

Static typing and dependency management for SOA

Nikita Gerasimov

Saint Petersburg University

Mathematics and Mechanics Faculty

Universitetsky prospekt, 28, Peterhof, St. Petersburg, Russia

Email: n.gerasimov@2015.spbu.ru

Abstract—Several problems related to work reliability appear while building service-oriented systems. The first problem consists in lack of static typing and lack of inter-service data type checking. The second one consists in high services connectivity. The article shows an example of strong and static polymorphic type system and a type check algorithm. Type system syntax and service-contract concept are described. Theoretic results were realized in a service form and were applied in practice in a real system, which improved its reliability. Also, technical realization decreased services connectivity which promoted system quality increase.

I. INTRODUCTION

A DEVELOPMENT of convenient multi-logic systems bases often on service or microservice oriented architectures (SOA). SOA means that application logic is divided into several self-sufficient components, providing separate tasks realization [8]. Every component has the single responsibility. The advantages of SOA are simplified maintenance, independence of single technology or programming language. Every logic change requires modifying only in-component realization and implementation of the current external interface.

However, SOA has also disadvantages, an inequality of providing and using interfaces and type checking in the whole system [5]. Various frameworks and approaches suggest the ways of system decomposition but don't suggest any ways of types consistency statically checking. Building analogy from dynamic-typing language this fact leads to in-production system instability increase.

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 allow 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.

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 Detection of outdated API usage can be nontrivial in complex systems with various components.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 possibility of important component disabling. The real case is that an outdated service A provides statistics collecting once per month by some mailing service. Logs analysis proves that there were no API calls during last 3 weeks that's why the service can be disabled.

Finally, we have 2 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 main goal of this research is to increase 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 statically check types
- develop service that should control API dependencies in the SOA system

Much of the work presented here is connected with 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 [4]. 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 defines the proposed type system and type checking algorithm to be realized in contract discovery service. Section 3 surveys the concept of contract and how contract-discovery service provides client to service linking. Section 4 describes our contract discovery service realization details. Section 5 illustrates our experience of application such service to the real microservice-oriented event-driven system.

II. TYPE SYSTEM

Data can be encoded with custom binary or text format: with XML or JSON while interoperation. 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 usually uses JSON as a data format and JSON Schema standard for validating and describing data structures.

All mentioned protocols are limited by using simple (integer, boolean, etc.) or complex (arrays and records) types [7][9]. For example, simplified JSON Schema type system [1] 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 compatibility of services we suppose the described type system to be

```

⟨t⟩ ::= ⟨arr⟩ | ⟨obj⟩ | ⟨num⟩ | ⟨str⟩ | boolean | ⟨p⟩ ⟨t⟩
⟨arr⟩ ::= {⟨t⟩} | ⟨ap⟩ ⟨arr⟩
⟨ap⟩ ::= additionalItems | maxItems | minItems
        | uniqueItems | contains
⟨obj⟩ ::= ⟨t⟩ ⟨t⟩ | ⟨op⟩ ⟨obj⟩
⟨op⟩ ::= maxProperties | minProperties | required
        | properties | patternProperties
        | additionalProperties | dependencies
        | propertyNames
⟨num⟩ ::= integer | real | ⟨np⟩ ⟨int⟩
⟨np⟩ ::= multipleOf | maximum | minimum
⟨str⟩ ::= string | ⟨sp⟩ ⟨str⟩
⟨sp⟩ ::= maxLength | minLength | pattern
⟨p⟩ ::= const | enum

```

Fig. 1. Simplified grammar of JSON Schema

structural one [2]. This statement allows us to assert that B is a subtype of A in $A <: B$ if for every future from A can be found equal one 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).

$$\begin{array}{l}
\Gamma \vdash A \\
\Gamma \vdash B \\
\Gamma \vdash A <: B
\end{array} \quad (1)$$

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

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

A. 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.

III. DESCRIPTION OF CONTRACT CONCEPT

We introduce the concept of a contract to describe communication between services. Service contract is an analogue of communication specification which describes one remote call or one session of information transfer. List of contracts forms

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

```

regular communication protocol (like OpenAPI or WSDL) if every item of the list is provided by 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 inter-service channel. The nonsynchronous one (event-driven design) requires a protocol to define only type of transmitting data.

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

User contract is compatible to provider contract if the chains of the first one occur to be subtypes of the second one. This means that if $A <: B$ is valid judgement than provider takes type A at input when 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 .

A. 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 must 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.

IV. REALIZATION 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 proving the idea we made the second realization with Golang. Service implements all requirements and all described functions. Daemon registers contract providers and users, performs regular alive checks and type checking.

We used described service for managing dependencies and for type checking in 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 improvement of some specific algorithm or data passing technique we have no ability to 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 would 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 complications:

- 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

V. CONCLUSION AND THE FUTURE WORK

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 usage of strong polymorphic typing allows 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.

From the other side presented approach sophisticates control over the current interaction of the system components. It also does not provide real static type checking for the communication of the elements.

Finally we did not gain the main goal: we did not strongly increase stability of the system.

Therefore we have new ideas on how to provide strong control over the services interaction. We suggest to specify all data types in a single file or project with a description of 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 would 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 [3]. Replacing dynamic contract discovery with service definition compiler would save listed advantages and decrease described disadvantages.

REFERENCES

- [1] 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>
- [2] B. Pierce “Types and Programming Languages”, London: MIT Press, 2002, pp. 251-254.
- [3] H. Kohei, V. T. Vasconcelos, M. Kubo “Language primitives and type discipline for structured communication-based programming” *Programming Languages and Systems, Lecture Notes in Computer Science*, vol 1381, Springer, Berlin, 1998
- [4] L. Sun, H. Dong, F. Hussain, O. Hussain, E. Chang “Cloud service selection: State-of-the-art and future research directions” *Journal of Network and Computer Applications*, Austria, October 2014.
- [5] N. Dragoni et al “Microservices: Yesterday, Today, and Tomorrow” *Present and Ulterior Software Engineering*, pp. 195-216, Springer, Cham, 2017
- [6] Contract checker, <http://github.com/tariel-x/cc>
- [7] OpenAPI Specification, <https://github.com/OAI/OpenAPI-Specification/blob/master/versions/3.0.1.md>
- [8] R. Rodger “The tao of microservices”, Manning publications, 2017, unpublished, pp. 17-19.
- [9] Web Services Description Language (WSDL), <https://www.w3.org/TR/wsdl>