Advances in Modelling and Analysis A
Vol. 55, No. 1, March, 2018, pp. 1-10
Journal homepage: http://iieta.org/Journals/AMA/AMA_A

Containerized microservices: Structures and dynamics

Gang Xue*, Jing Liu, Liwen Wu

School of Software, Yunnan University, Kunming, China

Corresponding Author Email: mess@ynu.edu.cn

Received: 16 November 2017
Accepted: 16 April 2018

Keywords:
microservices modelling, the category of mode-dependent networks, containerized microservices, microservice architecture

ABSTRACT

Microservice architecture (MSA) is a new kind of service-oriented architecture. The architecture requires that system components are implemented in the form of microservices. Microservices are message-driven services with small size, and they can be independently developed and automatically deployed. Microservices can be built and released in software containers, since a software container can provide an isolated and portable environment for executing programs. This paper observes structures and functions of containerized microservices from a formal point of view. A framework of categories is adopted for modelling structures and dynamics of containerized microservices. Some issues, which include interoperation, registration and accessing of microservices, are discussed in this paper, and relevant models are formed under the help of the category-based tools. The established models show that the adopted framework can be applied in modelling microservice-oriented applications, and it is able to bring functional and structural features of applications closer together.

1. INTRODUCTION

Microservice architecture (MSA) is an implementation approach for constructing service-oriented applications. The architecture requires that system components are implemented in the form of microserivces [7]. Microservices are fine grained services which have the following features in applications [1]: small in size, messaging enabled, bounded by contexts, autonomously developed, independently deployable, decentralized, built and released with automated processes. MSA has been applied in many fields or products, such as DevOps movement [3], cloud computing [3-4], internet of things (IoT) [5-6], Netflix OSS (https://netflix.github.io), SoundCloud (https:// www.soundcloud.com), and others.

A software container can provide an isolated environment with necessary resources for executing programs [2]. It is a kind of lightweight virtualized environment which runs on operating system kernel. Software containers can provide technical supports for building, testing, deploying and maintaining microservices. When microservices are implemented in software containers, they are called "containerized microservices".

This paper focuses on the topic of microservices modelling. A framework of categories is adopted for modelling main technical aspects of containerized microservices. The framework has two constituents: the category of mode-dependent networks (MDN) and its dynamical systems [8]. The category of mode-dependent networks is a wiring diagram [13] based language. The language can describe a system structure comprehensively, and by defining a dynamical system on the structural model, dynamics of the system can be revealed.

1.1 Related works

MSA and microservices have been discussed by different researchers or engineers, some results or conclusions can be found in [1, 10-12]. Some related topics or issues are suggested as open problems or future challenges, such as services cooperation, dependability, security, and others [7]. In the field of service-oriented applications, different tools are used in researching web services or service compositions, and some of them are Petri-net [18], Ontology [19], UML [20] and others. In [18], researchers applied Petri-net in static analysis of BPEL (Business Process Execution Language) [21] processes. Based on this, a comprehensive mapping from BPEL constructs to Petri-net structures is defined, and the mapping is applied in the implementation of a tool which can translate BPEL processes into Petri nets [18]. In [19], researchers proposed the Web Service Modeling Ontology, and it can be used in describing various aspects related to semantic web services.

Category theory is a powerful tool for investigating abstract concepts and their relationships. The category of mode-dependent networks (MDN) is a kind of symmetric monoidal category [8], and it provides wiring diagrams [13] for representing system structures. Wiring diagram is a kind of graphical language which is defined on the concept of operad [13], and the method has been extended in discussions about discrete-time processes [14] and open dynamical systems [15].

1.2 Organization of the text

The rest of this paper is organized as follows. Section 2 introduces containerized microservices, MDN and its dynamical systems. Section 3 discusses structural models of some containerized microservices. The discussion is organized as four parts which are simple services, interoperating services, service registration, and proxies. Dynamical systems on the established structural models are discussed in section 4. Section 5 introduces some software tools for constructing MSA applications, and section 6 concludes the whole paper.

2 PRELIMINARIES

This section introduces main concepts and foundations of the work in this paper. In the following content, the category of sets is denoted as Set; the category of typed finite sets is denoted as TFS; the category of wiring diagrams is denoted as WD, and the category of mode-dependent networks is denotes as MDN. In a category, the collection of objects is denoted as $Ob(\cdot)$; a morphism f from A to B is denoted as $f: A \to B$, and the morphism can also be written as $A \xrightarrow{f} B$. The composition of two morphisms f and g is written as $g \circ f$. The following operators are also used in this section: $\otimes$ (which is a tensor product operator), $\sqcup$ (which is a disjoin union operator), and $\times$ (which is a Cartesian product operator).

2.1 Containerized microservices

A microservice is defined as a cohesive, independent process interacting via messages [7]. Microservics can be independently developed, tested, deployed and maintained. Software container is a kind of operating system level virtualization implementation. Container-based virtualization is applicable to providing higher density of virtual environment and better performance [2]. There are several implementations of software containers, such as: Linux-VServer [16], OpenVZ (https://openvz.org), LXC (Linux Container, https://linuxcontainers.org/lxc/introduction/), docker [2] (https://www.docker.com), and others.

With the advantages of container-based virtualization, microservices can be built and released in software containers. In addition, containerized microservices could be organized as clusters. Generally, a container cluster contains multiple configurable nodes, and each node can be treated as a container ‘pool’. Similar to applying service-oriented architecture (SOA), when implementing a MSA-based system, the following aspects should be considered seriously: function, interface, messaging, deployment, registration, accessing, management, and so on.

2.2 MDN and dynamical systems

A category contains four constitutions [9, 17]: objects, morphisms, identity morphisms, and compositions of morphisms; in addition, the following laws must be obeyed: identity law and associative law. Informally, objects are things in a category. A morphism is a structure-preserving map from object to object, and an identity morphism is a special morphism from one object to itself. Under certain conditions, two morphisms can be composed as a new morphism, and the symbol $\circ$ is the operator of composition. A *functor* is a ‘bigger’ morphism whose domain and codomain are categories. The definition of a functor has to take the following components into account [9, 17]: objects and morphisms in two different categories; the computing must obey two rules [9, 17]: preservation of identity morphisms, and preservation of composition of morphisms.

Let symbol Set be the *category of sets*. Objects in Set are sets, and morphisms in Set are functions whose domain and codomain are sets. Based on the work of paper [8], the *category of typed finite sets*, which is denoted as TFS, can be defined. An object of TFS is a finite set with a *typing function* τ; formally, the collection of objects in TFS is [8]:
$Ob(TFS) \coloneqq \{(A, \tau) | A \in Ob(FinSet), \tau: A \to Set\}$

A morphism in TFS is defined as a typed function $f: (A, \tau) \to (A', \tau')$, which requires that there is a function $f': A \to A'$ such that $\tau = \tau' \circ f'$ [8]. For a typed finite set (A, τ), $\overline{(A, \tau)} \coloneqq \prod_{a \in A} \tau(a)$ is a *dependent product*, and the simplified form is $\overline{A}$ [8].

On the basis of TFS, the *category of wiring diagrams* (WD) can be defined. According to [8][13], an object of WD is defined as $X = (X^{in}, X^{out})$, where $X^{in}, X^{out} \in Ob(TFS)$ are typed input and output ports respectively; for two objects $X, Y \in Ob(WD)$, a morphism $\varphi: X \to Y$ is a pair of typed functions $\varphi = (\varphi^{in}, \varphi^{out})$, where $\varphi^{in}: X^{in} \to X^{out} \sqcup Y^{in}$ and $\varphi^{out}: Y^{out} \to X^{out}$. The composition of two morphisms is $\phi \circ \varphi = ((\phi \circ \varphi)^{in}, (\phi \circ \varphi)^{out}): X \to Z$, where $\varphi: X \to Y$ and $\phi: Y \to Z$ are morphisms in WD. More relevant discussions about WD can be found in [13].

Definition 1 (the category of mode-dependent networks) [8]: the category of mode-dependent networks (MDN) has the following major constitutions: (1) objects, (2) morphisms, (3) identities, (4) compositions. These constitutions are defined as follows: the collection of objects in MDN is represented as $Ob(MDN)$; an object is a pair (M, X), where $M \in Set$ is the mode set and $X: M \to Ob(WD)$ is the interface function. For two objects $(M, X), (N, Y) \in Ob(MDN)$, the set of morphisms from (M, X) to (N, Y) is denoted as $Hom_{MDN}((M, X), (N, Y))$; a morphism in $Hom_{MDN}((M, X), (N, Y))$ is a pair (ϵ, σ) , where $\epsilon: M \to Mor(WD)$ and $\sigma: M \to N$ are functions which make the following diagram commutes (in the diagram, $Mor(WD)$ is the set of morphisms in WD; map *dom* can get a domain of a morphism, and *cod* can get a codomain of a morphism):

$$\begin{tikzcd}
& M \arrow[dl, "X"'] \arrow[d, "\epsilon"] \arrow[rr, "\sigma"] & & N \arrow[d, "Y"] \\
Ob(WD) & Mor(WD) \arrow[l, "dom"'] \arrow[rr, "cod"] & & Ob(WD)
\end{tikzcd}$$

If $(M, X) \in Ob(MDN)$, (ϵ_{WD}, id_M) is an identity morphism on (M, X) , where $id_M: M \to M$ is an identity morphism on M, and $\epsilon_{WD}: M \to ids_{WD}$ is a function; in addition, if $m \in M$, then $X(m) \in Ob(WD)$, $id_{WD}: X(m) \to X(m)$ and $id_{WD} \in ids_{WD}$;

If $(\epsilon_0, \sigma_0) \in Hom_{MDN}\big((M_0, X_0), (M_1, X_1)\big)$ and $(\epsilon_1, \sigma_1) \in Hom_{MDN}$ $((M_1, X_1), (M_2, X_2))$ are two morphisms with $(M_0, X_0), (M_1, X_1), (M_2, X_2) \in Ob(MDN)$, the composition of (ϵ_0, σ_0) and (ϵ_1, σ_1) is defined as $(\epsilon_1, \sigma_1) \circ_{MDN} (\epsilon_0, \sigma_0)$, i.e. $(\epsilon, \sigma) \coloneqq (\epsilon_1, \sigma_1) \circ_{MDN} (\epsilon_0, \sigma_0)$, where $\sigma \coloneqq \sigma_1 \circ \sigma_0$ and $\epsilon: M_0 \to Mor(WD)$; in addition, if $m_0 \in M_0$, then $\epsilon(m_0) \coloneqq \epsilon_1 \sigma_0(m_0) \circ \epsilon_0(m_0)$. The computing of (ϵ, σ) is shown as follows:

$$\begin{tikzcd}
& M \arrow[dl, "X"'] \arrow[d, "\epsilon_0"] \arrow[rr, "\sigma_0"] & & M \arrow[dl, "X_1"'] \arrow[d, "\epsilon_1"] \arrow[r, "\sigma_1"] & M \arrow[d, "X_2"] \\
Ob(WD) & Mor(WD) \arrow[l, "dom"'] \arrow[r, "cod"] & Ob(WD) & Mor(WD) \arrow[l, "dom"'] \arrow[r, "cod"] & Ob(WD)
\end{tikzcd}$$

Since MDN is a category, the following category laws must be obeyed: the identity law and the associative law. In MDN, the monoidal structure of (M, X) and (M, Y) is defined as $(M \times N, X \sqcup Y)$, i.e. $(M, X) \otimes (N, Y) \coloneqq (M \times N, X \sqcup Y)$, where $X \sqcup Y$ is explained as:

$$M \times N \xrightarrow{X \times Y} Ob(WD) \times Ob(WD) \xrightarrow{\sqcup} Ob(WD)$$

□

Definition 2 (dynamical systems on MDN models) [8]: a dynamical system on a MDN model is defined by a lax functor $P: MDN \to Set$. To be more specific, P must be applied on objects and morphisms of the MDN model.

For an object $(M, X) \in Ob(MDN)$, a dynamical system is $P(M, X) := (S, q, f)$, where:

- S is the state set and $S \in Set$;
- $q: S \to M$ is the underlying mode function;
- if $s \in S$ and $m = q(s)$, then $f := (f^{in}, f^{out})$, where $f^{in}(s): \overline{X^{in}}(m) \to S$ is the state update function, and $f^{out}(s) \in \overline{X^{out}}(m)$ is the readout function.

For a morphism $(\epsilon, \sigma) \in Hom_{MDN}\big((M, X), (N, Y)\big)$, a dynamical system is $P(\epsilon, \sigma): P(M, X) \to P(N, Y)$. If (S, q, f) is a dynamical system of (M, X), then $P(\epsilon, \sigma)(S, q, f) := (S, r, g)$, where

- S is the state set of $P(M, X)$;
- $r = \sigma \circ q$, i.e., $S \xrightarrow{q} M \xrightarrow{\sigma} N$;
- for $s \in S$ and $m = q(s)$, the following functions can be given:
- $\overline{\epsilon^{in}}(m): \overline{Y^{in}}(m) \times \overline{X^{out}}(m) \to \overline{X^{in}}(m)$

$$\overline{\epsilon^{out}}(m): \overline{X^{out}}(m) \to \overline{Y^{out}}(m)$$

- $g := (g^{in}, g^{out})$, where

$$g^{out}(s) = \overline{\epsilon^{out}}(m)(f^{out}(s))$$
$$g^{in}(s)(y) = f^{in}(s)(\overline{\epsilon^{in}}(m)(y, f^{out}(s)))$$

For the lax monoidal structures, $P(M, X) \times P(N, Y) \to P(M \times N, X \sqcup Y)$ is formed by using the following Cartesian products: $S_{X \times Y} \coloneqq S_X \times S_Y$, $q_{X \times Y} \coloneqq q_X \times q_Y$, and $f_{X \times Y} \coloneqq f_X \times f_Y$.

Set, TFS, WD, MDN and dynamical systems on MDN models have been introduced by now. These tools could be used to model different aspects of containerized microservices. To be more specific,

- TFS is used to represent a system component; an element in the set represents a communication port of the component, and the typing function can specify a value domain of a port;
- WD is used to represent the structure of a system; since the objects of WD are typed finite sets, the nesting system structures and mappings of ports are defined as morphisms; a parallel structure of objects is represented as a monoidal structure;
- MDN has the mode set and wiring diagrams; a diagram in the category is indexed by an element of the mode set;
- A dynamical system on a MDN model can be used to describe messaging and functions in a system.

3 STRUCTURAL MODELS OF CONTAINERIZED MICROSERVICES

This section mainly introduces structural models of containerized microservices. The models are formed around some issues in applications.

3.1 Simple services

The simplest structure of a containerized mciroservice has two components: Container and Service, as shown in Figure 1. The figure also shows that the components have different ports. For the 'Service', i' is an input port and o' is an output port; but for the 'Container', i is an input port and o is an output port. Considering that container is an environment, inputs and outputs of the "Service" are passed through the ports of the 'Container'.

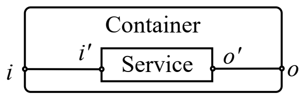

Figure 1. A simple microservice

According to **Definition 1**, the structure in Figure 1 can be represented as:

- $container := (M_c, I_c)$, and $M_c = \{'on'\}$;
- $service := (M_s, I_s)$, and $M_s = \{'working'\}$;
- $microservice := (\epsilon, \sigma): service \to container$ with $\sigma: M_s \to M_c$, $\epsilon: M_s \to \{\varphi\}$, and $\varphi \coloneqq (\varphi^{in}, \varphi^{out})$.

The morphism (ϵ, σ) defines that the object *service* is embedded in the object *container*. Function $\sigma: M_s \to M_c$ could be defined easily, since M_c or M_s is a mode set with one element. Interfaces of all components are defined as:

- $I_c('on') = (C^{in}, C^{out})$;
- $I_s('working') = (S^{in}, S^{out})$.

For the object *container*, C^{in} is the set of input ports, and C^{out} is the set of output ports. For the object *service*, S^{in} is the set of input ports, and S^{out} is the set of output ports. In Figure 1, port *i* is connected to port *i'*, and port *o* is connected to port *o'*. The connections are described as $\varphi^{in}: S^{in} \to C^{in}$ and $\varphi^{out}: C^{out} \to S^{out}$.

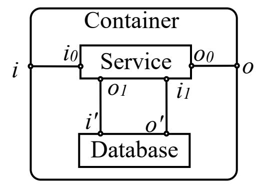

Figure 2. A microservice with a database component

A microservice may have a local database component, as shown in Figure 2. The 'Service' in Figure 2 has two input ports i_0, i_1 and two output ports o_0, o_1. In addition, i_0 is used to receive external inputs; i_1 is used for receiving database outputs; o_0 is a port for sending outputs, and o_1 is a port for sending database commands. The 'Container' in Figure 2 has an input port *i* and an output port *o*. For the 'Database', it uses input port *i'* for receiving commands, and it uses output port *o'* for sending outputs. The whole structure is represented as:

- $container := (M_c, I_c)$, and $M_c = \{'on'\}$;

- $service := (M_s, I_s)$, and $M_s = \{'working'\}$;
- $db := (M_d, I_d)$, and $M_d = \{'running'\}$;
- $ms := (\epsilon, \sigma): service \otimes db \rightarrow container$ with $\sigma: M_s \times M_d \rightarrow M_c$, $\epsilon: M_s \times M_d \rightarrow \{\varphi\}$, and $\varphi := (\varphi^{in}, \varphi^{out})$.

The morphism (ϵ, σ) defines that the object *service* and *db* are embedded in the object *container*. Function $\sigma: M_s \times M_d \rightarrow M_c$ can be defined easily, since M_d, M_c and M_s are one-element sets. Interfaces of all components are defined as:

- $I_c('on') = (C^{in}, C^{out})$;
- $I_s('working') = (S^{in}, S^{out})$;
- $I_d('running') = (D^{in}, D^{out})$.

The definition shows that (C^{in}, C^{out}), (S^{in}, S^{out}), (D^{in}, D^{out}) are the port sets of the object *container*, *service* and *db* respectively. In Figure 2, port i_0, i_1, and i' are connected to port i, o', and o_1 respectively; and port o is connected to port o_0. All connections are described as $\varphi^{in}: S^{in} \sqcup D^{in} \rightarrow C^{in} \sqcup D^{out} \sqcup S^{out}$ and $\varphi^{out}: C^{out} \rightarrow S^{out} \sqcup D^{out}$.

3.2 Interoperating services

A microservice is able to work with others. Figure 3 displays two containerized microservices, and the 'DB' is a service that can provide data persistence functions. The whole configuration of Figure 3 contains 5 components: a 'VM' (which is a virtual machine) with input port i and output port o; a 'Service' with input ports i_0, i_1 and output ports o_0, o_1; a 'DB' (which is a database component) with input port i' and output port o'; a 'Container$_1$' with input ports p_1, p_4 and output ports p_2, p_3; a 'Container$_2$' with input port p_{in} and output port p_{out}.

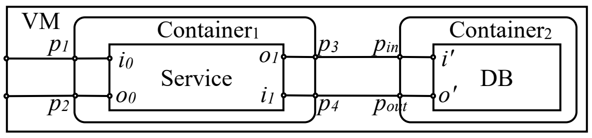

Figure 3. Two interoperating containerized microservices

According to **Definition 1**, the structure in Figure 3 can be defined as:

- $service := (M_s, I_s)$, and $M_s = \{'working'\}$;
- $db := (M_d, I_d)$, and $M_d = \{'running'\}$;
- $container_1 := (M_1, I_1)$, and $M_1 = \{'on_1'\}$;
- $container_2 := (M_2, I_2)$, and $M_2 = \{'on_2'\}$;
- $vm := (M, I)$, and $M = \{'up'\}$.

All objects are organized as:

$$service \otimes db \xrightarrow{(\epsilon_0, \sigma_0)} container_1 \otimes container_2 \xrightarrow{(\epsilon_1, \sigma_1)} vm$$

where (ϵ_0, σ_0) shows that the object *service* and *db* are embedded in the object *container$_1$* and *container$_2$*, and (ϵ_1, σ_1) shows that the object *container$_1$* and *container$_2$* are deployed in the object *vm*. (ϵ_0, σ_0) and (ϵ_1, σ_1) can be composed as a morphism: $services := (\epsilon, \sigma): service \otimes db \rightarrow vm$

where $(\epsilon, \sigma) := (\epsilon_1, \sigma_1) \circ_{MDN} (\epsilon_0, \sigma_0)$, and:

- $\sigma_0: M_s \times M_d \rightarrow M_1 \times M_2$;
- $\sigma_1: M_1 \times M_2 \rightarrow M$;
- $\epsilon_0: M_s \times M_d \rightarrow \{\varphi\}$, and $\varphi := (\varphi^{in}, \varphi^{out})$;
- $\epsilon_1: M_1 \times M_2 \rightarrow \{\psi\}$, and $\psi := (\psi^{in}, \psi^{out})$.

Function σ_0 and σ_1 can be defined easily. Interfaces of components are defined as the following functions: $I_s('working') = (S^{in}, S^{out})$, $I_d('running') = (D^{in}, D^{out})$, $I('up') = (V^{in}, V^{out})$, $I_1('on_1') = (C_1^{in}, C_1^{out})$, and $I_2('on_2') = (C_2^{in}, C_2^{out})$. The ports of 'Service', 'DB', 'Container$_1$' and 'Container$_2$' are interconnected, and the connections are summarized as $\varphi^{in}: S^{in} \sqcup D^{in} \rightarrow C_1^{in} \sqcup C_2^{in}$ and $\varphi^{out}: C_1^{out} \sqcup C_2^{out} \rightarrow S^{out} \sqcup D^{out}$. Similarly, ports of 'Container$_1$', 'Container$_2$' and 'VM' are interconnected, so the following functions can be given: $\psi^{in}: C_1^{in} \sqcup C_2^{in} \rightarrow V^{in} \sqcup C_1^{out} \sqcup C_2^{out}$, and $\psi^{out}: V^{out} \rightarrow C_1^{out} \sqcup C_2^{out}$.

3.3 Service registration

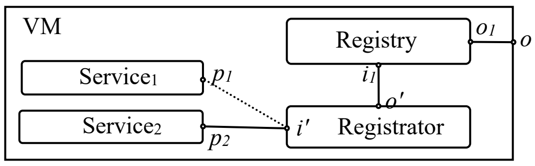

Figure 4. An example of service registration

Service registration is a process of recording states information about working services. The information is stored in a registry, and the data will be used during service discovery or service accessing. The registering work can be performed by a component, and basic tasks include: (a) collecting states of working services; (b) sending states to a registry. Figure 4 shows a simple scenario of service registration. The configuration contains: a 'VM' (which is a virtual machine) with output port o; two services with output ports p_1, p_2; a 'Registrator' (which is a registering component) with input port i' and output port o'; a 'Registry' with input port i_1 and output port o_1. In addition, "Service$_1$", "Service$_2$", "Registry", "Registrator" are containerized microservices.

The structure in Figure 4 is represented as:

- $service_1 := (M_1, I_1)$, and $M_1 = \{'working_1'\}$;
- $service_2 := (M_2, I_2)$, and $M_2 = \{'working_2'\}$;
- $registry := (M_r, I_r)$, and $M_r = \{'running'\}$;
- $registrator := (M_s, I_s)$, and $M_s = \{1, 2\}$;
- $vm := (M, I)$, and $M = \{'up'\}$;
- $sys := (\epsilon, \sigma): service_1 \otimes service_2 \otimes registry \otimes registrator \rightarrow vm$ with $\varphi_1 := (\varphi_1^{in}, \varphi_1^{out})$, $\varphi_2 := (\varphi_2^{in}, \varphi_2^{out})$, $\varphi_1, \varphi_2 \in Mor(WD)$, $\epsilon: M_1 \times M_2 \times M_r \times M_s \rightarrow Mor(WD)$, and $\sigma: M_1 \times M_2 \times M_r \times M_s \rightarrow M$.

The morphism (ϵ, σ) shows that the objects (which are *registrator*, *registry*, *service$_1$* and *service$_2$*) are embedded in the object *vm*. Function $\sigma: M_1 \times M_2 \times M_r \times M_s \rightarrow M$ could be defined easily. Considering that $\epsilon: M_1 \times M_2 \times M_r \times M_s \rightarrow Mor(WD)$, and M_s has two elements, the function ϵ can generate two wiring diagrams which are denoted as φ_1 and φ_2. The two diagrams are shown in Figure 4, the main difference is port i' can be connected to port p_1 or p_2. When port i' is connected to port p_1, the situation is represented as a wiring diagram φ_1; and for the situation that port i' is connected to port p_2, a wiring diagram φ_2 can be defined. Interfaces of components are defined as:

- $I('up') = (V^{in}, V^{out})$;
- $I_r('running') = (R^{in}, R^{out})$;
- $I_s = (S^{in}, S^{out})$;
- $I_1('working_1') = (S_1^{in}, S_1^{out})$;
- $I_2('working_2') = (S_2^{in}, S_2^{out})$.

(S_1^{in}, S_1^{out}), (S_2^{in}, S_2^{out}), (R^{in}, R^{out}), (S^{in}, S^{out}), (V^{in}, V^{out}) are input and output ports of *service₁*, *service₂*, *registry*, *registrator* and *vm* respectively. Since *vm*, *service₁* and *service₂* only have output ports, so $S_1^{in} = S_2^{in} = V^{in} = \emptyset$. The connections of ports are summarized as: $\varphi_1^{out}, \varphi_2^{out} : V^{out} \to S_1^{out} \sqcup S_2^{out} \sqcup S^{out} \sqcup R^{out}$ and $\varphi_1^{in}, \varphi_2^{in}: S^{in} \sqcup R^{in} \to S_1^{out} \sqcup S_2^{out} \sqcup S^{out} \sqcup R^{out}$. When port *i'* is connected to port *p₁*, φ_1^{in} is defined. Similarly, when port *i'* is connected to port *p₂*, φ_2^{in} can be given.

3.4 Proxies

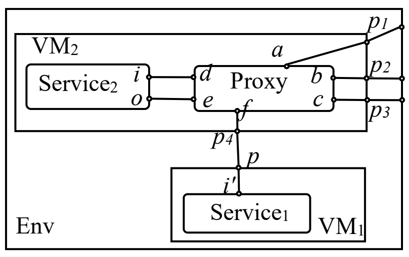

Figure 5. An example of using proxy

In applications, a proxy can be set up for isolating services from clients, and it can provide a unified accessing gateway for clients. Basic tasks of this component include: redirecting requests to different working services and delivering responses to different clients. Figure 5 shows a simple demonstration of using a proxy. The figure contains: a working environment named ‘Env’ with input ports *i₁*, *i₂* and output port *o₁*; two virtual machines (‘VM₁’ and ‘VM₂’) with input ports *p*, *p₁*, *p₂* and output ports *p₃*, *p₄*; three containerized services (which are ‘Proxy’, ‘Service₁’ and ‘Service₂’) with input ports *a*, *b*, *e*, *i*, *i'* and output ports *o*, *d*, *c*, *f*.

The demonstration is specified as follows: (1) ‘Service₁’ is deployed in ‘VM₁’, and it can be triggered by an input of port *i'*; (2) ‘Service₁’ and ‘Proxy’ are running in ‘VM₂’; (3) inputs of port *i₁* will be redirected to port *i'* through ports *p₁*, *a*, *f*, *p₄* and *p*; (4) inputs of port *i₂* will be redirected to port *i* through ports *p₂*, *b* and *d*; (5) outputs of port *o* will be send to port *o₁* through ports *e*, *c* and *p₃*. According to **Definition 1**, the structure in Figure 5 is represented as:

- $service_1 := (M_1, I_1)$, and $M_1 = \{'working_1'\}$;
- $service_2 := (M_2, I_2)$, and $M_2 = \{'working_2'\}$;
- $proxy := (M_p, I_p)$, and $M_p = \{'running'\}$;
- $vm_1 := (M', I')$, and $M' = \{'up_1'\}$;
- $vm_2 := (M'', I'')$, and $M'' = \{'up_2'\}$;
- $env := (M, I)$, and $M = \{'on'\}$.

All objects are organized as:

$$service_1 \otimes service_2 \otimes proxy \xrightarrow{(\epsilon_0, \sigma_0)} vm_1 \otimes vm_2 \xrightarrow{(\epsilon_1, \sigma_1)} env$$

where morphism (ϵ_0, σ_0) shows that *service₁*, *service₂* and *proxy* are embedded in the object *vm₁* and *vm₂*; morphism (ϵ_1, σ_1) shows that *vm₁* and *vm₂* are embedded in the object *env*. The following morphism can be defined:

$$system := (\epsilon, \sigma): service_1 \otimes service_2 \otimes proxy \to env$$

where $(\epsilon, \sigma) := (\epsilon_1, \sigma_1) \circ_{MDN} (\epsilon_0, \sigma_0)$, and:

- $\sigma_0: M_1 \times M_2 \times M_p \to M' \times M''$;
- $\sigma_1: M' \times M'' \to M$;
- $\epsilon_0: M_1 \times M_2 \times M_p \to \{\varphi\}$, and $\varphi := (\varphi^{in}, \varphi^{out})$;
- $\epsilon_1: M' \times M'' \to \{\psi\}$, and $\psi := (\psi^{in}, \psi^{out})$.

Function σ_0 and σ_1 could be defined easily. Interfaces of components are summarized as:

- $I_1('working_1') = (S_1^{in}, S_1^{out})$;
- $I_2('working_2') = (S_2^{in}, S_2^{out})$;
- $I('on') = (E^{in}, E^{out})$;
- $I'('up_1') = (V_1^{in}, V_1^{out})$;
- $I''('up_2') = (V_2^{in}, V_2^{out})$;
- $I_p('running') = (P^{in}, P^{out})$.

The ports of components are interconnected, so the following functions can be given:

- $\varphi^{in}: S_1^{in} \sqcup S_2^{in} \sqcup P^{in} \to V_1^{in} \sqcup P^{out} \sqcup V_2^{in} \sqcup S_2^{out}$, and $\varphi^{out}: V_2^{out} \to S_2^{out} \sqcup P^{out}$;
- $\psi^{in}: V_1^{in} \sqcup V_2^{in} \to E^{in} \sqcup V_2^{out}$, and $\psi^{out}: E^{out} \to V_2^{out}$.

4. DYNAMICS OF MICROSERVICES

This section discusses dynamics of models which are established in section 3. In this section, an isomorphism is denoted as $\cong$.

Table 1. A working process of containerized microservice in Figure 2

CURRENT STATES		OUTPUTS			INPUTS			NEXT STATES	
service	**db**	**o₀**	**o₁**	**o'**	**i₀**	**i₁**	**i'**	**service**	**db**
waiting	ready	null	null	null	get	null	null	conn	ready
conn	ready	null	open	null	null	null	open	conn	opened
conn	opened	null	open	connected	null	connected	open	querying	opened
querying	opened	null	query	connected	null	connected	query	querying	answering
querying	answering	null	query	data	null	data	query	disconn	answering
disconn	answering	data	close	data	null	data	close	disconn	ready
disconn	ready	data	close	null	null	null	close	waiting	ready

4.1 Behaviors of simple microservice

For the microservice in Figure 2, it is supposed that a procedure of data querying includes the following operations: (i) opening a database; (ii) selecting data; (iii) closing the database. Table 1 lists a working process of the containerized microservice, and the work have the following tasks: (1) ‘Service’ receives a request; (2) the ‘Service’ initiates a connection to ‘Database’; (3) the ‘Database’ returns the connection state; (4) the ‘Service’ sends a query command to the ‘Database’; (5) the ‘Database’ returns data; (6) the ‘Service’ closes the database connection, and responds to the request.

According to Table 1, interfaces of components in Figure 2 are specified as:

- $D^{in} := \{(\{i'\}, \tau_D^{in}) |\ \tau_D^{in}(i') = \{'null', 'open', 'query', 'close'\}\}$;
- $D^{out} := \{(\{o'\}, \tau_D^{out}) |\ \tau_D^{out}(o') =$

$\{'null', 'connected', 'data'\}\}$;

- $S^{in} \coloneqq \{(\{i_0, i_1\}, \tau_S^{in})\}$, and $\tau_S^{in}(i) \coloneqq \begin{cases} \{'get', 'null'\}, & if\ i = i_0 \\ \{'connected', 'data', 'null'\}, & if\ i = i_1 \end{cases}$
- $S^{out} \coloneqq \{(\{o_0, o_1\}, \tau_S^{out})\}$, and $\tau_S^{out}(o) \coloneqq \begin{cases} \{'data', 'null'\}, & if\ o = o_0 \\ \{'null', 'open', 'query', 'close'\}, & if\ o = o_1 \end{cases}$

For the object *db* (which is the model of the component 'Database'), a dynamical system can be defined as $P(M_d, I_d) \coloneqq (S_d, q_d, f_d)$, where the state set is $S_d = \{'ready', 'opened', 'answering'\}$; $q_d: S_d \to M_d$ is the underlying mode function, and $f_d \coloneqq (f_d^{in}, f_d^{out})$. The component has three states: 'ready', 'opened' and 'answering'. The 'ready' state indicates that the component is ready for work; the database is 'opened' after it is connected by others; and the 'answering' state shows that the database is making a response to its client. For a state $s \in S_d$ and an input $c \in \overline{D^{in}}$, the state update function is $f_d^{in}(s): \overline{D^{in}} \to S_d$, and $\overline{D^{in}} = \tau_D^{in}(i')$; specifically,

$$f_d^{in}(s)(c) \coloneqq \begin{cases} \text{'opened'}, & if\ s = \text{'ready'}, c = \text{'open'} \\ \text{'answering'}, & if\ s = \text{'opened'}, c = \text{'query'} \\ \text{'ready'}, & if\ s = \text{'answering'}, c = \text{'close'} \\ s, & otherwise \end{cases}$$

The readout function is $f_d^{out}: S_d \to \overline{D^{out}}$, where $\overline{D^{out}} = \tau_D^{out}(o')$; specifically, $f_d^{out}(s) \coloneqq \begin{cases} \text{'data'}, & if\ s = \text{'answering'} \\ \text{'connected'}, & if\ s = \text{'opened'} \\ \text{'null'}, & otherwise \end{cases}$

The dynamics of the object *service* (which is the model of the component 'Service') is given by $P(M_s, I_s) \coloneqq (S_s, q_s, f_s)$, where $q_s: S_s \to M_s$ is the underlying mode function; $S_s = \{'waiting', 'conn', 'querying', 'disconn'\}$ is the state set, and $f_s \coloneqq (f_s^{in}, f_s^{out})$. The 'Service' has four states. The 'waiting' state shows that the component is ready for work; the 'conn' state indicates that the component is opening a database; the 'querying' state shows that the component is working with a database component; the 'disconn' state indicates the component is sending a response. For a state $s \in S_s$ and an input $c \in \overline{S^{in}}$, the state update function is $f_s^{in}(s): \overline{S^{in}} \to S_s$, where $\overline{S^{in}} = \tau_S^{in}(i_0) \times \tau_S^{in}(i_1)$; specifically,

$$f_s^{in}(s)(c) \coloneqq \begin{cases} \text{'conn'}, & if\ s = \text{'waiting'}, c = (\text{'get'}, \text{'null'}) \\ \text{'querying'}, & if\ s = \text{'conn'}, c = (\text{'null'}, \text{'connected'}) \\ \text{'disconn'}, & if\ s = \text{'querying'}, c = (\text{'null'}, \text{'data'}) \\ \text{'waiting'}, & if\ s = \text{'disconn'}, c = (\text{'null'}, \text{'null'}) \\ s, & otherwise \end{cases}$$

The readout function is $f_s^{out}: S_s \to \overline{S^{out}}$, where $\overline{S^{out}} = \tau_S^{out}(o_0) \times \tau_S^{out}(o_1)$; specifically,

$$f_s^{out}(s) \coloneqq \begin{cases} (\text{'null'}, \text{'null'}), & if\ s = \text{'waiting'} \\ (\text{'null'}, \text{'open'}), & if\ s = \text{'conn'} \\ (\text{'null'}, \text{'query'}), & if\ s = \text{'querying'} \\ (\text{'data'}, \text{'close'}), & if\ s = \text{'disconn'} \end{cases}$$

For the object *container* (which is the model of the component 'Container'), it has interfaces $C^{in} \coloneqq \{(\{i\}, \tau_C^{in})\}$ and $C^{out} \coloneqq \{(\{o\}, \tau_C^{out})\}$. A dynamical system on the morphism $(\epsilon, \sigma): service \otimes db \to container$ can be summarized as $P(\epsilon, \sigma)(S, q, f) \coloneqq (S, r, g)$, where the state set is $S = S_s \times S_d$; $q = q_s \times q_d$ is the underlying mode function, and $\sigma: M_s \times M_d \to M_c$. The state update function is $f^{in} \coloneqq f_s^{in} \times f_d^{in}$, and the readout function is $f^{out} \coloneqq f_s^{out} \times f_d^{out}$. The function r is defined by $\sigma \circ q$, i.e., $S_s \times S_d \xrightarrow{q_s \times q_d} M_s \times M_d \xrightarrow{\sigma} M_c$. Function ϵ can be detailed as follows:

- $m = (\text{'working'}, \text{'running'})$;
- $\overline{\epsilon^{out}}(m): \tau_S^{out}(o_0) \times \tau_S^{out}(o_1) \times \tau_D^{out}(o') \to \tau_C^{out}(o)$;
- $\overline{\epsilon^{in}}(m): \tau_C^{in}(i) \times \tau_S^{out}(o_0) \times \tau_S^{out}(o_1) \times \tau_D^{out}(o') \to \tau_S^{in}(i_0) \times \tau_D^{in}(i') \times \tau_S^{in}(i_1)$.

Function $\overline{\epsilon^{out}}(m)$ shows that port *o* is depended on port *o₀*, *o₁*, or *o'*. Since port *o* is connected to port *o₀*, so outputs of port *o* are same as outputs of port *o₀*, so $\tau_C^{out}(o) \cong \tau_S^{out}(o_0)$ can be get. Function $\overline{\epsilon^{in}}(m)$ can be explained as follows: inputs of *i* are same as inputs of port *i₀*; outputs of *o₁* are same as inputs of port *i'*, and outputs of *o'* are same as inputs of port *i₁*. Therefore, $\tau_C^{in}(i) \cong \tau_S^{in}(i_0)$, $\tau_S^{out}(o_1) \cong \tau_D^{in}(i')$ and $\tau_D^{out}(o') \cong \tau_S^{in}(i_1)$ can be given.

Until now, $P(\epsilon, \sigma)(S, q, f) \coloneqq (S, r, g)$ can be defined according to Definition 2, and it is a dynamical system on the model of the microservice in Figure 2.

4.2 Interoperation of microservices

For Figure 3, the 'DB' can provide data persistence functions for others. It is supposed that the procedure of accessing 'DB' includes the following steps: (i) opening a database; (ii) selecting data; (iii) closing the database. Table 2 lists a working process which contains the following tasks: (1) 'Service' is triggered; (2) the 'Service' initiates a connection to 'DB'; (3) the 'DB' returns the connection state to the 'Service'; (4) the 'DB' receives a query command; (5) the 'DB' returns data; (6) the 'Service' closes the database connection, and responds to a request. According to the process, interfaces of 'Service' and 'DB' are detailed as follows.

- $S^{in} \coloneqq \{(\{i_0, i_1\}, \tau_S^{in})\}$, and $\tau_S^{in}(i) \coloneqq \begin{cases} \{'get', 'null'\}, & if\ i = i_0 \\ \{'connected', 'data', 'null'\}, & if\ i = i_1 \end{cases}$
- $S^{out} \coloneqq \{(\{o_0, o_1\}, \tau_S^{out})\}$, and $\tau_S^{out}(o) \coloneqq \begin{cases} \{'data', 'null'\}, & if\ o = o_0 \\ \{'null', 'open', 'query', 'close'\}, & if\ o = o_1 \end{cases}$
- $D^{in} \coloneqq \{(\{i'\}, \tau_D^{in}) \mid \tau_D^{in}(i') = \{'null', 'open', 'query', 'close'\}\}$;
- $D^{out} \coloneqq \{(\{o'\}, \tau_D^{out}) \mid \tau_D^{out}(o') = \{'null', 'connected', 'data'\}\}$.

Let dynamical systems $P(M_s, I_s)$ and $P(M_d, I_d)$ have the same definitions which are summarized in section 4.1. For the containers, interfaces of 'Container₁' and 'Container₂' are specified as:

- $C_1^{in} \coloneqq \{(\{p_1, p_4\}, \tau_1^{in})\}$, and $C_1^{out} \coloneqq \{(\{p_2, p_3\}, \tau_1^{out})\}$;
- $C_2^{in} \coloneqq \{(\{p_{in}\}, \tau_2^{in})\}$, and $C_2^{out} \coloneqq \{(\{p_{out}\}, \tau_2^{out})\}$.

Denote $P(\epsilon_0, \sigma_0)(S, q, f) \coloneqq (S, r, g)$ as a dynamical system on the morphism (ϵ_0, σ_0), where $S = S_s \times S_d$ is the state set; $q = q_s \times q_d$ is the underlying mode function, and $\sigma_0: M_s \times M_d \to M_1 \times M_2$. The state update function is $f^{in} \coloneqq f_s^{in} \times f_d^{in}$, and the readout function is $f^{out} \coloneqq f_s^{out} \times f_d^{out}$. The function r is defined by $\sigma_0 \circ q$, i.e., $S_s \times S_d \xrightarrow{q_s \times q_d} M_s \times M_d \xrightarrow{\sigma_0} M_1 \times M_2$. Function ϵ_0 can be detailed as follows:

- $m = (\text{'working'}, \text{'running'})$;

- $\overline{\epsilon_0^{out}}(m): \tau_S^{out}(o_0) \times \tau_S^{out}(o_1) \times \tau_D^{out}(o') \to \tau_1^{out}(p_2) \times \tau_1^{out}(p_3) \times \tau_2^{out}(p_{out})$;
- $\overline{\epsilon_0^{in}}(m): \tau_1^{in}(p_1) \times \tau_1^{in}(p_4) \times \tau_2^{in}(p_{in}) \to \tau_S^{in}(i_0) \times \tau_S^{in}(i_1) \times \tau_D^{in}(i')$.

Table 2. A working process of interoperating services in Figure 3

CURRENT STATES		OUTPUTS				INPUTS				NEXT STATES	
service	**db**	**o$_0$**	**o$_1$**	**o'**	**o**	**i$_0$**	**i$_1$**	**i'**	**i**	**service**	**db**
waiting	ready	null	null	null	null	get	null	null	get	conn	ready
conn	ready	null	open	null	null	null	null	open	null	conn	opened
conn	opened	null	open	connected	null	null	connected	open	null	querying	opened
querying	opened	null	query	connected	null	null	connected	query	null	querying	answering
querying	answering	null	query	data	null	null	data	query	null	disconn	answering
disconn	answering	data	close	data	data	null	data	close	null	disconn	ready
disconn	ready	data	close	null	data	null	null	close	null	waiting	ready

Function $\overline{\epsilon_0^{out}}(m)$ shows that outputs of 'Container$_1$' are same as outputs of 'Service', and outputs of 'Container$_2$' are same as outputs of 'DB'. So $\tau_1^{out}(p_2) \cong \tau_S^{out}(o_0)$, $\tau_1^{out}(p_3) \cong \tau_S^{out}(o_1)$, and $\tau_2^{out}(p_{out}) \cong \tau_D^{out}(o')$ can be given. Function $\overline{\epsilon_0^{in}}(m)$ shows that inputs of 'Service' are same as inputs of 'Container$_1$', and inputs of 'DB' are same as inputs of 'Container$_2$'. So $\tau_1^{in}(p_1) \cong \tau_S^{in}(i_0)$, $\tau_1^{in}(p_4) \cong \tau_S^{in}(i_1)$, and $\tau_2^{in}(p_{in}) \cong \tau_D^{in}(i')$ can be get. The dynamical system $P(\epsilon_0, \sigma_0)(S, q, f) := (S, r, g)$ can be defined according to Definition 2.

Interfaces of 'VM' are $V^{in} := \{(\{i\}, \tau^{in})\}$ and $V^{out} := \{(\{o\}, \tau^{out})\}$. Dynamical system on the morphism (ϵ_1, σ_1) is defined as $P(\epsilon_1, \sigma_1)(S, r, g) := (S, t, h)$, where $S = S_s \times S_d$ is the state set; r is the underlying mode function; $\sigma_1: M_1 \times M_2 \to M$ and $g := (g^{in}, g^{out})$. The function t is defined by $\sigma_1 \circ r$, i.e., $S_s \times S_d \overset{r}{\to} M_1 \times M_2 \overset{\sigma_1}{\to} M$. Function ϵ_1 can be detailed as follows:

- $m' = (\text{'}on_1\text{'}, \text{'}on_2\text{'})$;
- $\overline{\epsilon_1^{out}}(m'): \tau_1^{out}(p_2) \times \tau_1^{out}(p_3) \times \tau_2^{out}(p_{out}) \to \tau^{out}(o)$;
- $\overline{\epsilon_1^{in}}(m'): \tau^{in}(i) \times \tau_2^{out}(p_{out}) \times \tau_1^{out}(p_3) \times \tau_1^{out}(p_2) \to \tau_1^{in}(p_1) \times \tau_1^{in}(p_4) \times \tau_2^{in}(p_{in})$.

Function $\overline{\epsilon_1^{out}}(m')$ shows that outputs of 'VM' are same as outputs of port p_2, so $\tau^{out}(o) \cong \tau_1^{out}(p_2) \cong \tau_S^{out}(o_0)$. Function $\overline{\epsilon_1^{in}}(m')$ shows that inputs of port p_1 are same as inputs of port i, so $\tau^{in}(i) \cong \tau_1^{in}(p_1)$; inputs of port p_4 are same as outputs of p_{out}, so $\tau_2^{out}(p_{out}) \cong \tau_1^{in}(p_4)$; and inputs of port p_{in} are same as outputs of p_3, so $\tau_2^{in}(p_{in}) \cong \tau_1^{out}(p_3)$.

Until now, $P(\epsilon_1, \sigma_1)(S, r, g) := (S, t, h)$ can be defined according to Definition 2, and dynamics of the morphism $(\epsilon, \sigma): service \otimes db \to vm$ can be checked.

4.3 Registering microservices

For Figure 4, it is supposed that 'Service$_1$' and 'Service$_2$' is able to share their working states, so their interfaces are specified as:

- $S_1^{in} := \{(\emptyset, !) | !: \emptyset \to set, set \in Set\}$;
- $S_1^{out} := \{(\{p_1\}, \tau_1^{out}) | \tau_1^{out}(p_1) = \{\text{'}ok\text{'}, \text{'}failed\text{'}\}\}$;
- $S_2^{in} := \{(\emptyset, !) | !: \emptyset \to set, set \in Set\}$;
- $S_2^{out} := \{(\{p_2\}, \tau_2^{out}) | \tau_2^{out}(p_2) = \{\text{'}ok\text{'}, \text{'}failed\text{'}\}\}$.

'Service$_1$' and 'Service$_2$' have no input ports, and $!: \emptyset \to set$ is a special function that it sends an empty set $\emptyset$ to a given set. A dynamical system $P(M_1, I_1) := (S_1, q_1, f_1)$ can be defined on the object *service$_1$*, where $q_1: S_1 \to M_1$ is the underlying mode function; $S_1 = \{\text{'}error\text{'}, \text{'}no\_error\text{'}\}$ is the set of states, and $f_1 := (f_1^{in}, f_1^{out})$. It is obvious that the service has two different states: 'error' and 'no_error'. Function $q_1: S_1 \to M_1$ can be defined easily, since M_1 has one element. The state update function is $f_1^{in} := id_{s_1}: S_1 \to S_1$. The readout function is $f_1^{out}: S_1 \to \overline{S_1^{out}}$; specifically,

$$f_1^{out}(s) := \begin{cases} \text{'}ok\text{'}, & if\ s = \text{'}no\_error\text{'} \\ \text{'}failed\text{'}, & if\ s = \text{'}error\text{'} \end{cases}$$

Table 3. An example of registering microservices in Figure 4

CURRENT STATES		OUTPUTS				INPUTS		NEXT STATES	
registrator	**registry**	**p$_1$**	**p$_2$**	**o'**	**o$_1$**	**i$_1$**	**i'**	**registrator**	**registry**
(ok, 1)	recording	ok	failed	ok$_2$	done	ok$_2$	ok	(ok, 2)	recording
(ok, 2)	recording	ok	failed	ok$_1$	done	ok$_1$	failed	(failed, 1)	recording
(failed, 1)	recording	ok	failed	failed$_2$	done	failed$_2$	ok	(ok, 2)	recording

Dynamical system $P(M_2, I_2) := (S_2, q_2, f_2)$ for the object *service$_2$* is defined similarly as $P(M_1, I_1)$. In Figure 4, the work of service registration is performed by the component 'Registrator'. A working process is listed in Table 3, and it contains the following steps: (1) 'Registrator' collects state of 'Service$_1$', and reports last state of 'Service$_2$'; (2) the 'Registrator' collects state of the 'Service$_2$', and reports last state of the 'Service$_1$'; (3) the 'Registrator' collects state of the 'Service$_1$', and reports last state of the 'Service$_2$'. The 'Registrator' is defined as (M_s, I_s), where $M_s = \{1, 2\}$ and $I_s: M_s \to Ob(WD)$. For the component, *mode* 1 and *mode* 2 have the same interfaces, so $I_s(1) = I_s(2) = (S^{in}, S^{out})$, where

- $S^{in} := \{(\{i'\}, \tau_S^{in}) | \tau_S^{in}(i') = \{\text{'}ok\text{'}, \text{'}failed\text{'}\}\}$;
- $S^{out} := \{(\{o'\}, \tau_S^{out}) | \tau_S^{out}(o') = \{\text{'}ok_1\text{'}, \text{'}failed_1\text{'}, \text{'}ok_2\text{'}, \text{'}failed_2\text{'}\}\}$.

Let $P(M_s, I_s) := (S_s, q_s, f_s)$ be a dynamical system of the object *registrator*, where the state set is $S_s := \{\text{'}ok\text{'}, \text{'}failed\text{'}\} \times M_s$; the underlying mode function is $q_s: \{\text{'}ok\text{'}, \text{'}failed\text{'}\} \times M_s \to M_s$, and $f_s := (f_s^{in}, f_s^{out})$. The update function is $f_s^{in}(v, i)(v') := (v', (i\ mod\ 2) + 1)$, where $v', v \in \{\text{'}ok\text{'}, \text{'}failed\text{'}\}$ and $i, ((i\ mod\ 2) + 1) \in M_s$. The readout function is $f_s^{out}: S_s \to \overline{S^{out}}$; specifically,

$$f_s^{out}(v, i) := \begin{cases} \text{'}ok_1\text{'}, & if\ v = \text{'}ok\text{'}, i = 2 \\ \text{'}failed_1\text{'}, & if\ v = \text{'}failed\text{'}, i = 2 \\ \text{'}ok_2\text{'}, & if\ v = \text{'}ok\text{'}, i = 1 \\ \text{'}failed_2\text{'}, & if\ v = \text{'}failed\text{'}, i = 1 \end{cases}$$

Interfaces of 'Registry' are detailed as follows:

- $R^{in} := \{(\{i_1\}, \tau_R^{in}) | \tau_R^{in}(i_1) = \{'ok_1', 'failed_1', 'ok_2', 'failed_2'\}\}$;
- $R^{out} := \{(\{o_1\}, \tau_R^{out}) | \tau_R^{out}(o_1) = \{'done'\}\}$.

A dynamical system on the object *registry* is $P(M_r, I_r) := (S_r, q_r, f_r)$, where $S_r = \{'recording'\}$ is the state set; $q_r: S_r \to M_r$ is the underlying mode function, and $f_r := (f_r^{in}, f_r^{out})$. The state update function is $f_r^{in}(s): \overline{R^{in}} \to S_r$, where $f_r^{in}(s)(r) := s$ with $s \in S_r$ and $r \in \tau_R^{in}(i_1)$. The readout function is $f_r^{out}(s) :=$ '*done*'.

The last component is 'VM'. Let $V^{in} := \{(\emptyset, !) | !: \emptyset \to set, set \in Set\}$ and $V^{out} := \{(\{o\}, \tau_V^{out})\}$ be its interfaces. For $(\epsilon, \sigma): service_1 \otimes service_2 \otimes registry \otimes registrator \to vm$, let $P(\epsilon, \sigma)(S, q, f) := (S, r, g)$ be a dynamical system, where the state set is $S = S_1 \times S_2 \times S_r \times S_s$; the underlying mode function is $q = q_1 \times q_2 \times q_r \times q_s: S_1 \times S_2 \times S_r \times S_s \to M_1 \times M_2 \times M_r \times M_s$; and $\sigma: M_1 \times M_2 \times M_r \times M_s \to M$. The state update function is $f^{in} := f_1^{in} \times f_2^{in} \times f_r^{in} \times f_s^{in}$, and the readout function is $f^{out} := f_1^{out} \times f_2^{out} \times f_r^{out} \times f_s^{out}$. The function r is defined by $\sigma \circ q$, i.e., $S \xrightarrow{q} M_1 \times M_2 \times M_r \times M_s \xrightarrow{\sigma} M$. ϵ can be detailed as follows:

- $m_1 = ('working_1', 'working_2', 'running', 1)$;
- $m_2 = ('working_1', 'working_2', 'running', 2)$;
- $\overline{\epsilon^{out}}(m_1), \overline{\epsilon^{out}}(m_2): \tau_1^{out}(p_1) \times \tau_2^{out}(p_2) \times \tau_S^{out}(o') \times \tau_R^{out}(o_1) \to \tau_V^{out}(o)$;
- $\overline{\epsilon^{in}}(m_1), \overline{\epsilon^{in}}(m_2): ! \times \tau_1^{out}(p_1) \times \tau_2^{out}(p_2) \times \tau_S^{out}(o') \times \tau_R^{out}(o_1) \to ! \times ! \times \tau_S^{in}(i') \times \tau_R^{in}(i_1)$.

Function $\overline{\epsilon^{out}}$ shows that port *o* is depended on ports *p₁*, *p₂*, *o₁*, and *o'*. In fact, outputs of port *o* are same as outputs of port *o₁*, so $\tau_V^{out}(o) \cong \tau_R^{out}(o_1)$ can be given. Function $\overline{\epsilon^{in}}$ shows that ports *i'* and *i₁* are depended on *p₁*, *p₂*, *o₁*, and *o'*. In the case of *m₁*, outputs of port *p₁* are same as inputs of *i'*, therefore $\tau_1^{out}(p_1) \cong \tau_S^{in}(i')$ and $\tau_R^{in}(i_1) \cong \tau_S^{out}(o')$ can be get. In the case of *m₂*, outputs of port *p₂* are same as inputs of *i'*, so $\tau_2^{out}(p_2) \cong \tau_S^{in}(i')$ and $\tau_R^{in}(i_1) \cong \tau_S^{out}(o')$ can be given.

Until now, $P(\epsilon, \sigma)(S, q, f) := (S, r, g)$ can be defined according to Definition 2, dynamics of the morphism $(\epsilon, \sigma): service_1 \otimes service_2 \otimes registry \otimes registrator \to vm$ can be checked.

4.4 Accessing services

Table 4. A process of accessing services in Figure 5

CURRENT STATES			INPUTS AND OUTPUTS						NEXT STATES		
service₁	service₂	proxy	output port	c	d	f	o		service₁	service₂	proxy
			input port	a	b	e	i	i'			
ready	ready	free	output	null	null	null	null		ready	ready	set₁
			input	cmd	null	null	null	null			
ready	ready	set₁	output	null	null	cmd	null		executing	ready	free
			input	null	null	null	null	cmd			
executing	ready	free	output	null	null	null	null		ready	ready	set₂
			input	null	req	null	null	null			
ready	ready	set₂	output	null	req	null	null		ready	answering	set₂
			input	null	null	null	req	null			
ready	answering	set₂	output	null	req	null	resp		ready	answering	put₂
			input	null	null	resp	req	null			
ready	answering	put₂	output	resp	null	null	resp		ready	ready	free
			input	null	null	resp	null	null			

In Figure 5, interfaces of 'Service₁', 'Service₂' and 'Proxy' are specified as:

- $S_1^{in} := \{(\{i'\}, \tau^{in}) | \tau^{in}(i') = \{'cmd', 'null'\}\}$;
- $S_1^{out} := \{(\emptyset, !) | !: \emptyset \to set, set \in Set\}$;
- $S_2^{in} := \{(\{i\}, \tau_S^{in}) | \tau_S^{in}(i) = \{'req', 'null'\}\}$;
- $S_2^{out} := \{(\{o\}, \tau_S^{out}) | \tau_S^{out}(o) = \{'resp', 'null'\}\}$;
- $P^{in} := \{(\{a, b, e\}, \tau_P^{in})\}$;
- $P^{out} := \{(\{c, d, f\}, \tau_P^{out})\}$.

Table 4 lists a working process which contains the following steps: (1) a request is received from port *a*; (2) 'Proxy' redirects the request to port *i'*; (3) 'Service₁' executes, and a new request is received from port *b*; (4) the 'Proxy' redirects the request to port *i*; (5) 'Service₂' executes and returns a response; (6) the 'Proxy' redirects the response to port *o₁*.

According to Table 4, a dynamical system on the object *service₁* is $P(M_1, I_1) := (S_1, q_1, f_1)$, where $f_1 := (f_1^{in}, f_1^{out})$,

$q_1: S_1 \to M_1$ and $S_1 = \{'ready', 'executing'\}$. The state set S_1 shows that the ‘Service₁’ has two states: ‘ready’ and ‘executing’. Function $q_1: S_1 \to M_1$ can be defined easily, since *M₁* has only one element. For a state $s \in S_1$ and an input $c \in \overline{S_1^{in}}$, the state update function is $f_1^{in}(s): \overline{S_1^{in}} \to S_1$, where $\overline{S_1^{in}} = \tau^{in}(i')$; specifically,

$$f_1^{in}(s)(c) := \begin{cases} \text{‘}executing\text{’}, & if\ s = \text{‘}ready\text{’}, c = \text{‘}cmd\text{’} \\ \text{‘}ready\text{’}, & if\ s = \text{‘}executing\text{’}, c = \text{‘}null\text{’} \\ s, & otherwise \end{cases}$$

The readout function is $f_1^{out}: S_1 \to \overline{S_1^{out}}$, so $\overline{S_1^{out}} = set$ with $set \in Set$.

Let $P(M_2, I_2) := (S_2, q_2, f_2)$ be a dynamical system of the object *service₂*, where $f_2 := \left(f_2^{in}, f_2^{out}\right)$, $q_2: S_2 \to M_2$, and $S_2 = \{\text{‘}ready\text{’}, \text{‘}answering\text{’}\}$. Function $q_2: S_2 \to M_2$ can be defined easily. For a state $s \in S_2$ and an input $c \in \overline{S_2^{in}}$, the state update function is $f_2^{in}(s): \overline{S_2^{in}} \to S_2$, where $\overline{S_2^{in}} = \tau_S^{in}(\mathrm{i})$; specifically,

$$f_2^{in}(s)(c) := \begin{cases} \text{‘}answering\text{’}, & if\ s = \text{‘}ready\text{’}, c = \text{‘}req\text{’} \\ \text{‘}ready\text{’}, & if\ s = \text{‘}answering\text{’}, c = \text{‘}null\text{’} \\ s, & otherwise \end{cases}$$

The readout function is $f_2^{out}: S_2 \to \overline{S_2^{out}}$, where $\overline{S_2^{out}} = \tau_S^{out}(o)$; in particular,

$$f_2^{out}(s) := \begin{cases} \text{‘}resp\text{’}, & if\ s = \text{‘}answering\text{’} \\ \text{‘}null\text{’}, & otherwise \end{cases}$$

Dynamical system of the ‘Proxy’ is defined as $P\left(M_p, I_p\right) := \left(S_p, q_p, f_p\right)$, where the state set is $S_p = \{'free', 'set_1', 'set_2', 'put_2'\}$; the underlying mode function is $q_p: S_p \to M_p$, and $f_p := (f_p^{in}, f_p^{out})$. The component ‘Proxy’ has four states. The state ‘free’ indicates that the component is ready for work. State ‘set₁’ shows that the component is redirecting a request to ‘Service₁’, and ‘set₂’ shows that the component is redirecting a request to ‘Service₂’. ‘put₂’ means that the ‘Proxy’ is delivering a ‘Service₂’ response. For a state $s \in S_p$, the state update function is $f_p^{in}(s): \overline{S_p^{in}} \to S_p$, where $\overline{S_p^{in}} = \tau_P^{in}(a) \times \tau_P^{in}(b) \times \tau_P^{in}(e)$; in particular,

$$f_p^{in}(s)(c) := \begin{cases} \text{‘}set_1\text{’}, & if\ s = \text{‘}free\text{’},\ c = (\text{‘}cmd\text{’}, \text{‘}null\text{’}, \text{’}null\text{’}) \\ \text{‘}free\text{’}, & if\ s = \text{‘}set_1\text{’},\ c = (\text{‘}null\text{’}, \text{‘}null\text{’}, \text{’}null\text{’}) \\ \text{‘}set_2\text{’}, & if\ s = \text{‘}free\text{’},\ c = (\text{‘}null\text{’}, \text{‘}req\text{’}, \text{’}null\text{’}) \\ \text{‘}put_2\text{’}, & if\ s = \text{‘}set_2\text{’},\ c = (\text{‘}null\text{’}, \text{‘}null\text{’}, \text{’}resp\text{’}) \\ \text{‘}free\text{’}, & if\ s = \text{‘}put_2\text{’},\ c = (\text{‘}null\text{’}, \text{‘}null\text{’}, \text{’}resp\text{’}) \\ s, & otherwise \end{cases}$$

The readout function is $f_p^{out}: S_p \to \overline{S_p^{out}}$, where $\overline{S_p^{out}} = \tau_P^{out}(c) \times \tau_P^{out}(d) \times \tau_P^{out}(f)$; in particular,

$$f_p^{out}(s) := \begin{cases} (\text{‘}null\text{’}, \text{‘}null\text{’}, \text{‘}cmd\text{’}), & if\ s = \text{‘}set_1\text{’} \\ (\text{‘}null\text{’}, \text{‘}req\text{’}, \text{‘}null\text{’}), & if\ s = \text{‘}set_2\text{’} \\ (\text{‘}resp\text{’}, \text{‘}null\text{’}, \text{‘}null\text{’}), & if\ s = \text{‘}put_2\text{’} \\ (\text{‘}null\text{’}, \text{‘}null\text{’}, \text{‘}null\text{’}), & otherwise \end{cases}$$

Interfaces of ‘VM₁’ and ‘VM₂’ are detailed as follows:

- $V_1^{in} := \left\{\left(\{p\}, \tau_{V1}^{in}\right)\right\}$, and $V_1^{out} := \{(\emptyset, !) | !: \emptyset \to set, set \in Set\}$;
- $V_2^{in} := \left\{\left(\{p_1, p_2\}, \tau_{V2}^{in}\right)\right\}$, and $V_2^{out} := \{(\{p_3, p_4\}, \tau_{V2}^{out})\}$.

Let $P(\epsilon_0, \sigma_0)(S, q, f) := (S, r, g)$ be a dynamical system on the morphism (ϵ_0, σ_0), where $S = S_1 \times S_2 \times S_p$ is the state set; $q = q_1 \times q_2 \times q_p$ is the underlying mode function, and $\sigma_0: M_1 \times M_2 \times M_p \to M' \times M''$. The update function is $f^{in} := f_1^{in} \times f_2^{in} \times f_p^{in}$, and readout function is $f^{out} := f_1^{out} \times f_2^{out} \times f_p^{out}$. The function r is defined by $\sigma_0 \circ q$, i.e., $S_1 \times S_2 \times S_p \xrightarrow{q_1 \times q_2 \times q_p} M_1 \times M_2 \times M_p \overset{\sigma_0}{\to} M' \times M''$. Function ϵ_0 can be detailed as follows:

- $m = (\text{‘}working_1\text{’}, \text{‘}working_2\text{’}, \text{‘}running\text{’})$;
- $\overline{\epsilon_0^{out}}(m): \tau_S^{out}(o) \times \tau_P^{out}(c) \times \tau_P^{out}(d) \times \tau_P^{out}(f) \to \tau_{V2}^{out}(p_3) \times \tau_{V2}^{out}(p_4)$;
- $\overline{\epsilon_0^{in}}(m): \tau_{V1}^{in}(p) \times \tau_P^{out}(c) \times \tau_P^{out}(d) \times \tau_P^{out}(f) \times \tau_{V2}^{in}(p_1) \times \tau_{V2}^{in}(p_2) \times \tau_S^{out}(o) \to \tau^{in}(i') \times \tau_S^{in}(i) \times \tau_P^{in}(a) \times \tau_P^{in}(b) \times \tau_P^{in}(e)$.

Function $\overline{\epsilon_0^{out}}(m)$ shows that outputs of port *p₃* are same as outputs of port *c*, and outputs of port *p₄* are same as outputs of port *f*. So $\tau_{V2}^{out}(p_3) \cong \tau_P^{out}(c)$ and $\tau_{V2}^{out}(p_4) \cong \tau_P^{out}(f)$ can be get. Function $\overline{\epsilon_0^{in}}(m)$ shows that inputs of port *i'* are same as inputs of port *p*; inputs of port *a*, *b* are same as inputs of port *p₁*, *p₂* respectively; and inputs of port *e*, *i* are same as outputs of port *o*, *d* respectively. So $\tau_P^{in}(a) \cong \tau_{V2}^{in}(p_1)$, $\tau_P^{in}(b) \cong \tau_{V2}^{in}(p_2)$, $\tau_{V1}^{in}(p) \cong \tau^{in}(i')$, $\tau_P^{in}(e) \cong \tau_S^{out}(o)$, and $\tau_S^{in}(i) \cong \tau_P^{out}(d)$ can be get. $P(\epsilon_0, \sigma_0)(S, q, f) := (S, r, g)$ can be defined according to Definition 2.

Interfaces of ‘Env’ are defined as: $E^{in} := \left\{\left(\{i_1, i_2\}, \tau_E^{in}\right)\right\}$ and $E^{out} := \{(\{o_1\}, \tau_E^{out})\}$. A dynamical system on the morphism (ϵ_1, σ_1) is defined as $P(\epsilon_1, \sigma_1)(S, r, g) := (S, t, h)$, where $S = S_1 \times S_2 \times S_p$ is the state set; r is the underlying mode function; $\sigma_1: M' \times M'' \to M$ and $g := (g^{in}, g^{out})$. The function t is defined by $\sigma_1 \circ r$, i.e., $S_1 \times S_2 \times S_p \overset{r}{\to} M' \times M'' \overset{\sigma_1}{\to} M$. Function ϵ_1 can be detailed as follows:

- $m' = (\text{‘}up_1\text{’}, \text{‘}up_2\text{’})$;
- $\overline{\epsilon_1^{out}}(m'): \tau_{V2}^{out}(p_3) \times \tau_{V2}^{out}(p_4) \to \tau_E^{out}(o_1)$;
- $\overline{\epsilon_1^{in}}(m'): \tau_E^{in}(i_1) \times \tau_E^{in}(i_2) \times \tau_{V2}^{out}(p_3) \times \tau_{V2}^{out}(p_4) \to \tau_{V2}^{in}(p_1) \times \tau_{V2}^{in}(p_2) \times \tau_{V1}^{in}(p)$.

Function $\overline{\epsilon_1^{out}}(m')$ shows that outputs of ‘Env’ are same as outputs of port *p₃*, so $\tau_E^{out}(o_1) \cong \tau_{V2}^{out}(p_3)$ can be get. Function $\overline{\epsilon_1^{in}}(m')$ shows that inputs of port *p* are same as outputs of port *p₄*, i.e. $\tau_{V1}^{in}(p) \cong \tau_{V2}^{out}(p_4)$; inputs of port *p₁*, *p₂* are same as inputs of port *i₁*, *i₂* respectively, so $\tau_{V2}^{in}(p_1) \cong \tau_E^{in}(i_1)$ and $\tau_{V2}^{in}(p_2) \cong \tau_E^{in}(i_2)$ can be get.

Until now, $P(\epsilon_1, \sigma_1)(S, r, g) := (S, t, h)$ can be defined according to Definition 2, and dynamics can be checked.

5. A BRIEF DISCUSSION ABOUT APPLICATION IMPLEMENTATIONS

Conceptions or components, which are discussed in section 3 and 4, can be implemented by using software tools. For constructing a basic application environment, Vagrant (https://www.vagrantup.com) can be used for defining a configurable working environment. VirtualBox (https://www.virtualbox.org) or other products can be installed and configured as virtual machines in a working environment, and instances of software containers can be deployed and managed in virtual machines. For applications, many open source projects can be adopted and applied. For example, Registrator (https://gliderlabs.com/registrator/latest/) is able to provide registration functions for containerized microservices, and it supports different pluggable registries; Consul (https://www.consul.io) is a persistence system, and it can be set up as a registry for storing states of services; nginx (https://nginx.org) or other servers can be used as a reverse proxy.

6 CONCLUSIONS

This paper uses the category of mode-dependent networks and dynamical systems to model structures and functions of containerized microservices. Formal models are established around some issues of microservice-oriented applications. The results and relevant discussions show that structural and functional models of containerized microservices can be formed by using the category-based tools. Therefore, the modelling tools are applicable to modelling microservice-oriented applications, and they are able to contribute to clearly revealing important technical features of applications.

ACKNOWLEDGEMENT

This work is supported by the Open Foundation of Key Laboratory in Software Engineering of Yunnan Province under Grant No. 2015SE101.

REFERENCES

[1] Nadareishvili I, Mitra R, McLarty M, Amundsen M. (2016). Microservice Architecture: Aligning Principles, Practices, and Culture, O'Reilly Media.

[2] Bui T. (2016). Analysis of docker security, eprint arxiv: 1501.02967, https://arxiv.org/abs/1501.02967, accessed on 8 Nov 2017.

[3] Balalaie A, Heydarnoori A, Jamshidi P. (2016). Microservices architecture enables devops: an experience report on migration to a cloud-native architecture. IEEE Software 33(3): 42-52.

[4] Toffetti G, Brunner S, Blöchlinger M. et al. (2015). An architecture for self-managing microservices, In Proceedings of the 1st International Workshop on Automated Incident Management in Cloud (AIMC '15), Bordeaux, France, pp. 19-24.

[5] Vresk T, Cavrak I. (2016). Architecture of an interoperable IoT platform based on microservices, In Proceedings of the 39th International Convention on Information and Communication Technology Electronics and Microelectronics (MIPRO 2016), pp. 1196-1201.

[6] Namiot D, Sneps-Sneppe M. (2014). On micro-services architecture, International Journal of Open Information Technologies 2(9): 24-27.

[7] Dragoni N, Giallorenzo S, Lafuente AL, et al. (2016). Microservices: yesterday, today, and tomorrow, eprint arxiv: 1606.04036, https://arxiv.org/abs/1606.04036, accessed on Nov. 8, 2017.

[8] Spivak DI, Tan JZ. (2015). Nesting of dynamical systems and mode-dependent networks, eprint arxiv: 1502.07380, https://arxiv.org/abs/1502.07380, Nov. 8, 2017.

[9] Spivak DI. (2014). Category Theories for Sciences, 1st edition, The MIT Press.

[10] Newman S. (2015). Building Microservices, O'Reilly Media.

[11] Lewis J, Fowler M. (2014). Microservices, http:// martinfowler.com/articles/microservices.html, accesed on Nov. 8, 2017.

[12] Montesi F, Weber J. (2016). Circuit Breakers, Discovery, and API gateways in microservices, eprint arxiv: 1609.05830, https://arxiv.org/abs/1609.05830, Nov. 8, 2017.

[13] Spivak DI. (2013). The operad of wiring diagrams: Formalizing a graphical language for databases, recursion, and plug-and-play circuits, eprint arxiv: 1305.0297, https://arxiv.org/abs/1305.0297, accessed on Nov. 8, 2017.

[14] Rupel D, Spivak DI. (2013). The operad of temporal wiring diagrams: formalizing a graphical language for discrete-time processes, eprint arxiv: 1307.6894, https://arxiv.org/abs/1307.6894, accessed on Nov. 8, 2017.

[15] Vagner D, Spivak DI, Lerman E. (2015). Algebras of open dynamical systems on the operad of wiring diagrams, eprint arxiv: 1408.1598, https://arxiv.org/abs/1408.1598, accessed on Nov. 8, 2017.

[16] Soltesz S, Pötzl H, Fiuczynski ME, et al. (2007). Container-based operating system virtualization: A scalable, high-performance alternative to hypervisors, In Proceedings of the 2nd ACM SIGOPS/EuroSys European Conference on Computer Systems 2007 (EuroSys '07), Lisbon, Portugal, pp. 275-287.

[17] Lawvere FW, Schanuel SH. (2009). Conceptual Mathematics: A First Introduction to Categories, 2nd edition, Cambridge University Press.

[18] Ouyang C, Verbeek E, Van Der Aalst WMP, et al. (2007). Formal semantics and analysis of control flow in WS-BPEL, Science of Computer Programming 67: 162-198.

[19] Roman D, Keller U, Lausen H, et al. (2005). Web service modeling ontology, Applied Ontology 1(1): 77–106.

[20] Skogan D, Grønmo R, Solheim I. (2004). Web service composition in UML, In Proceedings of Enterprise Distributed Object Computing Conference, pp. 47-57.

[21] OASIS standard, Web Services Business Process Execution Language (WS-BPEL), https://docs.oasis-open.org/wsbpel/2.0/plnktype