Tópicos em Sistemas de Informação

Minicursos do XVI Simpósio Brasileiro de Sistemas de Informação (SBSI - 2020)

Organizadores

Valdemar Vicente Graciano Neto
Emilio de Camargo Francesquini
Flávio E. A. Horita
Carlos A. Kamienski

TÓPICOS EM SISTEMAS DE INFORMAÇÃO

Minicursos SBSI 2020

ISBN: 978-65-87003-21-4

Sociedade Brasileira da Computação

Porto Alegre

2020

TÓPICOS EM SISTEMAS DE INFORMAÇÃO

Minicursos SBSI 2020

ISBN: 978-65-87003-21-4

Sociedade Brasileira de Computação –SBC

CNPJ: 29.532.264/0001-78

Coordenação Geral

Flávio E. A. Horita
Carlos A. Kamienski

Coordenação do Comitê de Programa - Minicursos

Valdemar Vicente Graciano Neto
Emilio de Camargo Francesquini

Edição dos Anais

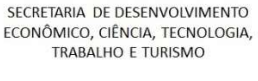
Davi Viana

Realização

Organização

Em cooperação

Apoiadores Institucionais



Fomento

Patrocinadores

Editores

Comitê técnico

Comissão Especial de Sistemas de Informação da SBC (CESI)

Comitê de Programa – Minicursos

Alex Borges (Universidade Federal de Juiz de Fora)
Alexandre Cidral (Universidade da Região de Joinville)
André de Oliveira (Universidade Federal de Juiz de Fora)
André Freire (Universidade Federal de Lavras)
Andre Martinotto (Universidade de Caxias do Sul)
Carla Merkle Westphall (Universidade Federal de Santa Catarina)
Carlos Eduardo Santos Pires (Universidade Federal de Campina Grande)
Carlos Eduardo de Barros Paes (Pontifícia Universidade Católica de São Paulo)
Claudia Cappelli (Universidade do Estado do Rio de Janeiro)
Clodoaldo Lima (Universidade de São Paulo)
Daniel Notari (Universidade Caxias do Sul)
Daniela Barreiro Claro (Universidade Federal da Bahia)
Davi Viana (Universidade Federal do Maranhão)
Edmundo Spoto (Universidade Federal de Goiás)
Eliomar Lima (Universidade Federal de Goiás)
Emanuel Coutinho (Universidade Federal do Ceará)
Fernanda Baião (Pontifícia Universidade Católica do Rio de Janeiro)
Flávio Soares Corrêa da Silva (Universidade de São Paulo)
Glauco Carneiro (Universidade Salvador)
Heitor Costa (Universidade Federal de Lavras)
Jorge Barbosa (Unisinos)
José David (Universidade Federal de Juiz de Fora)
Juliano Lopes de Oliveira (Universidade Federal de Goiás)
Leonardo Azevedo (IBM Research Brazil)
Leticia Peres (Universidade Federal do Paraná)
Marcos Chaim (Universidade de São Paulo)
Maria Istela Cagnin (Universidade Federal de Mato Grosso do Sul)
Morganna Diniz (Universidade Federal do Estado do Rio de Janeiro)
Patricia Vilain (Universidade Federal de Santa Catarina)
Paulo Sérgio Santos (Universidade Federal do Estado do Rio de Janeiro)
Renata Araujo (Universidade Presbiteriana Mackenzie)
Ricardo Choren (Instituto Militar de Engenharia do Rio)
Rita Suzana Pitangueira Maciel (Universidade Federal da Bahia)
Roberto Pereira (Universidade Federal do Paraná)
Rodrigo Santos (Universidade Federal do Estado do Rio de Janeiro)
Scheila de Avila e Silva (Universidade de Caxias do Sul)
Victor Stroele (Universidade Federal de Juiz de Fora)
Vladimir Rocha (Universidade Federal do ABC)

S612a Simpósio Brasileiro de Sistemas de Informação: Tópicos em Sistema de

Informação (16. : 2020 : São Bernardo do Campo, SP).

Anais do 16º Simpósio Brasileiro de Sistemas de Informação: Tópicos em Sistema de Informação [recurso eletrônico] / XVI Simpósio Brasileiro de Sistemas de Informação: Tópicos em Sistema de Informação, 3-6 novembro 2020. São Bernardo do Campo, Brasil ; organizado por Valdemar Vicente Graciano Neto [et al.]. – Porto Alegre : Sociedade Brasileira da Computação, 2020.

Disponível em: https://sol.sbc.org.br/livros/

ISBN: 978-65-87003-21-4

1 . Sistemas de Informação – Congressos. I. Valdemar Vicente Graciano Neto. II. Emílio de Camargo Francesquini. III. Flávio E. A. Horita. IV. Carlos A. Kamienski. V. Título.

CDU: 004.65

Prefácio Minicursos SBSI 2020

Este livro reúne trabalhos apresentados nos minicursos ministrados no XVI Simpósio Brasileiro de Sistemas de Informação (SBSI), realizado online e organizado pela Universidade Federal do ABC, no período de 03 a 06 de novembro de 2020. Participam do SBSI, fórum nacional de debates da área de Sistemas de Informação (SI), estudantes e pesquisadores com apresentação de trabalhos científicos e discussão de temas contemporâneos relacionados à área.

Neste ano, foram submetidas 5 propostas de minicursos válidas e duas foram selecionadas, tal qual decidido na última reunião da Comissão Especial de Sistemas de Informação. Tais propostas foram avaliadas por, no mínimo, três pesquisadores que fazem parte de um comitê composto por 38 professores pesquisadores.

Os dois minicursos contidos neste livro abordam tópicos de interesse da comunidade de Sistemas de Informação e correlatos ao tema da 16a. Edição do evento, intitulado "Sistemas de Informação na Transformação e Inovação Digital". O primeiro capítulo, "Utilizando Ciência de Redes no Desenvolvimento de Sistemas Complexos", ensina como o contexto dos sistemas complexos afeta o desenvolvimento das aplicações contemporâneas e como modelos em ciência de redes podem ser utilizados para melhor compreender a estrutura e características desses sistemas. O segundo capítulo, "Desenvolvimento de Soluções com Serviços: SOA, Cloud e Microsserviços", discute a sinergia entre os conceitos de Arquitetura Orientada a Serviços (SOA), Computação em Nuvem (Cloud) e Arquitetura de Microsserviços (MSA) no contexto do desenvolvimento de software de Sistemas de Informação, mostrando como aplicar estes paradigmas no desenvolvimento de aplicações distribuídas para sistemas de informação.

Esperamos que este livro auxilie estudantes, pesquisadores e profissionais da área de Sistemas de Informação na construção do conhecimento em temas específicos relacionados ao que foi aqui apresentado.

Valdemar Vicente Graciano Neto (UFG) e Emilio Francesquini (UFABC)
Coordenadores da Trilha de Minicursos do SBSI 2020
São Bernardo do Campo/SP, Novembro de 2020.

Coordenadores dos Minicursos do SBSI 2020

Valdemar Vicente Graciano Neto é professor adjunto no Instituto de Informática da Universidade Federal de Goiás, em Goiânia desde 2014. Ele recebeu seu título de doutor em dupla titulação em Ciências da Computação e Matemática Computacional pela Universidade de São Paulo, São Carlos, Brasil e em Informática pelo IRISA Labs na Universidade da Bretanha Sul, em Vannes, França (2018). Ele também é bacharel e mestre em Ciências da Computação pela Universidade Federal de Goiás (2009 e 2012, respectivamente). Ele já publicou mais de 80 estudos em conferências e periódicos arbitrados e capítulos de livros. Valdemar foi o coordenador do Comitê Especial de Sistemas de Informação (CESI) da Sociedade Brasileira de Computação (SBC) no período 2018-2019, tendo sido também membro eleito no período 2015-2017. Foi organizador geral do Simpósio Brasileiro de Sistemas de Informação (SBSI) em 2015 em Goiânia e já organizou outros três workshops nacionais e internacionais. Ele é membro de comunidades científicas, como a Sociedade Brasileira de Computação (2009-2019) e The Society For Modeling and Simulation International (2019-atual). Seus interesses de pesquisa incluem sistemas de informação, cidades inteligentes, sistemas de sistemas e sistemas de sistemas de informação, arquitetura de software e avanços em modelagem e simulação para engenharia de software.

Emilio Francesquini é professor adjunto no Centro de Matemática, Computação e Cognição da Universidade Federal do ABC, em Santo André desde 2018. Ele recebeu o seu título de doutor (em regime de dupla titulação) da Universidade de São Paulo e da Universidade de Grenoble-Alpes (França) em 2014. Ele também é mestre (2007) e bacharel (2003) em Ciência da Computação pela Universidade de São Paulo. Foi organizador de diversos eventos da CE-ACPAD como (SBAC-PAD 2017, WSCAD 2017, ERAD-SP 2018, 2019), é o coordenador de programa da ERAD-SP 2020 além de ser o coordenador geral do WSCAD 2020, WPerformance 2020 (evento associado à SBC) e ERAD-SP 2021. Seus interesses de pesquisa incluem programação de alto desempenho, arquitetura de computadores (em especial tecnologias de memória não volátil) e programação funcional paralela.

Organizadores Gerais do SBSI 2020

Flávio Eduardo Aoki Horita é Professor Adjunto A no Centro de Matemática, Computação e Cognição (CMCC) da Universidade Federal do ABC (UFABC) em Santo André/SP, Brasil. Pesquisador Visitante no Warwick Business School da Universidade de Warwick, Coventry, Inglaterra. Pesquisador Associado no Núcleo de Pesquisa NUVEM/UFABC, AGORA/USP e LabGRIS/UFABC. Doutor em Ciência da Computação e Matemática Computacional pelo Instituto de Computação e Matemática Computacional (ICMC) da Universidade de São Paulo (USP/São Carlos) tendo realizado estágio-sanduíche no Departamento de Sistemas de Informação e Supply Chain Management (Prof.-Ing. Bernd Hellingrath) do European Research Center for Information Systems (ERCIS) na Universidade de Münster em Münster na Alemanha. Mestre em Ciência da Computação pela Universidade Estadual de Londrina (UEL) (2013), Especialista em Engenharia de Software com UML pelo Centro Universitário Filadélfia (2011) e Bacharel em Sistemas de Informação pelo Centro Universitário de Lins (2008). Coordenador e pesquisador em diversos projetos de pesquisa financiados por agências nacionais, como o CNPq (Universal), CAPES (Pró-alertas) e FAPESP (FAPESP-IVA, entre outros), e internacionais, como o EPSRC (UK-Brazil). Atuou como pesquisador PCI no Centro Nacional de Monitoramento e Alerta de Desastres Naturais (CEMADEN-MCTIC) e pesquisador técnico no Departamento de Pesquisa, Desenvolvimento & Inovação da Agência Brasileira de Meteorologia (CLIMATEMPO). Membro da Sociedade Brasileira de Computação (SBC), International Association for Mathematical Geosciences (IAMG), Association of Information Systems (AIS) e Scrum Alliance. Certificação de Scrum Master (CSM). Tem experiência em áreas de Engenharia de Software, Sistemas de Informação e Geoinformação com foco em Sistemas de Apoio à Tomada de Decisão, Sistemas de Informações Geográficas, Modelagem de Processos de Negócios e Decisão, Gestão de Desastres, Arquiteturas de Softwares e Interoperabilidade de Sistemas. A relevância dos resultados de pesquisas obtidos ao longo da carreira acadêmica são evidenciados pelas publicações tanto em conferências de impacto, como AMCIS, HICSS e ISCRAM, quanto em periódicos reconhecidos internacionalmente, como Decision Support Systems (IF 2.604), Computers & Geosciences (IF 2.474) e International Journal of Disaster Risk Reduction (IF 1.608). Coordenador de trilhas em conferências internacionais (como AMCIS, ISCRAM) e coordenador geral de evento (SBSI 2020).

Carlos Alberto Kamienski possui graduação em Ciência da Computação pela Universidade Federal de Santa Catarina (1989), mestrado em Ciência da Computação pela Universidade Estadual de Campinas (1994) e doutorado em Ciências da Computação pela Universidade Federal de Pernambuco (2003). Atualmente é Professor Titular Livre da Universidade Federal do ABC (UFABC) onde atua desde 2006. Foi professor de educação profissional técnica e tecnológica por 10 anos. Seus interesses atuais de pesquisa incluem Internet das Coisas, computação em nuvem, softwarização de redes, redes sociais, cidades inteligentes e agricultura inteligente. Foi coordenador da Pós-Graduação em Engenharia da Informação da UFABC de junho de 2008 a fevereiro de 2010. Foi Pró-Reitor de Pós-Graduação de fevereiro de 2010 a fevereiro de 2014. Foi Assessor de Relações Internacionais de fevereiro de 2014 a março de 2018. É coordenador do Núcleo Estratégico NUVEM (Universos Virtuais, Entretenimento e Mobilidade) da UFABC desde dezembro de 2013.

Sumário

Capítulo 1

Utilizando Ciência de Redes no Desenvolvimento de Sistemas Complexos

Rodrigo Pereira dos Santos, Jefferson Elbert Simões, Everton Cavalcante

Resumo

Sistemas complexos são caracterizados como tal em razão dos diversos recursos envolvidos em seu ciclo de vida, bem como da interação com outros sistemas, atores e artefatos. Essas características afetam decisões de projeto, desenvolvimento, operação, governança e evolução desses sistemas, requerendo melhor compreensão dos relacionamentos entre seus elementos. Além disso, a demanda por gerir uma troca de volumes cada vez maiores de dados e a necessidade de se fazer uso eficiente de recursos computacionais e energéticos limitados são fatores críticos para esses sistemas complexos. Nessa perspectiva, este capítulo introduz como o contexto dos sistemas complexos afeta o desenvolvimento das aplicações contemporâneas e como modelos em ciência de redes podem ser utilizados para melhor compreender a estrutura e características desses sistemas. De forma mais específica, por meio do estudo de duas classes de sistemas complexos (ecossistemas de software e sistemas de sistemas), este capítulo busca contribuir para: (i) compreender as características desses sistemas, que inclusive refletem-se em alguns sistemas atuais; (ii) apresentar a área de ciência de redes, em termos de modelos, ferramentas e mecanismos de análise; (iii) avaliar problemas nesse contexto, e; (iv) identificar desafios e oportunidades de pesquisa.

Abstract

Complex systems are characterized as such due to the many resources involved in their life cycle, as well as their interactions with other systems, actors, and artifacts. These characteristics affect decisions on to the design, operation, management, governance, and evolution of such systems, thus requiring a a better understanding of of the relationships among their elements. Furthermore, the demand for managing the increasing exchange of large volumes of data and the need for efficient use of limited computational and energy resources are critical factors for these complex systems. This chapter introduces how the context of complex systems affects the development of

contemporary applications and how network science models can be used to better understand the structure and characteristics of these systems. More specifically, by studying two classes of complex systems (software ecosystems and systems-of-systems), this chapter seeks to contribute to: (i) understand the characteristics of these systems, which indeed are reflected into some of today's systems; (ii) present the field of network science in terms of models, tools, and analysis mechanisms; (iii) assess problems in this context; and (iv) identify research challenges and opportunities.

1.1. Introdução

Sistemas complexos exercem um papel importante na vida cotidiana, ciência e economia, nos mais variados domínios de aplicação [Graciano Neto *et al*. 2017b]. Tal complexidade advém dos diversos recursos (*hardware*, *software*, humanos, técnicos) e processos (processamento, monitoramento, gestão, comunicação etc.) envolvidos no ciclo de vida desses sistemas, bem como da interação com outros sistemas, atores e artefatos [Santos 2016]. Essas características sem dúvida afetam decisões de projeto, desenvolvimento, operação, governança e evolução, requerendo melhor compreensão dos relacionamentos entre os elementos que os compõem [Jansen *et al*. 2013].

Exemplos desses tipos de sistemas são os **ecossistemas de software** (ECOS) e os **sistemas de sistemas** (SoS), ambos objetos de pesquisa e desenvolvimento nos últimos anos, na academia e na indústria [Santos e Werner 2011a, Nielsen *et al*. 2015]. Um ECOS consiste de um sistema complexo formado por atores (empresas, desenvolvedores internos e externos e usuários) e artefatos que interagem com um mercado de *software* cujas relações são apoiadas por uma plataforma tecnológica e realizadas pela troca de informação, recursos e artefatos (*e.g*., SAP, Eclipse e Android) [Jansen *et al*. 2009]. Por sua vez, um SoS pode ser entendido como sistema complexo que tem como constituintes outros sistemas heterogêneos e independentes, cada um deles com seu(s) próprio(s) objetivo(s) e colaborando entre si para satisfazer objetivos maiores e globais (*e.g*., suporte a desastres e cidades inteligentes) [Maier, 1998].

O objetivo deste capítulo é apresentar como o contexto dos sistemas complexos afeta o desenvolvimento das aplicações contemporâneas e como modelos em **ciência de redes** podem ser utilizados para melhor compreender a estrutura e características desses sistemas. A ciência de redes visa estudar entidades físicas e/ou virtuais e os relacionamentos entre elas utilizando teorias e métodos advindos de múltiplas áreas de conhecimento, dentre elas Matemática, Física, Computação, Estatística e Sociologia. De forma mais específica, por meio do estudo de duas classes de sistemas complexos (ECOS e SoS[1]), este capítulo busca contribuir para (i) compreender as características desses sistemas, que inclusive se refletem em alguns sistemas atuais, (ii) conhecer a área de ciência de redes, em termos de modelos, ferramentas e mecanismos de análise, (iii) avaliar problemas existentes nesse contexto e (iv) identificar desafios e oportunidades. A temática abordada possui significativa relação com tópicos relevantes no âmbito da comunidade da área de Sistemas de Informação, a exemplo de sistemas de sistemas de informação e complexidade de sistemas de informação.

[1] Neste capítulo, as abreviações ECOS e SoS serão utilizadas tanto para o singular quanto para o plural: *ecossistema(s) de software* e *sistema(s) de sistemas*, respectivamente.

Além desta introdução, este capítulo contém as seguintes seções. A Seção 1.2 discorre sobre os fundamentos acerca dos conceitos de SoS, ECOS e ciência de redes. A Seção 1.3 explora alguns exemplos de como a ciência de redes pode ser aplicada na análise de SoS e ECOS. A Seção 1.4 traz alguns desafios e oportunidades de pesquisa neste contexto. Por fim, a Seção 1.5 conclui o capítulo com as considerações finais.

1.2. Sistemas Complexos e Ciência de Redes

Esta seção tem por objetivo apresentar as principais características de SoS e ECOS, bem como as bases teóricas da ciência de redes, seus fenômenos e principais ferramentas de modelagem e análise.

1.2.1. Sistemas de Sistemas (SoS)

Tanto sistemas tradicionais quanto SoS são sistemas que integram componentes a fim de prover funcionalidades e satisfazer missões. No entanto, SoS distinguem-se de outros sistemas complexos e de larga escala devido a um conjunto de características inerentes a essa classe de sistemas, **todas** elas devendo ser satisfeitas para se considerar um sistema como um SoS [Maier 1998, Firesmith 2010]. Essas características, descritas brevemente adiante, são as que tornam SoS distintos dos sistemas tradicionais, requerendo assim uma mudança de paradigma no desenvolvimento desses sistemas. Com efeito, pesquisas na área já reconheceram que métodos existentes na Engenharia de Sistemas possuem aplicabilidade limitada ou não se aplicam a SoS, fazendo com que o desenvolvimento dessa classe de sistemas represente um desafio significativo [Lana *et al*. 2016, Azani 2019].

Independência operacional dos sistemas constituintes. Sistemas constituintes em um SoS são operacionalmente independentes no sentido de que proveem suas próprias funcionalidades e satisfazem suas próprias missões mesmo quando não estão em cooperação com outros sistemas no escopo do SoS. Essa característica não se observa em sistemas tradicionais, uma vez que seus componentes constituintes operam conforme foram designados para prover suas funcionalidades.

Independência gerencial dos sistemas constituintes. Sistemas constituintes que participam de um SoS são frequentemente desenvolvidos por diferentes organizações, com seus próprios *stakeholders*, equipes de desenvolvimento, processos e recursos, além de apresentarem um ciclo de vida próprio e terem autonomia para gerenciar seus próprios recursos. Esse não é o caso de sistemas tradicionais, em que seus componentes constituintes estão atrelados a ele e todas as decisões são feitas sobre o sistema, de maneira centralizada.

Distribuição geográfica dos sistemas constituintes. Esta característica refere-se à dispersão dos sistemas constituintes, sendo necessária alguma forma de conectividade para permitir comunicação e troca de informações [Nielsen *et al*. 2015]. Assim, sistemas constituintes de um SoS estão fisicamente desacoplados e trocam apenas informações entre si.

Desenvolvimento evolucionário do SoS. Um SoS pode evoluir constantemente em resposta a mudanças, sejam estas relacionadas ao ambiente, aos seus sistemas constituintes, às suas funcionalidades ou às suas missões. Devido a sua independência gerencial, os sistemas constituintes podem evoluir por conta própria sem o controle do

SoS, que, por sua vez, deve absorver tais mudanças e minimizar possíveis efeitos colaterais indesejáveis.

Comportamentos emergentes no SoS. Em sistemas tradicionais, comportamentos e interações dos seus componentes constituintes são em sua grande maioria previsíveis e controláveis da perspectiva do sistema. No entanto, mesmo que os sistemas constituintes de um SoS sejam previsíveis, suas independências operacional e gerencial fazem com que o comportamento do SoS emerja em tempo de execução. Isso se coloca como um desafio significativo ao desenvolvimento dessa classe de sistemas, uma vez que SoS dependem fortemente de comportamentos emergentes para satisfazer suas missões e prover novas funcionalidades [Inocêncio *et al*. 2019]. Tais funcionalidades resultam da colaboração entre os sistemas constituintes e não podem ser providas por nenhum deles se considerados de forma isolada.

Por fim, uma característica adicional de SoS é a sua **inerente dinamicidade**, que faz com que o sistema esteja constantemente sujeito a mudanças, isto é, seus sistemas constituintes podem ser integrados, operados e reconfigurados em tempo de execução, na maioria das vezes de forma não planejada. Os sistemas constituintes concretos a integrarem um SoS em tempo de execução frequentemente são conhecidos de forma parcial ou mesmo desconhecidos quando do projeto desse SoS. Com isso, os sistemas constituintes devem ser descobertos, selecionados e integrados em tempo de execução para que sejam identificados os arranjos adequados desses sistemas que contribuem para alcançar os objetivos gerais do SoS com base em suas capacidades [Gomes *et al*. 2015].

1.2.2. Ecossistemas de Software (ECOS)

Para apoiar a globalização da indústria de *software*, avanços na criação de métodos para desenvolver sistemas de grande porte, de longo prazo, interconectados e sujeitos a rápida velocidade de implantação e evolução têm sido empreendidos [Bosch 2016]. Esse cenário requer profissionais com a habilidade de abstrair a complexidade do sistema como um todo e entender que um sistema é resultado da combinação entre *software*, *hardware* e *peopleware*, constituída sobre uma plataforma comum [Messerschmitt e Szyperski 2003, Santos *et al*. 2012], levando aos ECOS. Grandes empresas como Amazon, Apple, Google, Microsoft e SAP lideram o desenvolvimento de ECOS, o que contribuiu para a crescente pesquisa no tema [Manikas e Hansen 2013, Manikas 2016].

No mercado de *software* atual, empresas não trabalham mais como unidades independentes ou entregam produtos distintos, mais sim dependem de outros fornecedores de componentes e de infraestruturas vitais, tais como sistemas operacionais, bibliotecas, lojas de componentes e plataformas [Bosch e Olsson 2018]. Nesse sentido, ECOS traz uma visão holística, uma vez que a participação de atores externos, sejam eles conhecidos ou não, passou a fazer parte da rotina das empresas, gerando uma nova maneira de estruturar modelos de negócio e de desenvolvimento de *software* [Yang *et al*. 2017]. Os estudos relativos a esse contexto visam ainda explicar novos elementos que estão atrelados ao *software*. Por exemplo, ao invés de gerir a variabilidade dos produtos e serviços de *software* oferecidos aos seus clientes, as empresas passam a ter de prover formas para que eles customizem as suas próprias soluções segundo a necessidade/contexto (*e.g*., ECOS móveis) [Fontão *et al*. 2017].

Entre as definições de ECOS, há três principais. Jansen *et al*. (2009) definem ECOS como um conjunto de atores funcionando como uma unidade que interage com um

mercado distribuído entre *software* e serviços, juntamente com as suas relações frequentemente apoiadas por uma plataforma tecnológica ou por um mercado comum e realizadas pela troca de informação, recursos e artefatos. Bosch (2009) define ECOS como um conjunto de soluções de *software* que possibilitam, apoiam e automatizam as atividades e transações feitas por atores em um ecossistema social ou de negócios e por organizações que provêm soluções. Lungu *et al*. (2010) definem um ECOS como uma coleção de projetos de *software* que são desenvolvidos e evoluem em conjunto em um mesmo ambiente.

Para van den Berk *et al*. (2010), um ECOS possui basicamente três elementos: **centralizador** (*hub*), **plataforma** (tecnologia ou mercado) e **agentes de nicho** (*niche players*). Nesse contexto, o centralizador é o dono da plataforma e os agentes de nicho são os envolvidos que podem utilizá-la para gerar valor para si e para o ecossistema. Por exemplo, no caso do ECOS Windows, a Microsoft é o centralizador, o Windows é a plataforma e as demais empresas, desenvolvedores externos e usuários são os agentes de nicho, que usam a plataforma para construir/usar suas aplicações. Um problema que aparece neste contexto é a existência de muitas empresas e relacionamentos, que envolvem fornecedores, desenvolvedores externos e usuários, o que pode dificultar decisões no desenvolvimento da plataforma do ECOS [Serebrenik e Mens 2015].

Um dos benefícios para uma empresa ou desenvolvedor se tornar um membro de um ECOS está na oportunidade de explorar a inovação aberta para que atores colaborem a fim de alcançar benefícios regionais e globais. O esforço dos atores externos em desenvolver para uma plataforma de ECOS pode transformar a inovação em algo positivo para si e seus clientes, contribuindo também para o centralizador por estender/melhorar a plataforma e engajar mais envolvidos [Fontão *et al*. 2018]. A proximidade entre o centralizador e os atores externos é fundamental para o sucesso de um empreendimento de *software* em ECOS. Por outro lado, entre as dificuldades enfrentadas neste contexto, estão: (i) estabelecer e modelar relacionamentos entre os atores de um ECOS; (ii) definir estratégias de negócio; (iii) manter arquitetura da plataforma; (iv) monitorar a evolução do ECOS; (v) romper obstáculos de comunicação de requisitos em projetos distribuídos geograficamente, e; (vi) construir ferramentas para acelerar a interação social e melhorar o processo de tomada de decisão [Barbosa *et al*. 2013].

Algumas classificações foram desenvolvidas pela comunidade acadêmica visando facilitar a compreensão do conceito de ECOS a partir da modelagem e análise de seus elementos constituintes. Uma estratégia para se modelar e analisar ECOS foi introduzida por Campbell e Ahmed (2010) e explorada em detalhes por Santos e Werner (2011a, 2011b, 2012a, 2012b). Os autores apresentam uma visão de ECOS em três dimensões que visam entender as funções assumidas por cada um dos pilares de um ECOS (Figura 1.1):

- **Arquitetura** (ou dimensão técnica): envolve a plataforma (infraestrutura com tecnologias e aplicações) na qual o ecossistema vai estar inserido e explora questões da arquitetura de *software*, linha de produto e processos;
- **Negócio** (ou dimensão transacional): envolve conhecimento do mercado, de decisões que os atores devem tomar sobre modelos de negócio, de definição do portfólio de produtos do ECOS, de estratégias de licenças e de vendas;
- **Social**: envolve explorar como a rede de atores se relaciona dentro do ECOS para atingir os seus objetivos, bem como fomentar o crescimento do ECOS por meio de proposições de valor em que todos possam ser beneficiados.

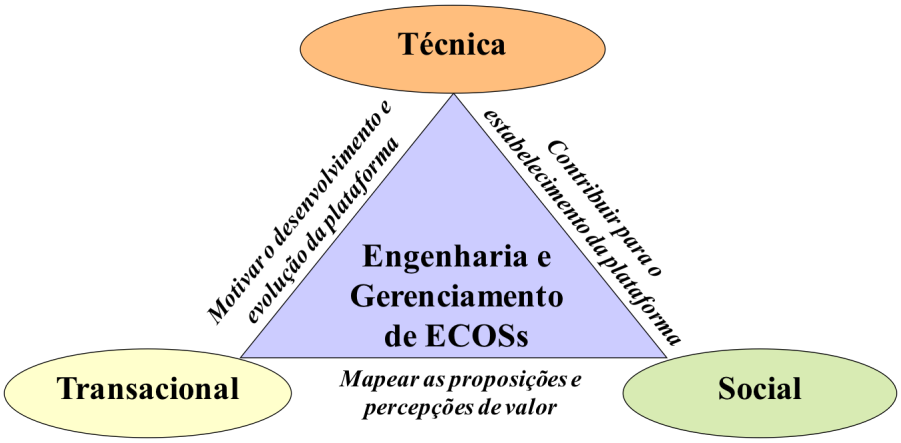


Figura 1.1: Visão de ECOS em três dimensões.
Fonte: Santos e Werner (2011a, 2011b, 2012a, 2012b)

Vale ressaltar ainda que há um número significativo de estudos que investigam ECOS centrados em plataformas do tipo *free open source software* (FOSS), seja pela disponibilidade de dados ou pela importância da colaboração entre os atores como elemento fortalecedor da plataforma [Franco-Bedoya *et al*. 2017]. Além disso, existem ECOS centrados em uma plataforma de ativos de *software*, representando uma linha de produtos de *software* [McGregor 2012], ou ainda voltados para um mercado de *software* e serviços de Tecnologia da Informação, formando um ecossistema de negócio para processos de inovação (aberta ou fechada). Alguns elementos comumente modelados e analisados em um ECOS são: (i) a arquitetura da plataforma, em termos de estabilidade, gerência de evolução, extensibilidade, segurança e confiabilidade [Amorim *et al*. 2016]; (ii) processos de negociação de requisitos, sendo necessário alinhar necessidades reais com soluções, componentes e portfólios [Fotrousi *et al*. 2015]; (iii) desenvolvimento colaborativo [van der Maas 2016]; (iv) modelos e métricas para redes de produção de *software* [Santos e Oliveira 2013], e; (v) modelos de negócio, como o de ECOS formado por organizações de pequeno e médio porte [Valença *et al*. 2018].

Dado que o contexto de ECOS reúne um conjunto de problemas em aberto para os diferentes domínios de sistemas, modelos e análises de casos reais têm utilizado recursos como UML, *i** e grafos [Coutinho *et al*. 2017]. É fundamental assim modelar elementos de arquitetura de modo que esta deva ser estendida para tratar gerenciamento, regras de negócio e restrições, integração e existência de múltiplas funcionalidades, seja para a plataforma central ou para as aplicações construídas sobre ela. Segurança e confiabilidade são atributos de qualidade críticos para equilibrar modularidade e flexibilidade na especificação da arquitetura da plataforma, focando em interoperabilidade e manutenção de interfaces de programação [Motta *et al*. 2017]. Além disso, inconsistências geradas por evoluções devem ser verificadas/validadas por meio das dependências de componentes e serviços da plataforma, o que requer mecanismos de coordenação [Graciano Neto *et al*. 2017a]. A constante evolução no contexto de um ECOS demanda processos adaptáveis para que o desenvolvimento tenha interoperabilidade, implantação e liberação de versões estáveis da plataforma. Como consequência, a análise e o projeto arquitetural passam a enfatizar estratégias para identificação de metas de negócio e de requisitos emergentes, além de táticas para

avaliação arquitetural. O processo de elicitação de requisitos torna-se um desafio, dado que novos atores se juntam àqueles comumente envolvidos no processo de desenvolvimento de *software*, criando um corpo numeroso e distribuído no ECOS [Silva *et al*. 2017]. Por fim, cadeias de valor e de resultados passam a apoiar a engenharia de requisitos [Yu e Deng 2011], além da modelagem, análise e visualização de redes sociais, técnicas e sociotécnicas para tratar redes de influência e interoperabilidade.

1.2.3. Ciência de Redes

Nos anos 1930, Jacob Levy Moreno apresentou uma série de representações de redes sociais, hoje conhecidas como **sociogramas**. Nessas representações, Moreno captura as relações sociais entre alunas de diversas turmas da New York Training School for Girls através de diagramas [Moreno 1934], como mostra a Figura 1.2. Nascia aqui a disciplina de análise de redes sociais, um ramo da Sociologia que visa realizar estudos quantitativos sobre redes sociais tendo como informação de base o padrão de conexões entre as pessoas que compõem tais redes.

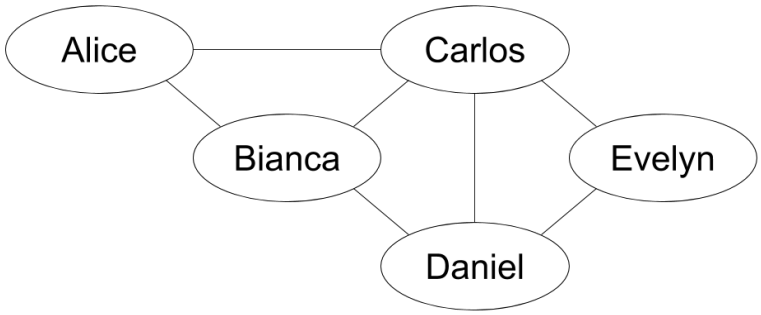


Figura 1.2: Exemplo de sociograma representando os relacionamentos de amizade entre cinco pessoas que constituem uma rede social. Neste exemplo, a rede apresentada possui sete relacionamentos no total.

Sociogramas implicitamente compreendem as relações sociais representadas como relações par-a-par, isto é, relações cuja existência ou ausência é observada entre pares de pessoas[2]. Assim como redes sociais, muitos outros sistemas podem ser intuitivamente vistos como uma combinação complexa de relacionamentos par-a-par. Um sistema de transporte metroviário pode ser visto como um conjunto de estações conectadas sucessivamente pelas linhas de metrô que as atravessam. Uma teia alimentar é constituída de relações predatórias entre as inúmeras espécies que habitam uma comunidade ecológica. A infraestrutura da Internet é formada por diversas organizações chamadas sistemas autônomos, que constituem uma infraestrutura própria de roteadores e negociam conexões com outros sistemas autônomos para troca de tráfego. Até mesmo o sistema nervoso humano, formado por bilhões de neurônios, pode ser interpretado como uma rede de conexões entre esses neurônios através das quais impulsos elétricos se propagam. Em todos esses contextos, um sistema real é modelado através da representação de um conjunto de objetos ligados uns aos outros por relacionamentos par-a-par de alguma natureza. Essa abstração é chamada de **rede**.

[2]Em alguns contextos, relações podem envolver mais do que duas partes, a exemplo de um contrato de aluguel envolvendo o inquilino, o proprietário e a imobiliária ou a transferência de um jogador de futebol de uma equipe para outra. Estes dois exemplos constituem relações **ternárias**.

Redes não são uma abstração nova em diversas áreas de conhecimento, em particular nas áreas de Matemática e Computação, nas quais são mais conhecidas por **grafos**. De fato, desde os primeiros resultados de teoria de grafos há quase 300 anos, esse objeto matemático se tornou um dos mais estudados em ambas as áreas. Para diversas comunidades acadêmicas que utilizam resultados derivados de teoria de grafos, os termos **grafo** e **rede** são utilizados de forma quase intercambiável, com leves diferenças de significado. Em ciência de redes, em geral utiliza-se o termo **rede** em referência à representação de objetos e relacionamentos observados no mundo real (rede social, rede metroviária, rede neuronal), enquanto o uso do termo **grafo** geralmente tem a conotação de que a rede representada não é relevante para a discussão em questão. Essa convenção será seguida neste texto.

A área de ciência de redes tem como objetivo central analisar e entender as redes que surgem nos mais diversos domínios de conhecimento, os processos que elas afetam e os processos que são afetados por elas. Naturalmente, a ciência de redes é fortemente interdisciplinar, pois converge conhecimentos, técnicas e objetivos de áreas como Matemática, Computação, Física, Estatística, Biologia, Economia e Sociologia. Além disso, sendo uma área de conhecimento com delimitação relativamente recente (cerca de 20 anos) e fortemente motivada pelo surgimento de grandes massas de dados extraídas dos mais diversos contextos, a ciência de redes também se apresenta como marcadamente empírica e quantitativa.

Em termos de terminologia, os objetos de uma rede são frequentemente chamados de **vértices** ou **nós** e as conexões/relacionamentos entre eles são ditos **arestas** ou **elos**. Dois vértices conectados diretamente por uma aresta são ditos **vizinhos**. De maneira geral, tanto os vértices quanto as arestas podem conter rótulos e/ou atributos que codificam alguma informação extraída da realidade. No caso das arestas, quanto um atributo identifica a intensidade do relacionamento codificado, tal atributo é denominado **peso** da aresta.

Inúmeras métricas topológicas já foram propostas e utilizadas para quantificar propriedades de redes e de seus vértices e arestas. A mais fundamental delas é o **grau** de um vértice, definido como a quantidade de vizinhos que esse vértice possui. O grau é o primeiro indicador do quão bem-conectado um vértice está em uma rede e, por isso, muitos resultados em teoria de grafos são obtidos em função dos graus mínimo ou máximo de uma rede. Apesar disso, o grau de um vértice ainda é um pedaço de informação muito simples para informar algo sobre aquele vértice individual e ainda menos relevante no contexto da rede como um todo. Em vez disso, interessa-se na frequência com que cada grau ocorre na rede, isto é, a **distribuição de grau**.

A representação mais simples e direta da distribuição de grau é um histograma indicando a quantidade de vértices com cada grau, mas em alguns casos é interessante também observar variantes em sua versão cumulativa (isto é, mostrando a quantidade de vértices com grau menor ou igual ao valor dado) ou cumulativa complementar (quantidade de vértices com grau maior ou igual). Também é frequente normalizar o gráfico pela quantidade total de vértices na rede, de forma que o valor plotado seja, na verdade, a proporção de vértices em vez do número absoluto. Um exemplo é apresentado na Figura 1.3.

A distribuição de grau é uma propriedade fundamental na caracterização da estrutura de uma rede. Em particular, como ilustrado na Figura 1.4, muitas redes reais

apresentam distribuição de grau do tipo “causa pesada”, caracterizada pela ocorrência de vértices, em proporção não-desprezível, com grau ordens de grandeza maior que a média. Isto indica uma forte desigualdade nos graus dos vértices, com número de *outliers* de grau extremamente alto concentrando a maior parte das arestas. Barabási e Albert (1999) mostraram que esse tipo de distribuição surge naturalmente de redes resultantes de um processo de crescimento, no qual novos vértices vão sendo gradualmente adicionados à rede, acoplado a um mecanismo de anexação preferencial, que confere a vértices com grau mais alto algum nível de favorecimento ao receber novas arestas.

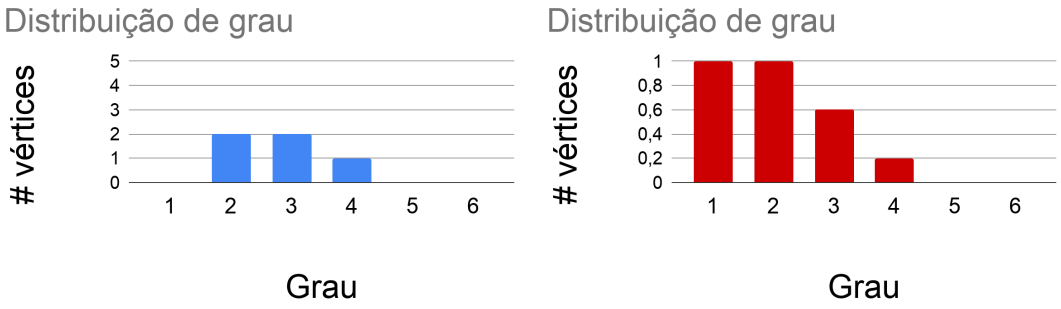


Figura 1.3: Distribuição de grau da rede apresentada na Figura 1.2. O histograma à esquerda apresenta a distribuição de grau em sua forma mais simples, em que cada barra indica a quantidade de vértices com o valor de grau correspondente. O histograma à direita apresenta a distribuição de grau cumulativa complementar: cada barra indica a quantidade de vértices (normalizada entre 0 e 1) com grau menor ou igual ao valor correspondente.

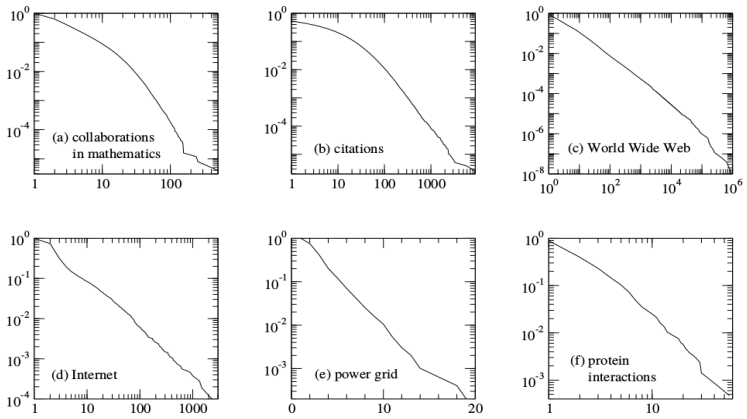


Figura 1.4: Distribuição de grau, na forma cumulativa complementar, de redes diversas: (a) rede de colaborações entre matemáticos; (b) rede de citações entre artigos científicos; (c) rede de páginas Web interligadas por *hiperlinks*; (d) infraestrutura da Internet; (e) malha elétrica; (f) rede de interações entre proteínas. Em todos os gráficos, exceto o (e), os eixos estão em escala logarítmica e a forma do gráfico é aproximadamente uma linha reta, o que sugere uma distribuição de grau de causa pesada, em particular do tipo lei de potência [Newman 2003].

Outra forma de analisar a estrutura de uma rede é observar as distâncias entre os vértices. Dados dois vértices quaisquer na rede, a distância entre eles é o comprimento do menor caminho que sai de um vértice e chega no outro, onde todos os passos desse caminho obrigatoriamente precisam seguir as arestas da rede. Um exemplo de fenômeno que pode ser estudado utilizando esta métrica é o chamado “efeito mundo pequeno”: a existência de caminhos extremamente curtos entre quaisquer dois vértices de redes muito

grandes. Por exemplo, uma análise da rede de amizades no Facebook, realizada em 2011, descobriu que, em média, 4,32 passos conectam dois usuários americanos aleatoriamente selecionados, número que aumenta para apenas 4,74 passos para dois usuários aleatórios no mundo todo [Backstrom *et al*. 2012]. Esse resultado parece contra-intuitivo, considerando que a quantidade média de amigos de uma pessoa qualquer no Facebook (o grau deste vértice) é muito menor do que a quantidade total de usuários cadastrados. O efeito mundo pequeno, observado experimentalmente em diversas redes sociais ao longo de várias décadas [Travers e Milgram 1969, Dodds *et al*. 2003, Backstrom *et al*. 2012], se tornou famoso na literatura de ciências sociais sob o nome de "seis graus de separação" e inspirou o número de Erdös[3], parte canônica do folclore matemático, e jogos como o "Six Degrees of Kevin Bacon"[4].

Conectado a este fenômeno, tem-se uma métrica fundamental, a **clusterização**. Intuitivamente, a clusterização indica uma tendência de dois vértices que possuam um vizinho comum serem, eles próprios, vizinhos também (intuitivamente, "os amigos dos meus amigos são meus amigos também"). Essa tendência é intuitivamente observada em diversas classes de rede, como observado, por exemplo, pela teoria de equilíbrio em ciências sociais. Quantitativamente, a clusterização é obtida normalizando a quantidade de triângulos em uma rede, isto é, a quantidade de trios de vértices dois-a-dois conectados uns aos outros. Algumas classes de redes, como as redes tecnológicas (*e.g*., malhas elétricas), apresentam clusterização próxima de zero enquanto outras, como as redes sociais, apresentam clusterização relativamente alta. Essa propriedade é, à primeira vista, conflitante com o efeito mundo pequeno, em redes com grau médio baixo. Isto porque, quanto mais arestas em uma rede têm o papel de conectar vértices a distâncias baixas, conferindo à rede um alto índice de clusterização, menos arestas estarão disponíveis para conectar esses vértices ao restante da rede e diminuir a distância média na rede como um todo. Propostas de resolução desta contradição foram feitas, por exemplo, em contextos como Sociologia [Granovetter 1973] e Neurologia [Gallos *et al*. 2012], com o modelo "mundo pequeno" de Watts e Strogatz (1998) ganhando expressiva notoriedade.

Para compreender o papel de vértices específicos em uma rede qualquer, uma classe extremamente importante de métricas são as **medidas de centralidade**. Na literatura de ciência de redes, o termo **centralidade** indica uma forma quantitativa de mensurar a importância de um determinado vértice em comparação com os demais, com vértices mais importantes recebendo um valor maior de centralidade. Naturalmente, existem diversas formas de conceber, em termos qualitativos, a importância de um vértice qualquer, dependendo da natureza da rede e também da aplicação em questão. Em razão disso, já foram propostas na literatura diversas medidas de centralidade. Algumas das mais importantes são listadas abaixo:

- **Centralidade de intermediação (*betweenness centrality*):** obtida, para um dado vértice da rede, contando quantos caminhos mínimos na rede (caminhos que ligam algum par de vértices da forma mais curta possível) passam por este vértice; um vértice com alta centralidade de intermediação é importante, pois serve como ponte para conectar uma quantidade muito grande de outros vértices e sua remoção da rede tende a causar o maior impacto em termos de aumento das distâncias entre os demais vértices;

[3] The Erdös Number Project: https://oakland.edu/enp/

[4] The Oracle of Bacon: http://oracleofbacon.org

- **Centralidade de proximidade (*closeness centrality*):** obtida, para um dado vértice da rede, somando as distâncias deste vértice para todos os outros e tomando o inverso desta soma; um vértice com alta centralidade de proximidade é importante, pois está, em média, mais perto dos outros vértices da rede em comparação com os demais;
- **Centralidade de autovetor:** formulação recursiva de centralidade que determina que um vértice é tão central quanto mais centrais forem seus vizinhos; nesta formulação, a centralidade de um vértice é proporcional à soma das centralidades de seus vizinhos, mas o valor específico atribuído a cada vértice é dependente de um parâmetro de normalização, com o vértice mais central da rede geralmente recebendo valor de centralidade igual a 1;
- **Centralidade de *PageRank*:** variante da centralidade de autovetor na qual a centralidade de um vértice depende da centralidade de seus vizinhos por um fator inversamente proporcional ao grau de cada um deles; a ideia por trás desta métrica de centralidade é a base do algoritmo de ranqueamento de páginas Web inovador apresentado pelo Google em 1998 e utilizado em seu motor de busca até hoje.

Por fim, é importante também mencionar as **métricas de assortatividade**. Estas métricas exigem que cada vértice da rede possua um atributo associado, que pode ser endógeno (calculado a partir da estrutura da rede, como o grau de cada vértice) ou exógeno à rede (obtido externamente, como a idade em uma rede social ou a quantidade de páginas dos artigos em uma rede de citações). Diz-se que uma rede apresenta **homofilia** com relação a um atributo se há uma tendência das arestas de uma rede a conectarem vértices com valores iguais ou próximos deste atributo, enquanto a tendência inversa, com arestas conectando vértices com valores distintos, corresponde a um cenário de **heterofilia**.

Por exemplo, a análise de uma rede de *blogs* políticos nos Estados Unidos, ilustrada na Figura 1.5 (na qual arestas indicam *hiperlinks* entre os blogs), identificou homofilia para o atributo exógeno "orientação política" [Interian e Ribeiro 2018]. Em contraste, redes tecnológicas como a Internet possuem uma estrutura fortemente hierárquica na qual vértices com grau alto tendem a se conectar a vértices com grau baixo, apresentando, assim, heterofilia com relação ao atributo endógeno "grau" [Newman 2003]. Métricas diferentes são utilizadas dependendo, por exemplo, se o atributo em questão é categórico, numérico discreto ou contínuo. Por outro lado, a correspondência entre os fenômenos de homofilia e heterofilia e valores específicos da métrica de assortatividade depende também da distribuição empírica do atributo na rede.

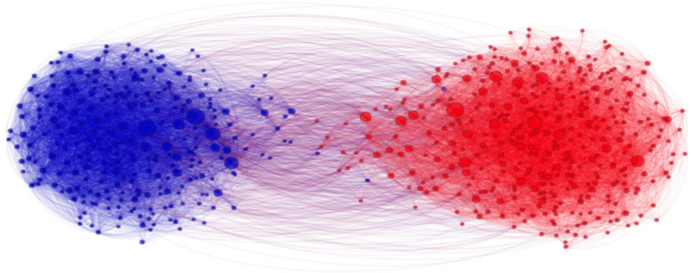


Figura 1.5: Rede de blogs políticos nos Estados Unidos. O agregado à esquerda corresponde a alinhamento com o Partido Democrata, enquanto o agregado à direita corresponde a alinhamento com o Partido Republicano [Interian e Ribeiro 2018].

1.3. Aplicando Ciência de Redes na Análise de SoS e ECOS

Ciência de redes, nas palavras de Barabási (2016), é "uma plataforma capacitante, fornecendo novas ferramentas e perspectiva para uma vasta gama de problemas científicos, de redes sociais a projeto de remédios". Em troca, ela exige que seja incorporada uma nova forma de pensar e enxergar a complexidade dos sistemas. É preciso compreender que "por trás de cada sistema complexo existe uma rede intrincada que codifica as interações entre os componentes do sistema" [Barabási 2016]. Esses componentes podem ser elementos computacionais, pessoas, partes de um organismo, países ou até mesmo segmentos de informação.

Dessa forma, em uma tentativa de linearizar o processo científico utilizando as ferramentas que ciência de redes oferece, pode-se afirmar que o primeiro passo consiste na identificação das redes relacionadas ao problema em mãos. Mais precisamente, deve-se identificar quais são os componentes do sistema e qual é a natureza do relacionamento entre esses componentes, os quais corresponderão, respectivamente, a vértices e arestas da rede. Este passo é fundamental, pois ele não apenas define quais dados deve-se obter para construir uma imagem da rede que possa futuramente ser quantitativamente analisada, mas também define a semântica subjacente a todas as métricas que serão computadas a partir destes dados.

É importante ressaltar que, de forma geral, existem diversos elementos de um sistema complexo que podem ser compreendidos como componentes interagentes, de forma que existem múltiplas redes que podem ser identificadas em um dado contexto. Por exemplo, ao estudar a dinâmica de produção científica, uma análise do ponto de vista do conhecimento produzido leva naturalmente à construção da rede de citações, na qual os vértices são artigos científicos e as arestas correspondem a citações entre eles, enquanto o emprego de uma abordagem sociológica torna mais interessante o estudo da rede de colaborações, em que os vértices são pesquisadores e as arestas são induzidas pela publicação de trabalhos em coautoria.

Além disso, a granularidade com que os dados podem ser obtidos também influencia a definição da rede. Por exemplo, a rede neuronal de organismos simples pode ser analisada a nível de neurônios (a rede neural do verme *Caenorhabditis elegans* foi totalmente mapeada em 1986 [White et al. 1986]), enquanto para organismos mais complexos como o ser humano é necessário construir a rede a nível de áreas funcionais. O estudo da Internet, uma rede de roteadores interligados por enlaces de comunicação, raramente desfruta de informação suficiente sobre enlaces específicos e a análise topológica da Internet geralmente é feita sobre a rede de sistemas autônomos (regiões topológicas da Internet sob administração única).

Seguem alguns exemplos de redes que podem naturalmente ser identificadas nos contextos de ECOS e SoS:

- um SoS pode ser diretamente modelado como uma rede de sistemas constituintes interagentes, em que os sistemas correspondem aos vértices da rede e as trocas de mensagens entre eles são representadas pelas arestas;
- em ECOS, uma plataforma de desenvolvimento e compartilhamento de artefatos de *software* induz naturalmente uma dinâmica de coutilização em projetos, que pode ser analisada através de uma rede de coutilização: os artefatos constituem os

vértices da rede e as arestas correspondem à utilização simultânea em um projeto, com um peso indicando a frequência de coutilização;

- tanto em ECOS quanto em SoS, qualquer *software* desenvolvido de forma modular pode ser visto como uma rede de dependências, com os vértices correspondendo aos módulos e as dependências diretas entre eles induzindo as arestas;
- tanto em ECOS quanto em SoS, a dinâmica de desenvolvimento de artefatos de *software* envolve interações entre desenvolvedores, o que torna interessante a análise da rede de colaborações entre desenvolvedores: os vértices são os desenvolvedores e as arestas indicam a quantidade de projetos em que eles colaboraram ou artefatos que eles produziram em conjunto.

O passo seguinte consiste em identificar métricas que capturem propriedades da rede relacionadas com o problema a ser analisado. Naturalmente é interessante utilizar métricas já propostas na literatura e com validade comprovada pela comunidade científica, mas, em muitos casos, as métricas existentes podem ser insuficientes para capturar o problema atacado de maneira significativa. Neste caso, é interessante que novas métricas, adequadas para o caso em questão, sejam propostas. Destaca-se que a discussão de como as características de ECOS e SoS anteriormente apresentadas podem ser analisadas por meio da ciência de redes, a fim de mostrar como seus modelos apoiam decisões de desenvolvimento, tem sido pouco relatada em artigos científicos publicados sobre sistemas complexos.

Mesmo as métricas mais básicas de ciência de redes podem ser interessantes para uma análise preliminar da rede em questão. Seguem alguns exemplos utilizando as métricas apresentadas na Seção 1.2:

1) Na rede de interações de um SoS, é interessante que nenhum sistema concentre uma grande parte das interações, evitando assim a formação de gargalos ou pontos únicos de falha. Essa hipótese pode ser verificada calculando a distribuição de grau desta rede e verificando se é observada uma distribuição bastante concentrada. A existência de um conjunto de vértices com grau muito maior do que a média sugere que tais vértices podem constituir gargalos e pontos de falha do SoS. Em outras palavras, a distribuição de grau da rede de interações serve de indicativo para a robustez do SoS como um todo;
2) Na rede de dependências de um *software*, como cada módulo idealmente possui um objetivo relativamente pontual, em que ele dever-se-ia relacionar com um conjunto limitado de outros módulos. Novamente, essa hipótese pode ser verificada calculando a distribuição de grau desta rede. Uma distribuição de grau muito desigual pode ser um indício de que um determinado módulo está concentrando um número excessivo de funções, evidenciando uma modularização inadequada;
3) A rede de colaborações de desenvolvedores codifica os padrões de interação existentes entre as pessoas envolvidas nos processos de desenvolvimento. Considere-se um cenário em que se gostaria de compor um comitê consultivo técnico para a análise de novos projetos a serem adicionados a uma plataforma, como ocorre em um ECOS. Um critério interessante para a seleção de pessoas para comporem esse comitê seria encontrar desenvolvedores que mais colaboram nesta comunidade e, com isso, conhecem melhor os critérios que constituem um

bom projeto de *software*. Esse critério sugere que se procure os desenvolvedores que possuem valor alto em alguma métrica de centralidade, como, por exemplo a centralidade de autovetor;

4) Considere-se ainda uma plataforma de desenvolvimento de *software* em que cada artefato possui um conjunto de rótulos (*tags*), indicando de maneira resumida seu contexto e seu propósito. Na rede de coutilização de artefatos desta plataforma, é esperado que artefatos semelhantes não sejam utilizados simultaneamente por não agregarem muita funcionalidade ao *software* já desenvolvido, como acontece na plataforma de um ECOS. Isso sugere que um grau alto de homofilia nesta rede é um indicador de redundância no conjunto de artefatos existentes, ou um sinal de que muitos artefatos só funcionam em conjunto e talvez devessem ser distribuídos como um único artefato.

É importante ter em mente que, quantitativamente falando, é natural que redes de naturezas diferentes apresentem propriedades topológicas diferentes. Por exemplo, é razoável que uma rede de colaborações de um ECOS, sendo uma rede social, apresente um índice alto de clusterização, ao contrário de uma rede tecnológica como a rede de interações em um SoS. Por isso, é interessante fornecer um *benchmark* para comparar as métricas extraídas das redes reais. Esse *benchmark* é tradicionalmente dado por um modelo estocástico para redes, do qual pode-se extrair um valor esperado para tais métricas analiticamente ou através de simulações computacionais. Inúmeros modelos já foram propostos na literatura, sendo alguns dos mais conhecidos o modelo Erdős–Rényi, o modelo de blocos estocástico, o modelo Barabási–Albert e o modelo Watts–Strogatz. Cada modelo é mais adequado para representar redes obtidas por diferentes processos de formação e, em casos específicos, também pode ser interessante propor um modelo adequado para a rede em questão.

1.4. Desafios e Oportunidades de Pesquisa

Como desdobramentos, sugere-se investigar alguns desafios e oportunidades de pesquisa no campo de ciência de redes para sistemas complexos como ECOS e SoS. Primeiramente, é importante investigar como a diversidade de tecnologias, sistemas, organizações e atores afetam o desenvolvimento de sistemas de informação modernos, dado que diferentes redes sociotécnicas são constituídas de maneira dinâmica e, para tanto, requerem técnicas e ferramentas para modelagem e análise que considerem essa característica dos tipos de sistemas complexos discutidos neste capítulo.

Outra linha de investigação refere-se a como decisões de projeto, desenvolvimento, operação, governança e evolução desses sistemas podem ser apoiadas por técnicas que combinem mineração de dados e visualização de informação, considerando os vários níveis de abstração, os diferentes pontos de vista dos *stakeholders* e as carências de uma gestão de conhecimento orgânica e institucionalizada. Mais ainda, é importante verificar como identificar, gerir e avaliar a demandas e necessidades de diferentes *stakeholders* em um cenário cuja troca de volumes cada vez maiores de dados e a necessidade de se fazer uso eficiente de recursos computacionais e energéticos limitados têm sido fatores críticos, dado que sistemas complexos possuem, em sua essência, comportamento emergente e dinamismo em constituir e desfazer determinadas configurações para cumprir missões, como em SoS.

Outro aspecto interessante a investigar é como apoiar uma melhor compreensão dos relacionamentos entre os elementos envolvidos em sistemas complexos, uma vez que se torna importante não apenas entender que elementos compõem as redes sociotécnicas, mas também os diversos tipos de relacionamentos que podem ocorrer entre tais elementos. Um primeiro passo pode ser partir de tipos reportados em mapeamentos sistemáticos e explorar técnicas da ciência de redes para sumarização de informação.

Por fim, mas não menos importante, estudos experimentais tais como estudos de caso de natureza interpretativa e qualitativa podem explorar informações de contexto e contribuir para uma base de conhecimento que possa guiar diferentes *stakeholders* em seus processos de tomada de decisão. Essa é uma questão relevante uma vez que não é uma tarefa fácil prover soluções passíveis de generalização para problemas enfrentados na gestão e engenharia desses sistemas, mas cujas experiências de diferentes cenários ajudam a gerar *insights*.

1.5. Considerações Finais

Este capítulo introduziu como o contexto dos sistemas complexos vem afetando o desenvolvimento das aplicações contemporâneas e como modelos em ciência de redes podem ser utilizados para melhor compreender a estrutura e características desses sistemas. De forma mais específica, por meio do estudo de duas classes de sistemas complexos (ECOS e SoS), buscou-se compreender as características desses sistemas (que inclusive refletem-se em alguns sistemas atuais), conhecer a área de ciência de redes (alguns modelos, ferramentas e mecanismos de análise), avaliar problemas e identificar desafios e oportunidades de pesquisa. Destaca-se que, apesar de ECOS e SoS serem sistemas que materializam o desafio de se tratar a complexidade com uma visão mais sistêmica, poucos trabalhos têm reportado a aplicação de técnicas e modelos de ciência de redes neste contexto, o que reforça o desafio para a pesquisa e prática. Isso evidencia desafios e abre oportunidades de pesquisa para a comunidade de sistemas de informação, uma vez que requer uma visão integrada de pessoas, processos e tecnologia.

Referências

Amorim, S. S., McGregor, J. D., Almeida, E. S., Chavez, C. F. G. (2017) "Software ecosystems' architectural health: Another view", Proceedings of the Joint 5th International Workshop on Software Engineering for Systems-of-Systems and 11th Workshop on Distributed Software Development, Software Ecosystems and Systems-of-Systems. USA, IEEE, pp. 66-69.

Azani, C. (2009) "An open systems approach to System of Systems Engineering". In: Jamshidi, M., ed. Systems of Systems Engineering: Innovations for the 21st Century. USA, John Wiley & Sons, Inc., pp. 21-43.

Backstrom, L., Boldi, P., Rosa, M., Ugander, J., Vigna, S. (2012) "Four degrees of separation", Proceedings of the 4th Annual ACM Web Science Conference. USA, ACM, pp. 33-42.

Barabási, A.-L. (2016) "Network Science". United Kingdom, Cambridge University Press.

Barabási, A.-L., Albert, R. (1999) "Emergence of scaling in random networks". Science, vol. 286, no. 5439, pp. 509-512.

Barbosa, O., Santos, R. P., Alves, C., Werner, C., Jansen, S. (2013) “A systematic mapping study on software ecosystems from a three-dimensional perspective”. In: Jansen, S., Brinkkemper, S., Cusumano, M. A. eds. Software ecosystems: Analyzing and managing business networks in the software industry. United Kingdom/USA, Edward Elgar Publishing, pp. 59-81.

Bosch, J. (2009) "From software product lines to software ecosystem", Proceedings of 13th International Software Product Line Conference. USA, ACM, pp. 1-10.

Bosch, J. (2016) "Speed, data, and ecosystems: The future of Software Engineering", IEEE Software, vol. 33, no. 1, pp. 82-88.

Bosch, J., Olsson, H. H. (2018) "Ecosystem traps and where to find them", Journal of Software: Evolution and Process, vol. 30, no. 11.

Campbell, P. R. J., Ahmed, F. (2010) "A three-dimensional view of Software Ecosystems", Proceedings of the 4th European Conference on Software Architecture: Companion Volume. USA, ACM, pp. 81-84.

Coutinho, E. F., Viana, D., Santos, R. P. (2017) "An Exploratory Study on the need for modeling software ecosystems: The case of SOLAR SECO", Proceedings of the 9th International Workshop on Modelling in Software Engineering. USA, IEEE, pp. 47-53.

Dodds, P. S., Muhamad, R., Watts, D. J. (2003) "An experimental study of search in global social networks", Science, vol. 301, no. 5634, pp. 827-829.

Firesmith, D. (2010) "Profiling systems using the defining characteristics of systems of systems (SoS)", Technical report. USA, Software Engineering Institute, Carnegie Mellon University.

Fontão, A., Ábia, B., Wiese, I., Estácio, B., Quinta, M., Santos, R. P., Dias-Neto, A. C. (2018) "Supporting governance of mobile application developers from mining and analyzing technical questions in stack overflow", Journal of Software Engineering Research & Development, vol. 6.

Fontão, A., Dias-Neto, A. C., Viana, D. (2017) "Investigating factors that influence developers' experience in mobile software ecosystems", Proceedings of the Joint 5th International Workshop on Software Engineering for Systems-of-Systems and 11th Workshop on Distributed Software Development, Software Ecosystems and Systems-of-Systems. USA, IEEE, pp. 55-58.

Franco-Bedoya, O., Ameller, D., Costal, D., Franch, X. (2017) "Open source software ecosystems: A systematic mapping", Information and Software Technology, vol. 91, pp. 160-185.

Gallos, L. K., Makse, H. A., Sigman, M. (2012) "A small world of weak ties provides optimal global integration of self-similar modules in functional brain networks", Proceedings of the National Academy of Sciences, vol. 109, no. 8, pp. 2825-2830.

Gomes, P., Cavalcante, E., Maia, P., Batista, T., Oliveira, K. (2015) "A systematic mapping on discovery and composition mechanisms for systems-of-systems", Proceedings of the 41st Euromicro Conference on Software Engineering and Advanced Applications. USA, IEEE, pp. 191-198.

Graciano Neto, V. V., Cavalcante, E., El Hachem, J., Santos, D. S. (2017a) "On the interplay of business process modeling and missions in systems- of-information systems", Proceedings of the Joint 5th International Workshop on Software Engineering for Systems-of-Systems and 11th Workshop on Distributed Software Development, Software Ecosystems and Systems-of-Systems. USA, IEEE, pp. 72-73.

Graciano Neto, V. V., Santos, R. P., Araujo, R. M., (2017b) "New challenges in the social Web: Towards systems-of-information systems ecosystems". In: Anais do VIII Workshop sobre Aspectos da Interação Humano-Computador na Web Social. CEUR Workshop Proceedings, vol. 2039, pp. 1-12.

Granovetter, M. S. (1973) "The strength of weak ties", American Journal of Sociology, vol. 78, no. 6, pp. 1360-1380.

Inocêncio, T. J., Gonzales, G. R., Cavalcante, E., Horita, F. E. A. (2019) "Emergent behavior in systems-of-systems: A systematic mapping study", Proceedings of the XXXIII Brazilian Symposium on Software Engineering. USA, ACM, pp. 140-149.

Interian, R., Ribeiro, C. C. (2018) "An empirical investigation of network polarization", Applied Mathematics and Computation, vol. 339, pp. 651-662.

Jansen, S., Brinkkemper, S., and Cusumano, M. A. (2013) "Software ecosystems: Analyzing and managing business networks in the software industry". United Kingdom/USA, Edward Elgar Publishing.

Jansen, S., Finkelstein, A., Brinkkemper, S. (2009) "A sense of community: A research agenda for software ecosystems", Proceedings of the 31st International Conference on Software Engineering. USA, ACM, pp. 187-190.

Lana, C. A., Souza, N. M., Delamaro, M. E., Nakagawa, E. Y., Oquendo, F., Maldonado, J. C. (2016) "Systems-of-systems development: Initiatives, trends, and challenges", Proceedings of the XLII Latin American Computing Conference. USA, IEEE, pp. 585-596.

Lungu, M., Lanza, M., Girba, T., Robbes, R. (2010) "The Small Project Observatory: Visualizing software ecosystems", Science of Computer Programming, vol. 75, no. 4, pp. 264-275.

Maier, M. W. (1998) "Architecting principles for systems-of-systems", Systems Engineering, vol. 1, no. 4, pp. 267-284.

Manikas, K. (2016) "Revisiting software ecosystems research: A longitudinal literature study", Journal of Systems and Software, vol. 117, pp. 84-103.

Manikas, K., Hansen, K. M. (2013) "Software ecosystems – A systematic literature review", Journal of Systems and Software, vol. 86, no. 5, pp. 1294-1306.

McGregor, J. D. (2012) "Ecosystem modeling and analysis", Proceedings of 16th International Software Product Line Conference - Volume 2. USA, ACM, p. 268.

Messerschmitt, D. G., Szyperski, C. (2003) "Software ecosystem: Understanding an Indispensable technology and industry". USA, MIT Press.

Moreno, J. L. (1934) "Who shall survive?: A new approach to the problem of human interrelations". Nervous and Mental Disease Publishing Co.

Motta, R. C., Oliveira, K. M., Travassos, G. H. (2017) "Rethinking interoperability in contemporary software systems", Proceedings of the Joint 5th International Workshop on Software Engineering for Systems-of-Systems and 11th Workshop on Distributed Software Development, Software Ecosystems and Systems-of-Systems. USA, IEEE, pp. 9-15.

Newman, M. J. (2003) "The Structure and Function of Complex Networks", SIAM Review, vol. 45, no. 2, pp. 167-256.

Nielsen, C. B., Larsen, P. G., Fitzgerald, J., Woodcock, J., Peleska, J. (2015) "Systems of Systems Engineering: Basic concepts, model-based techniques, and research directions", ACM Computing Surveys, vol. 48, no. 2.

Santos, R. P. (2016) "Managing and monitoring software ecosystem to support demand and solution analysis", PhD Thesis. Brazil, COPPE/UFRJ.

Santos, R. P., Oliveira, J. (2013) "Análise e aplicações de redes sociais em ecossistemas de software", Anais do IX Simpósio Brasileiro de Sistemas de Informação. Brasil, SBC, pp. 19-24.

Santos, R. P., Werner, C. M. L. (2011a) "A proposal for Software Ecosystems Engineering". In: Proceedings of the 3rd International Workshop on Software Ecosystems. CEUR Workshop Proceedings, vol. 746, pp. 40-51.

Santos, R. P., Werner, C. (2011b) "Treating business dimension in software ecosystems", Proceedings of the 3rd ACM/IFIP International Conference on Management of Emergent Digital EcoSystems. USA, ACM, pp. 197-201.

Santos, R. P., Werner, C. M. L. (2012a) "Treating social dimension in software ecosystems through ReuseECOS Approach", Proceedings of the 6th International Conference on Digital Ecosystem Technologies – Complex Environment Engineering. USA, IEEE, pp. 1-6.

Santos, R. P., Werner, C. (2012b) "ReuseECOS: An approach to support global software development through software ecosystems", Proceedings of the 7th IEEE International Conference on Global Software Engineering Workshops. USA, IEEE, pp. 60-65.

Santos, R. P., Werner, C., Barbosa, O., Alves, C. (2012) "Software ecosystems: Trends and impacts on Software Engineering", Proceedings of the 26th Brazilian Symposium on Software Engineering. USA, ACM, pp. 206-210.

Serebrenik, A., Mens, T. (2015) "Challenges in software ecosystems research", Proceedings of the 2015 European Conference on Software Architecture Workshops. USA, ACM.

Silva, R. T., Aguiar, L. G. F., Audacio, E. D., Genvigir, E. C. (2017) "Identifying actors to support software ecosystem health", Proceedings of the Joint 5th International Workshop on Software Engineering for Systems-of-Systems and 11th Workshop on Distributed Software Development, Software Ecosystems and Systems-of-Systems. USA, IEEE, pp. 76-77.

Travers, J., Milgram, S. (1969) "An experimental study of the small world problem", Sociometry, vol. 32, no. 4, pp. 425-443.

Valença, G., Alves, C., Jansen, S. (2018) "Strategies for managing power relationships in software ecosystems", Journal of Systems and Software, vol. 144, pp. 478-500.

van den Berk, I., Jansen, S., Luinenburg, L. (2010) "Software ecosystems: A software ecosystem strategy assessment model", Proceedings of the 4th European Conference on Software Architecture: Companion Volume. USA, ACM, pp. 135-142.

van der Maas, J. (2016) "Evolution of collaboration in open source software ecosystems", Master Thesis. Netherlands, Utrecht University.

Watts, D. J., Strogatz, S. H. (1998) "Collective dynamics of 'small-world' networks", Nature, vol. 393, pp. 440-442.

White, J. G., Southgate, E., Thomson, J. N., Brenner, S. (1986). "The structure of the nervous system of the nematode Caenorhabditis elegans", Philosophical Transactions of the Royal Society of London B Biological Sciences, vol. 314, no. 1165, pp. 1-340.

Yang, Z., Jansen, S., Gao, X., Zhang, D. (2016) "On the future of solution composition in software ecosystems". In: Bañares J., Tserpes K., Altmann J., eds. Proceedings of the 13th International Conference on Economics of Grids, Clouds, Systems, and Services. Lecture Notes in Computer Science, vol 10382. Switzerland, Springer International Publishing AG.

Yu, E., Deng, S. (2011) "Understanding software ecosystems: A strategic modeling approach". In: Proceedings of the 3rd International Workshop on Software Ecosystems. CEUR Workshop Proceedings, vol. 746, pp. 65-76.

Sobre os autores

Rodrigo Santos é Professor Adjunto do Departamento de Informática Aplicada (DIA) e membro efetivo do Programa de Pós-Graduação em Informática (PPGI) da Universidade Federal do Estado do Rio de Janeiro (UNIRIO), do qual foi Coordenador do Curso de Mestrado (2019-2020). Lidera o Laboratório de Engenharia de Sistemas Complexos da UNIRIO (LabESC). Doutor e Mestre em Engenharia de Sistemas e Computação pela COPPE/UFRJ, onde realizou também o seu Pós-Doutorado (2016), e Bacharel em Ciência da Computação pela UFLA. Atuou como pesquisador visitante na University College London (2014-2015). É editor-chefe da iSys: Revista Brasileira de Sistemas de Informação e organizou edições especiais em periódicos como iSys, JBCS e JISA, além de volume na série Springer CCIS. É membro da Sociedade Brasileira de Computação (SBC) desde 2006, atuando como Coordenador do Comitê Gestor da Comissão Especial de Sistemas de Informação (CE-SI), membro do Comitê Gestor da Comissão Especial de Jogos e Entretenimento Digital (CE-Jogos) e membro efetivo do Grupo de Interesse em Educação em Computação (GIEC). Tem experiência na área de Ciência da Computação, com ênfase em Engenharia de Software e Sistemas de Informação. Seus principais campos de atuação são Engenharia de Sistemas Complexos (especialmente ecossistemas de software e sistemas de sistemas) e Educação em Engenharia de Software. Foi coordenador científico de mais de vinte eventos (simpósios, trilhas e *workshops*) no Brasil e no exterior e proferiu comunicações (palestras, minicursos e tutoriais) em mais de vinte eventos nacionais. Publicou mais de 150 artigos em periódicos e congressos, sendo alguns deles premiados, e recebeu distinções acadêmicas como revisor de destaque em conferências e como orientador de trabalhos em concursos de teses e dissertações da SBC. *Link* para Currículo Lattes: http://lattes.cnpq.br/8613736894676086.

Jefferson Elbert Simões é Professor Adjunto do Departamento de Informática Aplicada (DIA) da Universidade Federal do Estado do Rio de Janeiro (UNIRIO). É Mestre e Doutor em Engenharia de Sistemas e Computação pela COPPE/Universidade Federal do Rio de Janeiro (UFRJ), com período de Doutorado Sanduíche na École Polytechnique Fédérale de Lausanne (EPFL), e Engenheiro de Computação e Informação *Magna cum Laude* pela UFRJ (2010). Possui interesse em ciência de redes, modelos probabilísticos, educação à distância, sistemas de computação, algoritmos e grafos. *Link* para Currículo Lattes: http://lattes.cnpq.br/6938694286834426.

Everton Cavalcante é Professor Adjunto do Departamento de Informática e Matemática Aplicada (DIMAp) da Universidade Federal do Rio Grande do Norte (UFRN), Natal, Brasil. Possui dupla diplomação de Doutor em Ciência da Computação pela Universidade Federal do Rio Grande do Norte e Docteur en Sciences et Technologies de l'Information et de la Communication pela Université Bretagne Sud, França (2016). Possui experiência na área de Ciência da Computação com ênfase em arquitetura de software e sistemas distribuídos, atuando principalmente nos seguintes temas: middleware, Computação em Nuvem, Computação Ubíqua, Internet das Coisas, cidades inteligentes, reconfiguração dinâmica de software, linguagens de descrição arquitetural e sistemas de sistemas. É docente permanente do Programa de Pós-Graduação em Sistemas e Computação (PPgSC) da UFRN e integra o Núcleo Integrador de Pesquisa e Inovação em Engenharia de Software (SETe) do Instituto Metrópole Digital (IMD) da UFRN, onde é coordenador do curso de Bacharelado em Engenharia de Software e vice-coordenador no Programa de Residência em Tecnologia da Informação. *Link* para Currículo Lattes: http://lattes.cnpq.br/5065548216266121.

Capítulo 2

Desenvolvimento de Soluções com Serviços: SOA, Cloud e Microsserviços

Leonardo Guerreiro Azevedo

Abstract

Service-Oriented Architecture (SOA), Cloud Computing and Microservices Architecture (MSA) are fundamental concepts, highly interconnected, which should be understood for the development of flexible, distributed, aligned to business and high-performance applications - essential characteristics of modern information systems. The SOA goal is to reduce costs and deadlines by developing applications using existing services composition. In Microservice Architecture, each microservice makes available specific functionalities of a domain. They are developed and deployed independent of each other. In Cloud Computing, processing, storage, data and resources of software are virtualized towards on-demand scaling and availability. This short course has theoretical and practical features. Its goal is to give expertise in distributed application development employing those paradigms for building information systems.

Resumo

Arquitetura Orientada a Serviços (SOA), Computação em Nuvem (Cloud) e Arquitetura de Microsserviços (MSA) são conceitos fundamentais, altamente interligados, que devem ser bem entendidos para o desenvolvimento ágil de aplicações distribuídas flexíveis, alinhadas ao negócio e com desempenho - características essenciais para os sistemas de informação modernos. SOA tem objetivo de reduzir custos e prazos pelo desenvolvimento de aplicações através da composição de serviços. Em MSA, cada microsserviço disponibiliza funcionalidades específicas de um domínio, sendo desenvolvido e implantado independentemente uns dos outros. Na Cloud, processamento, armazenamento, dados e recursos de software são virtualizados em busca de escalonamento e disponibilização sob demanda sendo seu uso um dos princípios de MSA. Este minicurso tem característica teórico-prática e seu objetivo é capacitar no desenvolvimento de aplicações distribuídas empregando estes paradigmas para construção de sistemas de informação.

2.1. Introdução

Arquitetura Orientada a Serviços (SOA - Service-Oriented Architecture) é uma abordagem para construir e integrar aplicações pela descoberta, invocação e composição de serviços distribuídos [Papazoglou and Van Den Heuvel 2007]. SOA permite reduzir custos e prazos pelo desenvolvimento de aplicações através da composição de serviços [Erl 2005]. SOA facilita interoperabilidade entre tecnologias de *middleware* heterogêneas e promove o acoplamento fraco entre consumidor de serviço (requisitantes, clientes) e provedor de serviço [Pautasso et al. 2008]. SOA tem se mostrado como um importante paradigma para desenvolvimento de aplicações flexíveis, distribuídas, alinhadas ao negócio, compostas por serviços independentes de plataforma e protocolo em um ambiente distribuído [Papazoglou and Van Den Heuvel 2007, Erl 2005].

Arquitetura de Microsserviços (MSA - Microservice Architecture) é visto de diferentes formas na literatura (como uma nova abordagem de SOA ou uma evolução de SOA ou uma implementação de SOA) para a utilização de tecnologias para construção de sistemas [Richardson 2016]. Organizações vêm obtendo sucesso na adoção de **MSA**, demonstrando ganhos de eficiência e escalabilidade pela construção de produtos providos por pequenas equipes multidisciplinares, em um ciclo de vida mais curto, e abertos à integração, porém independentes [Fowler and Lewis 2014]. Um desafio de MSA surge a partir da necessidade de integrar dados de origens distintas, o qual é um problema usual em muitas empresas, que já possuem diversas fontes de informação, como bancos de dados relacionais, NoSQL, planilhas, sistemas que gerenciam configurações de artefatos de software, e repositórios de dados não estruturados. Villaça *et al.* apresentam as principais soluções MSA para este problema [Villaça et al. 2018a] e apresenta como fazer este tipo de integração na prática [Villaça et al. 2018b].

Computação na Nuvem permite disponibilizar serviços encapsulando seus detalhes internos. A computação é virtualizada através da construção de componentes distribuídos, tais como, processamento, armazenamento, dados e recursos de software. Nesse ambiente, usuários têm acesso a grande poder de processamento de uma maneira totalmente virtualizada e escalável [Mell et al. 2011].

Este minicurso apresenta os principais conceitos destes três paradigmas de maneira prática. Seu objetivo é proporcionar a fundamentação teórica e prática de SOA, Microsserviços e Cloud. Material complementar está disponível no site deste minicurso [Azevedo 2020] com exemplos de código em diferentes níveis de complexidade, apresentações, exercícios etc.

O principal pré-requisito para o aluno é ter conhecimento em programação orientada a objeto, em especial na linguagem Java. Não é necessário conhecimento avançado na linguagem, uma vez que, quando os conceitos são empregados na parte prática, eles são explicados.

O minicurso está dividido da seguinte forma:

- A Seção 2.2 apresenta a Arquitetura Orientada a Serviços. Esta seção provê o fundamento necessário da área, o qual também é base para entender os conceitos relacionados a microsserviços. É uma seção em maior parte teórica, incluindo motivação, definição e princípios de SOA além de introduzir a tecnologia de Serviços

Web, que é a principal tecnologia para implementação de SOA. A parte prática desta seção refere-se a caracterizar o desenvolvimento de Serviços Web através das tecnologias SOAP e REST. As tecnologias relacionadas são apresentadas e exemplos de implementação são discutidos em detalhes.

- A Seção 2.3 detalha Arquitetura de Microsserviços, a qual é uma forma de implementação e implantação de serviços em SOA empregando práticas do estado da arte de engenharia de software [Zimmermann 2017]. Esta seção apresenta a comparação de Microsserviços com SOA e a comparação de microsserviços com arquitetura monolítica para desenvolvimento de aplicações. Além disso, ela apresenta os princípios que norteiam o paradigma e as principais tecnologias para implantação. Ao final, é apresentado um exemplo de implantação de microsserviço empregando Docker.

- A Seção 2.4 é dedicada à Cloud Computing, a qual é um modelo que permite acesso ubíquo, conveniente, sob demanda a um conjunto compartilhado de recursos de computação (por exemplo, rede, servidores, armazenamento, aplicações e serviços) que podem ser rapidamente provisionados e liberados com esforço mínimo de gerência ou interação com o provedor do serviço [Mell et al. 2011]. A seção apresenta as características essenciais de Cloud Computing, modelos de implantação e modelos de serviços. Finalmente, a seção apresenta o IBM Cloud, uma ferramenta para desenvolver, implantar, executar e gerenciar aplicações na Cloud.

- A Seção 2.5 apresenta os exercícios práticos a serem seguidos para aprofundamento dos conceitos apresentados nas seções anteriores. A seção aponta para a página do minicurso [Azevedo 2020] onde estão disponibilizados o cenário dos exercícios, passo-a-passo de instalação de ferramentas e de execução dos exercícios, arquivos fontes, apresentações e outros materiais de apoio. O objetivo é que esta página seja evoluída ao longo do tempo de acordo com as aulas lecionadas e o *feedback* dos alunos.

- Finalmente, a Seção 2.6 apresenta as considerações finais.

2.2. Arquitetura Orientada a Serviços

Esta seção está dividida da seguinte forma. Na Section 2.2.1, são apresentados a motivação, definição e princípios de SOA, além das características de serviços. https://overleaf.sl.cloud9.ibm.com/p
A Seção 2.2.2 apresenta a definição de Serviços Web (principal tecnologia para implementação de serviços) e dos seus principais tipos de implementação: Serviços Web SOAP e Serviços Web REST, ilustrando a implementação de cada um destes tipos de serviços.

2.2.1. Definição, princípios e motivação do paradigma

Organizações modernas precisam responder de forma efetiva e rápida às oportunidades do mercado. Suas aplicações necessitam se comunicar de forma integrada com o objetivo de atingir agilidade e simplificar processos de negócio, tornando-os mais produtivos, frente à crescente e a intensa competitividade.

O uso de serviços e de seus padrões de integração automatizada de negócios levou a grandes avanços na integração de aplicações. SOA (Service-Oriented Architecture

ou Arquitetura Orientada a Serviços) é um paradigma para a realização e manutenção de processos de negócio em um grande ambiente de sistemas distribuídos que são controlados por diferentes proprietários [Josuttis 2007]. Em SOA, funcionalidades do negócio são disponibilizadas para consumidores como serviços compartilhados e reutilizáveis em uma rede de TI [Marks and Bell 2008].

A nível conceitual, serviços são componentes de software providos através de um *endpoint* (ponto de acesso) acessível na rede [Vinoski 2002]. Serviços são módulos de negócio ou funcionalidades das aplicações que possuem interfaces expostas e que são invocados via mensagens [Erl 2005]. Um serviço recebe uma mensagem de entrada, a processa e retorna uma mensagem de resposta. Por exemplo, em um sistema bancário teríamos os seguintes serviços: serviço para buscar nomes e endereços; serviço de abertura de conta; serviço de balanço de contas; serviço de depósitos. Dentre as características (ou princípios) de serviços destacamos [Erl 2005]: (i) Deve estar alinhado ao negócio, isto é, fornecer funcionalidade de um processo da organização; (ii) Deve ser operacionalmente independente com alta coesão e acoplamento fraco; (iii) Deve ser sem estado, isto é, fornecer os mesmos resultados para uma mesma entrada; (iv) Deve permitir composição; (v) Deve ser atômico/auto-contido; (vi) Deve garantir consistência das informações; (vii) Deve ter pré e pós-condições bem definidas; (viii) Deve garantir interoperabilidade, ou seja, poder se comunicar com consumidores ou consumir outros serviços desenvolvidos empregando diversificadas tecnologias; (ix) Deve permitir reuso. Estes princípios podem ser relaxados para alcançar determinados objetivos como, por exemplo, serviços compostos não são atômicos, pois utilizam outros serviços para realizar suas tarefas.

Ciclo de vida de uma abordagem tradicional não se aplica diretamente a uma abordagem SOA. SOA herda problemas encontrados no desenvolvimento de software tradicional, como, por exemplo: não satisfação das necessidades dos usuários, uso de recursos além dos orçados, ultrapassagem do tempo estimado para o projeto. Também herda problemas existentes no desenvolvimento de software distribuído, como, por exemplo: complexidade operacional, falta de comunicação entre as pessoas envolvidas, falta de clareza nas tarefas de desenvolvimento, distribuição de responsabilidades. Além disso, SOA traz preocupações adicionais, tais como: cooperação entre os papéis arquiteturais de SOA, bom entendimento dos modelos de negócio e relacionamento entre os parceiros de negócio, como lidar com requisitos conflitantes como distribuir serviços através das fronteiras da organização, como produzir serviços de forma a possibilitar o reuso. Portanto, um ciclo de vida específico para SOA é necessário [Gu and Lago 2007].

Um ciclo de vida de SOA deve considerar os papéis arquiteturais (*stakeholders*): provedor, consumidor e *brokers*/registro de serviços. O *Provedor de Serviços* é o proprietário do serviço. Ele é responsável por desenvolver, publicar e manter serviços para serem consumidos. O *Consumidor de Serviços* é responsável por invocar serviços através de uma aplicação que atende aos requisitos de um usuário final a qual é construída e disponibilizada pelo *Servidor de Aplicações*. É razoável considerar que *Servidor de Aplicações* e *Consumidor de Serviços* atuam de forma combinada, pois o primeiro analisa os requisitos do usuário e projeta, implementa e testa a aplicação que o usuário final irá utilizar. No momento em que serviços são consumidos, este *stakeholder* assume o papel de *Consumidor de Serviços* e descobre serviços, orquestra, negocia, invoca e monitora os mesmos. Já o *Broker de Serviços* provê a informação da localização do serviço que

está contida em um registro de serviços que é mantido por ele. *Provedores de Serviços* utilizam o registro para publicar seus serviços e os *Consumidores de Serviços* para localizá-los. *Broker de Serviços* tem um papel cada vez mais proeminente na redução da lacuna entre os requisitos de negócio e tecnologia [Duan et al. 2014]. Fornecedores de tecnologia proveem diversas ferramentas para registro de serviços, tais como: IBM WebSphere Service Registry and Repository[1], Oracle Enterprise Repository[2], Anypoint service registry (MuleSoft)[3], WSO2 Governance Registry[4].

2.2.2. Serviços Web

Serviço Web é a principal tecnologia para implementação de uma arquitetura orientada a serviços [Josuttis 2007]. Um Serviço Web é um sistema de software projetado para apoiar interação interoperável máquina-a-máquina através de uma rede. Ele tem uma interface descrita em um formato processável por máquina. Outros sistemas interagem com o Serviço Web de acordo com o formato de suas mensagens empregando um protocolo de comunicação (tipicamente HTTP) [Booths et al. 2004]. Existem dois tipos de Serviços Web: Serviço Web SOAP (também conhecidos como Serviço Web WS-* ou "Big" Web Service) e Serviço Web RESTful [Pautasso et al. 2008].

2.2.2.1. Serviços Web SOAP

Serviços Web SOAP empregam um conjunto de tecnologias baseadas em XML[5] (Extensible Markup Language), tais como SOAP, WSDL, XSD e UDDI.

XML descreve uma classe de objetos de dados chamados de documentos XML e parcialmente descreve o comportamento de programas que os processam. Documentos XML são feitos de unidades de armazenamento chamadas de entidades, as quais contem caracteres "parsed" e caracteres não "parsed", alguns dos quais formam dados de caracteres e outros formam *markups*. *Markups* codificam uma descrição do *layout* e da estrutura lógica do armazenamento de documentos, provendo um mecanismo para definir restrições sobre os mesmos.

XSD (XML Schema Definition Language) oferece mecanismos para descrever a estrutura e restrições do conteúdo de documentos XML, incluindo aquele que exploram o mecanismos de Namespace. Namespace é um mecanismo para quebrar esquemas em subconjuntos de forma a obter definições reutilizáveis por mais de um projeto.

SOAP[6] (Simple Object Access Protocol) é um protocolo leve que tem o objetivo de troca de informações estruturadas em um ambiente descentralizado e distribuído. Ele usa as tecnologias XML para definir um *framework* de mensagens extensíveis, provendo um construto de mensagem que pode ser trocado sobre protocolos variados. Este

[1] https://developer.ibm.com/integration/docs/wsrr/

[2] https://www.oracle.com/middleware/technologies/enterprise-repository.html

[3] https://www.mulesoft.com/resources/esb/service-registry-repository

[4] https://wso2.com/products/governance-registry/

[5] https://www.w3.org/TR/xml/

[6] https://www.w3.org/TR/soap12-part1/

framework foi projetado para ser independente de qualquer modelo de programação particular e outas semânticas específicas de implementação.

WSDL[7] (Web Service Description Language) é uma linguagem baseada em XML para especificação do contrato de serviços SOAP. Um documento WSDL de um serviço descreve as operações disponíveis pelo serviço, os esquemas dos tipos de dados referentes aos parâmetros de entrada e de retorno das operações, os *endpoints* onde os serviços estão disponibilizados, o formato de troca de mensagens, o protocolo de comunicação etc. As definições de serviços em WSDL proveem documentação para clientes consumirem os serviços em um ambiente distribuído e serve como uma “receita"para automatizar os detalhes envolvidos na comunicação de aplicações.

A Listagem 2.1 apresenta um exemplo de implementação de Serviço Web SOAP em Java. Nas linhas 3, 4 e 5 temos os *imports* necessários para as anotações *@WebService*, *@WebMethod* e *@WebParam* do pacote JAX-WS (API java para Serviços Web XML). A especificação da API JAX-WS[8] define um padrão para fazer o mapeamento *Java-WSDL*, isto é, a especificação determina como um método Java é invocado e como a mensagem SOAP é mapeada nos parâmetros dos métodos. Este mapeamento também determina como o valor de retorno do método é mapeado em uma mensagem de resposta SOAP. JAX-WS usa anotações para simplificar o desenvolvimento e a implantação de Serviços Web e clientes de Serviços Web. JAX-WS 2.0 substituiu JAX-RPC buscando Serviços Web com mensagens no formato de documento [Hewitt 2009]. *@WebService* é utilizado (Linha 7) para definir a classe *StrManagementSOAPWS* como um serviço disponibilizado como serviço *StrManagementWS*. *@WebMethod* é utilizado (Linha 10) para anotar o método `concat` para ser invocado quando a operação *concatenate* for invocada. *@WebParam* é utilizado (linhas 11 e 12) para definir o nome dos parâmetros do método *str1* and *str2*, respectivamente.

```
1  package org.str.management;
2
3  import javax.jws.WebService;
4  import javax.jws.WebMethod;
5  import javax.jws.WebParam;
6
7  @WebService(serviceName = "StrManagementSOAPWS")
8  public class StrManagementSOAPWS {
9
10   @WebMethod(operationName = "concatenate")
11   public String concat(@WebParam(name = "str1") String str1,
12               @WebParam(name = "str2") String str2) {
13     return str1 + str2;
14   }
15 }
```

Listing 2.1. Exemplo de implementação de Serviço Web SOAP em Java.

Um cliente para o serviço implementado em Java é apresentado na Listagem 2.2. Tanto o serviço como o cliente foram implementados empregando o NetBeans[9], o qual gera o código e as anotações para o serviço e, para o cliente, gera os artefatos java necessários para invocar o serviço.

[7] `https://www.w3.org/TR/wsdl/`
[8] `https://en.wikipedia.org/wiki/Java_API_for_XML_Web_Services`
[9] `https://netbeans.org/`

```
1  package org.str.management;
2
3  public class StrManagementSOAPclient {
4
5    public static void main(String[] args) {
6      String str = concatenate("Curso ", "SOA");
7      System.out.println(str);
8    }
9
10   private static String concatenate(java.lang.String str1, java.lang.String str2) {
11     org.str.management.client.artifacts.StrManagementSOAPWS_Service service = new org
       .str.management.client.artifacts.StrManagementSOAPWS_Service();
12     org.str.management.client.artifacts.StrManagementSOAPWS port = service.
       getStrManagementSOAPWSPort();
13     return port.concatenate(str1, str2);
14   }
15 }
```

Listing 2.2. Exemplo de implementação de cliente em Java para o Serviço Web SOAP.

O código e os slides que explicam as implementações estão disponíveis na página GIT do curso [Azevedo 2020][10]. Neste mesmo local, estão disponíveis outros exemplos de código e aulas para implementação de Serviços Web SOAP, como, por exemplo, implementação de serviços empregando NetBeans e SGBD PostgreSQL[11].

2.2.2.2. Serviço Web REST

Serviço Web REST (Representational State Transfer) é uma alternativa mais simples a Serviços Web SOAP. O estilo REST define um conjunto de restrições/princípios que os serviços devem seguir a fim de alcançar usabilidade, simplicidade, escalabilidade e extensibilidade [Li and Chou 2011]. A ideia central deste estilo é projetar aplicações com baixo acoplamento que se baseiam em recursos, os quais correspondem a qualquer informação que pode ser acessada/manipulada.

REST foi introduzido em 2000 por Roy Fielding em sua tese de doutorado na Universidade de Califórnica, Irvine [Fielding 2000]. Não atraiu muita atenção na época, mas hoje é amplamente utilizado como, por exemplo, pelo Yahoo, Google e Facebook, e tem se estabelecido como o mecanismo para comunicação de sistemas distribuídos [Haupt et al. 2017];

REST[12] é apresentado pela W3C como sendo um subconjunto da Web baseado em HTTP nos quais agentes proveem semântica de interface uniforme - essencialmente criar, recuperar, atualizar e apagar - ao invés de interfaces arbitrárias ou específicas de aplicação, e manipulam recursos apenas pela troca de representações. Além disso, as interações REST são sem estado (*stateless*) no sentido de que o significado da mensagem não depende do estado da conversação.

REST não impõe restrições no formato da mensagem, como SOAP para Serviços Web SOAP, mas sim apenas no comportamento dos componentes envolvidos. Dessa forma, o desenvolvedor tem maior flexibilidade em optar pelo formato de mensagem que

[10] pasta *web-services-soap*
[11] `https://www.postgresql.org/`
[12] `https://www.w3.org/TR/ws-arch/\#relwwwrest`

atenda melhor as suas necessidades. Os formatos mais comuns são JSON[13] (Java Script Object Notation), XML e texto puro, mas em teoria qualquer formato pode ser usado. A principal característica de Serviços Web REST é usar explicitamente os métodos HTTP (POST, GET, PUT, DELETE) para denotar a invocação de diferentes operações.

Serviços Web REST se baseiam nos seguintes princípios [Varanasi and Belida 2015] [Fielding 2000]: (i) Separação clara entre as responsabilidades de cliente e servidor a fim de que eles possam evoluir independentemente; (ii) Usar explicitamente os métodos HTTP; (iii) Ser sem estado: chamadas para o serviços devem conter todas as informações necessárias para executá-lo. O servidor não deve armazenar qualquer contexto e dados de sessão devem ser armazenados do lado do cliente; (iv) Expor URIs como uma estrutura de diretório; (v) Respostas do serviço devem ser declaradas como *cacheable* ou não *cacheable*; (vi) Transferir XML, JSON, ou ambos.

A Listagem 2.3 apresenta um exemplo de implementação de Serviço Web REST em Java. Nas linhas 3 a 8, temos os *imports* de JAX-RS (API Java para Serviços Web REST) necessárias à implementação. JAX-RS usa anotações para desenvolvimento de Serviços Web REST as quais permitem o mapeamento de uma classe de recurso (um POJO - Plain-Old Java Object) como um recurso Web[14]. A anotação *@Path* (Linha 10) especifica o caminho relativo de uma classe ou método do recurso. A anotação *@Context* (Linha 13) retorna o contexto inteiro de um objeto, por exemplo, o código *@Context HttpServletRequest request*) retorna o contexto inteiro do objeto *HttpServletRequest*. A anotação *@GET* especifica o método HTTP GET. A anotação *@Produces* (Linha 23) especifica o tipo de mídia de resposta, por exemplo, *text/html*. A anotação *@QueryParam* faz a ligação de um parâmetro do método a um parâmetro (Query) de consulta HTTP.

Para que a uma classe que representa um recurso possa ser disponibilizada como um serviço a configuração apresentada na Listagem 2.4 é necessária. Este código define quais Serviços Web serão disponibilizados na aplicação, o que é feito via a adição da classe do recurso (Linha 17).

```
1  package strmanagement;
2
3  import javax.ws.rs.core.Context;
4  import javax.ws.rs.core.UriInfo;
5  import javax.ws.rs.Produces;
6  import javax.ws.rs.GET;
7  import javax.ws.rs.Path;
8  import javax.ws.rs.QueryParam;
9
10 @Path("strmanagement")
11 public class StrManagementResource {
12
13   @Context
14   private UriInfo context;
15
16   /**
17   * Creates a new instance of StrManagementResource
18   */
19   public StrManagementResource() {
20   }
21
22   @GET
23   @Produces("text/html")
```

[13] https://www.json.org/

[14] https://en.wikipedia.org/wiki/Java_API_for_RESTful_Web_Services

```
24    public String getHtml(
25      @QueryParam("str1") String str1,
26      @QueryParam("str2") String str2){
27        return "<html><body><h1>"+str1+str2+"</body></h1></html>";
28    }
29
30  }
```

Listing 2.3. Exemplo de implementação de Serviço Web REST.

```
1  package strmanagement;
2
3  import java.util.Set;
4  import javax.ws.rs.core.Application;
5
6  @javax.ws.rs.ApplicationPath("webresources")
7  public class ApplicationConfig extends Application {
8
9    @Override
10   public Set<Class<?>> getClasses() {
11     Set<Class<?>> resources = new java.util.HashSet<>();
12     addRestResourceClasses(resources);
13     return resources;
14   }
15
16   private void addRestResourceClasses(Set<Class<?>> resources) {
17     resources.add(strmanagement.StrManagementResource.class);
18   }
19 }
```

Listing 2.4. Configuração necessária para disponibilização Serviço Web REST.

Um cliente para o serviço REST implementado em Java é apresenta na Listagem 2.5 e na Listagem 2.6. A primeira listagem corresponde ao código necessário para configurar o acesso Serviço Web alvo (linhas 8 a 24) e para fazer a invocação (Linha 25). A segunda listagem, basicamente, é um exemplo de uso da implementação anterior através de um console Java.

```
1  package concatclientapplication;
2
3  import javax.ws.rs.ClientErrorException;
4  import javax.ws.rs.client.Client;
5  import javax.ws.rs.client.WebTarget;
6
7  public class StrManagementResourceClient {
8    private final WebTarget webTarget;
9    private final Client client;
10   private static final String BASE_URI = "http://localhost:8080/
       StrManagementRESTWSApp/webresources";
11
12   public StrManagementResourceClient() {
13     client = javax.ws.rs.client.ClientBuilder.newClient();
14     webTarget = client.target(BASE_URI).path("strmanagement");
15   }
16
17   public String getHtml(String str1, String str2) throws ClientErrorException {
18     WebTarget resource = webTarget;
19     if (str1 != null) {
20       resource = resource.queryParam("str1", str1);
21     }
22     if (str2 != null) {
23       resource = resource.queryParam("str2", str2);
24     }
25     return resource.request(javax.ws.rs.core.MediaType.TEXT_HTML).get(String.class);
26   }
27 }
```

Listing 2.5. Exemplo de implementação de cliente para o Serviço Web REST.

```
1  package concatclientapplication;
2
3  public class ConcatClientApplication {
4
5    public static void main(String[] args) {
6      StrManagementResourceClient strManagement=new StrManagementResourceClient();
7
8      String str = strManagement.getHtml("Barmenia", "Versicherungen");
9
10     System.out.println("concat: " + str);
11   }
12 }
```

Listing 2.6. Exemplo de implementação de invocação do Serviço Web REST.

Existem diferentes tecnologias para especificação de contratos de serviços REST, tais como: Swagger (ou OpenAPI)[15], WADL[16], WSDL2.0[17], e API Blueprint[18]. A principal delas é Swagger, cuja terceira versão foi renomeada para especificação OpenAPI (OAS - OpenAPI Specification) [OpenAPI initiative 2018]. OpenAPI é a linguagem mais popular para especificação de contratos REST [Tsouroplis et al. 2015] e a iniciativa OpenAPI provê um *framework* completo para criação da documentação e geração de código do cliente e do servidor a partir da especificação. Existem várias ferramentas para especificação de contratos REST em Swagger/OpenAPI. Santos *et al.* apresentam uma análise das ferramentas cujos resultados ajudar engenheiros de software a escolher as ferramentas mais adequadas e apresentam lacunas a serem tratadas em iniciativas de pesquisa [Santos et al. 2020].

Os códigos fontes e os slides que explicam os conceitos em detalhes e implementações de serviços REST estão disponíveis na página curso [Azevedo 2020][19].

2.3. Arquitetura de Microsservicos

Esta seção apresenta os principais conceitos de microsserviços, tais como: definição e como microsserviços e SOA estão relacionados (Seção 2.3.1); princípios (Seção 2.3.2); comparação entre arquitetura monolítica e arquitetura de microsserviços (Seção 2.3.3); algumas principais tecnologias para implantação de microsserviços (Seção 2.3.4); e, um exemplo de implantação de um Serviço Web em contêiner Docker (Seção 2.3.5).

2.3.1. Definição

A Arquitetura de Microsserviços (MSA), ou simplesmente microsserviços, surgiu empiricamente a partir de padrões arquiteturais utilizados no mundo real, onde sistemas são compostos por serviços que colaboram entre si para atingir seus objetivos, se comunicando a partir de mecanismos leves (como Web APIs) [Fowler and Lewis 2014, Newman 2015]. É uma proposta de construção de pequenas aplicações, desenvolvidas

[15] http://swagger.io
[16] https://www.w3.org/Submission/wadl/
[17] https://www.w3.org/TR/2007/REC-wsdl20-20070626/
[18] http://raml.org
[19] Pasta *web-services-rest*

de forma independente, que tendem a apresentar aspectos de processamento e de interoperabilidade eficientes, permite a implantação/entrega contínua de aplicações grandes e complexas. Ela permite que uma organização evolua em sua *stack* de tecnologia de maneira mais fácil.

Apoiadores de microsserviços afirmam que este é um novo estilo arquitetural [Richards 2015]; em contraste, aqueles que defendem SOA argumentam que microsserviços é uma abordagem de implementação e implantação de SOA [Zimmermann 2017]. Neste trabalho, seguimos a definição de que microsserviços é uma forma de implementação e implantação de serviços em SOA empregando práticas do estado da arte de engenharia de software de acordo com revisão feita na literatura [Zimmermann 2017]. Esta revisão concluiu que as diferenças entre microsserviços e tentativas anteriores de aplicar SOA não estão relacionadas ao estilo arquitetural (isto é, restrições e intenções de projeto e princípios de independência de plataforma e padrões), mas sim na realização da arquitetura (por exemplo, paradigmas de desenvolvimento e implantação e tecnologias).

O termo microsserviços apareceu inicialmente em blogs e artigos online a partir de 2014; Lewis e Fowler [Fowler and Lewis 2014] apresentam uma coleção de postagens sobre o tema. A partir de 2016 que microsserviços começaram a aparecer em workshops e conferências [Zimmermann 2017].

Arquitetura de Microsserviços é uma abordagem para o desenvolvimento de uma única aplicação formada por um conjunto de pequenos serviços (microsserviços), cada um executando em seu próprio processo (por exemplo, isolado em um contêiner) e se comunicando através de algum mecanismo de pequeno porte, geralmente uma API HTTP. Esses serviços são construídos em torno de uma parte específica do negócio (projetado usando técnicas de projeto orientadas ao domínio ou Domain-Driven Design -DDD) e são implantados de forma completamente automatizada (utilizando técnicas de automação e conteinerização). Eles podem ser escritos em diferentes linguagens de programação e usar diferentes tecnologias de banco de dados (isto é, múltiplos paradigmas e tecnologia que melhor se aplica como solução). Existe uma camada mínima centralizada de gerenciamento desses serviços [Fowler and Lewis 2014].

O estilo de disponibilização de funcionalidades como microsserviços é análogo aos utilitários UNIX, os quais são desenvolvidos cada um com uma finalidade específica e podem ser combinados para executar tarefas complexas. Por exemplo, combinar os utilitários *grep*, *cat* e *find* através de um *script shell* [Richardson 2016].

2.3.2. Princípios

Há sete princípios atribuídos a microsserviços [Zimmermann 2017, Fowler and Lewis 2014, Newman 2015]:

- Desenvolver microsserviços como unidades com **interfaces de granularidade fina** com responsabilidade única que encapsulam dados e lógica de processamento e são expostos remotamente, tipicamente através de APIs Web (por exemplo, recursos HTTP RESTful) ou filas de mensagens assíncronas. Estas unidades podem ser implantadas, alteradas, substituídas e escalonadas independentemente uma das outras.
- Usar **práticas de desenvolvimento e linguagens padronizadas orientadas ao ne-**

gócio paraidentificar e conceitualizar serviços como, por exemplo, o Projeto Orientado ao Domínio (Domain-Driven Desing - DDD) [Evans 2004]. Contexto bem definido (ou *Bounded Context*) é o padrão central do DDD [Vernon 2013]), o qual divide grandes domínios em contextos menores. Através do DDD equipes de desenvolvimento priorizam de forma sistemática os aspectos mais distintos e valiosos para a organização;

- Seguir **princípios de projeto baseados em computação em nuvem**, por exemplo, *IDEAL* (isolamento, distribuição, elasticidade, gestão automatizada e baixo acoplamento)[Fehling et al. 2014].

- Empregar **múltiplos paradigmas** de computação e armazenamento (isto é, usar estratégia de programação e persistência poliglota) [Wampler and Clark 2010] que melhor se apliquem ao desenvolvimento da solução.

- Usar **contêineres leves** para disponibilizar serviços para promover artefatos através de processos de entrega contínua (*e.g.*, Docker [Merkel 2014]).

- Empregar técnicas de **entrega contínua descentralizada** para promover um alto nível de automação e autonomia. Isto demanda maturidade para construir artefatos de forma automatizada, aliados a uma suíte de testes [Humble and Farley 2010].

- Empregar ***DevOps*** através do uso de técnicas e automatização de configuração, desempenho e gerenciamento de falhas, o qual estende práticas ágeis e incluem monitoramento de serviços [Hüttermann 2012].

2.3.3. Arquitetura Monolítica x Arquitetura de Microsserviços

Microsserviços são uma alternativa às aplicações monolíticas onde uma única aplicação contém toda a lógica do negócio e gerenciamento de dados [Richardson 2015]. Neste caso, mesmo a aplicação tendo uma arquitetura interna dividida em módulos, ela é empacotada e disponibilizada inteiramente, por exemplo, um WAR[20] para arquivos Java ou aplicações Java empacotadas como executáveis JAR.

Estes tipos de aplicações são fáceis de desenvolver por IDEs e outras ferramentas focadas em construir uma única aplicação. Elas também são fáceis de serem testadas e distribuídas. Inclusive elas podem ser escalonadas colocando-se múltiplas cópias em múltiplos servidores e empregando balanceamento de carga.

Problemas começam a surgir quando a aplicação "monolítica" cresce tornando-se extremamente complexa. Corrigir erros (*bugs*) e implementar novas funcionalidades corretamente tornam-se difíceis e consomem muito tempo. A carga cognitiva para entender o código é grande. Quando à implantação, a aplicação monolítica grande pode demorar a iniciar e alterações no código ao longo do dia, demandando reimplantação (que deve ser inteira) pode demandar muito tempo para a aplicação e suas cópias (quando há escalonamento) estarem ativas [Richardson 2015].

[20]Web application ARchive (WAR) é um formato de arquivo para distribuir, por exemplo, uma coleção de JavaServer Pages, Servlets Java, classe Java para ser implantado em servidor de aplicação como o Tomcat ou Glassfish.

Em aplicações monolíticas, seus módulos executam no mesmo processo. Logo, erro em um módulo pode derrubar toda a aplicação. A adoção de novos *frameworks* também torna-se um problema, por não poder facilmente ser executada em uma parte específica do código o qual não está inteiramente dependente.

Para solucionar estes problemas, muitas empresas, tais como, Amazon, Netflix, The Guardian e outros estão usando arquitetura de microsserviços [Di Francesco et al. 2017], cuja ideia é dividir a aplicação em um conjunto de serviços menores interconectados. Um microsserviço tipicamente implementa um conjunto de funcionalidades do negócio. Cada microsserviço é uma "mini-aplicação", construída independentemente expondo/-consumindo funcionalidades para/de outros microsserviços. Alguns microsserviços podem implementar interfaces Web permitindo implantar experiências distintas (independentes) para usuários ou dispositivos específicos ou para casos de uso específicos.

A fim de reduzir o acoplamento, em geral, cada microsserviço tem seu próprio banco de dados. Eventualmente, dados podem ter que ser replicados e/ou tecnologias de bancos de dados distintas serem empregadas, como, por exemplo, um banco de dados que executa consultas espaciais com alto desempenho [Sadalage and Fowler 2012].

A arquitetura de microsserviços trata o problema da complexidade através da decomposição da aplicação monolítica em um conjunto de microsserviços. Ela inverte a lei de Conway que afirma que organizações que projetam sistemas ficam limitadas a produzir projetos que são cópias da estrutura de comunicação da organização [Conway 1968]. Em microsserviços, times são organizados de forma a serem responsáveis por um único microsserviços desde o desenvolvimento até a implantação. Portanto, isto mitiga os problemas de comunicação de organizações trabalhando em bases de código grandes identificada na lei de Brook do mítico homem mês [Brooks Jr 1995].

Existem várias vantagens da abordagem de microsserviços. O escopo menor de cada microsserviço minimiza a carga cognitiva dos desenvolvedores porque o código base de trabalho é pequeno e fácil de entender. Código base menor facilita antecipar o impacto de mudanças. A equipe também tem liberdade de escolher a tecnologia que melhor resolve a tarefa; enquanto que, em um código base grande de uma aplicação monolítica, os desenvolvedores de um módulo podem escolher as melhores ferramentas desde que estejam de acordo com a linguagem e os *frameworks* do monólito. Como microsserviço foca em uma funcionalidade específica, fica mais fácil prever o seu comportamento quanto às características de execução, tais como, CPU e escrita em disco (*I/O*), quanto ao escalonamento de recursos de acordo com as requisições, quanto à sensibilidade à latência, se ele é sem estado e como ele é favorável a balanceamento de carga. Portanto, operadores tem mais informações para tomar decisões de como alocar recursos baseados nos características de cada microsserviço [Azevedo et al. 2019].

Existem também desvantagens para microsserviços. Do ponto de vista do desenvolvedor, perde-se a capacidade de usar a depurador da IDE[21] para observar todas as interações com outras partes do sistema. Microsserviços requer o uso pesado de ferramentas para *logging*, rastreamento e monitoramento de desempenho para dar a visibilidade do que está acontecendo. Do ponto de vista do operador, a gerência de configuração re-

[21] Integrated Development Environment

quer executar e atualizar dezenas ou centenas de de microsserviços colaborando o que é desafiante. Plataformas de orquestração de contêineres, *middlewares* e ferramentas de integração contínua e metodologias como DevOps ajudam a sobrecarga dos operados, mas trazem complexidade que deve ser tratada é ainda muito maior do que em arquiteturas monolíticas [Azevedo et al. 2019].

Não existe bala de prata e a arquitetura de microsserviços tem questões importantes a serem consideradas. Apesar de ser indicado que microsserviços sejam pequenos, o objetivo é que microsserviços correspondam a decomposições da aplicação a fim de facilitar desenvolvimento e implantação ágeis. Uma arquitetura de microsserviços corresponde a um sistema distribuído com comunicação inter processo, o que é mais complexo do que invocar métodos no nível da linguagem, sendo também mais complexo de testar, por exemplo, é necessário implementar testes de integração entre os microsserviços. Particionamento do banco de dados entre os microsserviços e persistência poliglota traz o problema de garantia de integridade dos dados distribuídos, entre outros desafios. Newman apresenta soluções para integração de dados com bancos de dados poliglotas [Newman 2015]. Villaça *et al.* analisam as soluções indicadas por Newman e indica o uso de modelo de dados canônico como uma possível solução para este problema. Eles se baseiam em padrões definidos antes do termo microsserviços ser definido, mas cujas características tem muita relação com este paradigma [Villaça et al. 2018a]. Villaça *et al.* ilustram o uso destes padrões [Villaça et al. 2018b] e Villaça *et al.* apresentam uma proposta de integração de dados provenientes de fontes de dados heterogêneas empregando modelo dados canônico [Villaça et al. 2020]. Além disso, Azevedo *et al.* [Azevedo et al. 2019] apresentam um exemplo de como fazer a integração de dados empregando o modelo CQRS (Command-Query Responsability Segregation) [Fowler 2011] em um cenário real na área de Óleo & Gás.

2.3.4. Tecnologias para implantação de microsserviços

Microsserviços são desenvolvidos como Serviços Web (Seção 2.2.2), principalmente, Serviços Web RESTful como ressaltado em no princípio "interfaces de granularidade fina" (Seção 2.3.2). Dentre as tecnologias mais utilizadas para implantação e orquestração de microsserviços temos, por exemplo, Docker, Docker Swarm, Kubernetes, Apache Mesos e ZooKeeper, que são tecnologias que seguem o princípio "contêineres leves". Ferramentas para Integração contínua (*continuous integration* - CI) e entrega contínua (*continuous deployment* - CD) são amplamente utilizadas, tais como, Jenkins, Hudson e Chef. Integração contínua envolve automatizar a integração de código em um repositório compartilhado uma vez ao dia ou mais e entrega contínua refere-se a automatizar a implantação de pacotes do software no ambiente de implantação - princípios "entrega contínua descentralizada'" e "DevOps". Ferramentas para monitoramento de serviços incluem, por exemplo, as ferramentas Elastic[22] - elasticsearch, logstack e kibana.

[22] `https://github.com/elastic/`

2.3.4.1. Contêineres

Contêiner é um termo que descreve uma alternativa mais leve às máquinas virtuais. Para isso é feito o encapsulamento da aplicação em um ambiente virtual que contém apenas os ativos necessários para o funcionamento. Os contêineres são isolados a nível de disco, memória, processamento e rede. Essa separação permite uma grande flexibilidade, onde ambientes distintos podem coexistir na mesma máquina hospedeira (*host*), sem causar problemas.

Comparando contêiner com maquina virtual, temos que cada máquina virtual requer um Sistema Operacional próprio além dos softwares e bibliotecas. Além disso uma camada intermediária (chamada *hypervisor*) gerencia a comunicação de cada máquina virtual com o sistema operacional hospedeiro. Já os contêineres acessam diretamente o sistema operacional hospedeiro e seus recursos (por exemplo, disco, memória, rede) para prover um ambiente virtual para as aplicações. Portanto, no ambiente de contêiner, é necessário instalar os requisitos (softwares, arquivos etc.) que o microsserviço precisa sem se preocupar com instalações de outro Sistema Operacional, além de não precisar do *hypervisor*.

Dentre as vantagens de contêineres temos: (i) flexíveis: permitem empacotar diversos tipos de aplicações; (ii) leves: compartilham o *kernel* da máquina hospedeira sendo mais rápidos do que máquinas virtuais; (iii) portáveis: são construídos localmente, mas podem ser implantados na Cloud e executar em qualquer lugar; (iv) têm baixo acoplamento: são alto suficientes e encapsulados, permitem serem substituídos e atualizados em impactar outros contêineres; (v) escaláveis: replicas de contêineres podem ser aumentadas e distribuídas automaticamente; (vi) seguros: possuem restrições e isolamento.

2.3.4.2. Docker

Docker[23] é uma plataforma aberta criada utilizando o modelo de contêiner para "empacotar" a aplicação, que após ser transformada em uma imagem Docker, poderá ser reproduzida em plataforma de qualquer porte. O objetivo é facilitar o desenvolvimento, implantação e execução de aplicações em ambientes isolados da forma mais rápida possível. Uma imagem Docker provê tudo o que é necessário para executar uma aplicação, por exemplo, código ou binário, *runtimes*, dependências e objetos de sistemas de arquivo necessários.

Na página do Docker, é apresentado um tutorial[24] para instalar o ambiente, construir uma imagem e executá-la como um contêiner, configurar e usar as ferramentas de orquestração Kubernetes e Swarm e compartilhar aplicações conteinerizadas no Docker Hub. Nesta seção, apresentamos algumas características que ajudam a realizar este tutorial.

Em geral, o fluxo de desenvolvimento de uma aplicação conteinerizada inclui os seguintes passos: (i) Criar e testar contêineres individualmente para cada componente de

[23] `https://www.docker.com/`
[24] `https://docs.docker.com/get-started/`

sua aplicação iniciando pela criação de imagens Docker; (ii) Combinar seus contêineres e infraestrutura de apoio em uma aplicação completa, expressa como um arquivo Docker (i.e., *docker stack file*) ou um arquivo YAML Kubernetes; (iii) Testar, compartilhar e implantar sua aplicação conteinerizada completa.

O Dockerfile é um arquivo que descreve o passo-a-passo para se construir uma imagem de contêiner Docker. Uma imagem sempre deve partir de uma imagem base. Portanto, o Dockerfile descreve de uma forma textual a diferença entre a imagem base e a imagem que se deseja criar, isto é, o Dockerfile contém a sequência de instruções necessárias para modificar a imagem base para que ela fique com as características desejadas.

A Listagem 2.7[25] apresenta um exemplo de Dockerfile. O comando da linha 1 indica que a imagem *node:6.11.5* é a imagem base. Esta imagem é uma imagem oficial, construída pelos fornecedores do noje.js e validade pelo Docker, contendo o interpretador node 6.11.5 e dependências básicas. O comando da linha 3 usa *WORKDIR* para indicar que todas as ações subsequentes serão feitas no diretório */usr/src/app* do sistema de arquivos da imagem (e nunca na máquina hospedeira). O comando da linha 4 usa o comando *COPY* para copiar o arquivo *package.json* para a raiz, isto é, */usr/src/app*. O comando da linha 5 executa o comando *npm install* dentro do sistema de arquivos da imagem, o qual lerá o arquivo *package.json* para identificar as dependências do código fonte da aplicação e instalá-las no sistema de arquivo da imagem. Na linha 6, o comando copia todo o restante do conteúdo do diretório corrente da máquina hospedeira para o sistema de arquivos da imagem. Finalmente, na linha 7, usando *CMD* são passados metadados para a imagem descrevendo como executar um contêiner baseado nesta imagem. Estes metadados informam que o processo conteinerizado que esta imagem apoia é *npm start*, ou seja, indica ao Docker para executar *npm start* quando o contêiner inicia. Existem várias outras diretivas[26] para construção de imagens Docker.

```
1  FROM node:6.11.5
2
3  WORKDIR /usr/src/app
4  COPY package.json .
5  RUN npm install
6  COPY . .
7
8  CMD [ "npm", "start" ]
```

Listing 2.7. Exemplo de Docker file.

Usando o comando *build*, a imagem é construída a partir do *Dockerfile* e o comando *run* inicia o container a partir da imagem e comando *rm* apaga o contêiner. (Listagem 2.8[27]). O parâmetro *publish* indica ao Docker para encaminhar para o contêiner na porta 8080 todo o tráfego que chega na porta 8000. O parâmetro *detach* indica que o contêiner deve executar em *background*. O parâmetro *name* define o nome *bb* como o nome do contêiner para ser usado em comandos futuros.

```
1  docker image build -t bulletinboard:1.0 .
2
3  docker container run --publish 8000:8080 --detach --name bb bulletinboard:1.0
4
```

[25] https://docs.docker.com/get-started/part2/
[26] https://docs.docker.com/engine/reference/builder/
[27] https://docs.docker.com/get-started/part2/

```
5 docker container rm --force bb
```

Listing 2.8. Exemplos de comandos para criar imagem docker e iniciar contêiner.

Outra ferramenta importante de Docker é o Docker Hub[28] o qual provê um recurso centralizado para descoberta, distribuição e gestão de mudanças de imagens de contêineres, colaboração de usuários, e automação de *workflow* através de um *pipeline* de desenvolvimento.

2.3.4.3. Kubernetes

Kubernetes proveem várias ferramentas para escalonar, colocar na rede, tornar seguro e manter uma aplicação conteinerizada. Todos os contêineres em Kubernetes são programados como *pods* que são grupos de contêineres co-localizados que compartilham alguns recursos. Objetos Kubernetes são descritos em manifestos chamados de arquivos *Kubernetes YAML*. Estes arquivos descrevem todos os componentes e configurações de uma aplicação Kubernetes e são usados para criar e destruir a aplicação em qualquer ambiente Kubernetes. Aplicações são programadas como *deployments*, os quais, em geral, são grupos de *pods* mantidos automaticamente pelo Kubernetes.

Um arquivo Kubernetes YAML, como ilustrado na Listagem 2.9[29], em geral, contém: *apiVersion*, que indica a API Kubernetes que faz o *parser* do objeto; *kind*, que indica o tipo de objeto, por exemplo, *Deployment* ou *Service*; *metadata*, como, por exemplo, nome de objetos; *spec*, que especifica os parâmetros e configurações do objeto.. No exemplo, o objeto *Deployment* define uma única réplica do pod, o qual é descrito a partir da chave *template*, em apenas um contêiner basedo na imagem *bulletinboard:1.0*. O serviço *NodePort* faz o roteamento do tráfego da porta 30001 da máquina hospedeira para a porta 8080 dentro dos pods permitindo alcançar a aplicação *bulletin board* a partir da rede.

```
1  apiVersion: apps/v1
2  kind: Deployment
3  metadata:
4    name: bb-demo
5    namespace: default
6  spec:
7    replicas: 1
8    selector:
9      matchLabels:
10       bb: web
11     template:
12       metadata:
13         labels:
14           bb: web
15       spec:
16         containers:
17         - name: bb-site
18            image: bulletinboard:1.0
19 ---
20 apiVersion: v1
21 kind: Service
22 metadata:
23   name: bb-entrypoint
```

[28] https://docs.docker.com/docker-hub/

[29] https://docs.docker.com/get-started/part3/

```
24    namespace: default
25  spec:
26    type: NodePort
27    selector:
28      bb: web
29    ports:
30    - port: 8080
31      targetPort: 8080
32      nodePort: 30001
```

Listing 2.9. Exemplos arquivo Kubernetes YAML.

A Listagem 2.10 apresenta comandos para: fazer a implantação da aplicação (*apply*); verificar se o pod está executando (*get deployments*); verificar se o serviço está executando (*get services*). Caso esteja executando, teste o serviço acessando *localhost:30001* no Browser; destruir a aplicação (*delete*).

```
1  kubectl apply -f bb.yaml
2
3  kubectl get deployments
4
5  kubectl get services
6
7  kubectl delete -f bb.yaml
```

Listing 2.10. Exemplos de comandos para implantação da aplicação com Kubernetes.

2.3.5. Implantando Serviços com Docker

A Listagem 2.11 e a Listagem 2.12 apresentam exemplo de código para disponibilizar os pacotes dos serviços (isto é, os arquivos *WAR* gerados quando do deploy dos serviços) em uma imagem Docker. Os arquivos *Dockerfile*, *start.sh* e *WAR* estão disponíveis na página do curso [Azevedo 2020] pasta *microservices* de exemplos de empacotamento de serviços via contêiner.

```
1
2  FROM glassfish:latest
3
4  COPY StrManagementRESTWSApp.war /
5  COPY start.sh /
6
7  EXPOSE 8080
8
9  ENTRYPOINT ["/start.sh"]
```

Listing 2.11. Implantação do serviço REST usando Docker.

```
1
2  #!/bin/sh
3
4  /usr/local/glassfish-4.1/bin/asadmin start-domain
5  /usr/local/glassfish-4.1/bin/asadmin -u admin deploy /aot.war
6  /usr/local/glassfish-4.1/bin/asadmin stop-domain
7  /usr/local/glassfish-4.1/bin/asadmin start-domain --verbose
```

Listing 2.12. Arquivo para iniciar o servidor.

```
1
2  docker image build -t glassfish:latest .
3
4  docker container run --publish 8000:8080 --detach --name wsrest glassfish:latest
```

Listing 2.13. Comandos para criar a imagem e iniciar o contêiner.

2.4. Cloud Computing

Esta seção apresenta a definição de Cloud Computing e suas características essenciais (Seção 2.4.1), os modelos de implantação (Seção 2.4.2), modelos de serviço (Seção 2.4.3) e um exemplo de plataforma de Cloud Computing (o IBM Cloud - Seção 2.4.4.)

2.4.1. Definição

Cloud Computing é um modelo para permitir acesso ubíquo, conveniente, sob demanda a um conjunto compartilhado de recursos de computação (por exemplo, rede, servidores, armazenamento, aplicações e serviços) que podem ser rapidamente provisionados e liberados com esforço mínimo de gerência ou interação com o provedor do serviço [Mell et al. 2011].

A característicasessenciais deste modelo são [Mell et al. 2011]:

- **Autoatendimento sob demanda**: As capacidades de computação (por exemplo, tempo de servidor e armazenamento) são provisionados automaticamente sem interação humana com o provedor do serviço.
- **Acesso amplo à rede**: Recursos são disponibilizados na rede e acessados através de mecanismos padronizados via plataforma cliente leve (por exemplo, telefones móveis, *tablet*, *laptops*).
- **Agrupamento de recursos**: Recursos são agrupados para servirem múltiplos consumidores usando um modelo multi-cliente. Eles são dinamicamente atribuídos de acordo com a demanda do consumidor.
- **Elasticidade rápida**: Capacidades podem ser provisionadas e liberadas elasticamente.
- **Serviço de medida**: O uso dos recursos é monitorado, controlado e comunicado, provendo transparência para ambos consumidor e provedor.

2.4.2. Modelos de Implantação da Cloud

Os modelos de implantação da Cloud são:

- **Cloud privada**: Cloud de uso exclusivo por uma única organização envolvendo múltiplos consumidores (por exemplo, unidades de negócio). A Cloud é de propriedade, gerenciada e operada pela própria organização ou por terceiro ou pela combinação de ambos. Ela pode estar hospedada na organização ou fora dela.
- **Cloud comunitária**: Cloud de uso exclusivo de uma comunidade composta por organizações com necessidades e preocupações compartilhadas (por exemplo, políticas de segurança). Cloud de propriedade, gerenciada e operada pela própria comunidade ou por terceiro ou pela combinação de ambos. Ela pode estar hospedada na organização ou fora dela
- **Cloud pública**: Cloud de uso aberto para o público em geral. Ela é de propriedade, gerenciada e operada por uma organização de negócio, acadêmica ou governamental ou pela combinação dos mesmos. Ela existe hospedada no provedor da Cloud.

- **Cloud híbrida**: Composição de duas ou mais infraestruturas distintas da Cloud que permanecem como entidades únicas. Infraestruturas são combinadas por tecnologia padronizada ou proprietária.

2.4.3. Modelos de Serviço da Cloud

Os principais modelos de serviço disponíveis em Cloud Computing são IaaS (Infrastructure as a Service), PaaS (Platform as a Service) e SaaS (Software as a Service) [Mell et al. 2011]:

- **Infrastructure-as-a-Service (IaaS)**: O provedor fornece recursos de computação fundamentais como, por exemplo, processamento, armazenamento e rede. Os consumidores ficam aptos a implantar e executar software arbitrário, tais como sistemas operacionais e aplicações. Exemplos de provedores IaaS são: IBM Cloud[30], Amazon Web Services[31] (AWS), Microsoft Azure[32], Google Cloud[33] OpenNebula[34].

- **Platform-as-a-Service (PaaS)**: O provedor fornece recursos a consumidores para criar ou implantar aplicações criadas usando estes recursos como, por exemplo, linguagens de programação, bibliotecas, serviços e ferramentas de implantação. Recursos compatíveis provenientes de outras fornecedores também podem ser usados. Exemplos de PaaS são : IBM Cloud, Google Cloud, SalesForce.com[35].

- **Software-as-a-Service (SaaS)**: O provedor fornece capacidades para consumidores usarem aplicações executando na infraestrutura da Cloud. Estas aplicações são acessíveis de vários dispositivos clientes usando uma camada de cliente leve (por exemplo, web browser ou API). Exemplos de SaaS são: Google docs[36], Office 365[37], Gmail[38].

O tempo é crítico para as aplicações de hoje em dia as quais devem ter velocidade e agilidade para atender às necessidades dos seus funcionários e/ou clientes e estarem a frente de seus competidores. A Tabela 2.1 ilustra os modelos de serviço em relação a o que é gerenciado pelo cliente e o que é gerenciado pelo provedor da cloud.

O núcleo de TI (ou ***core IT***) de uma organização representa tudo o que ela possui e gerencia em suas centrais de dados (*data centers*). Isto é ainda uma parte crítica da organização. Ela tem **benefícios**, tais como: (i) permite se ter uma versão estável e customizável de acordo com as necessidades do cliente, apesar de ser limitada aos custos; (ii) é necessária para alguns casos (por exemplo, processamento de certas transações); (iii) considera muito dos investimentos já feito pelas organizações (por exemplo, sistemas

[30] https://www.ibm.com/cloud
[31] https://aws.amazon.com/
[32] https://azure.microsoft.com/\textbf{text}
[33] https://cloud.google.com/
[34] https://opennebula.org/
[35] https://www.salesforce.com/
[36] https://docs.google.com/
[37] https://products.office.com/
[38] https://www.gmail.com/

Tabela 2.1. Recursos gerenciados pelo cliente (em fundo branco) e recursos gerenciados pelo provedor da Cloud (em fundo cinza) de acordo com os modelos de serviço da Cloud.

TI principal)	IaaS	PaaS	SaaS
Código	Código	Código	Código
Dados	Dados	Dados	Dados
Runtime	*Runtime*	*Runtime*	*Runtime*
Middleware	*Middleware*	*Middleware*	*Middleware*
S.O.	S.O	S.O.	S.O.
Virtualização	Virtualização	Virtualização	Virtualização
Servidores	Servidores	Servidores	Servidores
Armazenamento	Armazenamento	Armazenamento	Armazenamento
Rede	Rede	Rede	Rede

e dados). Por outro lado, há um grande **comprometimento de tempo** neste caso e como **limitações** temos: (i) tipicamente leva semanas para configurar e implantar uma aplicação inicial; (ii) a organização deve manter e atualizar hardware e software, por exemplo, *packs* de serviços, antivírus e *patches*; (iii) custo pode ser alto; (iv) é necessário equipe dedicada; (v) há dificuldades em experimentar novas tecnologias.

IaaS permite implantação rápida, configuração rápida do ambiente etc. ao abstrair a infraestrutura das tarefas do cliente. Como **benefícios** de IaaS temos: (i) rede, armazenamento, servidores e virtualização são gerenciados pelo provedor de serviços; (ii) oferta da Cloud é mais customizável; (iii) são soluções onde é necessário customização de sistema operacional, *middleware* e *runtime*. Por outro lado, o **comprometimento de tempo** é menor, pois leva-se minutos para prover uma nova máquina virtual e como **limitações** temos: (i) cliente configura e gerencia sistema operacional, *middleware* e *runtime* o que pode demandar dias de trabalho; (ii) manutenções e atualizações também são necessários, tais como, *packs* de serviços, antivírus e *patches*.

Existem casos em que clientes necessitam maior agilidade do que as possibilidades anteriores e não precisam gastar tempo gerenciando plataforma (por exemplo, máquinas virtuais, sistemas operacionais e *runtime*). Neste caso, **PaaS** é uma solução mais adequada. Ela traz como **benefícios**: (i) rapidez na configuração de ambientes e implantação de aplicações; (ii) o provedor do serviços gerencia infraestrutura e plataforma. Quanto ao **comprometimento de tempo**, leva-se minutos para configurar e implantar aplicações. O cliente foca nas aplicações e nos seus dados, ou seja, construir e manter aplicações.

Finalmente, em uma **SaaS**, encontramos como benefícios: (i) infraestrutura, plataforma e aplicações são gerenciados pelo provedor de serviços, o que inclui, por exemplo, atualizações de software, consertos de hardware e ajustes da rede; (ii) a aplicação está totalmente da Cloud e podem ser acessadas de diferentes *endpoints*, dependendo do provedor de serviços. Quanto ao **comprometimento de tempo**, o foco do cliente está no **uso** das aplicações.

2.4.4. IBM Cloud

IBM Cloud[39] corresponde a uma plataforma para desenvolver, implantar, executar e gerenciar aplicações. Ela combina PaaS e IaaS. Ela possui mais de 190 serviços para serem utilizados para construir aplicações. Contém um conjunto de dados e ferramentas avançadas de IA. A plataforma é capaz de apoiar tanto times pequenos de desenvolvimento e organizações de pequeno porte como negócios de grandes empresas.

Dentre os principais componentes da plataforma, temos:

- Console: Permite criar, visualizar e gerenciar recursos da cloud;
- Gerente de identidade e acesso: Permite autenticação segura de usuários;
- Catálogo de aplicações;
- Mecanismos de busca e rotulação: Facilitam filtrar e identificar recursos;
- Camada de disponibilização (*provisioning*): Permite controlar e gerenciar recursos;
- Cobrança: Permite gestão de conta unificada, precificação, medição e relatórios de uso.

O IBM Cloud oferece várias opções para servidores de aplicação, tais como:

- Servidores *Bare Metal*: servidores dedicados de apenas um *tenant* onde nada é compartilhado, incluindo recursos dos servidores, com outros clientes;
- Servidores virtuais: Servidores escaláveis que são comprados com allocação de memória e *cores*.
- Soluções VMware: Permite integrar ou migrar cargas de trabalho VMware da empresa (*on-premise* usando infraestrutura escalável, segura e de alto desempenho e tecnologia de virtualização híbrida da VMware;
- Serviços Kubernetes: Combina contêineres Docke, tecnologia Kubernetes e segurança e isolamento integrados para automatizar implantação, operação, escalonamento e monitoramento de aplicações conteinerizadas em um cluster de computação.
- IBM Cloud Foundry: Permite instanciar múltiplas plataformas Cloud Foundry sob demanda de forma isolada.
- Funções Cloud: Plataforma Function-as-a-Service (FaaS) baseado em Apache OpenWhisk.

Como opções de armazenamento, a plataforma oferece:

- Block Storage: Armazenamento iSCI com persistência em alto desempenho provida e gerenciada independente de instâncias de computação.

[39] `https://www.ibm.com/cloud`

- File Storage: Persistência rápida e flexível atrelada à rede baseada em NFS.

- IBM Cloud Object Storage: Armazenamento de informações usando tecnologia IBM Cloud Object Storage. O armazenamento é encriptado e espalhado em múltiplas localizações geográficas e acessado via API REST.

- IBM Cloud Master Data Management: Armazenamento *offload* de grandes quantidades de dados das centrais de dados da empresa para Cloud Object Storage.

- IBM Cloud Backup: Ferramenta para fazer backup de dados entre servidores localizados na rede da IBM Cloud. O agente de backup é automatizado e gerenciado através de uma ferramenta no Browser.

Para uso da plataforma, pode-se criar uma conta[40] "lite" com vários serviços livres que podem ser usados para experimentar o ambiente. Para serviços mais avançados é necessário um plano pago. No entanto, existe ferramenta para estimar custos[41]. A plataforma oferece *stencils*[42] (i.e., *templates*) para criar diagramas da arquitetura criada usando ferramentas populares de diagramação.

Após o usuário logar na plataforma[43], ele entra no *Dashboard* onde ele pode criar recursos, criar aplicações, adicionar usuários e aprender [44] sobre a plataforma. Uma boa forma de iniciar é através dos tutoriais disponíveis[45], por exemplo o tutorial "Aplicações Web Escaláveis no Kubernetes"[46] que engloba os conhecimentos apresentados neste tutorial ou o tutorial "Entrega Contínua para o Kubernetes"[47]. Também pode-se iniciar o aprendizado a partir de uma linguagem de programação como, por exemplo, Java[48].

No catálogo de aplicação existem várias categorias classificadas como serviços e/ou software, tais como:

- Inteligência artificial: (i) Tradutor de idiomas (Language Translator): tradutor de texto, documentos e páginas Web; (ii) Ideias para literatura médica (Insights for Medical Literature): pesquisa em um corpus da literatura médica e descobre conhecimento/ideias; (iii) Catálogo de Conhecimento (Knowledge Catalog): permite realizar descobertas, catalogar e compartilhar de forma segura os dados da empresa e auxilia em ciência de dados e conformidade de dados; (iv) Watson Studio: provê um conjunto de ferramentas e um ambiente colaborativo para cientistas de dados, desenvolvedores e especialistas de domínio. Pode-se utilizar ferramentas de código aberto como RStudio ou Jupyter Notebooks e serviços do Watson para criar modelos flexíveis. Trata as fases de preparação de dados, preparação dos dados

[40] `https://cloud.ibm.com/registration`
[41] https://github.com/ibm-cloud-architecture/infrastructure-design-decision-tool/
[42] `https://github.com/ibm-cloud-architecture/ibm-cloud-stencils`
[43] `https://cloud.ibm.com/login`
[44] `https://cloud.ibm.com/docs`
[45] `https://cloud.ibm.com/docs/tutorials/index.html`
[46] `https://cloud.ibm.com/docs/tutorials/index.html`
[47] https://cloud.ibm.com/docs/tutorials?topic=solution-tutorials-continuous-deployment-to-kubernetes
[48] `https://cloud.ibm.com/docs/java?topic=java-getting-started`

para treinamento e treinamento. Permite gerenciar dados, ativos analíticos e projetos na Cloud. (v) Aprendizado de Máquina (Machine Learning): disponibiliza frameworks de machine learning como serviços REST, tais como: TensorFlow, Keras, Caffe, PyTorch, Spark MLlib, scikit learn, xgboost and SPSS.

- Infraestrutura: bare metal server, virtual server, HPCaaS from Rescale, Power Systems Virtual Server, WebSphere Application Server, Kubernetes Service, Red Had OpenShitf Cluster, Container Registry.
- Armazenamento: Box, File Storage, IBM Cloud Backup, Object Storage, Osnexus, Portworx Enterprise.
- Banco de Dados: Bockchain Platform, Cloudant, Databases for PostgreSQL, Databases for Redis, Databases for Elasticsearch, Databases for MongoDB, Messages for RabbitMQ, Databases for etcd, Blockchain, Compose Enterprise, Compose for JanusGraph, Compose for MySQL, Db2, db2 Hosted, SQL Query etc.
- Ferramentas de desenvolvimento: Actifio Go, Availability Monitoring, Continous Delivery, Event Management etc.
- Ferramentas de rede: Load Balancers, Content Delivery Network, Direct Link Connect, Direct Link Dedicated etc.

O IBM Cloud possui um conjunto de coleções de código para iniciar ("Starter Kits") como, por exemplo, Java Microservice com Spring, Java Web App com Spring, Node.js Microservice com Express.js, Node.js Web App com Express.js, Watson Natural Language, Watson Visual Recognition etc.

Para o desenvolvimento de aplicações existem dois tipos de interface: IBM Cloud Web Console e IBM Cloud CLI (Command Line Interface ? Interface de Linha de Comando). O código desenvolvido deve ser portável, isto é:

(i) Deve estar em uma linguagem padronizada;

(ii) Deve executar na Cloud;

(iii) Deve conectar a serviços da Cloud;

(iv) Deve atender a um caso de uso específico.

Uma aplicação pode ser criada a desde o início na plataforma ou pode-se trazer código previamente implementado de uma aplicação existente para o IBM Cloud.

Em geral os passos para desenvolvimento são: (i) Criar aplicação a partir do portal do desenvolvedor; (ii) Criar uma cadeia de implantação DevOps (usando DevOps tools chain); (iii) Usar ferramentas de desenvolvimento local, i.e., suas próprias ferramentas para desenvolver; (iv) Enviar as atualizações para serem combinadas com o código no repositório remoto; (v) A implantação é feita usando a cadeia de DevOps.

Para iniciar o uso do IBM Cloud, por exemplo, execute os passos a seguir:

- Instale a ferramenta de linha de comando[49], por exemplo, a Listagem 2.14 apresenta o comando para instalar a ferramenta (Linha 1), aceitar a licença do *XCode*, caso ainda não tenha feito (Linha 3) e verificar se a instalação foi feita com sucesso (Linha 3).

- Em seguida, pode-se iniciar, criando uma aplicação (indo ao *Dashboard*, então *App* e *Create an App*) ou usando um *kit* de iniciante[50] ou a partir de uma linguagem de programação[51]. Vamos iniciar criando um microsserviço com o perfil MicroProfile do Eclipse e Java EE. Clique no link [52] e marque *Java + Liberty*. Em seguida, clique em *pattern* na caixa correspondente a este perfil.

```
1 curl -sL http://ibm.biz/idt-installer | bash
2
3 sudo xcodebuild -license accept
4
5 ibmcloud dev help
```

Listing 2.14. Comandos de configuração do ibm Cloud CLI no Mac/Linux

2.5. Prática

Esta seção descreve a parte prática a ser executada neste minicurso. Detalhes de instalação, *links* para páginas de softwares, cenário dos exercícios, exemplos de implementação, apresentações, dicas dentre outros recursos estão disponíveis na página do minicurso [Azevedo 2020]. O material está disponível no *bitbucket*[53] que é uma ferramenta Git[54] para gestão de código fonte. A seguir são apresentados informações sobre o material. Para acessar o material, o aluno pode ir diretamente na página do curso ou baixar o material localmente em seu computador usando um cliente Git.

- *Instalações*: Dicas e passo-a-passos de instalação estão disponíveis na pasta `instalacoes`. Alguns softwares a serem instalados são:
  - Git[55];
  - Maven[56];
  - Uma IDE (Integrated Development Environment) para implementação em Java, por exemplo, o NetBeans[57];
  - Docker Desktop[58];

[49] `https://cloud.ibm.com/docs/home/tools`

[50] `https://cloud.ibm.com/docs/apps/tutorials?topic=creating-apps-tutorial-starterkit`

[51] `https://cloud.ibm.com/docs/home/build`

[52] `https://cloud.ibm.com/developer/appservice/starter-kits`

[53] `https://bitbucket.org/product/`

[54] Git é um sistema de controle de versões distribuído, usado principalmente no desenvolvimento de software, mas pode ser usado para registrar o histórico de edições de qualquer tipo de arquivo.

[55] `https://git-scm.com/downloads`

[56] `a`

[57] Baixe de `https://netbeans.org/downloads/8.0.1` a versão Java EE

[58] `https://www.docker.com/products/docker-desktop`

– Postman[59] para realizar testes com os serviços implementados.

- *Cenário*: o cenário para realizar o trabalho está disponibilizado na pasta `trabalhos`. O objetivo é que o mesmo cenário seja usado para todos os exercícios e trabalhos com diferentes níveis de dificuldade.

- *Exercícios e trabalhos*: estão disponibilizados na pasta `trabalho`. O `readme` desta pasta indica a melhor ordem para aprendizado de alunos iniciantes, intermediários e avançados.

- *Materiais específicos por assunto*: as pastas `web-service`, `microservice` e `cloud` incluem apresentações, tutoriais, artigos e dicas sobre cada assunto. O `readme` de cada uma das pastas apresenta detalhes de cada um dos recursos disponibilizados nas mesmas.

2.6. Considerações Finais

Este minicurso apresentou uma visão teórica e prática de SOA, Arquitetura de Microsserviços e Cloud Computing servindo como guia prático para o aprendizado dos artefatos e ferramentas destas abordagens. Em geral estes conceitos aparecem espalhados na literatura em diversos documentos, livros, artigos, postagens de blogs etc. Neste minicurso foi dada uma visão geral, mas com nível de aprofundamento, que permite que o aluno adquira fundamentação nas três abordagens e tenha ponteiros para se aprofundar nos aspectos que mais lhe interesse.

O aprendizado destes paradigmas é essencial para o desenvolvimento de sistemas de informação modernos bem como a aplicação destes conceitos na indústria e na academia. Ainda há muitos desafios e questões a serem respondidos. No entanto, sem base teórica e prática as direções podem ser tomadas de maneira inadequada.

Dessa forma, este minicurso complementa disciplinas de graduação e pós-graduação na área de desenvolvimento de sistemas distribuídos. Ele não aborda diretamente todos os conceitos como, por exemplo, questões de desempenho, segurança, aplicação detalhada dos princípios das abordagens em diferentes domínios, os quais se constituem tópicos interessantes para serem tratados em trabalhos futuros.

Referências

[Azevedo 2020] Azevedo, L. G. (2020). Desenvolvimento de Soluções com Serviços. `https://bitbucket.org/leogazevedo/curso-soa`. Acessado em 21/02/2020.

[Azevedo et al. 2019] Azevedo, L. G., Ferreira, R. d. S., Silva, V. T. d., de Bayser, M., Soares, E. F. d. S., and Thiago, R. M. (2019). Geological Data Access on a Polyglot Database Using a Service Architecture. In *XIII Brazilian Symposium on Software Components, Architectures, and Reuse (SBCARS)*, pages 103–112. ACM.

[59] `https://www.postman.com/downloads/`

[Booths et al. 2004] Booths, D., Haas, H., McCabe, F., Newcomer, E., Champion, M., Ferris, C., and Orchard, D. (2004). Web Services Architecture. `http://www.w3.org/TR/ws-arch/`. Acessado em 21/02/2020.

[Brooks Jr 1995] Brooks Jr, F. P. (1995). *The Mythical Man-Month: Essays on Software Engineering*. Pearson Education India.

[Conway 1968] Conway, M. E. (1968). How do committees invent. *Datamation*, 14(4):28–31.

[Di Francesco et al. 2017] Di Francesco, P., Malavolta, I., and Lago, P. (2017). Research on Architecting Microservices: Trends, focus, and potential for industrial adoption. In *2017 IEEE International Conference on Software Architecture (ICSA)*, pages 21–30. IEEE.

[Duan et al. 2014] Duan, Y., Narendra, N. C., Du, W., Wang, Y., and Zhou, N. (2014). Exploring Cloud Service Brokering from an Interface Perspective. In *2014 IEEE International Conference on Web Services (ICWS)*, pages 329–336. IEEE.

[Erl 2005] Erl, T. (2005). *Service-Oriented Architecture: Concepts, Technology, and Design*. Pearson Education India.

[Evans 2004] Evans, E. (2004). *Domain-Driven Design: Tackling Complexity in the Heart of Software*. Addison-Wesley Professional.

[Fehling et al. 2014] Fehling, C., Leymann, F., Retter, R., Schupeck, W., and Arbitter, P. (2014). *Cloud Computing Patterns: Fundamentals to Design, Build, and Manage Cloud Applications*. Springer Science & Business Media.

[Fielding 2000] Fielding, R. T. (2000). REST: Architectural Styles and the Design of Network-Based Software Architectures. *Doctoral dissertation, University of California*.

[Fowler 2011] Fowler, M. (2011). Command Query Responsibility Segregation (CQRS). `https://martinfowler.com/bliki/CQRS.html`. Acessado em 21/02/2020.

[Fowler and Lewis 2014] Fowler, M. and Lewis, J. (2014). Microservices. `https://martinfowler.com/articles/microservices.html`. Acessado em 21/02/2020.

[Gu and Lago 2007] Gu, Q. and Lago, P. (2007). A stakeholder-driven service life cycle model for soa. In *2nd international workshop on Service oriented software engineering: in conjunction with the 6th ESEC/FSE joint meeting*, pages 1–7. ACM.

[Haupt et al. 2017] Haupt, F., Leymann, F., Scherer, A., and Vukojevic-Haupt, K. (2017). A framework for the structural analysis of REST APIs. In *Software Architecture (ICSA), 2017 IEEE International Conference on*, pages 55–58. IEEE.

[Hewitt 2009] Hewitt, E. (2009). *Java SOA Cookbook: SOA Implementation Recipes, Tips, and Techniques*. "O'Reilly Media, Inc.".

[Humble and Farley 2010] Humble, J. and Farley, D. (2010). *Continuous Delivery: Reliable Software Releases through Build, Test, and Deployment Automation (Adobe Reader)*. Pearson Education.

[Hüttermann 2012] Hüttermann, M. (2012). *DevOps for Developers*, volume 1. Springer.

[Josuttis 2007] Josuttis, N. M. (2007). *SOA in practice: the art of distributed system design*. O'Reilly Media, Inc.

[Li and Chou 2011] Li, L. and Chou, W. (2011). Design and describe REST API without violating REST: A Petri net based approach. In *2011 IEEE International Conference on Web Services (ICWS)*, pages 508–515. IEEE.

[Marks and Bell 2008] Marks, E. A. and Bell, M. (2008). *Service-Oriented Architecture (SOA): a planning and implementation guide for business and technology*. John Wiley & Sons.

[Mell et al. 2011] Mell, P., Grance, T., et al. (2011). The NIST Definition of Cloud computing. Technical report, Computer Security Division, Information Technology Laboratory, National Institute of Standards and Technology.

[Merkel 2014] Merkel, D. (2014). Docker: lightweight linux containers for consistent development and deployment. *Linux Journal*, 2014(239):2.

[Newman 2015] Newman, S. (2015). *Building Microservices: Designing Fine-grained Systems*. O'Reilly Media, Inc.

[OpenAPI initiative 2018] OpenAPI initiative (2018). OpenAPI Specification. `https://github.com/oai/openapi-specification/blob/master/versions/3.0.1.md`.

[Papazoglou and Van Den Heuvel 2007] Papazoglou, M. P. and Van Den Heuvel, W.-J. (2007). Service Oriented Architectures: approaches, technologies and research issues. *The VLDB journal*, 16(3):389–415.

[Pautasso et al. 2008] Pautasso, C., Zimmermann, O., and Leymann, F. (2008). RESTful Web Services vs. Big Web Services: making the right architectural decision. In *17th International Conference on World Wide Web (WWW 2008)*, pages 805–814. ACM.

[Richards 2015] Richards, M. (2015). *Microservices vs. Service-Oriented Architecture*. O'Reilly Media.

[Richardson 2015] Richardson, C. (2015). Introduction to Microservices. `https://www.nginx.com/blog/introduction-to-microservices/`. Acessado em 21/02/2020.

[Richardson 2016] Richardson, C. (2016). Microservice Architecture Patterns and Best Practices. `http://microservices.io`. Acessado em 21/02/2020.

[Sadalage and Fowler 2012] Sadalage, P. J. and Fowler, M. (2012). *NoSQL distilled: a brief guide to the emerging world of polyglot persistence*. Pearson Education.

[Santos et al. 2020] Santos, J. S., Azevedo, L. G., Soares, E. F. S., Thiago, R. M., and Silva, V. T. (2020). Analysis of Tools for REST Contract Specification in Swagger/OpenAPI. In *2nd International Conference on Enterprise Information Systems (ICEIS 2020)*. INSTICC.

[Tsouroplis et al. 2015] Tsouroplis, R., Petychakis, M., Alvertis, I., Biliri, E., and Askounis, D. (2015). Community-based API Builder to manage APIs and their connections with Cloud-based Services. In *CAiSE Forum*, pages 17–23.

[Varanasi and Belida 2015] Varanasi, B. and Belida, S. (2015). *Introduction to REST*, pages 1–13. Apress, Berkeley, CA.

[Vernon 2013] Vernon, V. (2013). *Implementing Domain-Driven Design*. Addison-Wesley.

[Villaça et al. 2018a] Villaça, L. H., Azevedo, L. G., and Baião, F. (2018a). Query strategies on polyglot persistence in microservices. In *2018 ACM SIGAPP*. ACM.

[Villaça et al. 2020] Villaça, L. H., Azevedo, L. G., and Moreno, M. (2020). Microservice Architecture for Multistore Database Using Canonical Data Model. In *Proceedings of the XIV Brazilian Symposium on Software Components, Architectures, and Reuse*. ACM.

[Villaça et al. 2018b] Villaça, L. H., Pimenta Jr., A. F., and Azevedo, L. G. (2018b). Construindo aplicações distribuídas com microsserviços. In *Tópicos em Sistemas de Informação: Minicursos XV Simpósio Brasileiro de Sistemas de Informação*. SBC.

[Vinoski 2002] Vinoski, S. (2002). Putting the Web into Web services. Web services interaction models, part 1. *IEEE Internet Computing*, 6(3):89–91.

[Wampler and Clark 2010] Wampler, D. and Clark, T. (2010). Multiparadigm programming: guest editors' introduction. *IEEE Software*, 27(5):2–7.

[Zimmermann 2017] Zimmermann, O. (2017). Microservices tenets. *Computer Science-Research and Development*, 32(3-4):301–310.

Biografia do autor

Leonardo G. Azevedo é pesquisador da IBM Research Brazil desde 2013. Ele é Doutor (2005) e Mestre (2001) pelo PESC/COPPE/UFRJ e Bacharel em Informática (2000) pela UFRJ. Ele foi professor da UniRio de 2006 a 2018, lecionando disciplinas em Banco de Dados e Engenharia de Software e atuou no Programa de Pós-Graduação em Informática (PPGI/UNIRIO). Leonardo tem mais de 20 anos de experiência em pesquisa e desenvolvimento de sistemas e vem atuando em projetos para diferentes empresas nacionais e internacionais e governo. Suas áreas de pesquisa incluem Sistemas Distribuídos, Arquitetura Orientada a Serviços (SOA), Microsserviços, Representação do Conhecimento, Ontologias, Gestão de Processos de Negócio (BPM), Computação Cognitiva, e Banco de Dados Espaciais. Para maiores detalhes http://researcher.ibm.com/researcher/view.php?person=br-lga e http://lattes.cnpq.br/7214791464543522.