REST using NDN for efficient inter-service communication

Harshavardhan Kadiyala, Gu RainEr ,
Susmit Japhalekar , Ravneet

I. Introduction

Microservices [?] a new trend in architecting large software systems wherein a system is designed as a set of microservices. Microservices can be developed, managed and scaled independently. There is typically some kind of routing fabric [?] that gets requests to a specific instance of a microservices; this routing fabric often provides load-balancing and can isolate microservices that are in a failed state. There are costs to this approach as well such as the computational overhead of running an application in different processes and having to pay network communication costs [?] rather than simply making function calls within a process.

Using Web standards is recognized as a common approach in building microservices application architectures. REST mechanism is widely used for data-interchange [?] in microservices. It is a useful integration method because of its comparatively lower complexity over other protocols. Some of the basic properties of a REST protocol is statelessness and cache-ability. These properties will facilitate performance and reliability improvements in applications using REST.

For an efficient communication between microservices using REST, we need certain basic services like name-resolution of service endpoints, load balancing between different instances of the same service and caching of REST objects. Currently, these services are not provided by the network, But they are provided by external systems like DNS and middleboxes like HTTP cache. This middleware can introduce extra latency to API call under certain conditions such as cache miss event.

The Named Data Networking (NDN) [?] project aims to develop a new Internet architecture that can capitalize on strengths and address weaknesses of the Internets current host-based, point-to-point communication architecture in order to naturally accommodate emerging patterns of communication. By naming data instead of their locations, NDN transforms data into a first-class entity. It also enables several radically scalable communication mechanisms such as automatic caching to optimize bandwidth.

In this project, we will apply NDN features like named routing and automatic caching using in-network storage to create a REST implementation on NDN without middleboxes that can introduce latencies. We will evaluate our implementation by showing the reduced end-to-end latency of a REST API in our sample microservices application.

In section II, we will introduce to existing solutions and NDN architecture. In section IV, we will give problem statement and motivation for the problem. Section V will contain a description of the proposed solution. Finally, in VI, we will specify our evaluation strategy.

II. Related work

A. Existing solutions for service discovery

There are multiple solutions for service discovery in microservices. The simplest solution to registration and discovery is to just put all of the service endpoints belonging to a microservice behind a single DNS name [?]. To address a service, we can contact it by DNS name and the request should get to a random back-end hosting the microservice. Main drawbacks of this approach are DNS suffers from propagation delays; even after a server failure is detected a de-registration command issued to DNS, there will be at least a few seconds before this information gets to the consumers. Also, due to the various layers of caching in the DNS infrastructure, the exact propagation delay is often non-deterministic. Another major problem is that a service is identified just by name, there is no way to determine which boxes get traffic. We will get the equivalent of random routing, with loads chaotically piling up behind some back-ends while others are left idle.

In contrast to DNS based approach, our NDN based service discovery can load balance the flows [?] among different instances of the same microservice at the NDN router. It has faster recovery from service failures. As soon as NDN router identify service failures; it will invalidate the link. if a new request comes to the same service our NDN router will simply use alternate path for propagation.

Etcd [?] is a key-value store that provides shared configuration and service discovery for Container Linux clusters. etcd runs on each node trying to form a cluster using Raft [?] with the others for achieving high availability and fault-tolerance. It also supports service reconfiguration by watching updates to key prefixes with the service name. Another work is Consul [?] that provide consistent key-value store to deliver service discovery and integrated health checking based Consul agents. These agents communicate with Consul servers where data is stored and replicated, to maintain cluster state. Synapse [?] is a system for service discovery whose heart is a HAProxy, a stable and proven routing component. Synapse runs the HAProxy on application servers that are used to route requests from application servers to service providers running in the cluster. In order to provide service discovery, Synapse comes with several watchers, which frequently check for changes to the service location to update the HAProxy configuration.

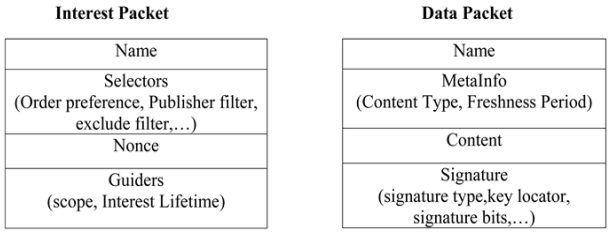

Fig. 1. NDN packets [**?**]

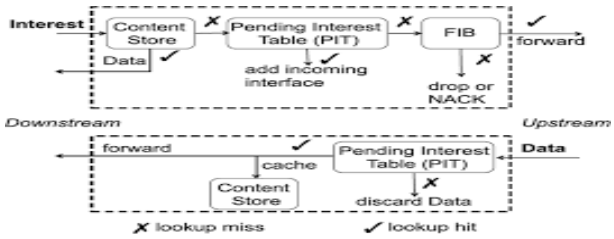

Fig. 2. Forwarding Process at an NDN Node [**?**]

Docker Swarm [**?**] is a native cluster system for Docker hosts. Docker Swarm decides where service images should be deployed to, thus each Swarm needs a key-value store which acts as a DNS server to store the location of each service. Each host has a Swarm agent running which is responsible for advertising its attributes to the key-value store.

Above-mentioned solutions use a consistent key-value store that is mainly achieved through cluster implementation. In practice, service discovery mechanisms that use the cluster to preserve the system reliability adds complexity to deployments. With the solutions that form a quorum to maintain clusters activities strongly depend on the number of servers. Without a sufficient number of cluster nodes, the server nodes cannot form a Raft-based quorum, thus the cluster cannot operate, and data loss can occur. Moreover, these solutions might occur high event advertising overhead for synchronizing the service state on every cluster nodes. Furthermore, there would be a significant increment of service records due to the number of available service instances. Therefore, it is very difficult to scale when applications have larger service popularity or applications grow larger.

III. Background

A. Named Routing

Communication in NDN [**?**] is driven by receivers i.e., data consumers, through the exchange of two types of packets: Interest and Data. Both types of packets carry a name that identies a piece of data that can be transmitted in one Data packet (see Fig.1).

To carry out the Interest and Data packet forwarding functions, each NDN forwarder [**?**] maintains three data structures: a Pending Interest Table (PIT), a Forwarding Information Base (FIB), and a Content Store (CS) (see Fig.2), as well as a Forwarding Strategy module that determines whether, when and where to forward each Interest packet. The PIT stores all the Interests that a router has forwarded but not satisfied yet. The Content Store is a temporary cache of Data packets the router has received. The forwarding information base (FIB) table use the names to route a request to next hop routers that are closer to the data provider by performing longest-prefix matching. The hierarchy also allows name resolution and data routing information to be gathered across similar data source names, which is the key point to enhance scalability and flexibility of the network architecture.

NLSR [**?**] is a routing protocol in NDN that populates NDNs Routing Information Base. The main design goal of NLSR is to provide a routing protocol to populate NDNs FIB. NLSR calculates the routing table using link-state or hyperbolic routing

IV. Problem Statement

REST protocol requires certain essential services from its transport layer in order to function efficiently. Some of the important services include name resolution of the endpoints [**?**] and caching of long-lived objects. Currently, such services are given by external systems and middle-boxes which are adding more complexity to network and additional network overhead to maintain. Though HTTP provides a feature for caching, it doesn't provide in-network caching. In order to use caching feature of HTTP, we need the request to go through an HTTP cache which is a middle-box that can introduce a delay upon a cache miss [**?**]. Every request to a new HTTP endpoint has to go through a DNS or a service discovery agent, this, in turn, adds more latency to the new API calls [**?**].

There is no way in the existing network to uniquely address an REST object and maintain a cached copy of it in the network without using additional complex middleware. In order to address this problem, we will use NDN as a transport protocol for REST objects. We will implement a python client that can convert REST commands into NDN packets and a content store for in-network storage of RESTful objects in the forwarder. We will deploy a sample microservices application in this new network to measure latency improvements that the new transport layer brings to the application using REST.

V. Proposed solution

In a typical microservices application, who is communicating is less important than the data being exchanged. To give priority to the data rather than to hosts, we will design an Information-centric network [**?**] between microservices using NDN network stack [**?**]. This new network design will facilitate two important services that can help microservices to exchange and cache REST objects efficiently without using any additional middleware. Two important goals of our new network.

A. Main features of new network

1) **Service discovery**
   a) Each REST service endpoint will have a unique global name in an application namespace.
   b) Every microservice inside an application namespace will have the capability to request objects of data from any other endpoint inside the same namespace.

c) The network will have a capability to propagate requested objects through the network to the requester.

2) **In-network cache**

a) We will provide a capability for in-network storage [**?**] of objects which can decrease latency and improve the reliability of the application.

To achieve above-said goals, we will describe the architecture of the new network model in next section.

B. Implementation

1) Overview of design : To transfer REST protocol objects efficiently on an NDN network, we need to develop a client library that can be attached to a microservice and helps in setting up connection, transferring of REST objects through NDN forwarder and closing of the NDN connection. We need a content store with REST specific caching policy so that we can take advantage of cacheable REST objects.

As part of our project, our main contributions will be

1) REST - NDN client library for python based microservice.
2) Design and implementation of a content store for REST objects.
3) Evaluate the performance of the REST protocol with NDN as transport.

2) REST - NDN client library: We will develop a REST-NDN client library in python which will give a capability of taking REST commands from application code through an API and sending NDN objects into the network. This library will also help clients to specify cache specific parameters like freshness which will be used by the content store. Each microservice will have a client library which will help in connecting to the NDN forwarding demon. Each physical node will have an NDN forwarding demon capable of routing NDN packets through the network (see figure 3).

3) Addressing schema: In our addressing scheme, each REST endpoint is addressable. Every REST endpoint will have a globally unique name.

Endpoint/instance_count/version

Naming structure of a endpoint

This addressing schema is simple enough but serves to achieve the required features.

1) By using only unique REST endpoint name and removing references to a microservice name, we can decouple endpoints from hosting microservices. This will help to decrease downtime and unnecessary changes when endpoints are moved between services.
2) Having an instance count can help identify and load balance among different instances of the same REST endpoint.
3) version helps in maintaining two versions of same endpoint available at the same time. This can facilitate rolling updates.

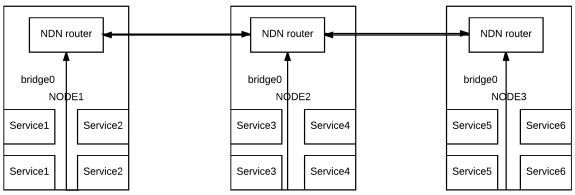

Fig. 3. Architecture of proposed network.

4) Connecting new microservice to the network: Newly created microservice using our REST-NDN client will broadcast in its own local container bridge [**?**] for local NDN forwarder IP address. Once it receives IP address it will use Named Data Link State Routing Protocol [**?**] to advertise all its objects to the local router. We will configure our NDN forwarder to only advertise longest common name for all other forwarders. In this way, we can have few updates while maintaining load balance between distributed endpoints.

5) Content store: We will design a caching system which will be attached to NDN forwarder. This caching system will have the capability to understand REST specific cache parameters set by REST-NDN clients. This cache will also improve the reliability microservices application by returning the most updated value of the REST object in case of network failure.

6) Named routing: There will be an NDN forwarding demon [**?**] [**?**] in every physical node (see Fig. 3). This forwarder will be connected to all other containers in the same node using a bridge [**?**]. We will replace default content store for REST specific content store. NDN uses NLSR - Named Data Link State Routing Protocol [**?**] to transfer new REST endpoint information among router overlay network.

VI. Evaluation

In this project, we will evaluate the new REST over NDN performance and compare it with existing REST over HTTP. We will show improvements in end-to-end latency and round-trip times of API calls as success criteria for our new stack.

A. Testbed

We will create a 3 node test bed. Each node is a VM with 2 VCPU s and 8 GB ram. We will deploy Cinema 3 [**?**], a microservice which uses python as the testing application.

B. Experiment A - REST-HTTP stack

In our first experiment, we will deploy cinema 3 [**?**] without any changes and we will measure end-to-end latency and round trip times for 5 of its APIs by load testing them.

C. Experiment B - REST-NDN stack

In this experiment, we will deploy cinema 3 [**?**] with new REST-NDN stack and NDN infrastructure. We will again measure end-to-end latency and round trip times for 5 of its APIs by load testing them.

D. Analysis

We will present an analysis of performance comparison between REST-HTTP stack and REST-NDN stack.