N+K: High Availability Solution for Stateful Service

Hongyu Yan
QingDao Institute of Technology
Fuzhou South Road 236, JiaoZhou, QingDao City,
ShanDong province, PRC
116546556@qq.com

Anmin Wang
Nokia Qingdao R&D
Zhuzhou Road 159, QingDao City, ShanDong province,
PRC
3217974@qq.com

***Abstract* — "1 active + 1 standby" model is often used to provide high availability for stateful service handling. However, this model is not very efficiency and it is not cloud native as more and more services are migrating to cloud. This article introduces another "N+K" model to achieve same purpose, which is more cost efficient and cloud native. This article describes the principle and implementation of "N+K" model, and discusses its advantages and challenges comparing with "1+1" model.**

Keywords: high availability, N+K, cloud native

I. BACKGROUND

A stateful[1] service is fulfilled by handling a set of requests, and the handling of each request is based on the result of the previous requests. So, a stateful service handler needs to store the result to handle next request. To provide high availability for stateful services, "1+1" redundancy model is often used. A pair of service handlers are deployed and only one handler in the pair is active, the other one is stand-by and is prepared to take over service in case the active one failed. When active handler receives an initial request, it creates a new session, the session context data is needed to process consequent requests. The session context data is synchronized to stand-by side at certain points.

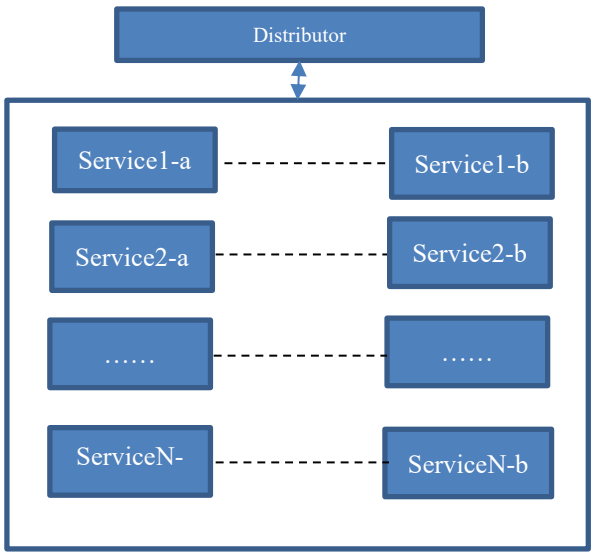

Figure 1-1 "1+1" model architecture

In above picture, we have:

1. Distributor: Component that interact with outside. It receives service requests from outside and distributes the requests to service handlers. Since this is the only interface to outside, it usually provides firewall functionality as well.
2. ServiceN-a/ServiceN-b: Service handler pair that handles requests from outside and provide service. Usually, ServiceN-a is active, and ServiceN-b is standby. When ServiceN-a is out of service somehow, ServiceN-b should become active automatically. The line between ServiceN-a ad ServiceN-b indicates there is data synchronization between active and standby handlers.

There are several problems with the "1+1" deployment model:

1. The number of service handlers to be deployed has to be doubled and stand-by handlers are not handling requests normally. So, this is a waste of resources normally.
2. Stand-by handlers are dedicated for specific active handlers only. If service 1 handlers duplex failed, even other stand-by handlers are ready, they cannot take service 1 handler's requests.
3. Not cloud native[2]. Active and stand-by handler need to be deployed in different physical machines to avoid duplex failure due to hardware faults. In cloud, it is not reasonable to have such kind of limitations.

To address the issues of "1+1" model, this article introduces a "N+K" model. We introduced the principle of "N+K" model and described our implementation in below sections.

II. "N+K" MODEL INTRODUCTION

At first, we'd like to give an overview of the "N+K" model and describe the functionality of the new components briefly. Then, we will discuss some typical "N+K" scenarios to illustrate how this solution works to provide high availability.

A. "N+K" overview

In "N+K" model, "N" is the minimum number of service handlers needed to be active to get the designed capacity. "K" is the number of extra service handlers

978-1-7281-8143-1/20/$31.00 ©2020 IEEE
DOI 10.1109/AEMCSE50948.2020.00018

deployed to provide high availability in case some handlers failed. In "N+K" model, all service handlers are active. If there is no failure, it can provide more capacity than designed. If one handler failed, the other handler should take the sessions owned by the failed handler to continue the service. As long as the number of failed handlers is not greater than "K", the designed capacity can be guaranteed.

B. "N+K" principle

In "N+K" model, all service handlers are providing service. When one service handler corrupted for some reason, any other service handlers can take over. As this is stateful service, session context data is needed to continue the service. In our implementation, all session context data is stored in a centralized database – Session DB. In this way, a service handler can get any session's context data when needed. Below picture illustrates the architecture of the "N+K" model:

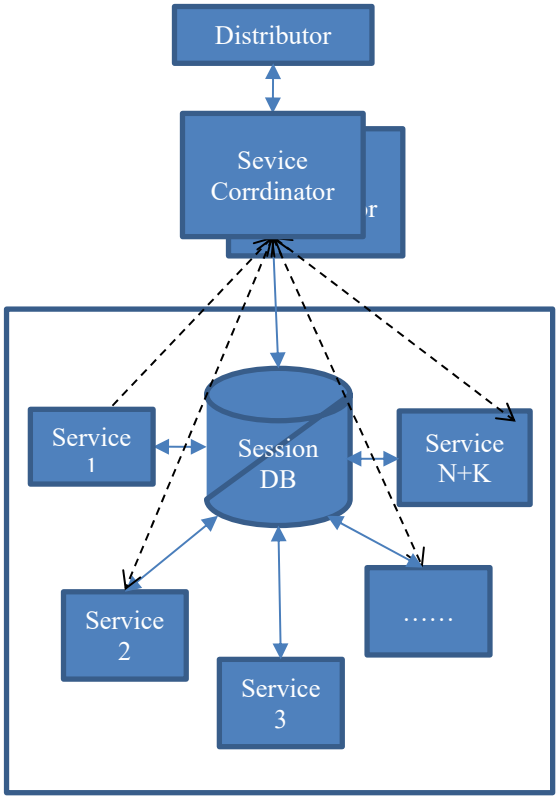

Figure 2-1 "N+K" model architecture

There are 2 new components are added comparing with the "1+1" model:

1 Session DB: This is a high-performance database which is used to store the session context data. After a service handler process a request, it will either create a new session or update the context of an existing session or delete a session. The session context data is stored in the service handler local memory. The service handler that owns this session should send the session context data to Session DB and make sure the data in Session DB is up-to-date. If another service handler needs to take over this session, it should retrieve the session context data from Session DB and keep it in local memory for further use.

2 Service Coordinator: Service Coordinator monitors all service handlers. Service Coordinator knows each service handler's status and all sessions each service handler is handling. When Service Coordinator detects a service handler out-of-service, it is responsible for informing other service handlers to take over the sessions owned by the failed handler. Service Coordinator can be an independent component or part of service handler. In the architecture, we made it an independent component to see it clearly.

C. "N+K" typical scenarios

We are trying to cover "N+K" typical scenarios here. More specific scenarios need to be considered to get this into production, especially considering the features of different services.

1) Sunny day scenario1: Service handler handle initial request

1. Distributor receives an initial request of the service.
2. Any service handler can handle an initial request. Distributor choose a service handler based on its own load balancing algorithm and sends the request to the service handler.
3. Service handler creates a new session and processes the initial request. Service handler writes the session context data to Session DB and informs Service Coordinator.
4. Service handler sends response back and waits for the next request.

2) Sunny day scenario 2: Service handler handling consequent requests

1. Distributor receives a consequent request of an existing session.
2. Distributor figures out the service handler that owns this session either consulting with Service Coordinator or reading Session DB and forward the request to the service handler.
3. The owning service handler finds the session context data in its local memory. Then service handler processes the requests. If any session context data changed, service handler shall update the Session DB. In some race conditions, service handler cannot find the session context data in its local memory, we will discuss this in race condition scenarios.
4. Service handler sends response back and waits for the next request.

3) Rainy day scenario: A service handler failed

1. A service handler became out-of-service somehow.
2. Service Coordinator detects the failure (the failed service handler may inform service coordinator if possible or service coordinator can detect that via some mechanism like heart-beating...)
3. Service Coordinator retrieves the list of sessions owner by the failed service handler.
4. Service Coordinator selects the active service handlers and sends them the list of session IDs to take over. Service Coordinator can select one or more active service handlers to take over the sessions based on implementations.
5. When the active service handler receives the takeover message from service coordinator, it retrieves the session context data from Session DB and save session context data in its local memory. The service handler is ready to handle the consequent requests of this session. In this way, the service availability is guaranteed even a service handler is out-of-service.
6. Active service handler response to service coordinator once completed.

4) Rainy day scenario 2: A service handler back to service

1. A service handler comes back to service after fixing the software or hardware problems.
2. The service handler informs service coordinator that it is in-service now. (It is also possible Service Coordinator detects that initiatively, depends on the implementation)
3. Service Coordinator mark that service handler ready.

5) Race scenario 1: Requests come before the handler failure detected

1. A service handler became out-of-service
2. Before service coordinator detects that, an incoming request comes.
3. Distributor sends the request to failed service handler. In this case, distributor won't get any acknowledge. After distributor tries several times, distributor will ask service coordinator again. Service coordinator should already detect the failure and will select another service handler.

6) Race scenario 2: Requests comes before takeover completed

1. A service handler became out-of-service.
2. Service coordinator detected that and informed other service handlers to take over the sessions owned by the failed one.
3. An incoming request comes while the takeover is in progress.
4. Distributor asks Service Coordinator for the new service handler. Then sends the request to the new service handler.
5. New service handler receives the request but didn't have session context data in local memory yet because the takeover is still in progress. In this case, we just let the service handler read Session DB to get the session context data and continue processing the request.
6. While the takeover continues, service handler may find the context data of a session to be taken over is already in its local memory. In this case, service handler just skips this session and continues to take over other sessions.

D. "N+K" challenges

As we can see from above, "N+K" model addressed the issues of "1+1" model. However, it brings new challenges as well. Below we describe the challenges and how we deal with them in our implementation.

1. There is DB write operation for every session context data update. When there is service handler failure, a lot of DB read operations happened in a very short time. All of this requires a high-performance DB application. Redis DB is chosen here because it works with an in-memory dataset and has outstanding performance [4]. There are different configurable options available to persist the data to achieve an outstanding performance [3][5].
2. Session DB is critical resource. We need to ensure the reliability of Session DB. Redis Sentinel and Redis Cluster is chosen to provide high availability in our implementation [1].
3. More complicated logic is required. More race conditions need to be considered.
4. It would take some effort to ensure the data consistency between the service handler local memory, Session DB and Service Coordinator. Probably need to have routine audit functionality to check the data consistency and do some auto-correction when possible.

III. COMPARISON OF "1+1" AND "N+K" MODEL

We compared the "1+1" model and "N+K" model in below aspects:

High Availability: We think "N+K" model provides higher availability. In "N+K" model, a service handler can take over the sessions owned by any other service handler. In "1+1" model, a service handler can only take over the sessions owned by service handler in the same pair. If both handler in a pair go out-of-service, the availability degraded even there is other standby handlers ready.

Capacity: "N+K" model provides higher capacity. In "N+K" model, all service handlers are providing services. If all service handlers are working fine, it provides more capacity than designed. In "1+1" model, even 2*N service handlers are deployed, only "N" of them are providing services.

Footprint: "N+K" model has smaller footprint. Normally "K" is less than "N". Need to deploy "N+K" service handlers to provide designed capacity. In "1+1"

model, “N+N” service handlers need to be deployed to provide designed capacity.

Independency: “1+1” model is more independent. “N+K” model highly depends on high-performance database application.

Cloud Native: “N+K” model is more cloud native. In “1+1” model, it is required to deploy the two service handlers in one pair in different physical machines to ensure the high availability. “N+K” model has no such limitation.

Implementation: In general, “N+K” model is more complicated. There are more race conditions to be considered. It is also a challenge to keep the data consistence between database and service handler. Similar scenarios occur in “1+1” model as well, but less complicated.

IV. SUMMARY

In summary, “N+K” model for stateful services provides higher capacity and has smaller footprint. And, it is more cloud native and should be considered while more and more services are moving to cloud. However, “N+K” model depends on high performance database application and brings more complexities. Investigation on “N+K” model is meaningful to reduce cost, especially with the background that more and more services are moving to cloud. More work is needed to come out some common practices to deal with the race conditions and to configure dependent database. Hope this helps.

REFERENCES

[1] Yan Chen, Xinyuan Tan, Wenjun Yang, Kai Chen, Guozhi Xu, “Stage based parallel programming model for high concurrency, stateful network services: internals and design principles”, International Journal of High Performance Computing and Networking 2005 – Vol. 3, No.1 pp. 33-44.

[2] Balalaie A, Heydarnoori A, Jamshidi P. Mocriservices architecture enables DevOps: migration to a cloud-native architecture[J]. IEEE Software, 2016, 33(3): pp. 42-52.

[3] Asay, Matt. NoSQL is still the cool kid in class[J]. ProQuest Journal, 2015, 30(1), pp.138-157.

[4] RedisLabs. Redis Cluster Specification https://redis.io/topics/cluster-spec/.

[5] Redis Performance Analysis. https://redis.io/topics/benchmarks