Segregating Signaling and Media Planes into Different containers

Dr. Kiran V, IEEE Senior Member, Associate Professor, Rahul Raj D N, Undergraduate Student, Electronics and Communication Engineering, R V College of Engineering, Bangalore

Abstract— **Scalability is an important aspect of communication networks. With the ascent of SIP and associated modern real-time protocols, IP telephony has become a revolutionary technology in connecting users through real time voice communications with enabled video and instant messaging capabilities. B2BUA is a SIP server that provides call management and authentication functionality by reformulating the request and routing the traffic to other user agent in the network. It comprises of signaling and media entities that handles all control signaling messages and real time data(media) information respectively. The signaling and entities run as different processes in the same container. Such an architecture encounters a large CPU utilization after a specific number of maximum calls due to increase traffic flowing within the same node. Further Packet processing is CPU intensive and there is need for architecture that scales well with increasing traffic without hitting the CPU performance. The paper presents the design of decoupled architecture for Signaling and Media entities by running both the processes in different containers. With such an approach, one Signaling entity can communicate with multiple Media entities or vice versa thereby providing a suitable scalable solution to deal with the increased traffic and further maintaining the system efficiency. The paper is concluded by highlighting the difference between Kubernetes and OpenStack for the proposed architecture**

Index Terms—**Back to Back User Agent(B2BUA), Docker, Kubernetes, OpenStack, Session Initiation Protocol (SIP).**

I. Introduction

Voice over Internet Protocol (VoIP) refers to sending voice and unified communications over an IP-based network. It differs from PSTN which forms a dedicated circuit connection for each call. IP telephony is more versatile and enables the transfer of voice data and video to multiple devices including smart phones laptops tablets and iPhones at a very low cost. They use Internet Protocol address (IP addresses) which defines rules for how computers and devices converse with each other on the Internet. Apart from making calls, VoIP service providers handle outgoing and incoming calls routing through existing telephone networks land lines and cell phones rely on the public switched telephone network PSTN.

VoIP was founded around 1995 by a company located in Israel called VocalTec to create a way to save money on long distance in international telephone charges they developed a product called Internet phone an application that offered computer to computer voice calls using a microphone and speaker.

The most common devices and network elements that participate in VoIP communication is depicted in figure 1. IP telephony calls can be generated directly by a special VoIP phone such as SIP phone, VoIP enabled PC or PC with necessary software which is connected to cable modem that enables high active internet connection. Other network elements include proxy to route the traffic to different network and gateways that connect IP based network to PSTN. Further, session border controllers that are responsible for authenticated service are embedded with such devices or present as independent entity.

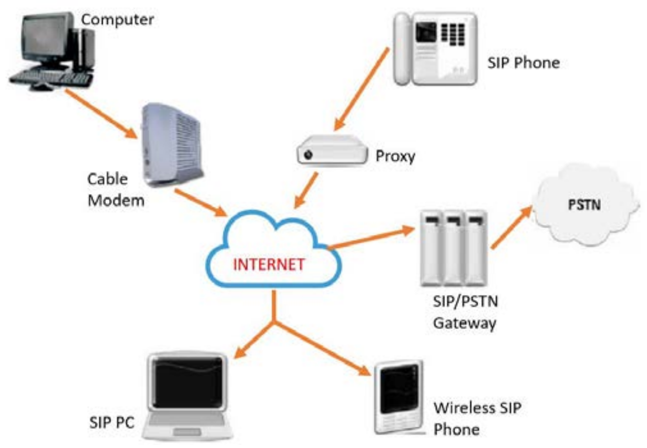


Fig. 1. Typical example of VoIP Technology

VoIP replaces the traditional analogue copper lines by internet connection for the communication. It uses a set of codecs at both ends to modify or convert into a pattern suitable for transmission and retrieve original data at receiving end. Main advantages of VoIP include low cost and easier accessibility.

SIP is the one of the most celebrated protocol in VoIP technology. It is primarily involved in handling sessions i.e., by initiating, managing and tearing down multimedia sessions. The protocol defines the rules with the set of messages that are involved in primary signaling before the exchange of actual audio/video packets. SIP network typically involves User Agents (endpoints), proxies (to forward all kind of signaling messages) and different servers to keep track of identity and location information of user agent.

II. BACKGROUND

A back-to-back user agent is an intermediate network element in SIP that takes request from on end, reformulates the request and forwards it to other end. It is similar to proxy server but in addition to the forwarding functionality it does the reformulation of request by adding network and media related information. It can be thought of as a composition of client and server. It behaves as a server when accepting the request and as a client when sending out the modified request. B2BUA creates a dialog state and involves in the entire duration of the dialog. It also captures the complete state information of calls throughout the session.

The functions provided by B2BUA include management of calls with support for transfer and disconnection of calls. Further it provides abstraction by hiding network topology. These are present as integral part of PBX and gateways. Session Border Controller can be considered as one of the most common B2BUA.

III. RELATED WORK

The unfolding of the Session Initiation Protocol (SIP) promised a simple and effective way for multimedia session handling among multiple users. In paper [1], SIP-based VoIP system was designed to ensure provision for a wide range of services. It emphasized on the cost savings of VoIP over traditional PSTN network through which organizations incurred toll charges. Main contributions of this paper include support for multi-conferencing along with point-to-point VoIP call. Paper [2] dealt with Asterisk, which is a unique open source PABX and the implementation of VoIP on it. It showed the configuration of Asterisk to implement normal calls, voice mail, and conferences on a local network with soft phones.

Security aspects and the challenges faced by SIP trunks are dealt in [3]. Adding PBX and SIP trunking service on top of exiting network does not provide the SIP packets to pass through. This is overcome when SBC is introduced at the edge of the network that allow only authorized calls to enter through organization. Further by applying real time security policies, SIP controls VoIP traffic.

Paper [4] presented a detailed survey on detection procedures for DoS and DDoS attacks in the context of VoIP network. DoS attack by flooding the SIP server with different SIP-messages, analysing the performance by SIP server by considering different performance metrics such as CPU and memory utilization is highlighted in [5]. Asterisk is used as SIP-server and the capture of voice packets on both ends is accomplished by Wireshark tool. It was observed that call initiation failed after a maximum number which in this case was 1387 calls. Further quality of VoIP calls is analysed by bombarding the server with only 2000 packets, 2000 packets with 100 simultaneous calls, 2000 packets with 200 simultaneous calls. With stress, it is observed that quality goes down in terms of jitter and delay.

The work in [6] elaborated on SDN to have a decoupled architecture for control and data planes. Basically, a controller with a centralised approach exhibits a tuned control over the underlying hardware/switch (data). This provides a better abstraction over the underlying hardware with more focus on scalability and enhanced performance. Several important aspects of SDWN and its relevance in wireless technology was presented. In paper, [7], the formal flow of SIP which involves initialization, registration and authentication, and the challenges faced with different attacks are discussed.

Authors in [8], presented OpenSIP which used proliferating technologies, such as software-defined networking (SDN) and network function virtualization (NFV). One of the main problems with SIP network is the overload incurred by SIP proxy. Paper [9] presented a comparative study between hypervisor and docker. The hypervisor was chosen as Xen; further the overhead involved in virtualization in HPC and OLTP were discussed. Platform independent isolated development with container technology and the way of incorporating dependent libraries were discussed in paper [10]; further different network architecture in the context of industrial automation were highlighted.

Developing application cloud using docker, Kubernetes, google cloud was surveyed in [11]. The specification related to docker daemon, architecture of Kubernetes and features of Kubernetes and its relevance in container technology with regard to health checks were highlighted. Leveraging Kubernetes for IoT applications has been emphasized in paper [12], [13]. The authors proposed the KEIDS scheduler that does two functions namely synchronization and scheduling that keeps check on desired state of cluster and schedules accordingly. With such a schedular, energy Utilization saw an improvement on the desired application by 14.42% with least interference.

Kubernetes engine scaler was proposed in paper [14] that is based out on machine learning. Various algorithms compete within the scaler to direct to a method that best suits for driving the traffic in situations of continuously varying requests. The network plugins and different network interface implementations were highlighted in [15].

IV. SIP CALL FLOWS

SIP is open standard and text based signalling protocol. Its purpose is to setup, modify and tear down sessions by following a request-response transaction model. SDP is text based description protocol that works in association with SIP. It defines parameters related to media. This is advertised by the user agents in the session/conference to describe parameters such as, the name of the owner of the session, the name of the session, the coding, the media, protocols, codec formats, timing, transport information etc.

Depending on the location of SDP message there are 3 types of call flows. SDP is message to describe media streams in a format understood by participants. Depending on this description party decides whether to join the conference or when or how to join the conference. Description involves information such as name of owner, name of session, media protocols, coding, timing, the codec formats.

Early Offer-Early Offer (EO-EO) call-flow: This is the case where caller sends on codec related information in SDP that is a embedded within the initial invite message. The callee then has a privilege to look into this codec and has the provision to negotiate and to use the codec of choice as shown in figure 2. This type of scenario in which caller is offering and callee is answering with codec it is going to use is called as offer- answer model.

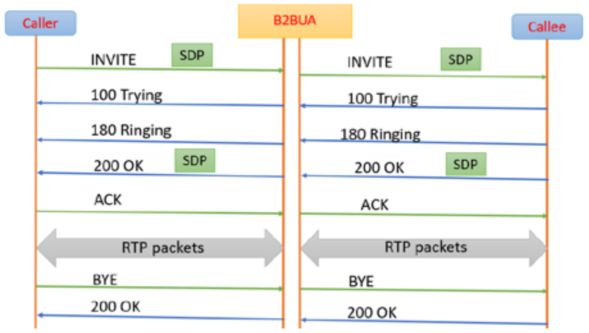


Fig. 2. Depiction of EO-EO call flow

Delayed offer-Delayed Offer (DO-DO) call flow: Here the SDP message is sent in the 200 OK response from callee. this is forwarded by B2BUA 2 caller. Caller then sends and negotiates codec by sending SDP within acknowledgement to B2BUA as shown in figure 3.

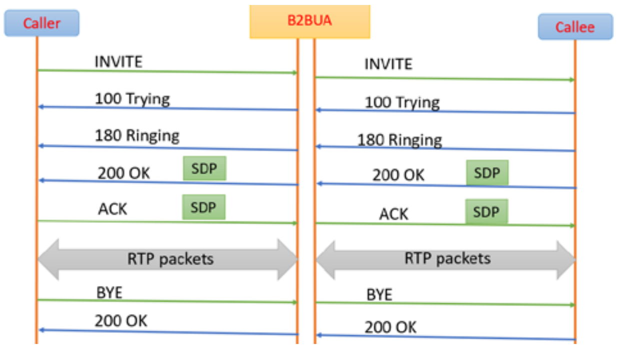


Fig. 3. Depiction of DO-DO call flow

Delayed Offer-Early Offer (DO-EO): This is accomplished by setting a parameter called forced early offer as true in the configuration file. On one call leg this appears as delayed offer while on the other leg the B2BUA adds the SDP and hence it behaves as early offer as shown in figure 4.

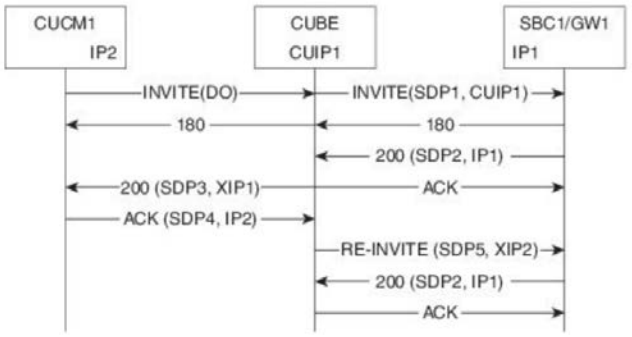


Fig. 4. Depiction of DO-EO call flow

V. Docker

Docker is a framework that provides the ability to have isolated environments to develop and package applications. Its main purpose is to containerize applications, ship them to different environments, and run applications on remote hosts without any requirements/dependencies.

In a scenario where an application stack has to be deployed with different applications such as MySQL, Redis, MongoDB, etc., there is a need to explicitly take care of versions of each application and its compatibility with the underlying OS. Each service may require libraries or dependencies of different versions. With docker, each component can be run in separate containers with its own dependencies and libraries, all on the same OS but in separate environments. To bring up an application there is just a need to run a simple docker run command.

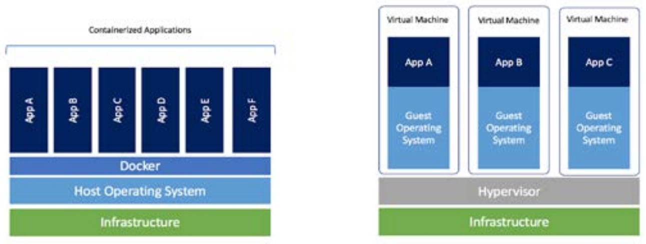


Fig. 5. Docker vs Virtualization

As shown in the figure 5, docker systems will have underlying hardware on which operating systems are running. Docker engine is installed on top of the OS and containers with necessary libraries and dependencies run on top of the docker engine. Whereas in virtual machines hypervisors such as VMware or VirtualBox is installed on the underlying hardware and virtual machines run on top of them. This overhead causes higher utilization of underline resources as there are multiple operating systems and kernels running. Virtual machines consume high disk space as each of the machines is heavy (in GB) and also they take minutes to boot up as it needs to boot up the entire kernel. Docker containers are light and they run in seconds. Other differences include deployment is easy in the case of docker and it's easily portable. Whereas in VM since they are completely isolated and don't rely on the underlying OS they provide complete isolation and the ability to run different applications on different OS such as Windows, Mac, and Linux flavours.

Docker containers are created using images that are formed from Dockerfile. Containers are running instances of the formed images.

Docker networking enables user to link the docker container to as many networks as desired and provides complete isolation. The network drivers supported by docker are:

- BRIDGE Network: It is a private default network created on the host. Docker daemon created virtual ethernet bridge and performs operation by automatically delivering packets across network interfaces.
- HOST Network: It is public network that uses host IP and TCP port to display services running inside containers. But this type of networking doesn't provide complete isolation and hence multiple containers cannot be run.
- OVERLAY Network: It is used to create internal private network particularly in orchestration tools such as Docker swarm cluster
- MACVLAN Network: This network assigns a MAC address to the Docker container and routing of traffic is based on this address.
- None: Total networking functionality is disabled for container

VI. Kubernetes

Kubernetes developed by Google is a container orchestration tool which is open-source. It basically helps in managing containerized applications that are made of a large number of containers and helps us manage in different environments like physical machines, virtual machines, cloud, and also the hybrid environment. Kubernetes comes to action after the containerized application has been deployed and takes care of automating scheduling and managing the deployed container.
The rise of microservices has increased the usage of container technologies because containers actually offer the perfect post for small independent applications like microservices. This surge in usage of microservices or containers has resulted in applications now comprised of hundreds or thousands of containers and managing those containers across multiple environments using scripts and self-made tools is really complex so there is a need for orchestration tools such as Kubernetes.

Features of an orchestration tool include high availability which means there would not be downtime for application our application is always accessible by users. Secondly, high scalability and hence high performance mean applications load faster with higher response time. In scenarios when used demand increases the traffic can be load-balanced across different nodes by simply replicating the pod instances.

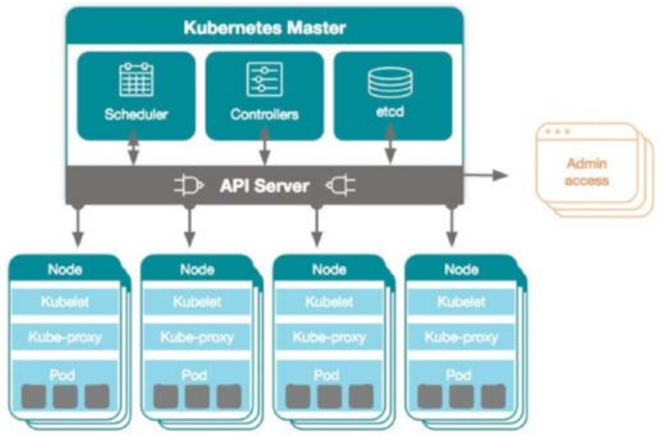


Fig. 6. Kubernetes Architecture (referenced from [11])

Four processes run on master node as shown in figure 6:
To deploy any new application or to schedule a pod on the remote cluster one needs to interact with API server with some client. API server acts cluster gateway or gatekeeper for authentication. Scheduler is one which schedules new pod; request is forwarded from API server to scheduler in order to start pod in one of node. It has intelligence to decide on which node to deploy the pod by checking the resources available or by checking which is least busy. Next important process is the controller manager which detects cluster changes such as pod crashing.
Worker node is there note that actually does the work it involves 3 tools or processes. Application pods have container running in it. So, container runtime has to be installed in every node. But the process that actually schedules those containers is kubelet. It interacts with both container and nodes. It gets request from scheduler to start the pod with containers which then start the node and assign resources to it.
Etcd is one more process that is responsible for storing the data. Every change in cluster such as new pod coming or any board crashing will be logged in etcd. it will store all the data in the form of key value store.

Deployment: Pods are the smallest deployable unit in Kubernetes. They provide abstractions over containers end enables the user to interact with only Kubernetes and its layers. Usually pods run one application within it sometimes it is also possible to run helper application within the same pod.

Deployments are the frameworks to define blueprint for pods. They are similar to replica sets. It can create multiple replicas of pods. The deployment provides us with capabilities to upgrade the underlying instances seamlessly using rolling updates, undo changes, and pause and resume changes to deployments as shown in figure 7.

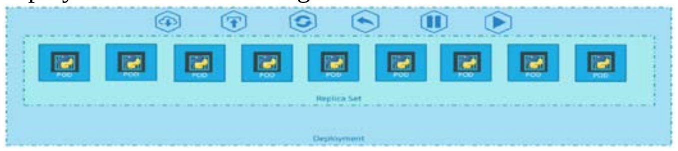


Fig. 7. Kubernetes Deployment

Services: Deploying pods over cluster gives them their own IP addresses but pods are ephemeral i.e., they are destroyed frequently. When a pod dies and a new pod comes up, it gets a new IP address. This is not desired and hence there is a need to have some service that provides stable or permanent IP addresses. Services provide load balancing across multiple replicas of pods and also provide good abstraction for loose coupling or communication within and outside cluster. 3 kinds of service are possible:
Cluster IP services is only accessible within the cluster. No external traffic can directly access cluster service. Whereas node port service creates service that is accessible on static port on each node port. Load balancing service becomes accessible externally through external cloud proxy servers such as Google cloud proxy Azure services etc.

VII. GRPC AND PROTOCOL BUFFERS

Today's trend is to build microservices and these microservices are in different languages and involves functions for user needs. For microservices to exchange information they must agree on API to exchange data, data format, error pattern, load balancing. One popular choice for building API is REST(HTTP-JSON. GRPC is one such framework that does all these in the backend. It is a free and open-source RPC framework that can run on any environment. It allows us to define requests and responses for RPC and handles all the rest by itself. It is modern, fast, efficient, and built on top of HTTP 2. Other features include low latency, supports streaming, and language independence.
Use of GRPC as a communication framework for 2 remote hosts is illustrated in figure 8. GRPC is basically used to efficiently connect services in and across data centers.

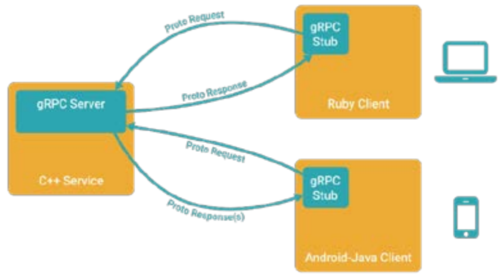


Fig. 8. GRPC client-server Paradigm

At the core GRPC uses protocol buffers to define messages and services. GRPC generates the skeleton code for us which defines classes and the implementation of server and client has to be done using those classes. Protocol buffers are the core stone of GRPC, where messages and services are defined to model the API endpoints.

Protocol buffers allow data to be compressed automatically and they are 3-10 times smaller and 20 -100 times faster than XML. The message is for the protocol buffers are defined in the profile and go generated code involves classes for implementation of the interfaces in many languages such as Java, C++, Go, Python, etc. Data serialization simply means transforming data from one format to another and deserialization is bringing back to original form.

Protocol buffers offer a very easy method to write message definition. The definition of API is independent of the implementation. Protocol buffers are used by GRPC and which is built on HTTP2. GRPC leverages HTTP 2 for backbone communications. It addresses some common pitfalls of HTTP 1.1. JSON also has a schema to transport data from client to server. But here it is sent over HTTP.

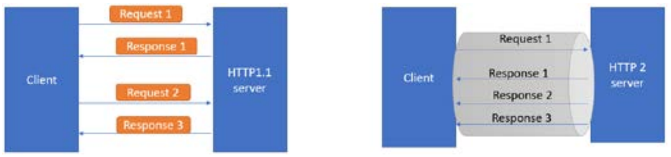


Fig. 9. HTTP1.1 vs HTTP2

A new TCP connection is opened by HTTP1.1 for every request to an endpoint as shown in figure 9. It does not compress headers. It is based on request and response mechanisms. This inefficiency adds latency and increases network packet size. HTTP 1.1 makes it easy for debugging but it is not efficient for transport over the network. Whereas HTTP2 supports multiplexing that is server and client can push messages in parallel over the same TCP correction. Latency is thus minimal and supports multiple messages streaming for one request. It also supports header compression and since it's binary it is more secure than the previous topologies.

GRPC supports 4 types of API or RPC calls. Unary streaming is the basic one which is similar to traditional request-response service server streaming is one in which the client is expecting a streamed response from the server. A stream of continuous requests is sent from client end to server in client streaming. Finally, bidirectional streaming is the most advanced in which the client and server are involved in streamed requests and responses. GRPC serves as asynchronous by default which means they do not block threads on requests. Whereas GRPC clients can be implemented as either asynchronous or synchronous; this is decided by the client upon implementation of the respective architecture.

GRPC uses protocol buffers that are smaller and faster and it is built on top of HTTP to offers the least latency. Whereas rest interface is based on JSON which is text-based and hence slower and occupies large space. JSON is based on HTTP1.1 and supports the client to server requests only that is it supports only request-response services. GRPC on the other hand supports streaming which is referred to as server push. And also it is API oriented which basically means it has very few constraints and it only thinks of what has to be implemented. Whereas REST is CRUD (create- retrieve- update) oriented using POST, GET, PUT and DELETE respectively. These are the action verbs that are used while accessing the URL or the desired service.

VIII. METHODOLOGY

The methodology followed to meet the design objectives of the project is shown in figure 10 Any communication between Signaling and media happens via Unix Socket which is present as a library. The project implementation involves the design for decoupled architecture for signaling and media entities and communication between them via GRPC. The Unix socket interface for the communication between different processes within the same VM/container is to be replaced with GRPC.

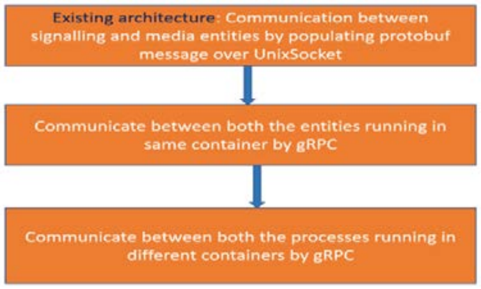

Existing architecture: Communication between signalling and media entities by populating protobuf message over UnixSocket
Communicate between both the entities running in same container by gRPC
Communicate between both the processes running in different containers by gRPC

Fig. 10. Design Methodology

Unix Socket doesn't provide the ability for communication between two separate hosts/containers. With GRPC, communication between different containers/VMs is possible because of the client server paradigm.

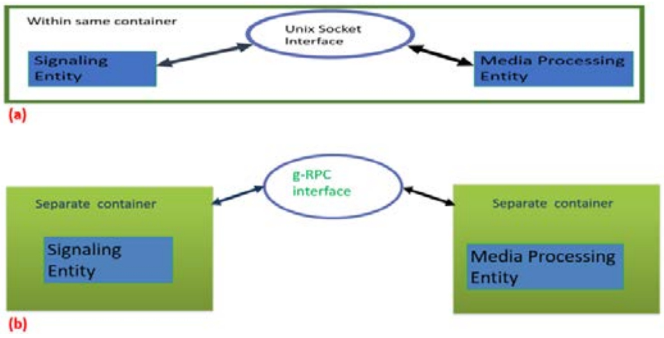


Fig. 11. Decoupled architecture for B2BUA

As shown in figure 11(a), the first goal of the project is to bring about the communication between two processes running on the same container via GRPC. Later decoupled architecture as proposed in figure 11(b) is to be implemented for signaling and media processes in different containers. The design will be implemented on Kubernetes platform as communication between separate pods since it provides a framework that can be easily portable in many public data centers.

IX. DESIGN

Design of Deployment and Service

In the case of deployment, the replicas, the metadata with labels are specified. This is important as any request coming from the outside or the ones that the service directs has to match those labels to direct the traffic. Next, to have deployment as an abstraction overpowered, the template is defined that specifies the regular pod specification with the images and the port at which the application has to be exposed. The DNS server monitors the Kubernetes API server and when a new service is created its name becomes available for easy resolution for requesting application. Kubernetes Ingress exposes HTTP and HTTPS roots from outside the cluster to services within the cluster. Traffic routing is controlled by rules defined on the ingress resource. An ingress controller is responsible for fulfilling the ingress. It is basically a daemon deployed as a Kubernetes pod that watches is the API server for updates to the ingress resource.

Design of GRPC Communication

For GRPC the design of communication between two processes was started by simple implementation of a client-server program. The GRPC library along with necessary 3rd party tools was installed in the container which is based out on GCC. Similar steps are followed in another GCC container. The communication between the 2 processes running in different containers was brought about by calling the API on GRPC which triggers the message. The messages are defined in a proto file which generates skeleton code where structures and protobuf messages are defined. The implementation code was written using these messages.

Next, the implementation of a similar case was done in a bidirectional way where the GRPC client sends an argument to the method on a server end server acknowledges the request by sending the response to the client with the result. This was implemented in 2 different containers and shows the interactive way of communication between them.

Further, the actual goal of communicating between 2 pods was done by GRPC. This was achieved by creating an API service to serve REST API response to the client route the request. This was done by container rising the actual application and also the service to deploy in the Kubernetes cluster. The images were pushed to the remote docker hub repository further the objects were configured in Kubernetes for deployment and service to manage the desired status of the pod and to drive traffic. Services provide the fixed addresses to access those pods. note port services were used since it is accessible from outside of the Kubernetes cluster.

Next dealing with the communication between signaling and media entities, the first integration was done by replacing an API that is responsible for clearing the active streams on media entities. This basically dealt with a message sent from signaling to media process whenever the B2BUA suffers a crash and restarts as shown in figure 12. This may result in some active sessions still running on the media entity. So, the goal here is to make sure that signaling will notify the media entity to remove all the sessions whenever the process starts.

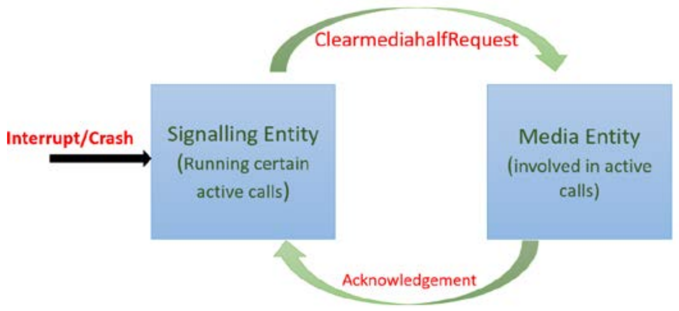

ClearmediahalfRequest
Interrupt/Crash
Signalling Entity (Running certain active calls)
Media Entity (involved in active calls)
Acknowledgement

Fig. 12. Clear Request message between the entities

X. IMPLEMENTATION

The calls discussed earlier are implemented by testing the SIPp calls. Each of the call scenarios can be independently tested by writing a separate set of XML files on the client and server sides. The general flow of signaling and the corresponding messages are specified in XML files. Each of the dynamic entries such as IP addresses and ports is present as placeholders in XML files. These placeholders are filled with the suitable values of the passed command line arguments. The XML files of the client and server are run as separate independent threads. Further, the server and client ports to be used for communication are passed as arguments while running SIPp calls. The number of calls can also be scaled by using -m option while running the calls. This makes the SIPp an amazing tool for SIP traffic generation and then to validate the flow of messages as specified in SIP XML scenarios.

To run a normal SIPp calls (assuming EO-EO call), the command is as follows:

```
sipp_64 -sf uas.xml -i  172.200.1.10 -p  7979 -t u1 -nr
sipp_64  -sf  uac.xml  -s   345  -i  172.200.1.10  -p   6012
172.200.1.20:5060 -t u1 -nr -m  50
```

Here IP address 172.200.1.20:5060 specifies the address of intermediary B2BUA that is present at the middle and form a separate call legs on either side. IP address 172.200.1.10:7979 is the specification for server(callee) while 172.200.1.10:6012 is the specification of the client(caller).

Implementation of Kubernetes Service:

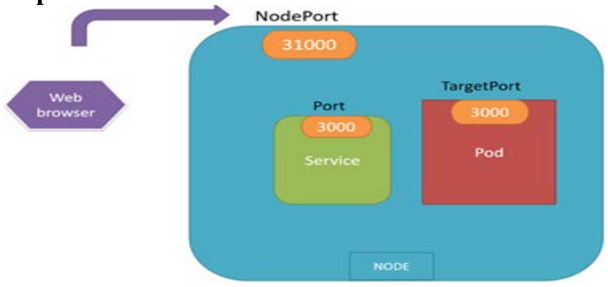


Fig. 13. Kubernetes NodePort service

As shown in the figure 13, the nodePort service creates a ClusterIP Port by default where the application can be accessed directly at a particular targetPort. Therefore the the traffic from outside world is directed first to clusterIP and then to the deployed application. To create any new deployment, it only needs an YAML file with suitable configuration and kubectl command to bring up such entities.

Implementation of GRPC

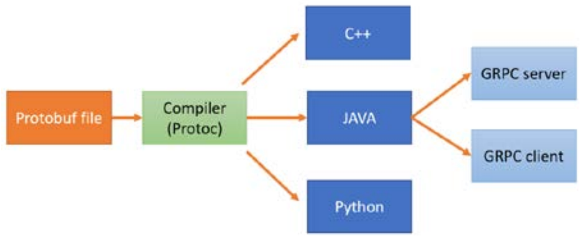
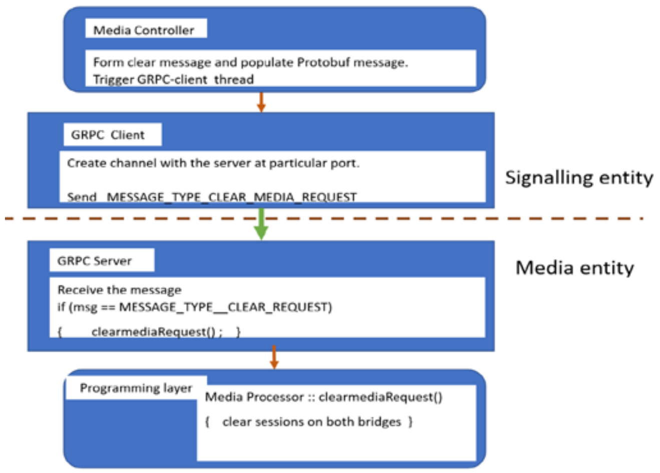


Fig.14. GRPC workflow

GRPC communication between the containers was achieved by simply installing all the GRPC libraries and third party tools in a container that is based on GCC and python. With these containers, the client running on GCC container was made to communicate by GRPC to the server running in python container. The basic workflow of GRPC is show in the figure 14.

Implementation of GRPC on B2BUA

The communication between signaling and media entities is challenging and hence requires a careful design procedure. This necessarily involves the replacement of UnixSocket's way of communication. Further, the GRPC as a library has to integrate into the B2BUA. An open-source RPM of GRPC that was compatible with CentOS 7 with all required dependencies was unavailable. So, one of the goals was to have GRPC RPM (that was custom build) to be integrated with all necessary dynamic and static libraries and not causing any conflict to the existing topology of B2BUA.



```
Form clear message and populate Protobuf message.
Trigger GRPC-client  thread

Create channel with the server at particular port.
Send  MESSAGE_TYPE_CLEAR_MEDIA_REQUEST

Receive the message
if (msg == MESSAGE_TYPE__CLEAR_REQUEST)
{       clearmediaRequest() ;   }

Media Processor :: clearmediaRequest()
{   clear sessions on both bridges  }
```

Fig.15. Flow for implementation of GRPC

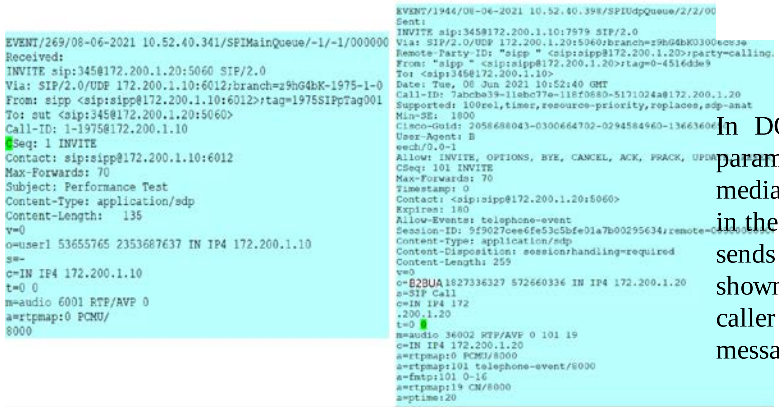
With this done the clearmediaRequest message to be communicated between signaling and media entities followed the flow as in figure 15. The GRPC client is run on the signaling side which sends the clearmediaRequest message to the GRPC server. This GRPC server runs in the form of a thread on the media process. On receiving this request the server calls to a suitable function to clear the active streams on both the bridges.

XI. RESULTS AND DISCUSSION

Analysis of all the designs mentioned and implemented in previous sections are discussed. Further detailed outputs for each of the case scenarios are illustrated. Most of the programs are compiled using GCC and CMake and their respective outputs are observed in the terminal. For Kubernetes-based deployments, outputs are shown in the web browser.

Simulation Results of Call flows

As observed from the results obtained in the log file after running the EO-EO calls from XML files as shown in figure 16, the codec negotiation happens on the second leg as the caller initially sends the timing and media-related information in the SDP message within INVITE.



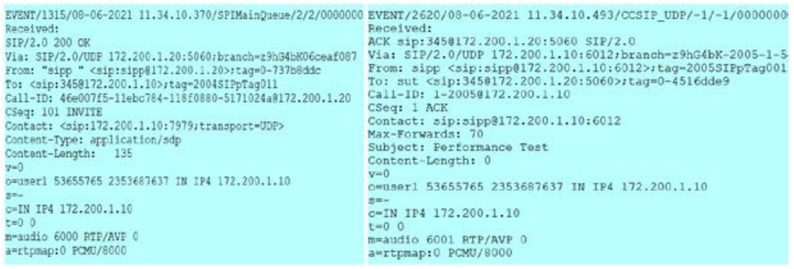
```
EVENT/269/08-06-2021 10.52.40.341/SPIMainQueue/-1/-1/000000
Received:
INVITE sip:345@172.200.1.20:5060 SIP/2.0
Via: SIP/2.0/UDP 172.200.1.10:6012;branch=z9hG4bK-1975-1-0
From: sipp <sip:sipp@172.200.1.10:6012>;tag=1975SIPpTag001
To: sut <sip:345@172.200.1.20:5060>
Call-ID: 1-1975@172.200.1.10
CSeq: 1 INVITE
Contact: sip:sipp@172.200.1.10:6012
Max-Forwards: 70
Subject: Performance Test
Content-Type: application/sdp
Content-Length:   135
v=0
o=user1 53655765 2353687637 IN IP4 172.200.1.10
s=-
c=IN IP4 172.200.1.10
t=0 0
m=audio 6001 RTP/AVP 0
a=rtpmap:0 PCMU/
8000
```

```
EVENT/1944/08-06-2021 10.52.40.398/SPIUdpQueue/2/2/00
Sent:
INVITE sip:345@172.200.1.10:7979 SIP/2.0
Via: SIP/2.0/UDP 172.200.1.20:5060;branch=z9hG4bK0300ec83e
Remote-Party-ID: "sipp " <sip:sipp@172.200.1.20>;party=calling.
From: "sipp " <sip:sipp@172.200.1.20>;tag=0-4516dde9
To: <sip:345@172.200.1.10>
Date: Tue, 08 Jun 2021 10:52:40 GMT
Call-ID: 7abcbe39-11ebc77e-118f0680-5171024a@172.200.1.20
Supported: 100rel,timer,resource-priority,replaces,sdp-anat
Min-SE:  1800
Cisco-Guid: 2058688043-0300664702-0294584960-1366360650
User-Agent: B
eech/0.0-1
Allow: INVITE, OPTIONS, BYE, CANCEL, ACK, PRACK, UPDA
CSeq: 101 INVITE
Max-Forwards: 70
Timestamp: 0
Contact: <sip:sipp@172.200.1.20:5060>
Expires: 180
Allow-Events: telephone-event
Session-ID: 9f9027cee6fe53c5bfe01a7b00295634;remote=0
Content-Type: application/sdp
Content-Disposition: session;handling=required
Content-Length: 259
v=0
o=B2BUA 1827336327 572660336 IN IP4 172.200.1.20
s=SIP Call
c=IN IP4 172
.200.1.20
t=0 0
m=audio 36002 RTP/AVP 0 101 19
c=IN IP4 172.200.1.20
a=rtpmap:0 PCMU/8000
a=rtpmap:101 telephone-event/8000
a=fmtp:101 0-16
a=rtpmap:19 CN/8000
a=ptime:20
```

Fig.16. Simulation Result for EO-EO case

The second party after analysing the SDP within INVITE does codec negotiation sends the codec-related details it is going to use in the 200 OK response to the B2BUA. The B2BUA then forwards this parameters within the 200 OK response it sends to the first party. The above outputs and the direction of messages are with respect to B2BUA.

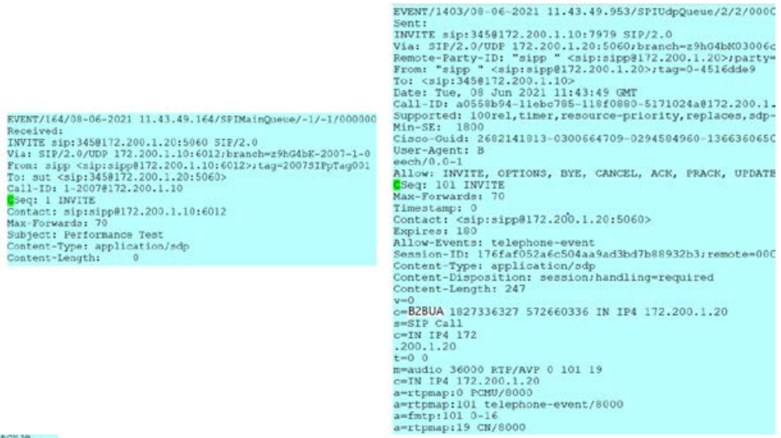
Unlike EO-EO as shown in figure 17, here the sender waits for codec related information from other party and hence does not send SDP in the INVITE. The callee party sends the codec related and media type information through SDP in the 200 OK response to the intermediate agent which is then forwarded to the intermediate agent.



```
EVENT/1315/08-06-2021 11.34.10.370/SPIMainQueue/2/2/0000000
Received:
SIP/2.0 200 OK
Via: SIP/2.0/UDP 172.200.1.20:5060;branch=z9hG4bK06ceaf087
From: "sipp " <sip:sipp@172.200.1.20>;tag=0-737b8ddc
To: <sip:345@172.200.1.10>;tag=2004SIPpTag011
Call-ID: 46e007f5-11ebc784-118f0880-5171024a@172.200.1.20
CSeq: 101 INVITE
Contact: <sip:172.200.1.10:7979;transport=UDP>
Content-Type: application/sdp
Content-Length:   135
v=0
o=user1 53655765 2353687637 IN IP4 172.200.1.10
s=-
c=IN IP4 172.200.1.10
t=0 0
m=audio 6000 RTP/AVP 0
a=rtpmap:0 PCMU/8000
```

```
EVENT/2620/08-06-2021 11.34.10.493/CCSIP_UDP/-1/-1/0000000
Received:
ACK sip:345@172.200.1.20:5060 SIP/2.0
Via: SIP/2.0/UDP 172.200.1.10:6012;branch=z9hG4bK-2005-1-5
From: sipp <sip:sipp@172.200.1.10:6012>;tag=2005SIPpTag001
To: sut <sip:345@172.200.1.20:5060>;tag=0-4516dde9
Call-ID: 1-2005@172.200.1.10
CSeq: 1 ACK
Contact: sip:sipp@172.200.1.10:6012
Max-Forwards: 70
Subject: Performance Test
Content-Length: 0
v=0
o=user1 53655765 2353687637 IN IP4 172.200.1.10
s=-
c=IN IP4 172.200.1.10
t=0 0
m=audio 6001 RTP/AVP 0
a=rtpmap:0 PCMU/8000
```

Fig.17. Simulation Result for DO-DO case



```
EVENT/164/08-06-2021 11.43.49.164/SPIMainQueue/-1/-1/000000
Received:
INVITE sip:345@172.200.1.20:5060 SIP/2.0
Via: SIP/2.0/UDP 172.200.1.10:6012;branch=z9hG4bK-2007-1-0
From: sipp <sip:sipp@172.200.1.10:6012>;tag=2007SIPpTag001
To: sut <sip:345@172.200.1.20:5060>
Call-ID: 1-2007@172.200.1.10
CSeq: 1 INVITE
Contact: sip:sipp@172.200.1.10:6012
Max-Forwards: 70
Subject: Performance Test
Content-Type: application/sdp
Content-Length:      0
```

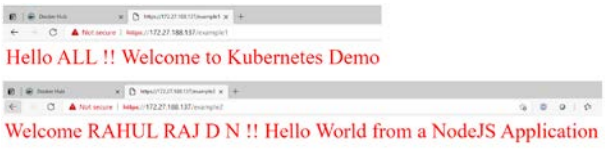
```
EVENT/1403/08-06-2021 11.43.49.953/SPIUdpQueue/2/2/000C
Sent:
INVITE sip:345@172.200.1.10:7979 SIP/2.0
Via: SIP/2.0/UDP 172.200.1.20:5060;branch=z9hG4bK03006c
Remote-Party-ID: "sipp " <sip:sipp@172.200.1.20>;party=
From: "sipp " <sip:sipp@172.200.1.20>;tag=0-4516dde9
To: <sip:345@172.200.1.10>
Date: Tue, 08 Jun 2021 11:43:49 GMT
Call-ID: a0558b94-11ebc785-118f0880-5171024a@172.200.1.
Supported: 100rel,timer,resource-priority,replaces,sdp-
Min-SE:  1800
Cisco-Guid: 2682141813-0300664709-0294584960-136636065C
User-Agent: B
eech/0.0-1
Allow: INVITE, OPTIONS, BYE, CANCEL, ACK, PRACK, UPDATE
CSeq: 101 INVITE
Max-Forwards: 70
Timestamp: 0
Contact: <sip:sipp@172.200.1.20:5060>
Expires: 180
Allow-Events: telephone-event
Session-ID: 176faf052a6c504aa9ad3bd7b88932b3;remote=00C
Content-Type: application/sdp
Content-Disposition: session;handling=required
Content-Length: 247
v=0
o=B2BUA 1827336327 572660336 IN IP4 172.200.1.20
s=SIP Call
c=IN IP4 172
.200.1.20
t=0 0
m=audio 36000 RTP/AVP 0 101 19
c=IN IP4 172.200.1.20
a=rtpmap:0 PCMU/8000
a=rtpmap:101 telephone-event/8000
a=fmtp:101 0-16
a=rtpmap:19 CN/8000
```

Fig.18. Simulation Result for DO-EO case

In DO-EO case, the first sender does not send the SDP parameters in the initial INVITE. But the B2BUA adds the media and codec related information to be used for the session in the second leg. The callee party analses the parameters and sends its SDP parameters in 200 OK response to B2BUA as shown in figure 18 which is then forwarded to caller end. The caller analyses these parameters and responds with SDP message in ACK.

Simulation Results of Kubernetes with Ingress

The implementation of Kubernetes ingress is implemented which does the path routing and drives the traffic to two different services. The ingress controller serves this request as defined in the ingress resource. Depending on the path of the external request the traffic is routed to correct service which in turn will be matched to correct application based on match labels.



Not secure | https://172.27.188.137/example1

Hello ALL !! Welcome to Kubernetes Demo

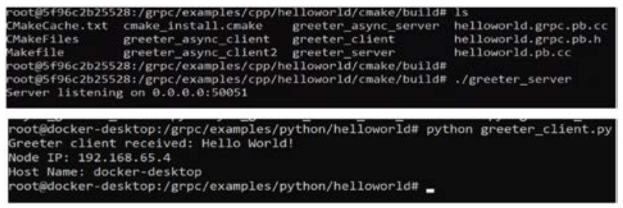
Not secure | https://172.27.188.137/example2

Welcome RAHUL RAJ D N !! Hello World from a NodeJS Application

Fig.19. Path Routing using Kubernetes Ingress

As shown in the figure 19, depending on the paths, the application accessed is different in both cases. This is based on path routing as implemented by ingress controller.

For GRPC programs, the communication between two different containers is brought about by utilizing the custom build GRPC RPMs. This is accomplished by creating a container that is based on python and GCC. That is the language-agnostic property of GRPC is leveraged and demonstrated here between python and C++ programs as shown in figure 20.

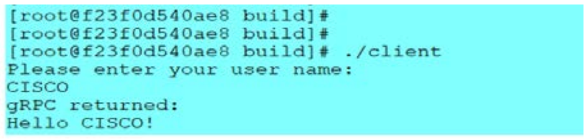


```
root@5f96c2b25528:/grpc/examples/cpp/helloworld/cmake/build# ls
CMakeCache.txt  cmake_install.cmake   greeter_async_server  helloworld.grpc.pb.cc
CMakeFiles      greeter_async_client  greeter_client        helloworld.grpc.pb.h
Makefile        greeter_async_client2 greeter_server        helloworld.pb.cc
root@5f96c2b25528:/grpc/examples/cpp/helloworld/cmake/build#
root@5f96c2b25528:/grpc/examples/cpp/helloworld/cmake/build# ./greeter_server
Server listening on 0.0.0.0:50051
```

```
root@docker-desktop:/grpc/examples/python/helloworld# python greeter_client.py
Greeter client received: Hello World!
Node IP: 192.168.65.4
Host Name: docker-desktop
root@docker-desktop:/grpc/examples/python/helloworld# _
```

Fig.20. GRPC communication (Unary)

Bidirectional streaming in which a sequence of requests and responses are enabled is implemented. The results of such an implementation is shown in figure 20.

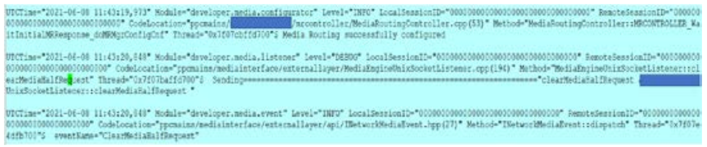


```
[root@f23f0d540ae8 build]#
[root@f23f0d540ae8 build]#
[root@f23f0d540ae8 build]# ./client
Please enter your user name:
CISCO
gRPC returned:
Hello CISCO!
```

Fig.21. GRPC communication (Bidirectional)

Here the client-server model involves the client sending a string and the server greets the client with Hello prefix. Further the client side implementation receives responses continuously.

Simulation Results of GRPC on B2BUA



```
UTCTime="2021-06-08 11:43:19,973" Module="developer.media.configurator" Level="INFO" LocalSessionID="00000000000000000000000000000002" RemoteSessionID="00000
0000000000000000000000000" CodeLocation="ppcmains/[redacted]/mrcontroller/MediaRoutingController.cpp(53)" Method="MediaRoutingController::MRCONTROLLER_Wa
itInitialMRResponse_doMRMgrConfigCnf" Thread="0x7f07cbffd700"$ Media Routing successfully configured

UTCTime="2021-06-08 11:43:20,848" Module="developer.media.listener" Level="DEBUG" LocalSessionID="00000000000000000000000000000000" RemoteSessionID="00000000
0000000000000000000000" CodeLocation="ppcmains/mediainterface/externallayer/MediaEngineUnixSocketListener.cpp(194)" Method="MediaEngineUnixSocketListener::cl
earMediaHalfRequest" Thread="0x7f07baffd700"$  Sending=================================================================="clearMediaHalfRequest [redacted]
UnixSocketListener::clearMediaHalfRequest "

UTCTime="2021-06-08 11:43:20,848" Module="developer.media.event" Level="INFO" LocalSessionID="00000000000000000000000000000000" RemoteSessionID="000000000000
0000000000000000000" CodeLocation="ppcmains/mediainterface/externallayer/api/INetworkMediaEvent.hpp(27)" Method="INetworkMediaEvent::dispatch" Thread="0x7f07e
4dfb700"$  eventName="ClearMediaHalfRequest"
```

Fig 22. GRPC results for B2BUA

The above outputs as in figure 22, the clearmediaRequest message being sent when the processes start off to clear any active sessions on the media entity side to make sure both the processes are in line with the new set of connections.

XII. CONCLUSION

The main motive of the project was to have a communication between signaling and media processes in the decoupled architecture. Scalability is an important aspect in the domain of networking. The existing topology used Unix Socket which is maintained as a library to bring about communication. But with the decoupled approach, UnixSocket is not a good candidate as Unix Socket is only applicable for communication within the same VM or container.

The goal of the project was to run signaling and media entity in separate VMs or containers. GRPC can run on different Inter-Process Communication such as by Unix Socket, shared memory, etc., But the goal is to have HTTP/1.0 way of communication. The sample application of Deployment, service are simulated over Kubernetes. Further the communication between the two containers using different IPs and ports. Later the GRPC communication between signaling and media entities was accomplished in the context of clearmediaRequest. This was the first integration of GRPC over the existing UnixSocket. The ultimate agenda of deploying signaling and media in different containers/VMs is to have a scalable framework where one signaling entity talks to many other media entities or vice versa. This way it can handle more calls without having to concern about CPU utilization.

XIII. FUTURE SCOPE

With the integration of GRPC over the B2BUA node, the processes within it can communicate with each other via GRPC. Decoupled architecture gives the ability to handle multiple calls without risking the CPU. Further, all this deployment is presently undertaken over docker containers running over the Openstack platform. Such a platform restricts our deployments to Webex data centers and hence does not provide the ability to be deployed in public data centers such as GCP, AWS. While Kubernetes is one such platform that allows having a framework to directly port to other platforms as it is supported by many other platforms. Deploying applications over Kubernetes and communicating between the pods via GRPC is challenging and requires a careful design procedure.

Although OpenStack is agile in build cloud infrastructure, it does not support portability and it restricts our deployments only to Webex data centers. Due to a very dynamic range of attributes, it lacks organized support. Since Kubernetes is widely accepted, our deployed application can be easily ported to public data centers such as AWS, Google Cloud, etc. Further having a decoupled architecture and communication by running the application as separate pods is feasible in Kubernetes. Also, down the line, the Open Stack and Kubernetes features can be complemented and run one over the other to achieve significant advantages.

REFERENCES

[1] S. Zeadally and F. Siddiqui, “Design and implementation of a sip-based VoIP architecture,” in18th International Conference on Advanced Information Networking and Applications, 2004. AINA 2004., vol. 2, 2004, 187–190 Vol.2.doi:10.1109/AINA.2004.1283783.

[2] L. Tian, N. Dailly, Q. Qiao, J. Lu, J. Zhang, J. Guo, and J. Zhang, “Study of sip protocol through VoIP solution of “asterisk”,” in2011 Global Mobile Congress,2011, pp. 1–5.doi:10.1109/GMC.2011.6103925..

[3] A. N. Jabel, S. Manickam, and S. Ramdas, “A study of sip trunk security and challenges,” in2012 IEEE International Conference on Electronics Design, Systems and Applications (ICEDSA), 2012, pp. 239–243.doi:10.1109/ICEDSA.2012.6507806.

[4] W. Nazih, W. Elkilani, H. Dhahri, and T. Abdelkader, “Survey of counteringdos/ddos attacks on sip based VoIP networks,” Electronics, vol. 9, p. 1827, Nov.2020.doi:10.3390/electronics9111827..

[5] A. Bansal, P. Kulkarni, and A. R. Pais, "Effectiveness of sip messages on sip server," in 2013 IEEE Conference on Information Communication Technologies, 2013, pp. 616–621.doi:10.1109/CICT.2013.6558168.

[6] M. Feng, S. Mao, and T. Jiang, "Enhancing the performance of future wireless net-works with software defined networking," Springer Frontiers of Information Technology and Electronic Engineering Journal, vol. 17, pp. 606–619, Jul. 2016.doi:10.1631/FITEE.1500336.

[7] P. Dhillon and S. Kalra, "Secure and efficient ecc based sip authentication scheme for VoIP communications in internet of things," Multimedia Tools and Applications, vol. 78, pp. 22 199–22 222, Aug. 2019.doi:10.1007/s11042-019-7466-y.

[8] A. Montazerolghaem, M. H. Y. Moghaddam, and A. Leon-Garcia, "OpenSIP: To-ward software-defined sip networking," IEEE Transactions on Network and Service Management, vol. 15, no. 1, pp. 184–199, 2018.doi:10.1109/TNSM.2017.2741258.49

[9] B. Wang, Y. Song, X. Cui, and J. Cao, "Performance comparison between hypervisor-and container-based virtualizations for cloud users," in2017 4th International Conference on Systems and Informatics (ICSAI), 2017, pp. 684–689.doi:10.1109/ICSAI.2017.8248375..

[10] M. Sollfrank, F. Loch, S. Denteneer, and B. Vogel-Heuser, "Evaluating docker for light weight virtualization of distributed and time-sensitive applications in industrial automation," IEEE Transactions on Industrial Informatics, vol. 17, no. 5, pp. 3566–3576, 2021.doi:10.1109/TII.2020.3022843

[11] J. Shah and D. Dubaria, "Building modern clouds: Using docker, kubernetes google cloud platform," in2019 IEEE 9th Annual Computing and Communication Work-shop and Conference (CCWC), 2019, pp. 0184–0189.doi:10.1109/CCWC.2019.8666479.

[12] K. Kaur, S. Garg, G. Kaddoum, S. H. Ahmed, and M. Atiquzzaman, "Keids: Kubernetes based energy and interference driven scheduler for industrial IoT in edge-cloud ecosystem," IEEE Internet of Things Journal, vol. 7, no. 5, pp. 4228–4237,2020.doi:10.1109/JIOT.2019.2939534.

[13] L. Toka, G. Dobreff, B. Fodor, and B. Sonkoly, "Machine learning-based scaling management for Kubernetes edge clusters," IEEE Transactions on Network and Service Management, vol. 18, no. 1, pp. 958–972, 2021.doi:10.1109/TNSM.2021.3052837.

[14] S. Qi, S. G. Kulkarni, and K. K. Ramakrishnan, "Assessing container network interface plugins: Functionality, performance, and scalability," IEEE Transactions on Network and Service Management, vol. 18, no. 1, pp. 656–671, 2021.doi:10.1109/TNSM.2020.3047545.

[15] A. Jeffery, H. Howard, and R. Mortier, "Rearchitecting Kubernetes for the edge," in Proceedings of the 4th International Workshop on Edge Systems, Analytics and Networking, ser. EdgeSys '21, Online, United Kingdom: Association for Computing Machinery, 2021, pp. 7–12,isbn: 9781450382915.doi:10.1145/3434770.3459730