing approach. In: *Proceedings of the 3rd International Conference on Self-Adaptive and Self-Organizing Systems (SASO)*, San Francisco/CA - USA: IEEE, 2009.

Nii, H. P. Blackboard systems part two: Blackboard application systems. *AI Magazine*, v. 7, n. 3, 1986.

Pereira, J. D.; Silva, R.; Antunes, N.; Silva, J. L. M.; de França, B.; Moraes, R.; Vieira, M. A platform to enable self-adaptive cloud applications using trustworthiness properties. In: *Proceedings of the 15th International Symposium on Software Engineering for Adaptive and Self-Managing Systems (SEAMS)*, New York, NY, USA: ACM, 2020.

Qin, Y.; Xu, C.; Yu, P.; Lu, J. Sit: Sampling-based interactive testing for self-adaptive apps. *Journal of Systems and Software*, v. 120, 2016.

Ramirez, A. J.; Cheng, B. H. C. Design patterns for developing dynamically adaptive systems. In: *Proceedings of the 2010 ICSE Workshop on Software Engineering for Adaptive and Self-Managing Systems (SEAMS)*, ACM Press, 2010.

Reichstaller, A.; Gabor, T.; Knapp, A. Mutation-Based Test Suite Evolution for Self-Organizing Systems. In: *Proceedings of the International Symposium on Leveraging Applications of Formal Methods (ISoLA)*, Limassol – Cyprus: Springer International Publishing, 2018.

Reichstaller, A.; Knapp, A. Risk-Based Testing of Self-Adaptive Systems Using Run-Time Predictions. In: *Proceedings of the 12nd International Conference on Self-Adaptive and Self-Organizing Systems (SASO)*, Trento – Italy: IEEE, 2018.

Ross, D. T.; Schoman, K. E. Structured Analysis for Requirements Definition. *IEEE Transactions on Software Engineering*, v. SE-3, n. 1, 1977.

Rothermel, G.; Untch, R. H.; Harrold, M. J. Test case prioritization: an empirical study. In: *Proceedings of the International Conference on Software Maintenance – Software Maintenance for Business Change - 1999 (ICSM)*, 1999.

Salehie, M.; Tahvildari, L. Self-adaptive software: Landscape and research challenges. *ACM Trans. Auton. Adapt. Syst.*, v. 4, 2009.

Sama, M.; Elbaum, S.; Raimondi, F.; Rosenblum, D. S.; Wang, Z. Context-aware adaptive applications: Fault patterns and their automated identification. *IEEE Transactions on Software Engineering*, v. 36, n. 5, 2010.

Sama, M.; Rosenblum, D. S.; Wang, Z.; Elbaum, S. Model-based fault detection in context-aware adaptive applications. In: *Proceedings of the 16th International Symposium on Foundations of Software Engineering (FSE)*, Atlanta, GA, USA: ACM, 2008.

Sampaio Jr., A. R.; Rubin, J.; Beschastnikh, I.; Rosa, N. S. Improving microservice-based applications with runtime placement adaptation. *Journal of Internet Services and Applications*, v. 10, n. 4, 2019.

Schmerl, B.; Cámara, J.; Moreno, G. A.; Garlan, D.; Mellinger, A. *Architecture-based self-adaptation for moving target defense.* Technical Report CMU-ISR-14-109, Institute for Software Research, School of Computer Science, Carnegie Mellon University, Pittsburgh, PA, USA, 2014.

Serikawa, M. A.; Landi, A. d. S.; Siqueira, B. R.; Costa, R. S.; Ferrari, F. C.; Menotti, R.; De Camargo, V. V. Towards the characterization of monitor smells in adaptive systems. In: *Proceedings of the Brazilian Symposium on Software Components, Architectures and Reuse (SBCARS)*, 2016.

Sharifloo, A. M.; Metzger, A. Mcaas: Model checking in the cloud for assurances of adaptive systems. In: de Lemos, R.; Garlan, D.; Ghezzi, C.; Giese, H., eds. *Software Engineering for Self-Adaptive Systems III. Assurances*, Cham: Springer, 2013.

Shaw, M. "self-healing": Softening precision to avoid brittleness: Position paper for woss '02: Workshop on self-healing systems. In: *Proceedings of the 1st Workshop on Self-Healing Systems (WOSS)*, New York, NY, USA: ACM, 2002.

Siqueira, B. R.; Costa Júnior, M.; Ferrari, F. C.; Santibáñez, D. S. M.; Menotti, R.; Camargo, V. V. Experimenting with a Multi-Approach Testing Strategy for Adaptive Systems. In: *Proceedings of the* 17^{th} *Brazilian Symposium on Software Quality (SBQS)*, Curitiba, PR, Brazil: ACM, 2018.

Siqueira, B. R.; Ferrari, F. C.; Serikawa, M. A.; Menotti, R.; Camargo, V. V. Characterisation of Challenges for Testing of Adaptive Systems. In: *Proceedings of the* 1^{st} *Brazilian Symposium on Systematic and Automated Software Testing (SAST)*, Maringá, PR, Brazil: ACM, 2016.

Siqueira, B. R.; Ferrari, F. C.; Souza, K. E.; Camargo, V. V.; de Lemos, R. Testing of adaptive and context-aware systems: approaches and challenges. *Software Testing, Verification and Reliability*, v. 31, n. 7, e1772 stvr.1772, 2021. Disponível em `https://onlinelibrary.wiley.com/doi/abs/10.1002/stvr.1772`

Siqueira, B. R.; Ferrari, F. C.; Souza, K. E.; Santibáñez, D. S. M.; Camargo, V. V. Fault Types of Adaptive and Context-Aware Systems and Their Relationship with Fault-based Testing Approaches. In: *Proceedings of the* 15^{th} *International Workshop on Mutation Analysis (Mutation)*, Porto – Portugal: IEEE, 2020a.

Siqueira, B. R.; Ferrari, F. C.; Vogel, T.; de Lemos, R. Micro-controllers: Promoting Structurally Flexible Controllers in Self-Aware Computing Systems. In: *Proceedings of the 1st Workshop on Self-Aware Computing (SeAC)*, Washington DC, USA: IEEE, 2020b.

Vassev, E.; Hinchey, M.; Nixon, P. Automated test case generation of self-managing policies for nasa prototype missions developed with assl. In: *Proceedings of the* 4^{th} *International Symposium on Theoretical Aspects of Software Engineering (TASE)*, Taipei - Taiwan: IEEE, 2010.

Vogel, T.; Giese, H. Adaptation and abstract runtime models. In: *Proceedings of the 2010 ICSE Workshop on Software Engineering for Adaptive and Self-Managing Systems (SEAMS)*, Cape Town, South Africa: ACM Press, 2010.

Vogel, T.; Giese, H. A language for feedback loops in self-adaptive systems: Executable runtime megamodels. In: *Proceedings of the 7th International Symposium on Software Engineering for Adaptive and Self-Managing Systems (SEAMS)*, Zurich, Switzerland: IEEE, 2012.

Vogel, T.; Giese, H. Model-driven engineering of self-adaptive software with EUREMA. *ACM Transactions on Autonomous and Adaptive Systems (TAAS)*, v. 8, n. 4, 2014.

Weyns, D.; Schmerl, B.; Grassi, V.; Malek, S.; Mirandola, R.; Prehofer, C.; Wuttke, J.; Andersson, J.; Giese, H.; Göschka, K. M. On patterns for decentralized control in self-adaptive systems. In: de Lemos, R.; Giese, H.; Müller, H. A.; Shaw, M., eds. *Software Engineering for Self-Adaptive Systems II: International Seminar, Dagstuhl Castle, Germany, October 24-29, 2010 Revised Selected and Invited Papers*, Heidelberg, Germany: Springer, 2013.

Wohlin, C.; Runeson, P.; Höst, M.; Ohlsson, M. C.; Regnell, B.; Wesslén, A. *Experimentation in Software Engineering.* 1st ed. Springer: Springer, 2012.

Xu, C.; Cheung, S. C.; Ma, X.; C., C.; Lv, J. Adam: Identifying defects in context-aware adaptation. *Journal of Systems and Software*, v. 85, 2012.

Yoo, S.; Harman, M. Regression testing minimization, selection and prioritization: A survey. *Software Testing, Verification & Reliability*, v. 22, 2012.

Yu, L.; Tsai, W. T.; Perrone, G. Testing context-aware applications based on bigraphical modeling. *IEEE Transactions on Reliability*, v. 65, 2016.

Yuan, E.; Malek, S.; Schmerl, B.; Garlan, D.; Gennari, J. Architecture-based self-protecting software systems. In: *Proceedings of the 9th international ACM SIGSOFT Conference on Quality of Software Architectures (QoSA)*, Vancouver, BC, Canada: ACM Press, 2013.