Implementação de Broker para alta disponibilidade no consumo de software externo via API REST

Pedro Figueiredo Dias[1], Nathan Silva Macena[1], Ye Wei Jiang[1], Claudio Ramos Siqueira Júnior[1], Ana Claudia Rossi[1]

[1]Faculdade de Computação e Informática - Universidade Presbiteriana Mackenzie - São Paulo - SP - Brasil

{pedro.figueiredo.dias, nathan.macena, ye.jiang, claudio.siqueira}@mackenzista.com.br, anaclaudia.rossi@mackenzie.br

Abstract. *Use third-party services through application programming interfaces (API) in a software reduces development time and allows focus on the business specific needs. This practice has become very common between developers and architects, but depending on a single provider can cause financial losses on eventual unavailability of their system. In this research, we propose a solution for passive replication for external software consumed through API REST. We intend to facilitate the usage of multiple providers for the same demand and contribute to maintain high availability goals and user satisfaction based on principles of site reliability engineering (SRE).*

Resumo. *Utilizar serviços de terceiros através de interfaces de programação de aplicações (API) em um software poupa tempo de desenvolvimento e possibilita focar em necessidades próprias do negócio. Essa prática se tornou bastante comum entre os desenvolvedores e arquitetos, porém a dependência em um único provedor pode causar prejuízos em eventual indisponibilidade deste sistema. Nesta pesquisa propomos uma solução de replicação passivo para softwares externos disponibilizados via API REST. Pretendemos facilitar a utilização de múltiplos provedores para uma mesma demanda e contribuir para manter metas de alta disponibilidade e satisfação do cliente, seguindo princípios de engenharia de confiabilidade de sites (SRE).*

1. Introdução

Em engenharia de software, delegar certas funcionalidades que não são o núcleo do negócio para provedores externos permite focar no que realmente pode agregar ao produto [Badidi, 2016]. Por exemplo, funções de disparo de mensagens e pagamento em um e-commerce.

Para viabilizar a contratação, os provedores precisam garantir o bom funcionamento do serviço, com altos níveis de disponibilidade e outros compromissos nos chamados *Service Level Agreements* (SLA), como a definição de penalidades em não cumprimento dos acordos.

A utilização de um único provedor para cada necessidade, no entanto, implica em pontos de falha única no sistema, dado que a garantia de funcionamento em 100% do tempo não é uma realidade, o que pode causar prejuízos [Beyer 2018]. Com base no estudo realizado por [Cerin *et. al.* 2014] sobre o prejuízo que empresas sofreram por conta da indisponibilidade, é possível perceber que os danos chegam a ser catastróficos, mesmo com altas taxas de disponibilidade. A taxa de disponibilidade de um sistema geralmente é calculada com base na quantidade de falhas que interrompem a operação. Um sistema disponível 99% do tempo, por exemplo, teria o total de 3 dias e 15 horas fora do ar no período de um ano. Por mais que seja um número pequeno comparado ao total, serviços com disponibilidades ainda maiores sofreram prejuízo maior, o que é uma característica que varia de acordo com o negócio. Um software com disponibilidade considerada boa tem, no mínimo, "5 noves": 99,999%, o que significa que teria 5 minutos de indisponibilidade por ano [Bass *et al.* 2021].

O objetivo, portanto, é desenvolver uma solução para gerenciar múltiplos provedores e contribuir para manter o nível de disponibilidade desejado por sistemas que dependem de parceiros externos em eventual falha destes.

2. Referencial Teórico

2.1 API REST

Representational State Transfer, ou REST, é uma arquitetura para sistemas de hipermídia distribuídos proposta por Roy Thomas Fielding, baseada nos conceitos de Cliente e Servidor, Stateless (sem estado), Cache, Interfaces Uniformes e Sistemas em camadas [Fielding 2000]. Atualmente esta arquitetura é vastamente utilizada na implementação de APIs (Application Program Interfaces) WEB por meio do protocolo HTTP.

2.2 Proxy

De acordo com [Buschmann *et. al.* 2011] um proxy é um componente que “encapsula toda a funcionalidade de manutenção de componentes em um substituto separado do componente - o proxy - e permita que os clientes se comuniquem apenas por meio do proxy, e não com o próprio componente”. Portanto um proxy expõe a mesma interface pública que um componente para um cliente, porém ocultando todos os processos de manutenção necessários para ambos. E quando expor uma interface idêntica não é possível, utiliza-se do Object Adapter para realizar o mapeamento entre as interfaces.

Há diversos tipos de proxy, tais como: Client Proxy, Business Delegate, Thread Safe Interface, Counting Handle, Virtual Proxy e Firewall Proxy.

2.3 Middleware de aplicação distribuída

De acordo com a obra A pattern language for distributed computing, “middleware é um software de infraestrutura de distribuição que reside entre os aplicativos e os sistemas operacionais subjacentes, pilhas de protocolos de rede e hardware. Sua função principal é preencher a lacuna entre os programas aplicativos e a infraestrutura de hardware e

software de baixo nível, para coordenar como as partes dos aplicativos são conectadas e como elas interoperam" [Buschmann *et. al* 2011]. Sendo este tipo de software útil para a facilitação do desenvolvimento de aplicações, devido ao fato de poder abstrair detalhes de certas implementações de baixo nível, assim como facilitar a integração de recursos e componentes desenvolvidos por diferentes fornecedores.

Dentre as implementações de middlewares de comunicação, destacam-se três padrões utilizados:

- *Messaging*: no qual os diferentes serviços se comunicam através de mensagens que são enviadas em canais e gerenciadas por roteadores
- *Publisher-Subscriber*: onde serviços ou componentes se comunicam de forma assíncrona através de eventos, de maneira que os publishers são responsáveis por gerar eventos, enquanto os subscribers se preocupam em consumir tais eventos.
- *Broker*: neste padrão os componentes interagem através da invocação de métodos remotos, sendo que um conjunto de brokers é capaz de gerenciar os detalhes de comunicação, resultados e exceções entre componentes.

A Tabela 1 apresenta um resumo das características de comunicação e dependência de componentes para os padrões de middlewares de comunicação citados no parágrafo anterior. A exemplo, o padrão *broker* se comunica através da invocação de método remoto, mantendo uma relação de um para um, apresentando dependência da interface dos componentes envolvidos.

Tabela 1. Comparação entre padrões de middleware de aplicação distribuída.

Padrão	Estilo de comunicação	Relacionamento da comunicação	Dependência de componentes
Broker	Invocação de método remoto	Um para um	Interface de componentes
Messaging	Mensagem	Muitos para um	Endpoints de comunicação e formato das mensagens
Publisher-Subscriber	Eventos	Um para muitos	Formato dos eventos

Fonte: [Buschmann *et. al.* 2011]

2.4 Broker

Como descrito anteriormente, *Broker* é um dos padrões de arquitetura de middlewares de aplicação distribuída, para que os componentes se comunicam por invocações de métodos remotos. Porém vale ressaltar que tais invocações são expostas de maneira a torná-las o mais próximo possível de um invocação de um componente local, abstraindo os detalhes de comunicação do cliente. [Buschmann *et. al.* 2011] Um broker também utiliza de outros padrões de arquitetura, para executar suas funções.

Na Figura 1 são listados alguns dos padrões de arquitetura, assim como suas respectivas áreas de atuação dentro do *broker*, tal como o Adaptador de Objetos, que

atua na área de despacho de requisições abstraindo do cliente a decisão de escolha de uma implementação de uma interface, para a adaptação de objetos.

2.5 Modelos de replicação

Replicação é um dos conceitos principais para a alta disponibilidade, em que dados ou recursos são provisionados em diferentes locais (servidores, proxys, caches etc), para torná-los disponíveis caso ocorra a indisponibilidade do seu provedor original [Coulouris 2013]. Há dois principais modelos de replicação:

- Modelo passivo: onde há um provedor principal que atende toda a carga, e o seu backup somente é utilizado em caso de falha;
- Modelo ativo: onde a carga passa por um servidor principal que a distribui para os diferentes provedores de réplicas.

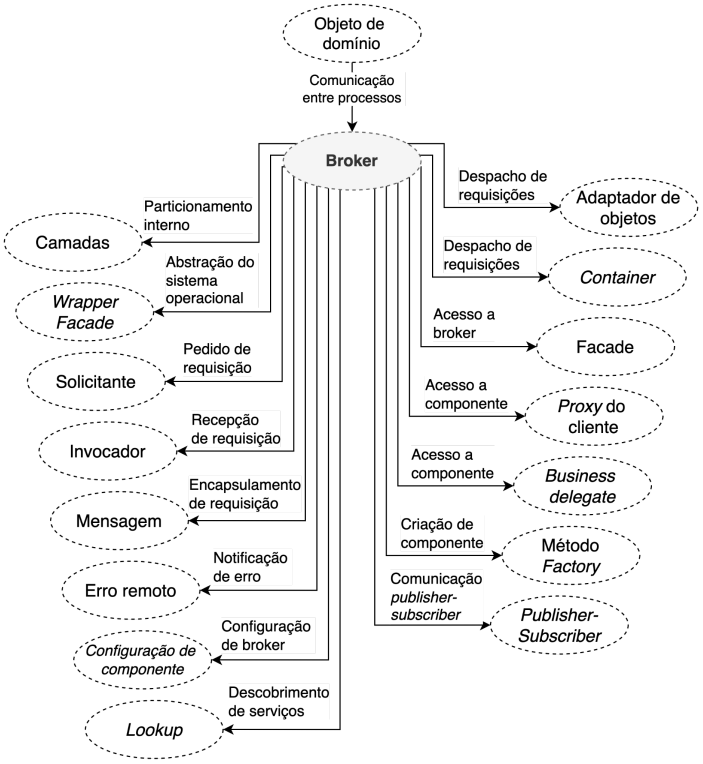

Figura 1. Diagrama das relações do padrão *Broker* com outros padrões de computação distribuída.

Fonte: [Buschmann *et. al.* 2011]

2.6 Engenharia de confiabilidade de sites

Para gerenciar um serviço é necessário conhecer as características importantes deste serviço, a fim de estabelecer níveis de qualidades. Para tal cenário o livro [Beyer *et. al.* 2016] estabelece três importantes conceitos:

- *Service Level Indicator* (SLI): uma métrica quantitativa a respeito do serviço, tal como porcentagem de erro, latência, taxa de transferência e disponibilidade;

- *Service Level Objective* (SLO): uma meta estabelecida com base em um SLI, tal como o tempo de resposta do serviço ser menor que 200 ms;

- *Service Level Agreement* (SLA): um acordo, com os clientes ou usuários finais do produto, das consequências do cumprimento, ou não cumprimento, dos objetivos do serviço (SLO). Entretanto há duas formas de SLA, sendo uma explícita para os usuários e outra implícita.

2.7 Métricas de monitoramento

Para manter a disponibilidade de um sistema é necessário haver métodos de prevenção e recuperação de falhas. Antes que qualquer ação possa ser tomada, é necessário que a falha seja detectada, por exemplo, a técnica *Ping/Echo* [Bass et al. 2021]. Esta técnica consiste em requisições entre sistemas para determinar a acessibilidade e o tempo de resposta. Normalmente essas requisições são efetuadas por um componente de monitoramento e possuem um tempo máximo de resposta configurado. container dos usuário

2.8 Monitoramento não intrusivo

Na proposta de arquitetura de monitoramento não intrusivo para sistemas baseados em microsserviços [Pina *et. al.* 2018] é apresentada uma estratégia de monitoramento onde as métricas, especificamente de requisições HTTP, não são coletadas a partir do cliente, mas sim de um componente por onde todas as requisições passam.

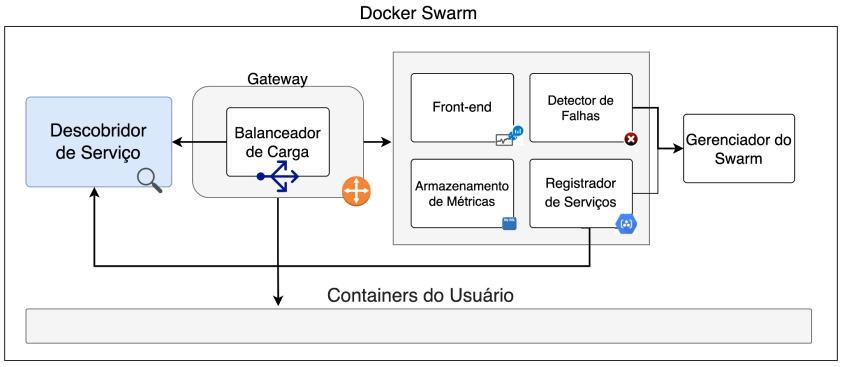

Figura 2. Arquitetura proposta por [Pina et. al. 2018] para monitoramento não intrusivo de microsserviços.

Fonte: [Pina et. al. 2018]

A Figura 2 esquematiza a solução elaborada, baseada na utilização de um *gateway* à frente dos microsserviços para a coleta de métricas, que são encaminhadas para um serviço de armazenamento e análise dos dados. Como resultado, ao utilizar essa estratégia não é necessário realizar modificações em cada microsserviço novo ou modificado.

2.9 Banco de dados de séries temporais

Time Series Database é um banco especializado no armazenamento de séries temporais, que consistem em uma sequência de dados medida em intervalos de tempos, normalmente regulares [Dantale 2012]. Como exemplo pode se tomar uma tabela em que cada linha é identificada por um instante de tempo.

2.10 Arquitetura baseada em espaço

Space-Based Architecture é um padrão de arquitetura de software com foco na escalabilidade de aplicações, a partir da remoção de uma unidade centralizada de armazenamento de dados, e adoção de grids de dados replicados em memória [Richards 2015].

Neste padrão há dois componentes principais, como apresentado na Figura 3:

- *Processing Unit* (Unidade de processamento): Este componente contém todos os módulos da aplicação, junto a estrutura de dados em memória.
- *Virtualized middleware* (Middleware virtualizado): As principais responsabilidades deste componente são controlar a sincronização de dados e gerenciar as requisições, junto às tarefas de escalonamento e orquestração das unidades de processamento, quando necessário.

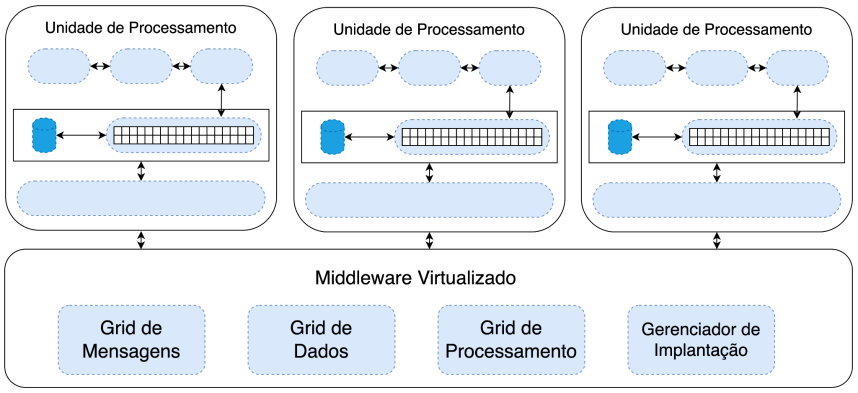

Figura 3. Diagrama da arquitetura de software baseada em espaço.

Fonte: [Richards 2015]

O middleware virtualizado possui os sub-componentes:

- *Messaging Grid* (Grade de Mensagens): responsável por gerenciar requisições e sessões;
- *Data Grid* (Grade de Dados): o principal sub-componente, que mantém a replicação de dados entre as diferentes unidades de processamento;
- *Processing Grid* (Grade de Processamento): sendo opcional, realiza mediação e orquestração das requisições entre diferentes unidades de processamento, quando necessário;

- *Deployment Manager* (Gerenciador de Implantação): gerencia o escalonamento das unidades de processamento, controlando a inicialização e finalização de cada instância.

3. Metodologia de Pesquisa

Essa pesquisa foi feita baseada na metodologia de *design science* proposta por [Wieringa 2014], que consiste em um ciclo de estudo separado por atividades de *design* e de investigação para solucionar um problema do mundo real. O ciclo é um processo racional de resolução de problemas, que compõe-se das seguintes tarefas:

- Investigação do problema
- Proposta de solução para o problema
- Validar a solução
- Implementação do artefato
- Avaliação da implementação

A etapa de Investigação do problema tem como objetivo especificar um problema que pode ser melhorado, além de preparar os dados para serem utilizados na etapa de proposta de solução para o problema.

No processo de proposta de solução para o problema, um ou mais artefatos são propostos para solucionar o problema descrito na primeira parte do ciclo. E a validação do artefato consiste em um período de questionamento sobre se o artefato proposto é capaz de solucionar o empecilho. Um artefato é caracterizado como um algoritmo, método, notação ou técnica com objetivo de resolver algum problema prático.

A implementação do artefato se trata da parte do Design Cycle responsável por desenvolver a aplicação solução do problema original do contexto.

Os passos consistem na iteração dos artefatos dentro do problema do contexto, e essas fases por sua vez são documentadas em especificações, sendo que um design é uma decisão a ser tomada e uma especificação é a documentação dessa decisão.

Por fim, a etapa de Avaliação da implementação qualifica o resultado dos tratamentos anteriores, e tem como objetivo medir o problema do contexto, assim como na etapa de Investigação do problema, porém agora após que os outros passos foram realizados. E com base nos resultados, é possível que ciclo se encerre, ou então comece novamente.

Assim como dissertado na introdução, o problema alvo deste artigo é a interrupção de serviços devido a indisponibilidade de recursos externos. Especificamente este trabalho trata de recursos disponibilizados por meio de APIs REST. Estes recursos podem exercer diversas funções para um sistema, tal como câmbio de moedas, consulta de CEP, envio de emails ou SMS etc.

A interrupção de serviços, mesmo que pequena, pode gerar grandes prejuízos [Cerin *et. al.* 2014]. O que afeta empresas, times de desenvolvimento, times de infraestrutura e clientes do serviço indisponível. Para estes a solução proposta na seção 4 apresenta os benefícios de possuir um mecanismo para mitigação de falhas, menor tempo de desenvolvimento voltado à prevenção de indisponibilidade, menor tempo de investigação e recuperação de falhas e menor impacto aos clientes.

4. Arquitetura Proposta

A proposta de solução é um broker de serviços para gerenciar APIs REST externas. Uma aplicação de e-commerce foi escolhida para exemplificar a interação com o sistema proposto. A Figura 4 exemplifica possíveis ações do cliente com o sistema elaboradas a partir dos casos de uso listados na implementação feita por Suthendra e Pakereng (2020) e contextualiza como a solução proposta interage com o sistema do usuário final.

Os termos recurso e provedor serão usados para descrever a arquitetura proposta. Recurso é uma ação necessária no sistema de e-commerce que é delegada a um parceiro externo, como envio de notificações por email ou gerenciar pagamentos de compras. Provedor é o parceiro capaz de executar a ação. A solução proposta contempla o cadastro de diferentes provedores para cada recurso, que serão acionados com base no modelo de replicação passiva, mencionado por Satheler (2021) e Coulouris (2013).

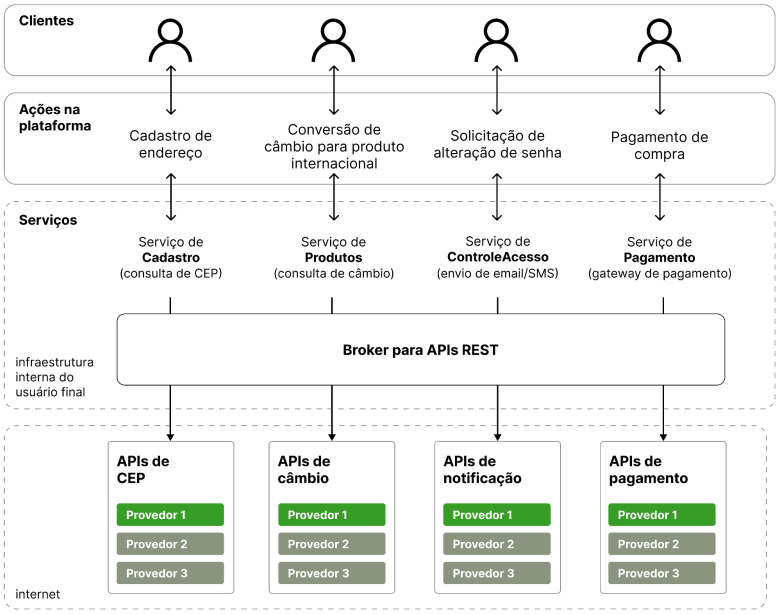

Figura 4. Exemplo da interação de um sistema de e-commerce com o Broker para API REST proposto.

Fonte: Autores.

4.1 Componente Broker

O componente *broker* introduz tolerância a falhas [Ahluwalia e Jain 2006] e faz interface com os serviços da aplicação do usuário final. Ele abstrai para si o

conhecimento de todos os provedores, de modo que os serviços consumidores não precisem ter conhecimentos específicos das APIs REST que irão acessar.

Os serviços da aplicação do usuário final farão requisições HTTP ao broker e deverão especificar o recurso que desejam acessar no formato de URL (*Uniform Resource Locator*): "http://broker/?recurso=pagamento", assim como os cabeçalhos e o corpo da requisição necessários para acessar o recurso.

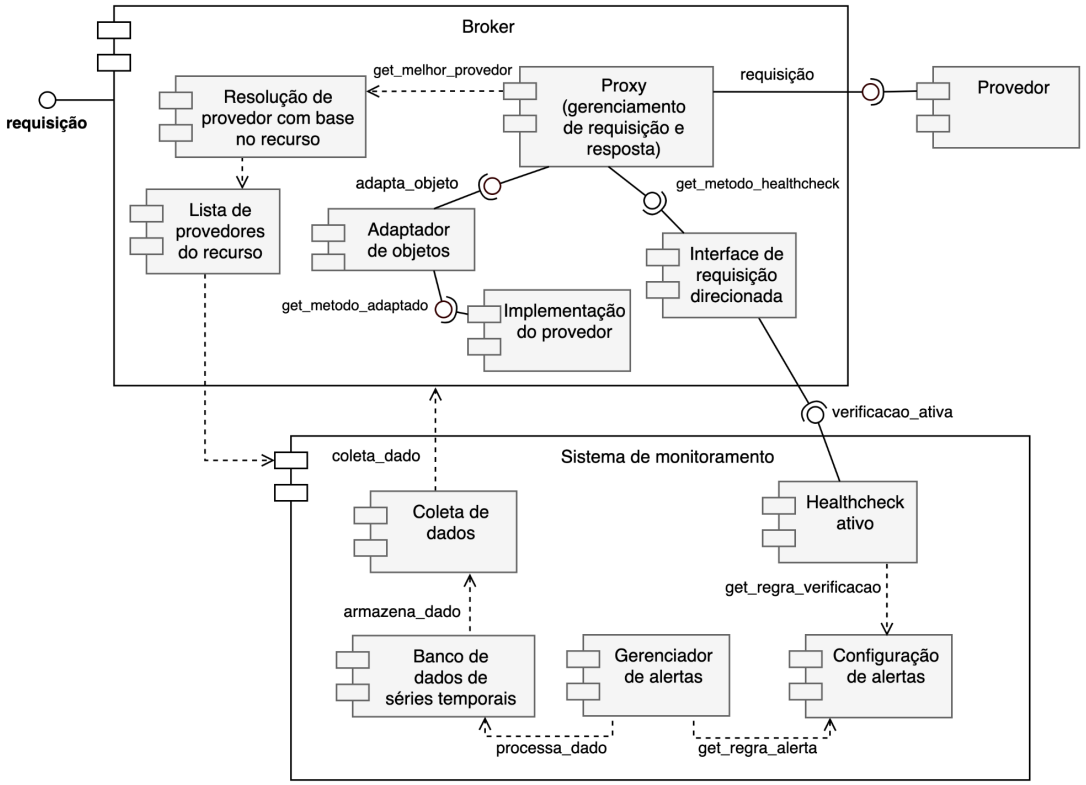

Figura 5. Diagrama de componentes do sistema proposto.

Fonte: Autores.

Os subcomponentes da Figura 5 mostram as necessidades que cada componente precisa para atender às necessidades do serviço consumidor. A Figura 6 detalha o processamento realizado pelo *broker* quando recebe uma requisição.

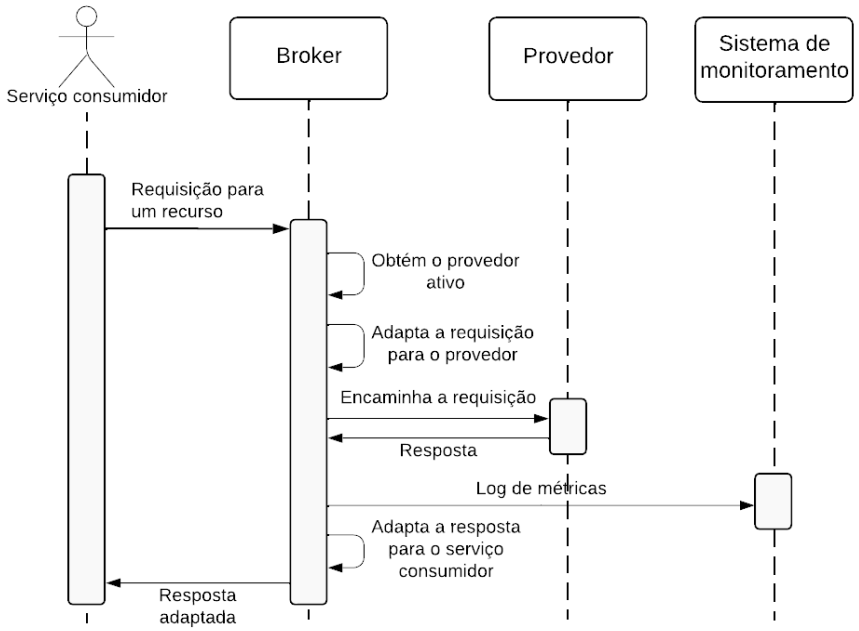

Figura 6. Diagrama de sequência de uma requisição do serviço consumidor no componente broker da arquitetura proposta.

Fonte: Autores.

Para permitir a integração com múltiplos provedores, será necessário também a adaptação dos objetos enviados e recebidos. O usuário final configura o *broker* com um objeto genérico de requisição e resposta, que será utilizado pelos serviços internos. Porém, cada provedor determina uma forma específica de comunicação, como mostra a Figura 7. O *broker* então faz a adaptação do objeto genérico para o específico através de um algoritmo do tipo *Strategy Pattern*, como implementado em [Satheler 2021].

```
Provedor A
{
  "cep": "01001-000",
  "logradouro": "Praça da Sé",
  "bairro": "Sé",
  "localidade": "São Paulo",
  "uf": "SP"
}
```

```
Provedor B
{
  "cep": "89010025",
  "street": "Rua Doutor Luiz",
  "neighborhood": "Centro",
  "city": "Blumenau",
  "state": "SC",
}
```

Figura 7. Exemplo da diferença entre o corpo JSON da resposta retornada por dois provedores para consulta de CEP. Adaptado de ViaCep (2022) e BrasilAPI (2022).

Fonte: Autores.

4.1.1 Padrão de arquitetura do componente Broker

Para o componente broker, os seguintes requisitos são importantes:

- Performance, para não interferir negativamente na latência das requisições dos serviços do usuário final aos provedores

- Escalabilidade, para suportar alto volume de requisições dos serviços consumidores
- Facilidade de adaptação a mudanças, como para incluir um novo provedor para um recurso

Segundo [Richards 2015] o padrão de arquitetura de software baseado em espaço atende a esses requisitos. Esse padrão tem como princípios a utilização de dados em memória, para atingir alta performance, e um *middleware* virtualizado para gerenciar unidades de processamento.

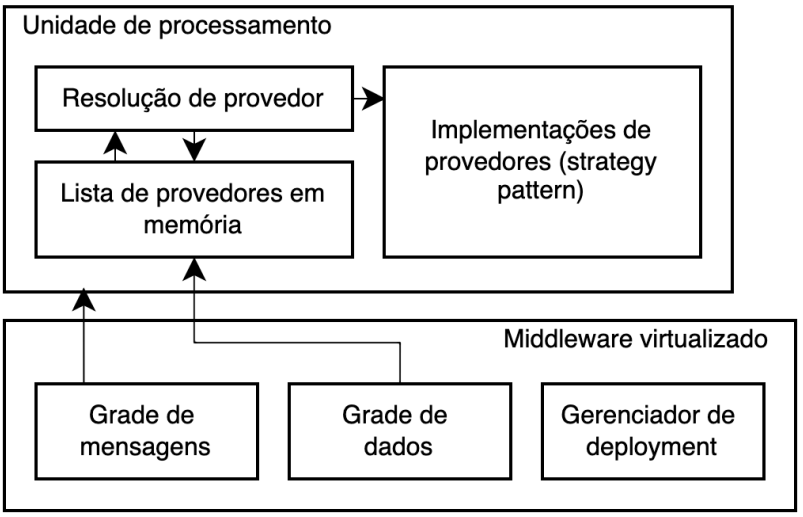

Figura 8. Diagrama da arquitetura baseada em espaço do componente broker.

Fonte: Autores.

A unidade de processamento é o que implementa a solução proposta para selecionar o melhor provedor com base em uma lista em memória. Para possibilitar escalabilidade no broker proposto, essa arquitetura utiliza da grade de mensagens para balancear as requisições entre as instâncias de unidades de processamento e escala pelo componente de grade de *deployment*. A grade de dados é a estrutura que tem a informação original da lista de provedores, que é replicada em memória nas unidades de processamento.

A resolução de provedor consiste em obter o provedor ativo da lista de provedores em memória. Em uma lista ordenada, o provedor ativo é o primeiro. Com essa informação a unidade de processamento é capaz de selecionar a implementação desse provedor com base no *Strategy Pattern*.

4.2 Componente de monitoramento

O componente de monitoramento é responsável pelo gerenciamento de falhas [Ahluwalia e Jain 2006]. O gerenciamento é dividido em identificação e correção das falhas. Para manter a informação de provedor ativo atualizada no componente *broker*, as requisições devem gerar métricas para análise em tempo real. Quando identificada uma

indisponibilidade do provedor com base nas necessidades do usuário final, um alerta deve ser disparado para substituição do provedor.

A substituição do provedor ativo pode ocorrer por indisponibilidade do servidor, tempo de resposta maior do que o desejado pelo usuário final ou erros do cliente. A indisponibilidade do servidor é identificada através de código de status de resposta HTTP entre 500 e 599 [Mozilla, 2022], assim como erros do cliente entre 400 a 499. O código de status *429 Too Many Requests*, por exemplo, significa que o cliente atingiu o número máximo de requisições permitidas em certo período, o que indicaria a necessidade de troca. A Tabela 2 apresenta alguns exemplos de códigos considerados para eventual substituição de um provedor.

Tabela 2. Exemplos de códigos de status de resposta HTTP.

Código	Nome	Significado
500	*Internal Server Error*	O servidor não foi capaz de lidar com determinada situação
503	*Service Unavailable*	O servidor não está apto para manipular a requisição no momento da requisição
504	*Gateway Timeout*	O servidor está atuando como um gateway e não processou a resposta a tempo
429	*Too Many Request*	O cliente fez muitas requisições em um determinado período de tempo

O tempo de resposta é também relevante, pois impacta na experiência do cliente. Por exemplo, um time de engenheiros de software de um e-commerce pode definir que o tempo de resposta desejado para o carregamento da tela de catálogo de produtos é de 400ms. Uma das etapas para a exibição dessa tela é a consulta a uma API de câmbio, para converter o preço de produtos internacionais e, considerando o tempo para processamento das demais etapas, a consulta a API externa não pode ultrapassar 200ms. Pela filosofia de SRE, o time pode definir também que 97% das requisições devem atender ao requisito [Beyer *et. al.* 2018]. Se o provedor de API de câmbio não atender às necessidades definidas pelos engenheiros, o objetivo poderá não ser atingido, logo a importância de redundância de provedores. A solução proposta avalia o tempo de resposta com base no que é configurado pelo usuário final, detalhado na Tabela 3.

Tabela 3. Configurações do usuário final que o sistema proposto avalia para determinar substituição de provedores.

Limite de requisições que ultrapassaram o tempo de resposta desejado
Tempo de resposta desejado para cada recurso
Limite de requisições com erros de servidor retornados pelo provedor
Limite de requisições com erros do usuário retornados pelo provedor

Para possibilitar a substituição de provedores em tempo suficiente para atingir alta disponibilidade deve haver monitoramento e disparo de alertas em tempo real.

Serão aplicadas as estratégias de checagem de saúde do provedor de forma ativa e passiva [Ramirez 2021]. A checagem de saúde, ou *health check*, passiva é a avaliação do tráfego real, ou seja, para cada requisição que passa pelo *broker*, métricas são coletadas e alertas são disparados para o caso de que algum limite configurado seja ultrapassado.

Health check ativo, por outro lado, é a execução periódica de consulta aos provedores para verificar se estão disponíveis. Como Pina *et. al.* (2018) utilizou, o *health check* é uma requisição HTTP GET ao provedor como "https://www.provedor.com/". Caso essa consulta retorne um código de erro do servidor, o provedor pode ser desligado no *broker* com antecedência, antes que uma requisição real seja direcionada para ele.

[Pina *et. al.* 2018] desenvolveu um sistema de monitoramento não intrusivo, isto é, que não requer adaptações tanto nos provedores quanto na arquitetura do usuário final. Essa estratégia se adequa ao sistema proposto para permitir que seja acoplado facilmente a um sistema.

O monitoramento não intrusivo vai considerar o processamento realizado pelo componente broker. As métricas utilizadas serão: código de status de resposta HTTP e tempo de resposta do provedor. Para, em tempo real, executar a coleta de dados do *broker* e disparar alertas para substituição de provedores é necessária uma ferramenta eficiente, com baixa latência e capacidade de lidar com alto volume de dados. [Boncea e Bacivarov 2016] listaram ferramentas open-source para gerenciar o monitoramento em IoT, que tem requisitos semelhantes aos do sistema proposto.

Assim como destaca Prometheus (2022), o *InfluxDB* é uma ferramenta viável para acoplar a arquitetura proposta para registrar os logs do componente *broker.* O *InfluxDB* é um banco de dados de séries temporais robusto e de alta performance que possui componentes para coleta de dados pelo modelo *push* (*Telegraf*) e gerenciamento de alertas em tempo real (*Kapacitor*). O servidor do *InfluxDB* pode ser disponibilizado via *Docker* e a interação com a arquitetura proposta é mostrada na Figura 9 [Influx Data 2022].

Como apresentado na Figura 6, o componente "gerenciador de alertas" será implementado pelo *Kapacitor*. A criação de alertas é feita através de arquivos de configuração no servidor do componente de monitoramento, com base nas configurações do usuário final. O componente "sistema de monitoramento" é implementado pelo *InfluxDB* e o *Telegraf* age como intermediário para possibilitar a coleta de dados. Os dados e os alertas são enviados através de requisições HTTP entre o servidor do componente broker e o servidor do componente de monitoramento.

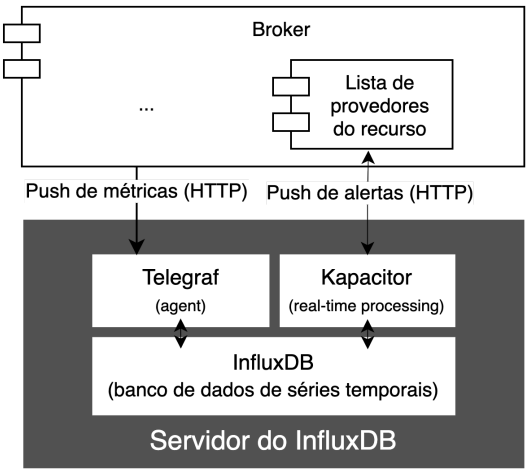

Figura 9. Diagrama do InfluxDB adaptado para o sistema proposto.

Fonte: Autores

O esquema em alto nível do sistema proposto é mostrado na Figura 10, com exemplos de APIs reais de provedores externos que podem ser utilizadas em um sistema de ecommerce. Os itens destacados em amarelo representam pontos em que o usuário final irá interagir com o sistema. Dado que a proposta da pesquisa é desenvolver um software *open-source*, qualquer parte do sistema poderá ser alterada. Apenas a implementação das funções para comunicação com os provedores e a configuração dos objetivos de nível de serviço desejadas são essenciais para o funcionamento da solução proposta.

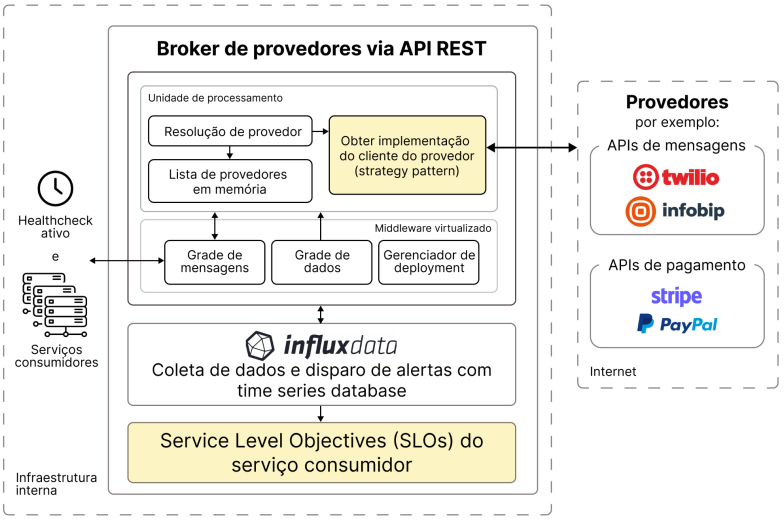

Figura 10. Esquema completo da arquitetura do broker proposto.

Fonte: Autores

5. Cronograma

As próximas fases da pesquisa serão a implementação e validação do artefato proposto. As etapas estão descritas na Tabela 4 e detalhadas na sequência.

Tabela 4. Cronograma das próximas atividades a serem desenvolvidas em TCC II.

Etapas	FEV 2023	MAR 2023	ABR 2023	MAI 2023
Etapa 1: Definição das tecnologias para implementação				
Etapa 2: Implementação da solução proposta				
Etapa 3: Definição da metodologia de testes				
Etapa 4: Execução dos testes de validação da solução				
Etapa 5: Análise dos resultados				
Etapa 6: Elaboração do artigo científico final				

Etapa 1: definir o conjunto de tecnologias que ainda não foram especificadas na arquitetura proposta para implementação do artefato e esquematização da arquitetura completa.

Etapa 2: implementar a solução proposta, que consiste nos componentes broker e de monitoramento.

Etapa 3: definir a metodologia de validação da solução, o que inclui definir os cenários de teste e o ambiente para execução.

Etapa 4: realizar os testes para validação.

Etapa 5: analisar os resultados e avaliar alterações na arquitetura proposta, caso necessário.

Etapa 6: elaborar o artigo científico final do trabalho com os resultados obtidos.

6. Referências

Ahluwalia, K. S. and Jain, A. (2006). High availability design patterns. In Proceedings of the 2006 conference on Pattern languages of programs - PLoP ’06. ACM Press. http://portal.acm.org/citation.cfm?doid=1415472.1415494, [acessado em nov. 17].

Badidi, E. (30 apr 2016). A Broker-based Framework for Integrated SLA-Aware SaaS Provisioning. International Journal on Cloud Computing: Services and Architecture, v. 6, n. 2, p. 01–19.

Bass, L., Clements, P. and Kazman, R. (2021). Software Architecture in Practice, 4th Edition. Addison-Wesley Professional.

Beyer, B., Jones, C., Petoff, J. and Murphy, N. R. [Eds.] (2016). Site reliability engineering: how Google runs production systems. First edition ed. Beijing ; Boston: Oreilly.

Beyer, B., Murphy, N. R., Rensin, D. K., Kawahara, K. and Thorne, S. (22 jun 2018). The Site Reliability Workbook. p. 508.

Boncea, R. and Bacivarov, I. (2016). A System Architecture for Monitoring the Reliability of IoT. Proceedings of the 15th International Conference on Quality and Dependability, p. 7.

Brasil API (2022). https://brasilapi.com.br/docs#tag/CEP, [accessed on Nov 17].

Buschmann, F., Henney, K. and Schmidt, D. C. (2011). A pattern language for distributed computing. Reprinted August 2011 ed. Chichester: Wiley.

Cerin, C., Coti, C., Delort, P., et al. (2014). Downtime Statistics of Current Cloud Solutions. International Working Group on Cloud Computing Resiliency, Tech. Rep

Coulouris, G. F. [Ed.] (2013). Sistemas distribuídos : conceitos e projeto. 5th ed ed. Porto Alegre, RS: Bookman Companhia Editora Ltda.

Dantale, V. S. (2012). Solving business problems with informix TimeSeries. Poughkeepsie, NY: IBM, International Technical Support Organization.

Fielding, R. T. (2000). Architectural Styles and the Design of Network-based Software Architectures. University of California, Irvine. p. 180.

Influxdata (2022). InfluxDB OSS 2.5 Documentation. https://docs.influxdata.com/influxdb/v2.5/, [accessed on Nov 28].

Mozilla (2022). Códigos de status de respostas HTTP - HTTP | MDN. https://developer.mozilla.org/pt-BR/docs/Web/HTTP/Status, [accessed on Nov 28].

Pina, F., Correia, J., Filipe, R., Araujo, F. and Cardroom, J. (nov 2018). Nonintrusive Monitoring of Microservice-Based Systems. In 2018 IEEE 17th International Symposium on Network Computing and Applications (NCA). IEEE. https://ieeexplore.ieee.org/document/8548311/, [accessed on Nov 26].

Prometheus (2022). Comparison to alternatives | Prometheus. https://prometheus.io/docs/introduction/comparison/, [accessed on Nov 27].

Ramirez, N. (14 sep 2021). How to Enable Health Checks in HAProxy? (Guide). HAProxy Technologies. https://www.haproxy.com/blog/how-to-enable-health-checks-in-haproxy/, [accessed on Nov 27].

Richards, M. (2015). Software architecture patterns: understanding common architecture patterns and when to use them. California: O'Reilly Media.

Satheler, G. (6 sep 2021). Alta Disponibilidade de Funções como Serviço em Ambiente de Múltiplas Nuvens de Computação. Universidade Federal do Pampa.

Suthendra, J. A. and Pakereng, M. A. I. (16 dez 2020). Implementation of Microservices Architecture on E-Commerce Web Service. ComTech: Computer, Mathematics and Engineering Applications, v. 11, n. 2, p. 89–95.

Time Series Database (TSDB) Guide | InfluxDB (2022). InfluxData. https://www.influxdata.com/time-series-database/, [acessado em nov. 17].

ViaCEP - Webservice CEP e IBGE gratuito (2022). https://viacep.com.br/, [accessed on Nov 17].

Wieringa, R. J. (2014). Design Science Methodology for Information Systems and Software Engineering. Berlin, Heidelberg: Springer Berlin Heidelberg.