A Fault Tree based Microservice Reliability Evaluation Model

A Fault Tree based Microservice Reliability Evaluation Model

Zhigang Zang[1*], Qiaoyan Wen[2], Kangming Xu[2]

[1] Zhongshiruian (Beijing) network Technology Co., Ltd., Beijing 100876, China

[2] Network Security Institute, Beijing University of Posts and Telecommunications, Beijing 100876, China

Email of all the authors: *blue_oceans@163.com*, *wqy@bupt.edu.cn*, kangmingxu@126.com

* Corresponding Author: Zhigang Zang; email: blue_oceeans@163.com*;*
phone: 13811521772.

Abstract. The system based on microservice architecture is a trend of software system development in the future. As there are many independent micro services in the system based on micro services, the reliability evaluation of these micro services and the impact of their reliability on the system reliability have become a problem worth studying. Based on the reliability evaluation of the system based on the service dependency graph and the fault tree, this paper proposes a scheme of automatically generating the service dependency graph with the help of the service registry, and improving the reliability model of service fault tree. When establishing the fault tree, the influence of the system's fault-tolerant mechanism and different probability of execution path on the system reliability is taken into account, thus improving the accuracy of the model analysis system fault rate.

1. Introduction

The design philosophy of the microservice architecture is based on the concept of an application-level interactive workflow of SOA [4]. It allows developers to freely choose the development framework, configuration environment and configuration methods during development, deployment, and debugging [1]; and dynamically adjust the number of corresponding microservice instances based on service load [2]. The systems based microservice are often a resilient system, and when one service fails, the system may lose some of its functionality and the rest will still function properly. Since a microservice tends to take on a small function, it doesn't have much code, which greatly simplifies developer development and debugging. Today, Microservice architectures are being used by a growing number of companies, such as Netflix[3], eBay, Twitter, Amazon.

Because there are many independently deployed microservices in the system, each microservice can fail. How do these failures manifest and affect other microservices in the system? How to spread and influence the final user experience in the system? [15].

One existing solution is to establish a service dependency graph and construct a fault tree according to the service dependency graph [5] to analyze the impact of the microservice fault on the system. The service dependency graph reflects the dependencies between the microservices. The fault tree is a logical diagram showing the relationship between the key events in the system and the cause of the event [6], and the fault tree analysis is the process of constructing the fault tree.

This paper proposes a service dependency graph automatic generation scheme and fault tree model. In this paper, the service registry is used to obtain the dependencies between microservices. The fault tree model is designed for the system's error tolerance scheme, and the model is quantitatively analyzed according to the execution probability of each execution path. The model has improved the accuracy of the task execution simulation in the system, and can effectively analyze how the fault propagates in the microservice architecture-based system and how it affects the entire system.

2. Related work

2.1. Service dependency graph generation scheme

A system based microservice contains many independently deployed micro services [7], which is certainly good if the various components in the system can run successfully without any other components. But in most practical situations, But in most practical situations, it is an island if there is no other microservice. In a real user usage scenario, after a service is started, it can respond to API calls on its own, but when implementing a specific function, it often requires coordination and cooperation of several services.

The service dependency graph is essentially a directed graph, originally used to reflect the data flow and control flow between various system modules. Each vertex in the graph represents a microservice in the system; the directed edge points from the service consumer to the service provider, indicating the dependencies between services. The microservices architecture is an abstract software architecture. Each system can be defined by a number of abstract operations at many different levels [14]. The specific granularity for the system description can be determined by the participants. Generally speaking, in a directed graph, each vertex represents a microservice in the system. The content of the dependency graph in Figure 1 is:

Service A depends on Service B and Service C;

Service C depends on Service D and Service E.

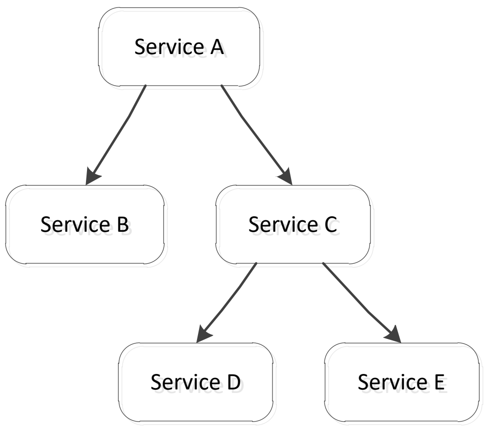

Figure 1. Example of a service dependency graph

2.2. Fault tree model construction algorithm

In his research [5], Johan Uhle refers to several elements in the fault tree element system of the Fuzzed Editor [8], and constructs a fault tree model based on the service dependency graph. The fault tree elements used in the model include: TOP event, AND-gate, OR-gate, Basic event, Intermediate event, etc. Each event represents a node in the fault tree and is connected through a logic gate. The TOP event indicates the system failure to be investigated, and the event that may cause the TOP event to occur is defined as a new event. These new events are connected to the TOP event through the AND-gate and

OR-gate. In the actual environment, the new event can be performed as shown above. Recursive operation, thus analyzing the failure of any granularity of the system. The leaf nodes in the fault tree are called Basic events; the nodes that are neither TOP events nor Basic events are called intermediate events, the Intermediate events.

The specific steps to build a fault tree model based on the service dependency graph are as follows:

1. Convert the root vertex to a TOP event;
2. Create an OR-gate and connect it to the TOP event;
3. Create a basic event called root vertex and connect it to the OR-gate;
4. Follow each output edge of the current vertex (starting from the root vertex) to the next vertex:

a) Create an intermediate event with the next vertex name and connect it to the OR-gate of the current vertex;

b) Create an OR-gate and connect it to an intermediate event;

c) Create a basic event with the next vertex name and connect it to the OR-gate;

d) Recursively repeat step 4 for the next vertex.

2.3. Introduction to the Eureka Framework

Eureka is a REST-based open source framework developed by Netflix [9]. The Eureka framework usually consists of the Eureka server and the Eureka client. The Eureka server provides service registration service to the outside. After each service in the system is started, it will be registered with the Eureka server [10]. In this way, you can see information about all the services in the system on the Eureka server. Each service sends a registration message to the service registry at the time of startup, and sends a heartbeat packet to the service registry during the running of each service. When the service is stopped, the service will send the unregistered information to the service registry. When the service exits abnormally, the service registration information will also be invalid because the service registry cannot receive the heartbeat packet sent by the service.

For Java applications, the Eureka framework provides a way to register Java client services with the Restful API-based service registration method. The Java client registration method can be implemented only by adding annotations to the application code [11]. For the service registration method of the Restful API, each service needs to use the REST operation mode for service registration and service discovery. By periodically sending messages to the Eureka service registration center for service registration, the registration related configuration information of each service can be written in each service. The configuration file includes, for example, the Eureka server address, service name, and so on. After the registration of each service in the system is completed, the service registration center can view the related information of each service in the system. The Eureka framework is shown in Figure 2 [12]. By default, the Eureka framework uses JSON as the communication protocol between Server and Client. You can also choose to use your own implementation of the protocol instead, with very good scalability.

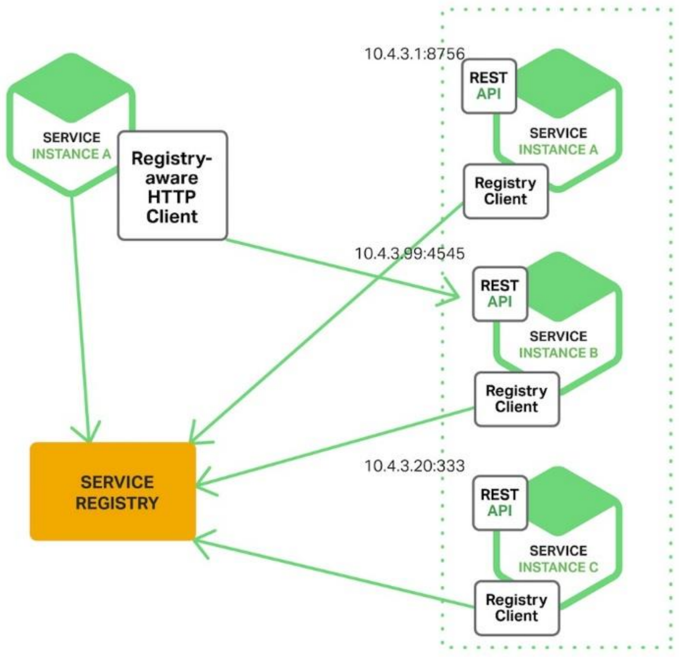

Figure 2. Eureka framework architecture

The service registration and discovery methods based on the Eureka framework have good stability. The system is highly available, flexible, and scalable through heartbeat detection, health check, and client-side caching. In addition, the Eureka framework provides users with a monitoring interface that allows users to visually see all registered service instances in the system. It is also very convenient to cluster management of services in the system.

2.4. *Graphviz open source toolkit*

The input of Graphviz[13] open source toolkit is a description script written in DOT language. Through the analysis of the description script, the points, edges and various custom graphics are analyzed, and then drew according to the properties of each module. The author only needs to consider the relationship between the nodes, and does not need to pay too much attention to the layout of the graphics and the relative position of each node. Especially for more complex graphics, the automatic generation of the Graphviz toolkit can save the user a lot of unnecessary trouble. This article uses the Graphviz toolkit to complete the drawing of service dependency graphs and quantitative service fault trees.

3. Microservice reliability assessment model

If Service A depends on Service B, then the failure of Service B will also cause Service A to fail. However, there is often a error tolerance mechanism in the system, so that even if service A depends on service B, service B fails, and service A does not necessarily fail due to the existence of error tolerance. In this paper, an evaluation model is designed based on the impact of error tolerance scheme on microservice reliability.

3.1. *Automated generation of service dependency graphs*

This article automatically generates a service dependency graph based on the system service registry and the configuration files of each microservice. The service registration center used in this paper adopts the implementation strategy of client discovery, that is, the service registration module is deployed in each service. After each micro service is started, the registration message is sent to the service registration center to complete the service registration. The client directly queries the service registry for the location of the service it wants to retrieve, and the service registry returns the location of the service. In the Eureka framework used in this article, if the microservice instance terminates or the service registry does not receive a periodic heartbeat signal from the microservice instance, it will trigger the service registry to delete the record corresponding to the instance in the registry, thereby enabling the service registry service. The list can reflect the actual situation of the system in real time.

This article adds a custom parameter to the service registration message to describe the dependencies of the current service and the error tolerance scheme designed for the service. This article divides the commonly used error tolerance schemes in the system into the following two cases:

1. Service A depends on Service B, but the success or failure of Service B does not affect the execution of Service A; it is marked with ET-no.

2. Service A depends on Service B and Service C. Service B and Service C can execute normally as long as they have a normal execution; they are marked with ET-or.

The dependency graph generation module can obtain dependencies between microservices from the service registry. The specific design architecture is shown in Figure 3. The process of automating the generation of service dependency graphs is shown in Algorithm 1.

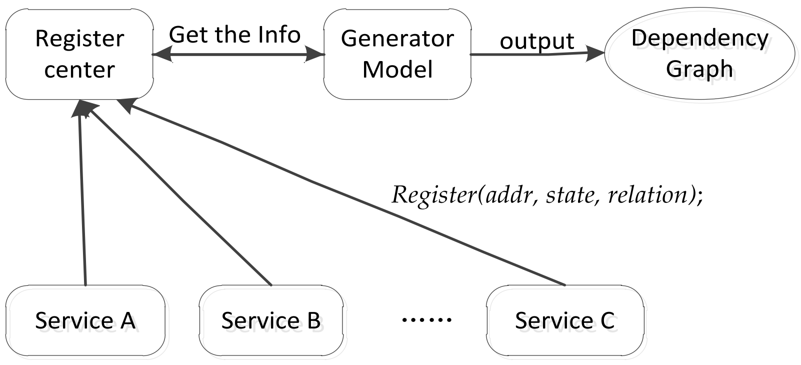

Figure 3. System architecture design

Algorithm 1 Auto-generation of service dependency graph

1. Microservices start and send register messages to the Register center: *Register(addr, state, relation)*; *addr* is the address where the service deployed, *state* is the service's running state; *relation* is the dependent relationship of the service.
2. Obtain the microservices' dependency relationship. The dependent graph generate module send request to the Register center for the number of microservices and *dependency* of each microservice. Generate the dot file according to the dependent relationships; add *start* tag to the start of the system and *end* tag to the end of the system, and if there exists error tolerance schema, add error-tolerance tag *ET-no* or *ET-or* according to the schema.
3. Generate the dependency graph. Use the dot file generated in 2 as input, call the java API of Graphviz to generate the dependency graph and output the graph to the specified directory.

3.2. Qualitative fault tree model

This section describes how to convert the service dependency graph generated in the previous section into a qualitative service fault tree. First, the dependency relations between microservices are obtained from the service dependency graph. Since the dependency graph is a directed graph, we create a fault graph according to Algorithm 2:

```
Algorithm 2 Fault tree analysis
1. Add TOP event to the top of the root node, create an OR gate and connect its
   output edge to the TOP event.
2. Create an event with name of the root node, connect this event to the above OR
   gate as an input.
3. Create an event with name of the current node (start with the root node), if the
   current node has outgoing edge(s), add a logic gate for the outgoing edge(s):
   (1) If there exists no error-tolerance tag:
       1) Create an AND gate,
       2) Connect the event with name of current node to this AND gate as its
   output,
       3) Create event(s) with name of the node(s) that the outgoing edge(s) of the
   current node points to and use the event(s) as input(s) of the AND gate.
   (2) If there exists error-tolerance tag(s):
       1) Create an OR gate,
       2) Connect the event with name of the current event to this OR gate as its
   output,
       3) Create events with name the nodes that the outgoing edges of the current
   node point to;
       In the case of ET-no tag, take these event(s) and additionally 1 as input(s) of
   the OR gate, as shown in Fig 4;
       In the case of ET-or tag, take these events with name of nodes that the current
   node points to as inputs of the OR gate, as shown in Fig 5.
      Repeat this step recursively for next node (namely the nodes that the outgoing
   edges of current node point to).
4. Mark the execution path existing in the system, mark the microservices in the
   same path with the same tag.
```

The main idea of Algorithm 2 is: starting from the root node, until all nodes in the dependency graph are converted into events, and connected through the AND-gate or OR-gate, the OR-gate between multiple services that are dependent on the same service. The selection method is that if multiple services that are dependent are located on different execution paths, different paths are often only executed once; therefore, when one service depends on multiple services, and multiple services belong to different execution paths, different For service input on the path, you need to choose to use the OR-gate to connect; otherwise, you choose to use the AND-gate to connect. At this point, the generation of a qualitative fault tree has been completed.

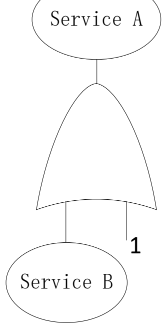

Figure 4

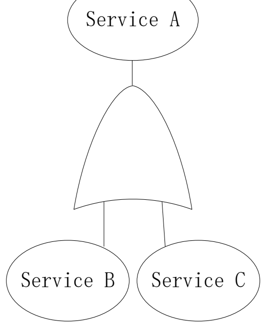

Figure 5

This article describes the fault tree using the DOT language. The DOT description file of the fault tree generated by the above steps is used as an input of the Graphviz open source toolkit to draw a graph of the system fault tree.

Here, taking the service dependency diagram shown in FIG. 1 as an example, the specific execution process of the step of going out is explained, and the process of constructing the fault tree is as shown in FIG. 6.

1. Add a TOP event to the output outside the root node (application A) and connect the TOP event to the root node with an OR-gate;
2. Add a logic gate for the outgoing edge of application A. Here, application A has two outgoing edges, that is, application A depends on two services, application B and application C, and has no error tolerance flag, so create an AND-gate here. Application A is used as the output of the AND-gate, and application B and application C are used as the inputs of the AND-gate;
3. For application B, since it has no edge, it does not need to continue the process.
4. For Service C, since it has two outbound edges, that is, application C depends on two services, application D and application E, and has no error tolerance markup, it is connected to the dependent two services through the AND-gate, here Create an AND-gate, use application C as the output of the AND-gate, and use application D and application E as input to the AND-gate.

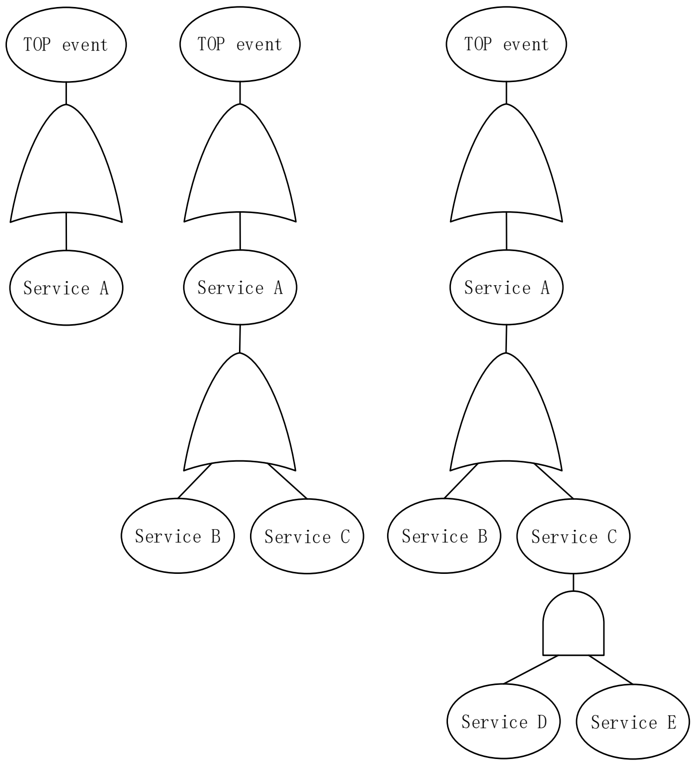

Figure 6 Process of building a fault tree

3.3. *Quantitative fault tree model*

Combining the historical data of the system operation, we estimate the probability of success of the basic events in the qualitative fault tree, and generate a quantitative fault tree for calculating the probability of success of the TOP event. The probability of success of the TOP event can be used to evaluate the probability of success of the entire system. It can also analyze how the service corresponding to the event affects the entire system by analyzing the impact of the probability of success of the basic event on the probability of success of the TOP event.

The model of the quantitative service fault tree in this section additionally considers the impact of different execution path probabilities on the final TOP event in the system. The quantitative calculation is as follows:

The probability of failure of each event is recorded as the probability of failure of the TOP event, which is recorded as $P(TOP\,Event)$. In order to facilitate the calculation, the probability of success of the output event $P(out)$ is calculated first, and the probability of failure is $P(Event) = 1 - P(out)$.

For the case of the connection with the AND-gate, the probability of success of the output is the product of the probability of success of each event input, as shown in Equation 1.

$$P(out) = P(in1) * \ldots * P(inN) \tag{1}$$

For the case of an OR-gate connection, the probability of success of the output satisfies Equation 2 and Equation 3, $P(in*)$ represents the probability of success of the underlying event of the input, $probability*$ indicates the probability that the service was executed in the system:

$$\begin{aligned} P(out) = P(in1) * \ pobability1 + \cdots + \\ P(inN) * \ probabilityN; (when\ P(inN)!= 1) \end{aligned} \tag{2}$$

$$P(out) = 1; (when\ there\ exists\ P(inN) = 1) \tag{3}$$

In a system based on microservice architecture design, there are often many execution paths. During the system operation, the probability of execution of different execution paths is also different. This paper obtains the execution probability of different execution paths by the following methods. Since a service is deployed and the assigned task is executed on the service, the execution status of the task on the service can be obtained by monitoring, and the historical data of the service execution task can be obtained through execution for a period of time. The historical data of each service operation in the integrated system can obtain the probability that each execution path in the system is executed. The probability that each path in the historical data is executed is added to the quantitative service fault tree. The specific steps in using the path execution probability for the quantitative service fault tree are:

Step 1: Through the historical running data of the system, the execution probability of each execution path in the system is obtained. For the basic events with multiple outgoing edges, the execution probability of the path is marked on each of the outgoing edges;

Step 2: If there is a case of the error tolerance scheme ET-no, the execution probability of the path is set to 1.

At this point, the generation of a quantitative service fault tree has been completed. The quantitative service fault tree can be used to analyze the impact of basic events in the system on system operation. By obtaining the success rate and failure rate of basic event execution, the success rate and failure rate of the TOP event of the quantitative service fault tree can be calculated.

4. Summary

This paper proposes a scheme for automatically generating a service dependency graph. The custom service registration request body saves the service dependency to the service registry. The service dependency graph automatic generation module obtains the service dependency from the service registry, and draws the service accordingly. The dependency graph can also obtain the service running status of the current system from the service registry and dynamically update the service dependency graph. Based on the above automatic generation service dependency graph scheme, this paper also improves the fault tree model. Based on the fault tree model, the impact of the error tolerance mechanism and the execution probability of different execution paths in the system on system reliability is further considered.

References

[1] Merkel D. Docker: lightweight linux containers for consistent development and deployment [J]. Linux Journal, 2014, 2014(239): 2.

[2] Gabbrielli M, Giallorenzo S, Guidi C, et al. Self-reconfiguring microservices[M] //Theory and Practice of Formal Methods. Springer International Publishing, 2016: 194-210
[3] Cockcroft A. Migrating to microservices [J]. QCon London, 2014: 16-18.
[4] MacKenzie C M, Laskey K, McCabe F, et al. Reference model for service oriented architecture 1.0[J]. OASIS standard, 2006: 18.
[5] Uhle J, Tröger P. On dependability modeling in a deployed microservices architecture [D]. Master’s thesis, University of Potsdam, Germany, 2014.
[6] Marvin Rausand and Arnljot Hø yland. System Reliability Theory: Models, Statistical Methods, and Applications. 2nd Edition. Wiley-Interscience, 2003, p.664.
[7] Viennot N, Bell J, Geambasu R, et al. Synapse:a microservices architecture for heterogeneous-database web applications[C]// 2015:1-16
[8] FuzzEd (Accessed on 28/05/2014). URL: http://fuzzed.org.
[9] Hunter II T. Service Discovery[M]//Advanced Microservices. Apress, 2017: 73-87
[10] Stubbs J, Moreira W, Dooley R. Distributed Systems of Microservices Using Docker and Serfnode[C]// International Workshop on Science Gateways. IEEE, 2015:34-39.
[11] Sharma S. Mastering Microservices with Java[M]. Packt Publishing Ltd, 2016.
[12] Chris Richardson. Service Discovery in a Microservices Architecture[OL]. [2017-1-20]. http://www.codes51.com/article/detail_317884.html.
[13] Ellson J, Gansner E, Koutsofios L, et al. Graphviz—open source graph drawing tools[C]//International Symposium on Graph Drawing. Springer, Berlin, Heidelberg, 2001: 483-484.
[14] Fielding R T, Taylor R N. Architectural styles and the design of network-based software architectures[M]. Doctoral dissertation: University of California, Irvine, 2000.
[15] Li L, Liu J, Zhou Z, et al. Causal Inference Based Service Dependency Graph for Statistical Service Fault Localization [C]//Semantics, Knowledge and Grids (SKG), 2014 10th International Conference on. IEEE, 2014: 41-48.