New Approach to Typified Microservice Composition and Discovery

Submitted: 8th October 2018; accepted: 7th February 2019

Nikita Gerasimov

DOI: 10.14313/JAMRIS_1-2019/10

Abstract:
Several problems related to work reliability appear while building service-oriented systems. The first problem consists of the lack of static typing and the lack of inter-service data type checking. The second one consists of the high connectivity of services. The article shows an example of the strong and static polymorphic type system and the type check algorithm. The service-contract and the contract discovery concepts for universal service linking and type verification are described. After theoretic results had been realized in a service form, they were applied in practice in the real system, which improved its reliability. Also, technical realization decreased services connectivity, which promoted system quality increase. However, the increased complexity of the resulting system leveled advanced reliability.

Keywords: *microservice architecture, static typing, SOA, services composition, service contract*

1. Introduction

Development of modern, convenient multi-logic systems often bases on service or microservice oriented architectures (SOA). SOA means that application logic is divided into several self-sufficient components, providing separate tasks realization [11]. Every component has a single responsibility.

The advantages of SOA get obvious when developing high-loaded web-services: separate components encourage horizontal scaling. Every service, as usual, does not require various dependencies or specific configuration. Separate components with a limited range of tasks have a lower cost of maintenance and delivery to production. Also, logic separation motivates developers to design scalable services.

However, SOA has also disadvantages: separation of service logic leads to the development of a communication layer between components. Other disadvantages are dependency management, service linking, type consistency of interfaces, an inequality of providing and using interfaces [5]. During the development of a solid application programming language's features solve mentioned issues. After any application function moves outside the main project, interoperation problems may appear.

Various frameworks and approaches suggest the ways of system decomposition but do not suggest any ways for static checking of types consistency. Just as at dynamic-typing language this fact leads to an increase in working system instability.

RPC-frameworks like Google Protobuf or Apache Thrift partially solve static type checking by providing client and server code generation based on API definition. Mentioned solutions enable ensuring at development stage that client and server would use the identical protocol. However code generation becomes less trivial while using JSON/XML-PRC, REST or using event-driven architectures. Moreover, they all provide synchronous calls and responses.

Next problem is less critical. Detection of outdated API usage can be nontrivial in complex systems with various components. Lack of automatized control over API usage leads to the possibility of important component disabling. Let us consider the real case: an outdated service A provides statistics collection and sending with mailing service once per month. Logs analysis proves that there were no API calls during last three weeks (for example); that is why the service can be disabled.

Finally, we have two main problems:

- the absence of strong type system with static checking for SOA that leads to potential stability decrease
- the absence of dependency control for SOA leading to possible breaking system in runtime after disabled outdated APIs

Therefore, the primary goal of this research is to increase the stability of SOA-based systems and decrease runtime errors.

There are two stages to reach the goal:

- improve the existing approach to service API typification to check types statically
- develop service that should control API dependencies in the SOA system

Much of the work presented here touches the description of a new tool providing the achievement of formulated goals. New service purpose is close to service-discovery systems purpose: to detect suitable components over the network automatically [13]. The main objective of the new service is checking of type consistency for providing and using API definitions. We call it "contract discovery".

The next section demonstrates state of the art. The third section defines the proposed type system and the type checking algorithm to be realized in contract discovery service. Section 4 surveys the concept of contract and how contract-discovery service provides a client to service linking. Section 5 describes our realization details of contract discovery service. Section 6 illustrates our experience in application such service

to the real microservice-oriented event-driven system.

This paper is an extended version of conference paper "Static typing and dependency management for SOA" [7].

2. State of the Art

2.1. Approaches to Service Composition

Since the microservice concept is the development of the service one, we can look over methods and ideas of composition for SOA.

Several well-known standards of organizing service communication exist, e.g. WS-BPEL, WS-CDL, BPML, ebXML, OWL-S, WSMF, etc. Though the mentioned definitions suit business process specification, they cover various complex scenarios, for example, WS-BPEL or WS-CDL allows defining user roles [12]. According to several definitions, microservices do not cover hole processes, but they implement limeted logic operations. Therefore microservice composition can not be expressed in terms of mentioned standards.

Except for well-known specifications like mentioned above, serveral research projects exist: eFlow, WISE, SOA4All, etc. Most of them have the same disadvantages. Another projects (e.g., METEOR-S, BCDF, SCENE) requires custom runtime environment or custom service executor [9]. We suppose such environment to be superfluous.

The most fitting project we found are SWORD and ASTRO. The first one determines service interface with lightware domain-specific language that is more simple than XML from previous examples. Project's compiler automatically verifies new service scheme to be compatible with current running services. This guarantees that a new update of a service would not break the whole system or that new service would work with another.

All described projects seem to be too complicated or too limited according to our requirements. Firstly, no one supports nonsynchronous communication, for example, event-driven architectures. Secondly, most of them suit for a description of business-process, not for a description of the behaviour of small services. Thirdly, we consider underlying XML format too verbose and not enough compact for our purposes.

2.2. Linking of Microservices

One of the modern popular ways to link microservices is the service discovery one. The matter of the way is finding dependent service by the name in the central services registry. The central registry provides registering of instances, checks the state of already registered ones, and provides access to information about services addresses.

However, service discovery does not check the compatibility of acquired and acquiring services.

2.3. Interface Consistency

Developer can define an interface of synchronous microservice with OpenAPI for REST [4], WSDL for SOAP [3], Protobuf for GRPC [2] or Apache Thrift. Data validation is usually performed with XML-Schema or JSON-Schema.

Except Thrift or Protobuf, Apache Avro [1] is another project attempting not only to describe a way of data encoding and validation but also attempting to control interfaces compatibility and versioning.

All mentioned, Thrift, Protobuf, Avro, WSDL, and OpenAPI are created to support synchronous RPC or REST.

3. Type System

Data can be encoded with custom binary or text format while interoperation: with XML or JSON. Encoded data satisfies restrictions of communication protocol: SOAP, XML-RPC (XML); REST, JSON-RPC (JSON); Protobuf, Thrift (binary) and so on. APIs based on the communication protocols can be described with formal specifications: WSDL for SOAP, OpenAPI for REST, etc. Event-driven SOA often uses the message broker system like Apache Kafka, RabbitMQ or NATS which transfers text-encoded messages. Among the mentioned ways of data representation, JSON is the most popular format for service communication. Validating received data becomes a simple task with JSON Schema validation or Apache Avro validation.

We suppose JSON Schema to be more actual than Avro. However, it is only the validation standard without any subtyping or polymorphism support, so we make our own subtype checking algorithm.

All mentioned above protocols are limited by using simple (integer, boolean, etc.) or complex (arrays and records) types [4] [3]. For example, simplified JSON Schema type system [14] can be expressed as presented at the figure 1.

Described grammar is simplified because it does not cover complex predicates containing boolean logic. Also, the grammar does not cover specific type formats.

According to standardization and popularity, we took JSON Schema as a base for our type system. To improve the compatibility of services, we suppose the described type system to be structural one [10]. This statement enables us to ensure that B is a subtype of A in $A <: B$ if for every parameter from A there is an equal parameter from B (1). We assert that types predicates are equal if their names and parameters conform. An induction rule is used to specify the subtype with predicates relation (2).

$$\begin{aligned} &\Gamma \vdash A \\ &\Gamma \vdash B \\ &\Gamma \vdash A <: B \end{aligned} \tag{1}$$

$$\frac{\Gamma \vdash AP_1 \quad \Gamma \vdash BP_2 \quad \Gamma \vdash P_1 = P_2}{\Gamma \vdash AP_1 <: BP_2} \tag{2}$$

Finally, we did not change JSON Schema syntax for compatibility with existing software purposes.

$\langle t \rangle$ ::= $\langle arr \rangle$ | $\langle obj \rangle$ | $\langle num \rangle$ | $\langle str \rangle$ | **boolean** | $\langle p \rangle$ $\langle t \rangle$

$\langle arr \rangle$ ::= {$\langle t \rangle$} | $\langle ap \rangle$ $\langle arr \rangle$

$\langle ap \rangle$ ::= **additionalItems** | **maxItems** | **minItems** | **uniqueItems** | **contains**

$\langle obj \rangle$ ::= $\langle t \rangle$ $\langle t \rangle$ | $\langle op \rangle$ $\langle obj \rangle$

$\langle op \rangle$ ::= **maxProperties** | **minProperties** | **required** | **properties** | **patternProperties** | **additionalProperties** | **dependencies** | **propertyNames**

$\langle num \rangle$ ::= **integer** | **real** | $\langle np \rangle$ $\langle int \rangle$

$\langle np \rangle$ ::= **multipleOf** | **maximum** | **minimum**

$\langle str \rangle$ ::= **string** | $\langle sp \rangle$ $\langle str \rangle$

$\langle sp \rangle$ ::= **maxLength** | **minLength** | **pattern**

$\langle p \rangle$ ::= **const** | **enum**

Fig. 1. Simplified grammar of JSON Schema

Algorithm 1 Type checking

Require: $type1, type2$
  $subtype \leftarrow true$;
  if $type1$ is scalar **then**
    $subtype \leftarrow type1! = type2 || type1.p! = type2.p$;
  else {$type1$ is object}
    for all $type1.f$ **do**
      $subtype \leftarrow subtype \&\& self(type1.f, type2[type1.f])$;
    end for
  end if

3.1. Algorithm of Subtype Checking

Our type checking algorithm 1 verifies that every field from the type A is equal to the same one from the type B. Record $type.p$ returns all predicates from the type $type$. Code $type2[field]$ takes from $type2$ subfield with the name $field$ and code $type2.f$ takes all fields from $type2$. The algorithm does not try to analyse predicates, it just checks identity of the name and the parameter. Types of the JSON Schema object are checking recursively. List of subtype required fields must be equal to the parent type one.

3.2. Example of Subtyping

Define 2 types: A at listing 1 and B at listing 2.

Listing 1. Type A

```
{
  "title": "Person",
  "type": "object",
  "properties": {
    "firstName": {
      "type": "string"
    }
  },
  "required": [
    "firstName"
  ]
}
```

Listing 2. Type B

```
{
  "title": "Person",
  "type": "object",
  "properties": {
    "firstName": {
      "type": "string"
    },
    "secondName": {
      "type": "string"
    }
  },
  "required": [
    "firstName",
    "secondName"
  ]
}
```

Here type A requires document to contain field string $firstName$ and therefore any document containing $firstName$ is suitable for this schema. Type B also requires this string field to be in a descibing document. Thus we can assert that B is the subtype of A: $A <: B$.

Though $B <: A$ can not be true because B applies one more required restriction for a document: string field $secondName$.

4. Description of Contract Concept

We introduce the concept of a contract to describe communication between services. A service contract is an analog of communication specification which describes one remote call or one session of information transfer. List of contracts forms regular communication protocol (like OpenAPI or WSDL) if every item of the list is provided with the same service or the same endpoint.

Interoperation of services divides into two categories: a synchronous and nonsynchronous one. The synchronous communication (RPC, REST) requires a protocol to define the way of call, the way of response and optionally an error definition. Custom protocols can specify complex sequences of data units passing to an inter-service channel. The nonsynchronous one (event-driven design) requires a protocol to define the only type of transmitting data.

In order to level differences between the methods, we define a contract as a sequence of message types. Thus, HTTP call would be a chain of two messages while an event would be a chain consisting of one element. A contract also contains:

- an endpoint of service which provides contract realization (provider)
- an address to check the contract provider urgency

- a direction of every chain unit - is the message incoming or outgoing for provider

In opposite to provider of contract, the user one claims that a service needs any provider to work correctly. User contract has the same format as the provider one but does not specify endpoint. User contracts ensure that the system's services have all dependencies and work correctly.

The user contract is compatible with provider contract if the chains of the first one occur to be subtypes of the second one. It means that if $A <: B$ is the valid judgment, then the provider takes type A at the input when a client can call it with type B. The provider service must be ready for input data with type B and must process it like data with type A.

4.1. Description of Conceptual Contract Discovery Service

Services must declare their requirements themselves because they contain all related API information. There are two targets for pushing declarations:

- all other services (e.g. broadcast notification)
- central service delegated to manage contracts

Notification of all other services requires broadcast messaging and storing information about the whole system in each one. Moreover, broadcast notification would require implementation of type and contract checking in every service. Therefore, central control is preferable.

Services which collect information about system components, provide their addresses and watch for their state are called "service discovery". Since our tool manages contract providers we call it "contract discovery" service. Prospective realization should have following features:

1) register contract provider
2) register contract user
3) watch for providers and users to be alive
4) deliver on demand information about contract providers for contract users
5) verify that all dependencies are resolved and show dependency problems
6) warn after disabling all providers of the contract that is still used

Providers send information to the service at their startup moment or at their deploy moment. Users get their dependencies also at the start by registering their dependencies or by separate call.

5. Realization of Contract Discovery and Testing

We implemented the first version of the contract discovery service as a proof-of-concept PHP daemon built on top of ReactPHP [6]. The daemon was used within a test suite containing stub services. After having proved the idea, we made the second realization with Golang. Service implements all requirements and all described functions. Daemon registers contract providers and users perform regular alive checks and type checking.

We used described service for managing dependencies and for type checking in the existing event-driven system. Services in this system register their contracts at their start. They also gain their own requirements via contract discovery. While services use message broker and do not expect any result of the call, all registered contracts consist of no more than one schema. Users obtain routing keys for dispatching messages from matched provider contracts.

Though the proposed approach does not suppose an improvement of some specific algorithm or data passing technique, we cannot present any numeric metrics. However, after registering automatization had been made, we noticed that the process of adding new services to the system became easier. Advances that we found are:

- inter-service integration became easier as the result of inter-service strong typing - service will not start while dependencies are not resolved
- contract-first development makes positive influence on service building speed
- developers do not need to keep track of the service dependencies in configuration

We also noticed several complicities:

- maintenance of all types consistency is complicated - there is no one place to store all actual contracts. Contract discovery stores only registered at present time items.
- lack of information about actual data routes
- all system depends on the central component
- since contract discovery checks only online services it does not provide real static type system
- contract discovery guarantees consistency of contract types but not identity of contracts and real interfaces

6. Conclusion

As the result we have replaced direct static services linking with detection of the most suitable contract provider. This kind of interaction allows us to ensure that enabled service would work correctly and have all required dependencies. Also, the usage of strong polymorphic typing enables us to ensure that APIs of interacting services are compatible. Contract discovery service ensures that a system does not have any dependency problems at the moment and helps to trace usage of an outdated interfaces.

From the other side, the presented approach sophisticates control over current interaction of a system components.

It also does not provide real static type checking for communication of an elements: contract discovery does not guarantee the identity of contracts and real interfaces and does not guarantee service compatibility before new service is enabled.

Finally, we did not gain the main goal: stability of the system has not increased significantly.

Therefore, we have new ideas on how to provide strong control over the services interaction. We suggest specifying all data types in a single file or project with a description of the whole services communication design. Moreover, such definition is expected to resemble a source code on any functional programming language and can also introduce instructions for deploying services. We expect that such source code will be assembled into container configuration files and the translator would perform static type checking. The concept that we are developing now recalls behavioural and session types [8]. Replacing dynamic contract discovery with service definition compiler would save listed advantages and decrease described disadvantages.

AUTHOR

Nikita Gerasimov – Mathematics and Mechanics Faculty, Saint Petersburg University, Universitetsky prospekt, 28, Peterhof, St. Petersburg, Russia, e-mail: n.gerasimov@2015.spbu.ru.

REFERENCES

[1] Apache Software Foundation, "Apache Avro™ 1.9.0 Documentation", http://avro.apache.org/docs/current/, Accessed on: 2018-11-10.

[2] Google, "Developer Guide | Protocol Buffers", https://developers.google.com/protocol-buffers/docs/overview, Accessed on: 2018-11-10.

[3] W3C, "Web Services Description Language (WSDL) Version 2.0 Part 1: Core Language", https://www.w3.org/TR/wsdl, Accessed on: 2018-04-26.

[4] OpenAPI Initiative, "The OpenAPI Specification", https://github.com/OAI/OpenAPI-Specification/blob/master/versions/3.0.1.md, Accessed on: 2018-04-26.

[5] N. Dragoni, S. Giallorenzo, A. L. Lafuente, M. Mazzara, F. Montesi, R. Mustafin, and L. Safina. "Microservices: Yesterday, Today, and Tomorrow". In: M. Mazzara and B. Meyer, eds., *Present and Ulterior Software Engineering*, 195–216. Springer International Publishing, Cham, 2017.

[6] N. Gerasimov. "Contract checker", http://github.com/tariel-x/cc, Accessed on: 2018-05-07.

[7] N. Gerasimov, "Static typing and dependency management for SOA". In: *Annals of Computer Science and Information Systems*, vol. 16, 2018, 105–107.

[8] K. Honda, V. T. Vasconcelos, and M. Kubo. "Language primitives and type discipline for structured communication-based programming". In: G. Goos, J. Hartmanis, J. van Leeuwen, and C. Hankin, eds., *Programming Languages and Systems*, volume 1381, 122–138. Springer Berlin Heidelberg, Berlin, Heidelberg, 1998.

[9] A. L. Lemos, F. Daniel, and B. Benatallah, "Web Service Composition: A Survey of Techniques and Tools", *ACM Comput. Surv.*, vol. 48, no. 3, 2015, 33:1–33:41, DOI: 10.1145/2831270.

[10] B. C. Pierce, *Types and Programming Languages*, The MIT Press: Cambridge, 2002.

[11] R. Rodger, *The Tao of Microservices*, Manning Publications: Shelter Island, New York, 2017.

[12] Q. Z. Sheng, X. Qiao, A. V. Vasilakos, C. Szabo, S. Bourne, and X. Xu, "Web services composition: A decade's overview", *Information Sciences*, vol. 280, 2014, 218–238, DOI: 10.1016/j.ins.2014.04.054.

[13] L. Sun, H. Dong, F. K. Hussain, O. K. Hussain, and E. Chang, "Cloud service selection: State-of-the-art and future research directions", *Journal of Network and Computer Applications*, vol. 45, 2014, 134–150, DOI: 10.1016/j.jnca.2014.07.019.

[14] A. Wright, H. Andrews, G. Luff, "JSON Schema Validation: A Vocabulary for Structural Validation of JSON", http://json-schema.org/latest/json-schema-validation.html, Accessed on: 2018-04-26.