Kubow: Um Serviço de Autoadaptação Baseada em Arquitetura para Aplicações Nativas da Nuvem

Carlos M. Aderaldo, Nabor C. Mendonça

[1] Programa de Pós-Graduação em Informática Aplicada, Universidade de Fortaleza,
Av. Wasington Soares, 3121, 60.811-905 Fortaleza, CE

{carlosmendes,nabor}@unifor.br

Abstract. *This paper presents Kubow, an instantiation of the Rainbow architecture-based self-adaptation framework for containerized cloud native applications managed by Kubernetes.*

Resumo. *Este artigo apresenta Kubow, uma instanciação do arcabouço de autoadaptação baseada em arquitetura Rainbow para aplicações nativas de nuvem implantadas em contêineres gerenciados pelo Kubernetes.*

Link para o vídeo de demonstração da ferramenta:
https://youtu.be/8aukViIFosM

1. Introdução

A crescente utilização de tecnologias de virtualização baseada em contêineres, como Docker,[1] e de orquestradores de contêineres, como Kubernetes,[2] representa uma importante mudança de paradigma na adoção da computação em nuvem, cujo foco sai da necessidade de gerenciar máquinas virtuais para gerenciar contêineres e serviços [Pahl et al. 2017]. Kubernetes, em particular, tem rapidamente se tornado o padrão *de facto* para gerenciar aplicações nativas da nuvem, já sendo oferecido como serviço pelos principais provedores de nuvem do mercado [Brewer 2018].

Um aspecto fundamental do gerenciamento de aplicações nativas de nuvem é a *autoadaptação*, ou seja, a capacidade das aplicações monitorarem e adaptarem dinamicamente seu comportamento, de modo a preservar seus atributos de qualidade diante de incertezas causadas por mudanças em seu ambiente de execução, nos recursos computacionais disponíveis, ou nos interesses de seus usuários [Kephart and Chess 2003]. Embora a literatura da área esteja repleta de soluções de autoadaptação, até o momento relativamente poucas técnicas de autoadaptação, como gerenciamento automático de servidores e autoescalabilidade de aplicações, têm sido efetivamente utilizadas pela indústria [Weyns 2019].

Uma das possíveis razões que explica o uso ainda limitado de soluções de autoadaptação é que a grande maioria das soluções propostas na literatura foi projetada visando ambientes de produção convencionais, ou seja, pré-nuvem, estando à margem, portanto, da nova geração de tecnologias de virtualização e gerenciamento de contêineres que vem sendo cada vez mais adotada pelas organizações. Exemplo disso é que tradicionais arcabouços de autoadaptação baseados em modelos de arquitetura, como Rainbow [Garlan et al. 2004], cujo projeto tem a reutilização como principal requisito, na

[1] https://www.docker.com
[2] https://kubernetes.io

prática têm se mostrado difíceis de customizar para ambientes de produção nativos da nuvem [Mendonça et al. 2018].

Para suprir essa lacuna, este artigo apresenta Kubow, uma instanciação do arcabouço de autoadaptação Rainbow para aplicações nativas da nuvem implantadas em contêineres Docker e gerenciadas pelo Kubernetes. A principal motivação para o desenvolvimento do Kubow é contribuir para um uso mais sistemático de ferramentas de autoadaptação pelas organizações, oferecendo uma solução extensível e alinhada com as principais tecnologias de nuvem atuais. O restante do artigo descreve a arquitetura e as principais funcionalidades do Kubow, seguidas de um exemplo de uso e uma breve comparação com alguns trabalhos relacionados.

2. Kubow

O Kubow compreende um conjunto de tecnologias que permite aos desenvolvedores de aplicações do Kubernetes adicionar a capacidade de monitoramento, planejamento, e adaptação às suas aplicações. Para isso, o Kubow estende e customiza os componentes de adaptação providos pelo arcabouço de autoadaptação Rainbow [Garlan et al. 2004], cuja arquitetura, implementada em Java, segue o clássico modelo de referência MAPE-K para sistemas autoadaptativos [Kephart and Chess 2003].

Os componentes originais do Rainbow incluem (ver Figura 1a): um *Model Manager*, que mantém o modelo de arquitetura da aplicação alvo, definido pelo usuário; um conjunto de *Probes* e *Gauges*, que monitoram o comportamento da aplicação alvo e atualizam os atributos de seu modelo de arquitetura; um *Architecture Evaluator*, que analisa o modelo de arquitetura e decide se alguma adaptação é requerida; um *Adaptation Manager*, que seleciona a melhor estratégia de adaptação a ser executada a partir de um conjunto de táticas e estratégias, também definido pelo usuário; um *Strategy Executor*, que gerencia a execução da estratégia selecionada; e um conjunto de *Effectors*, que provê os mecanismos necessários para alterar o comportamento da aplicação alvo em seu ambiente de execução. *Probes*, *Gauges* e *Effectors* constituem a camada de tradução do Rainbow, a qual deve ser instanciada para cada nova aplicação ou ambiente de execução, com os demais componentes sendo integralmente reutilizados. Todos os componentes se comunicam de forma assíncrona, por meio de um mecanismo próprio de publicação e notificação distribuída de eventos.

Além dos componentes do Rainbow, o Kubow utiliza vários serviços específicos do Kubernetes. Esses serviços são invocados por meio de duas APIs REST: *Metrics API*, que estende a API básica do Kubernetes, oferecendo uma forma padronizada de acesso a métricas coletadas tanto pelos componentes internos do Kubernetes quanto por ferramentas externas de monitoramento, como é o caso do Prometheus;[3] e *Kubernetes API*, que é a principal forma de acesso aos recursos gerenciados pelo Kubernetes, como *Pods*, *Deployments* e *Services*. A *Metrics API* permite definir *Probes* e *Gauges* no Kubow para monitorar virtualmente qualquer tipo de métrica, seja de sistema (e.g., consumo de CPU e memória) ou específica da aplicação alvo (e.g., tempo de resposta e vazão). Já a *Kubernetes API* permite definir *Effectors* para modificar qualquer atributo dos diferentes tipos de recursos gerenciado pelo Kubernetes (e.g., a versão da imagem do contêiner utilizado para executar um *Pod*, e as quantidades mínima e máxima de *Pods* de um *Deployment*)

[3] `https://prometheus.io`

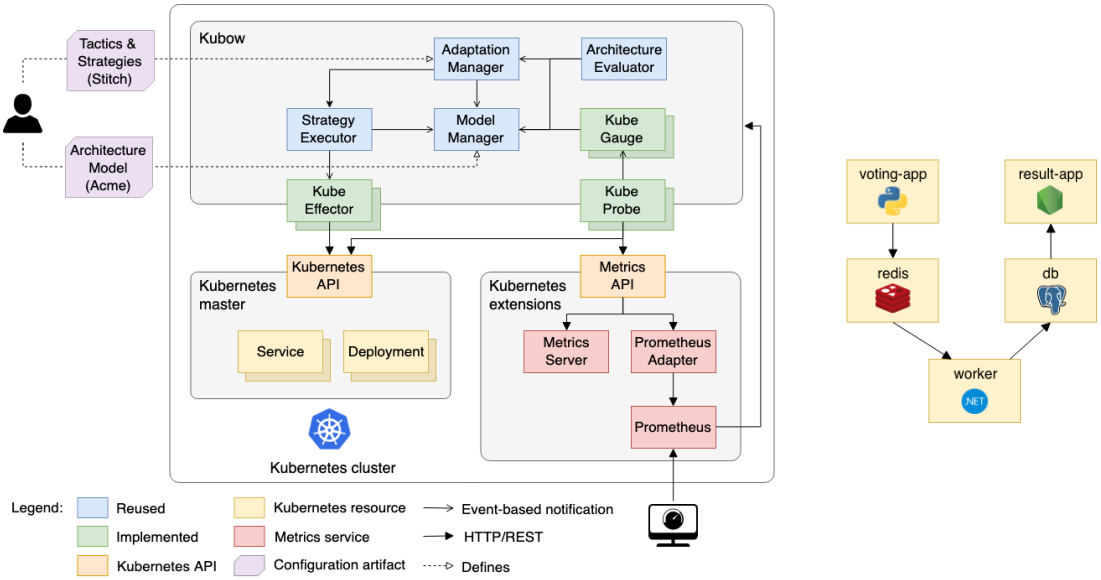

Figura 1. (a) Arquitetura do Kubow; (b) arquitetura da Voting App.

Em tempo de implantação, todo os componentes do Kubow são empacotados em um único contêiner Docker, o qual é então executado e gerenciado como um serviço nativo do Kubernetes.

3. Exemplo de Uso

Como exemplo de uso do Kubow, foi escolhida a Voting App, que faz parte do conjunto de aplicações de demonstração do Docker.[4] A Voting App implementa um simples sistema de votação online. Sua arquitetura é composta por cinco componentes (ver Figura 1b): um aplicativo web de votação (voting-app), implementado em Python, que coleta os votos dos usuários; uma fila de mensagens (redis), implementada em Redis, onde o aplicativo de votação publica os votos coletados; um processador de votos em segundo plano (worker), implementado em .NET, que consome os votos da fila de mensagens; um banco de dados relacional (db), implementado em Postgres, onde os votos processados são armazenados; e um segundo aplicativo web (result-app), implementado em Node.js, que permite aos usuários visualizar os resultados da votação em tempo real.

O primeiro passo para a integração do Kubow a uma aplicação é a definição do modelo de arquitetura da aplicação. No Rainbow, o modelo de arquitetura é definido utilizando a linguagem Acme, que oferece vários recursos sintáticos para a descrição de elementos arquiteturais em termos de componentes e conectores [Garlan et al. 2000].

A Figura 2 ilustra a definição do componente voting-app da Voting App em Acme. A definição desse componente referencia três tipos de elementos arquiteturais, quais sejam, DeploymentT (linha 3), ServiceT (linha 7), e LabelSelectorConnectorT (linha 10), os quais compõem a família ou estilo arquitetural previamente definido no Kubow para aplicações do Kubernetes. Os dois primeiros tipos correspondem a recursos típicos do Kubernetes, no caso, *Services* e *Deployments*, enquanto o terceiro tipo representa a definição de um seletor de rótulos do Kubernetes, o qual é utilizado para conectar um recurso

[4] https://github.com/dockersamples/example-voting-app

```
1 import families/Kubernetes.acme
2 System VotingAppSystem: KubernetesFam = new
      KubernetesFam extended with {
3   Component resultD = new DeploymentT
        extended with {
4     Property name = "result";
5     Property namespace = "votingapp";
6   }
7   Component resultS = new ServiceT extended
        with {
8     Property name = "result";
9   }
10  Connector resultSC = new
        LabelSelectorConnectorT extended with
        {
11    Property selectors = <[name: string = "
        app"; value: string = "result";]>;
12  }
13  Attachment resultS.redirectPort to resultSC
        .callee;
14  Attachment resultD.redirectPort to resultSC
        .caller;
15 }
```

Figura 2. Definição do componente result-app da VotingApp.

```
1 define boolean cHiRespTime = M.resultS.
      latency > M.MAX_RESPTIME;
2 define boolean cLoRespTime = M.resultS.
      latency < M.MIN_RESPTIME;
3 define boolean canAddPods = M.resultD.
      maxReplicas > M.resultD.desiredReplicas;
4 define boolean canRemovePods = M.resultD.
      minReplicas < M.resultD.desiredReplicas;
5 tactic addReplicas(int count) {
6   condition {
7     cHiRespTime && canAddPods;
8   }
9   action {
10    M.scaleUp(M.resultD, M.resultD.
          desiredReplicas + count);
11  }
12  effect {
13    M.resultD.maxReplicas >= M.resultD.
          desiredReplicas;
14  }
15 }
16 tactic removeReplicas(int count) {
17   condition {
18     cLoRespTime && canRemovePods;
19   }
20   action {
21     M.scaleDown(M.resultD, M.resultD.
           desiredReplicas - count);
22   }
23   effect {
24     M.resultD.minReplicas <= M.resultD.
           desiredReplicas;
25   }
26 }
```

Figura 3. Definição de táticas para o result-app.

do tipo *Service* a um recurso do tipo *Deployment* (linhas 13-14). Os demais componentes do modelo de arquitetura da Voting App foram definidos de maneira similar.

A partir da definição do modelo de arquitetura da aplicação alvo, podem ser definidos os demais elementos necessários para o Kubow poder monitorar (*Probes* e *Gauges*), analisar (táticas e estratégias) e adaptar (*Effectors*) o comportamento da aplicação em tempo de execução. Devido a limitações de espaço, neste artigo são mostrados apenas trechos dos conjuntos de táticas e estratégias definidos para autoescalar o número de réplicas (*Pods*) do serviço result-app da Voting App.

No Rainbow, táticas e estratégias são definidas em Stitch, uma linguagem de domínio específico para arcabouços de autoadaptação [Cheng and Garlan 2012]. As Figuras 3 e 4 mostram, respectivamente, os conjuntos de táticas e estratégias definidos em Stitch para o result-app. Na Figura 3, são definidas duas táticas, para adicionar (linhas 5-15) e remover (linhas 16-26) réplicas (*Pods*) do serviço, respectivamente. Note que cada tática invoca uma diferente operação de adaptação, no caso, scaleUP (linha 10) e scaleDown (linha 21), diretamente no elemento do modelo representando o serviço a ser adaptado. Em tempo de execução, essas operações são realizadas pelo componente *Strategy Executor* do Rainbow, acionando os respectivos *Effectors* do Kubow responsáveis pela adição e remoção de *Pods* no Kubernetes.

Na Figura 4, por sua vez, são definidas duas estratégias, para reduzir o tempo de resposta (linhas 1-6) e reduzir o custo (linhas 7-12) do serviço, respectivamente. Note que a estratégia de redução do tempo de resposta invoca a tática de aumento do número de réplicas apenas quando o tempo de resposta estiver acima do máximo permitido, e

```
1 strategy ReduceRespTime [ cHiRespTime ] {
2   t0: (cHiRespTime && canAddPods) ->
        addReplicas(1) @[30000 /*ms*/] {
3     t0a: (success) -> done;
4   }
5   t2: (default) -> TNULL;
6 }
```

```
7 strategy ReduceCost [ cLoRespTime ] {
8   t0: (cLoRespTime && canRemovePods) ->
        removeReplicas(1) @[30000 /*ms*/] {
9     t0a: (success) -> done;
10  }
11  t1: (default) -> TNULL;
12 }
```

Figura 4. Definição de estratégias para o result-app.

ainda for possível adicionar novas réplicas (linha 2). Já a estratégia de redução do custo invoca a tática de redução do número de réplicas apenas quando o tempo de resposta estiver abaixo do mínimo permitido, e ainda for possível remover réplicas (linha 8). Para cada invocação de tática, é definida uma latência de até 3 segundos, que corresponde ao intervalo de tempo que o *Strategy Executor* irá aguardar antes de verificar se a execução da tática foi bem sucedida.

É importante ressaltar que todos esses artefatos, i.e., modelo de arquitetura, táticas e estratégias, são definidos de forma declarativa pelo usuário do Kubow, em diferentes arquivos de configuração, podendo ser alterados a qualquer momento sem a necessidade de alterar ou recompilar o código fonte da ferramenta. O vídeo que acompanha este artigo demonstra os passos envolvidos na implantação do Kubow no Kubernetes para monitorar e alterar dinamicamente o quantidade de réplicas do serviço result-app da Voting App, em resposta a variações em sua carga de trabalho. Maiores detalhes sobre o desenvolvimento, uso e avaliação do Kubow podem ser obtidos em [Aderaldo 2019].

4. Trabalhos Relacionados

Recentemente, vários trabalhos têm sido publicados na literatura propondo soluções de autoadaptação para aplicações que utilizam tecnologias de contêineres. Em [Florio and Di Nitto 2016], os autores propõem uma abordagem descentralizada para escalabilidade horizontal de aplicações baseadas na arquitetura de microsserviços implantadas em contêineres Docker. Em [Delnat et al. 2018], os autores descrevem uma solução extensível para implementar e avaliar diferentes abordagens de autoescalabilidade de aglomerados de banco de dados implantados no Kubernetes. Já em [Sampaio Jr et al. 2018], os autores propõem um mecanismo para realocação dinâmica de microsserviços implantados no Kubernetes. Por fim, em [Rodriguez and Buyya 2018], os autores descrevem um novo mecanismo de gerência de contêineres para o Kubernetes, cujo objetivo é consolidar dinamicamente os contêineres da aplicação no menor número de máquinas virtuais possível.

Diferentemente do Kubow, todas as soluções descritas acima são limitadas quanto ao seu escopo de autoadaptação, não permitindo, por exemplo, a criação de novas táticas e estratégias de adaptação de forma declarativa, sem precisar alterar seu código fonte. A mesma limitação se aplica às soluções de autoadaptação atualmente em uso no mercado, como CloudWatch,[5] da Amazon, e o próprio serviço de autoescalabilidade horizontal de *Pods* (HPA) provido pelo Kubernetes.[6] Em geral, essas soluções suportam a configuração de novas métricas de monitoramento específicas da aplicação alvo, mas não permitem a criação de novos mecanismos de adaptação por parte dos desenvolvedores das aplicações.

[5] https://aws.amazon.com/cloudwatch/

[6] https://kubernetes.io/docs/tasks/run-application/horizontal-pod-autoscale/

5. Conclusão

Este artigo apresentou e exemplificou o uso do Kubow,[7] um serviço para autoadaptação de aplicações nativas da nuvem gerenciadas pelo Kubernetes. Embora o exemplo mostrado no artigo seja limitado ao uso do Kubow para autoescalar um dos serviços de uma aplicação, a ferramenta suporta a definição de outros tipos de adaptação (e.g., atualização da imagem do contêiner de um *Pod* em tempo de execução), de forma totalmente declarativa, oferecendo, assim, uma solução mais fácil de estender e customizar do que as soluções de autoadaptação atualmente disponíveis para o Kubernetes.

Agradecimentos

Este trabalho é financiado pelo CNPq (proc. 313553/2017-3 e 424160/2018-8).

Referências

Aderaldo, C. M. (2019). Kubow: Um Serviço de Autoadaptação Baseada em Arquitetura para Aplicações Implantadas com Docker e Kubernetes. Master's thesis, Universidade de Fortaleza, Programa de Pós-Graduação em Informática Aplicada.

Brewer, E. (2018). Kubernetes and the New Cloud (Invited Keynote). In *SIGMOD'18*.

Cheng, S.-W. and Garlan, D. (2012). Stitch: A language for architecture-based self-adaptation. *Journal of Systems and Software*, 85(12):2860–2875.

Delnat, W. et al. (2018). K8-scalar: a workbench to compare autoscalers for container-orchestrated database clusters. In *IEEE/ACM SEAMS'18*, pages 33–39. IEEE.

Florio, L. and Di Nitto, E. (2016). Gru: An approach to introduce decentralized autonomic behavior in microservices architectures. In *IEEE ICAC'16*, pages 357–362. IEEE.

Garlan, D. et al. (2004). Rainbow: Architecture-based self-adaptation with reusable infrastructure. *Computer*, 37(10):46–54.

Garlan, D., Monroe, R. T., and Wile, D. (2000). Acme: Architectural description of component-based systems. *Foundations of Component-Based Systems*, 68:47–68.

Kephart, J. O. and Chess, D. M. (2003). The vision of autonomic computing. *Computer*, 36(1):41–50.

Mendonça, N. C. et al. (2018). Generality vs. reusability in architecture-based self-adaptation: The case for self-adaptive microservices. In *ECSA'18: Comp. Proceedings*.

Pahl, C. et al. (2017). Cloud container technologies: a state-of-the-art review. *IEEE Trans. Cloud Computing*.

Rodriguez, M. A. and Buyya, R. (2018). Containers orchestration with cost-efficient autoscaling in cloud computing environments. *arXiv preprint arXiv:1812.00300*.

Sampaio Jr, A. R. et al. (2018). Improving Microservice-based Applications with Runtime Placement Adaptation. *Journal of Internet Services and Applications*.

Weyns, D. (2019). *Software Engineering of Self-adaptive Systems*, pages 399–443. Springer International Publishing.

[7] `https://github.com/ppgia-unifor/kubow/`.