Design and Implementation of distributed and fault-tolerant cellular core network

Project Dissertation

Submitted in partial fulfillment of requirements for the degree of

Master of Technology
Computer Science and Engineering

By

Vaishali Jhalani
173050033

Guided By

Prof. Mythili Vutukuru

Department of Computer Science and Engineering
INDIAN INSTITUTE OF TECHNOLOGY, BOMBAY

June, 2019

Acknowledgements

I would like to thank **Prof. Mythili Vutukuru** for project guidance and constant support. All the meetings were highly useful in understanding the topic in depth. I would also like to thank **Priyanka Naik** and **Sagar Tikore** for their valuable inputs.

Contents

List of Figures

Chapter 1

Introduction

With the decrease in the cost of electronic gadgets and the immense convenience provided by handheld devices, almost everyone today is expected to have a smart mobile device. This scenario has led to a continuously growing mobile data traffic all around the world. This data traffic is mainly handled by the cellular networks. Currently, 4G is the main wireless service which is provided by all mobile network companies. Main components in 4G lTE-EPC architecture are User Equipment (UE), Radio Access Network (RAN), Mobile Management Entity(MME), Serving Gateway (SGW), Home Subscriber Server (HSS) and many more. If we consider 4G which is still largely built over customized-hardware. The apparent solution to handle this growing data is to scale the core components but scaling in hardware needs a large amount of investment. Also, the failure requires a complete replacement of the hardware boxes. As the cellular operators are trying to attempt a technology known as NFV (Network Function Virtualization) which is able to virtualize the core network, it is easy to apply conventional distributed systems techniques to achieve scalability and fault-tolerance. In distributed system, for one server we can have many replicas and the load on all the replicas should be equal. To distribute the traffic equally there are many load balancing techniques. To make the system available all the time, we can use multiple replicas or can store the data in some reliable datastore.
The next generation cellular network is 5G which represents the next mobile wireless standards. Based upon the 4G LTE standards, 5G will also allow people to download and transfer photos, videos, etc. 5G core network also has many components like Radio Access Network (RAN), Session Management function (SMF), Access and Mobility Management function (AMF), User plane function (UPF), Policy Control Function (PCF), Authentication Server Function (AUSF), Unified Data Management (UDM) and many more. While significantly increase the speed of data and handling concurrent users at the same time. Main reason behind the efficiency of 5G is the use of technologies like NFV, SDN (Software defined networking). Here in this report, we only talk about NFV. To further increase the efficiency of 5G core network, scaling the main components of 5G core is a must. In the same way to ensure availability of the network, failure handling is also essential.

1.1 Motivation

We present the design and implementation of a distributed and fault-tolerant cellular core network. As the 4G/5G core components are stateful, therefore scaling and failure handling is a bit complicated because during these operations user and session states have to be maintained. In this project, we considered main component of the network for scaling and failure handling, MME in case of 4G and AMF in case of 5G core.
In 4G LTE-EPC, for each request from the user, MME (a component of EPC) stores the context of that user like IMSI(International Mobile Subscriber Identity), Tracking Area Identify, network capability, authentication key, location, connection state, etc. We call this information User Context. Throughout the session with respect to a user, MME uses and updates this information.
Same in 5G core network, for every request of the user AMF (component of 5G core) store context of that user like UE(user equipment) state, UserID, SUCI(Subscription Concealed Identifier), GUTI(Globally Unique Temporary ID), security context, etc. During whole registration and deregistration process, AMF uses these data to processes the request and also updates. While scaling or failure, these user contexts/states need to be handled properly. If a connection is diverted to a new node, these states need to be available to the new node otherwise the session would simply be dropped. We have explained about the user/session states in the later part of the report.
To make the design stateless (by not storing the user context on the node), we are using reliable data storage which can store state with respect to each UE(User Equipment). At the time of scaling or failure, other instances can fetch information from the data storage. We plan to explore different design options for scaling and failure handling: whether to use data storage or not for state synchronization, at what granularity should we save states in persistent key-value data storage. By using data storage, users may have better experience in terms of availability because the in-between states are not lost. This approach can help to reduce the recovery time after a failure, as the UEs will immediately migrate. But storing the context in data storage requires added communication leading to an increase in response time.
By comparing the above design options, we will be able to choose a suitable design to make the cellular networks reliable and horizontally scalable towards an increasing load.

Rest of the report structure is as follows: Next chapter contains a brief overview of LTE-EPC architecture, 5G design principles and core architecture, NFV technology, etc. We then discuss some related work in the area of distributed EPC and explained the relevance of our work. In chapter 3 we explain our design and implementation of distributed and fault-tolerant EPC. In chapter 3 we explain our design and implementation of distributed 5G core Network. Chapter 5 contains an evaluation of both the systems in terms of control plane performance. In the last chapter, we conclude and explain future work.

Chapter 2

Background and Related Work

2.1 Network Function Virtualization

NFV is a technique to virtualize network functions which traditionally run on a proprietary and customized hardware. Operations like routing, load balancing, network translation, etc are built as software which can be run on virtual machines and these are known as virtual network functions. These functions can be instantiated at various locations in a network without any new equipment. NFV has the capacity to increase performance and decrease cost as compared to specific hardware.

Maintaining and updating the equipment of network infrastructure is complex. Hardware boxes like routers, firewalls, load balancers, switches, etc are quite inflexible and costly to maintain. In place of costly and inflexible ASIC hardware systems, it would be easier to virtualize the network functions on a general purpose x86 hardware which is - easy to install, at a lower risk and consumes less power. With increasing load, it is difficult to scale any hardware equipment but NFV takes the benefit of distributed system techniques for maintaining scalability and reliability.

NFV with standard hardware may not be able to achieve the performance which a hardware box can. But, by using some advanced technologies like kernel bypassing with DPDK/Netmap, kernel optimization, etc, NFV can outperform customized hardware.

2.2 LTE EPC Architecture

The 21st century has seen tremendous growth in the number of smartphones. Nowadays LTE is the standard for high-speed smartphone communication. As it provides high bandwidth, low latency, backward compatibility. The Network architecture of LTE has 3 component: User Equipment, The Evolved UMTS Terrestrial Radio Access

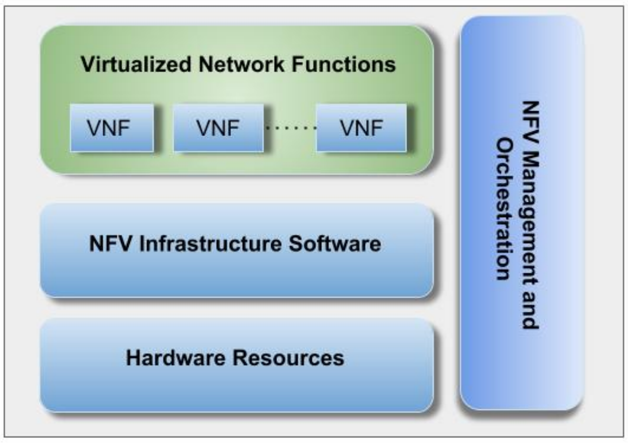

Virtualized Network Functions
VNF
VNF
VNF
NFV Infrastructure Software
Hardware Resources
NFV Management and Orchestration

Figure 2.1: Network Function Virtualization

Network (E-UTRAN) and The Evolved Packet Core (EPC). The E-UTRAN contains a group of Evolved NodeB. Figure 2.2 shows the architecture of LTE EPC.

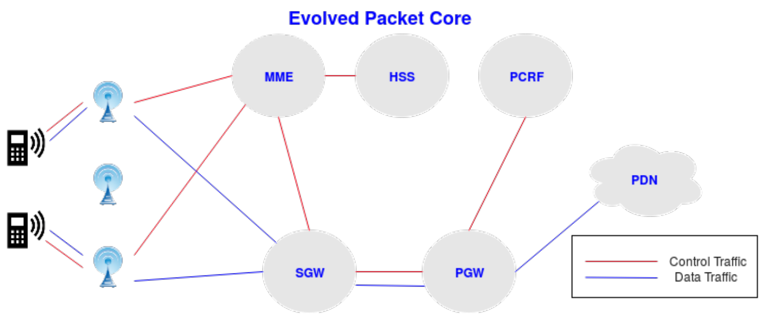


Figure 2.2: LTE EPC Architecture [?]

The EPC has many components: Mobile Management Entity (MME), Serving Gateway (SGW), Packet Data Network Gateway (PGW), Policy Control Rules Function (PCRF) and Home Subscriber Server (HSS).

All the EnodeB are interconnected to each other, also connected to the EPC. As EnodeB communicates to the UE(mobile equipment) wirelessly, the E-UTRAN acts as a communication medium between UE and EPC.

MME is responsible for handling the mobility of user equipment which sends the connection request to MME. The after data transfer tear down session is also handled by MME. Both PGW and SGW behave as a routing device and are responsible for forwarding packets between UE and Packet data network. The P-GW communicates with the outer network (PDN). P-GW also contains a component called Policy Control and Charging Rules Function (PCRF). PCRF is responsible for charging and Quality of Service (QoS). When a user moves from one place to other SGW is responsible for maintaining the connection.

2.3 LTE-EPC Procedure

There are various procedures carried out in LTE EPC when UE sends a connection request. The most common procedures include attach, data transfer and detach. Attach involves attaching a UE to the EPC using MME, SGW, and PGW. Data transfer involves a transfer of data between a UE and a server. Detach procedure is used to detach a UE from the EPC. These procedures are described in the subsequent sections.

2.3.1 UE Attach

As soon as the user turns on his equipment, the attach process begins. The attach procedure has some sub-procedures which need to be done sequentially. When the first step completes successfully only then the next step starts, otherwise user will not be able to connect to the network.

2.3.1.1 Authentication Request

The user sends an attach request to MME. On receiving this request the MME asks for this UE equipment from the HSS. HSS has a database which stores information about the UEs. If HSS gives positive response then only the connection procedure continues further. If HSS gives the expected authentication response to MME for UE which the MME stores, then MME sends the authentication request to the user equipment. When UE sends the authentication response to MME, it checks this response with the stored response. For successful completion of this process, these two responses should match.

2.3.1.2 Security setup

When the authentication completes successfully, two procedures, Non Access Stratum (NAS) and Access Stratum (AS) need to be executed to ensure secure communication. During this process, some encryption and integrity keys are exchanged between UE, eNodeB, and MME.

2.3.1.3 Location update

When the security setup is done successfully, MME updates the location of UE in HSS.

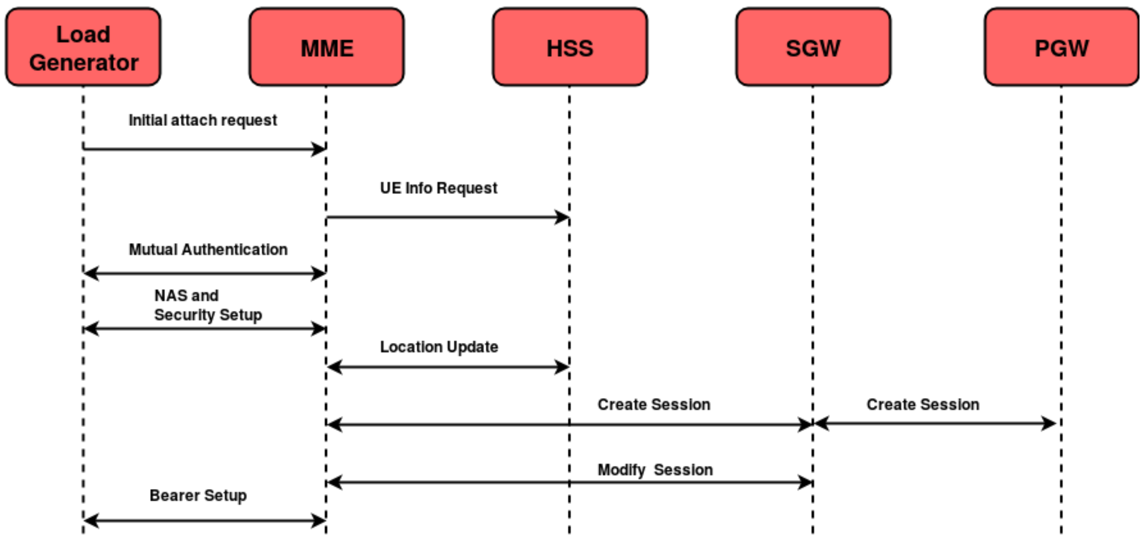


Figure 2.3: Attach Procedure

2.3.1.4 EPS Session/Bearer setup

After the location update, the MME would proceed with Create Session and Modify Session procedures for this UE by communicating with SGW and eNodeB. These sessions are required to create a setup for sending data through EPC to the PDN.

2.3.2 Data transfer

After the EPS session, the data is transferred in both uplink and downlink directions. The data transferred through GTP tunnels starts from UE then goes to the eNodeB, then to SGW, PGW and at the end to PDN. The process of tunneling is essential to identify user packets irrespective of their IP address.

2.3.3 UE Detach

Detach procedure is initiated by the UE. When UE sends a request to detach the connection, the MME first sends a request to SGW to delete the session. Then the SGW sends the same request to PGW and the responses come back in the opposite direction. In the end, MME sends the detach response to UE.[?]

2.4 5G Architecture

The fourth generation of mobile connectivity started in the late 2000s. 4G made mobile internet speeds up to 500 times faster than 3G and allowed support for HD TV on mobile, high–quality video calls, and fast mobile browsing.

4G with Long term evolution technology is an evolution of existing the 3G network. 4G is now deployed worldwide and massively used. But things like Internet-of-things, gaming, augmented reality required something more. That is where the 5G came into the picture. 5G stands for the fifth generation and represents the mobile wireless standard.

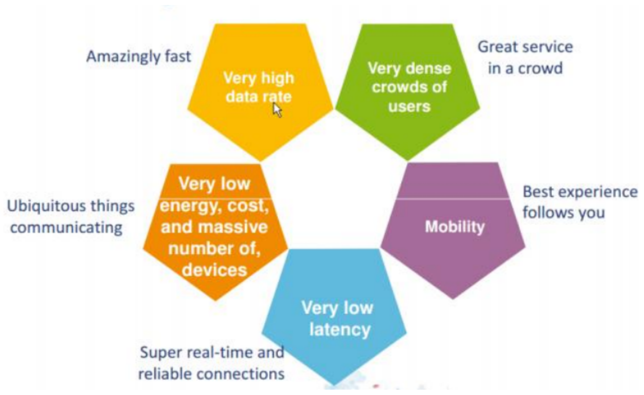

Amazingly fast
Very high data rate
Very dense crowds of users
Great service in a crowd
Ubiquitous things communicating
Very low energy, cost, and massive number of, devices
Mobility
Best experience follows you
Very low latency
Super real-time and reliable connections

Figure 2.4: Challenges and scenarios

5G fulfilled some of the requirements like latency reduction to less than one second, increment in data rates of at least one gigabit for thousands of users simultaneously, increment in energy efficiency, etc. In the absence of these features, IOT devices will not work correctly and seamlessly.

2.4.1 5G core Network: design principles and architecture

One of the main goals of 5G is to provide flexibility. It should be possible to modify the network structure according to the requirement, add new specifications to the network, adapt the changing traffic pattern, etc. Here NFV and SDN come into the picture. SDN and NFV are the two key technologies for realizing the 5G network. Other than driven by software, 5G also ensures End-to-End latency which is essential in all real-time applications. New air interface with the shorter transmission time interval can help to meet this challenge. 5G is also cost effective by only implementing l1/l2 functionality at the base station. All upper layer specifications will be there in the network cloud. Reducing the number of functionalities results in cost-effective management and deployment. Next, we will explain the service-based architecture of the 5G network. Control plane functions are connected via a service-based interface. The Access/session Management Functions are connected to the user-plane nodes over N1, N2 and N4 interface to manage subscriber attachment, sessions, and mobility. The N2 and N3 interfaces are determined by how the 5G radio presents itself to the core and, therefore, are dependent on the 5G RAN architecture. Below is the description of 5G network components:

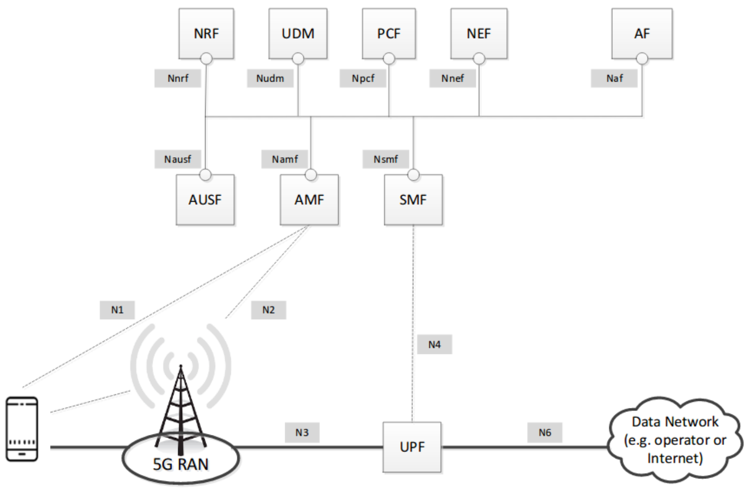


Figure 2.5: 5G Core Service-based Architecture

2.4.1.1 Access and Mobility Management Function (AMF)

: AMF manages registration, mobility and access control. It is the core component of the 5G network. AMF also collects mobility-related policies from the PCF (for example, mobility restrictions) and forwards them to the user equipment. AMF fully supports 4G interoperability with the interface to the 4G MME node.

2.4.1.2 Session Management Function (SMF)

: SMF sets up and manages sessions, according to network policy. It allocates IP addresses to the user equipment and selects and controls the UPF for data transfer. SMF also acts as the external point for all communication related to the various services offered and enabled in the user plane and how the policy and charging treatment for these services are applied and controlled.

2.4.1.3 User Plane Function (UPF)

: UPFs can be deployed in various configurations and locations, according to the service type. UPF is designed as a separate network functions virtualization (VNF) that provides a high-performance forwarding engine for user traffic.

2.4.1.4 Policy Control Function (PCF)

: This provides a policy framework incorporating network slicing, roaming and mobility management.

2.4.1.5 Unified Data Management (UDM)

: It Stores subscriber data and profiles. Similar to an HSS in 4G, but will be used for both fixed and mobile access.

2.4.1.6 NF Repository Function (NRF)

: This is a new functionality that provides registration and discovery functionality so that Network Functions (NFs) can discover each other and communicate via APIs.

2.4.1.7 Network Exposure Function (NEF)

: An API gateway that allows external users, such as enterprises or partner operators, the ability to monitor, provision and enforce application policy, for users inside the operator network.

2.4.1.8 Authentication Server Function (AUSF)

: As the name implies, this is an authentication server. AUSF contains mainly the EAP authentication server functionality.

2.5 Related Work

In this section, we are using different design terms like state synchronization and state partitioning. We use **state synchronization** when MME replicates states to its replica (SCALE [?], DMME [?]) or stores states to some persistent data storage (Premsankar et al.[?]).
State partitioning is used when multiple UE connections are distributed between instances of MME. Using some load balancing technique, state partitioning can be done. Premsankar et al.[?] used consistent hashing to distribute UEs across multiple MMEs.

SCALE [?] implement distributed EPC by using multiple MME replicas (state synchronization). MLB (MME LoadBalancer) distributes the UE requests to MMPs (MME processing entity). SCALE uses two instances of MME. As each UE can be mapped to any MME, when the request is processed, MMP chooses to replicate the state to the other MMP. For efficient load balancing consistent hashing mechanism is used (state partitioning).
Premsankar et al.[?] implement a 3-tier architecture: frontend, worker threads, and data storage. FE is a proxy which employs a round robin method to distribute traffic. To make the design stateless, UE context is saved to data storage at the end of a call flow (state synchronization), thereby reducing the latency but decreasing resilience. Also, the extra frontend component increases the overall latency.

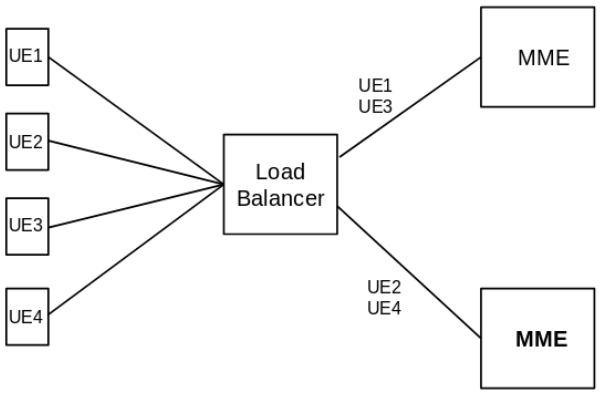


Figure 2.6: State partitioning

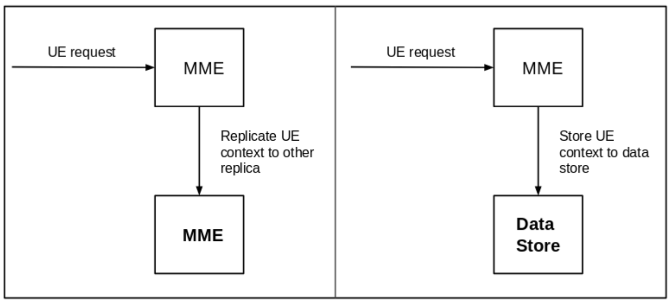

UE request
MME
Replicate UE context to other replica
MME
UE request
MME
Store UE context to data store
Data Store

Figure 2.7: State Synchronisation

The main objective of DMME [?] is to divide the job of processing control plane among multiple servers(state synchronization). The design has multiple DMME replicas for state replication.
Our design has similarities with the above implementations as we have used persistent data storage same as Premsankar et al.[?] for **state synchronisation** and for **state partitioning** we are using consistent hashing [?]. For reliability both SCALE [?] and DMME [?] use multiple replicas of MME. But they do not deal with the case when the active MME replicas fail. As we are using persistent data storage event if one of the MME fails, the UE states will not be lost. Premsankar et al.[?] store the state after the end of the call but they did not consider to store the states in between the attach call because the failure of MME in between attach can cause inconsistencies. Also both Premsankar et al.[?] and DMME [?] has not implemented with high loads. In our design, we parallelly send many UE request and measure latency and throughput. As attach procedure itself has multiple steps, we will explore that in-between attach procedure itself, how many times we can store the state in data storage so that lost of UE states is minimum. Also, all the above design only focus on scaling the MME but as MME replicas increase, SGW/PGW could be the bottleneck.

2.6 Contribution

We have an open source implementation of NFV based LTE-EPC core architecture [?]. We contributed to make the interface between RAN and AMF (S1AP interface) standard complient. Also the core component of EPC which is MME, a server with blocking network I/O operations. We implemented the asynchronous epoll based MME server with non-blocking network I/O.

We present a design for distributed 4G LTE-EPC, state partition and state partition + state synchronization. Mobility Management Entity is the main EPC component which handles mobile network control traffic. It manages UE authentication, mobility, and communication to other EPC components. So we are using MME for scaleup and failure handling. As it handles data coming from many other entities, therefore chances of MME becoming a bottleneck are high.

To distribute the UE traffic among multiple MME replicas, load balancing logic is embedded in the RAN. In RAN, each worker thread behaves as a UE and main thread behaves as a load balancer to the MME. For state synchronization we have used Redis key-value datastore. Based on when to store context in datastore we further compared the designs.

We are able to scale when computing capacity of an MME is reached and in case of failure of one of the MMEs, we are able to divert all the connections to the active MME pool.

Evaluation of our system is on the control plane traffic. We calculated the time that EPC takes to stabilize in case of scale-up and failure. We applied different load balancing techniques and compared them with respect to the number of UEs migrated in case of scale-up.

We have a running project on 5G testbed at IIT Bombay. We contributed in that project to make 5G testbed scalable using state partition + state synchronization. For state partitioning we used consistent hashing. We implemented a network function called as UDSF(Unstructured Data Storage network function) to store the user context. We made changes in the core network component AMF(Access and Mobility Management Function) to make it scalable. Evaluation of 5G core is on the control plane traffic. We measured throughput and latency to observe the effect of UDSF. Also measured the change in throughput while scaling.

Chapter 3

Design and Implementation of distributed and fault-tolerant 4G-LTE EPC

We had a monolithic NFV based LTE-EPC [?] design by Sadagopan N S. This design is based on NFV where each component of EPC behaves as a Virtual Network Function. In this design for feasibility, UEs and ENodeB combined to as a single entity called RAN. Other components are the same as specified by 3GPP standard. As monolithic design does not provide scalability and reliability, we choose this design to make it distributed and fault-tolerant. In this chapter, the overall design of the system, scaling and failure handling with/without using a persistent data storage is explained.

3.1 Overall Design of the System

RAN is designed in master-worker thread paradigm. We started with a single MME instance and as soon as this replica is fully utilized, the new replica is added to the pool of active MMEs. Each MME connects to RAN through a single TCP link. In the following sections, we describe the design of each component.

3.1.1 RAN design and Implementation

RAN consists of multiple worker threads and the main thread. A worker thread behaves as a User Equipment and sends data to the main thread. Domain sockets have been used to communicate between UE threads and main thread because of their bidirectional nature. Unix domain sockets are a method to communicate between multiple

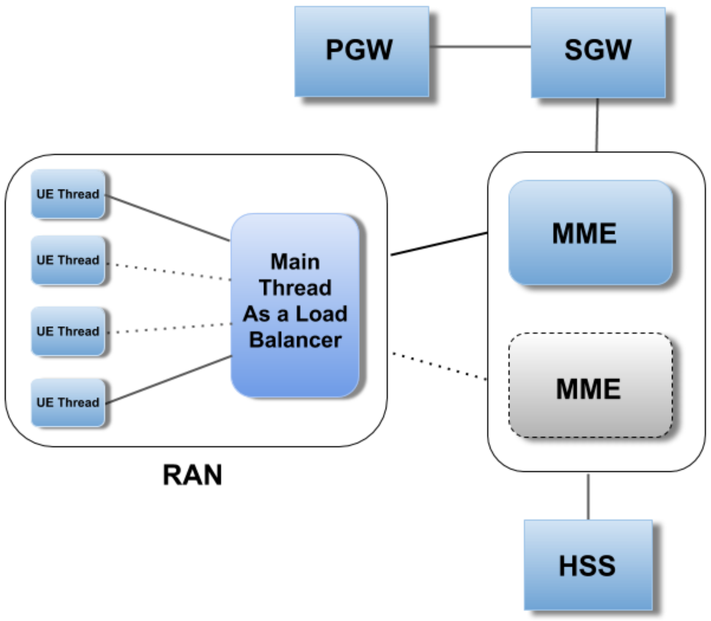


Figure 3.1: Overall Design of System

processes. These sockets may be created as a byte stream or a datagram sequence same as the UDP/TCP protocols. By using AF_UNIX, we created domain sockets (blocking) at each worker thread. As in TCP/UDP protocol sockets are bind to the IP address and port number, domain sockets are bind differently i.e. to a unique file path. This socket binding takes place on both sides: in the main thread and in multiple worker threads. Each worker thread sends an attach-detach request to the main thread over these Unix sockets. Worker threads are working in a closed loop environment. All threads simultaneously send a request and before sending again wait for a definite amount of time. Main thread works as a load balancer and diverts packets coming from UEs to the MME pool. It works in an event-driven non-blocking environment, containing an epoll loop which listens to the incoming events. Main thread also contains a mapping of UE_ID to domain socket. When a packet is received from MME, main thread extracts domain socket file descriptor corresponding to its UE_ID and sends the packet to that socket.

3.1.2 A step towards standard complaint EPC

3GPP specifications define the communication protocol between E-UTRAN and EPC. ENodeB to MME interface consists of IP, SCTP, and S1AP. S1AP is the signaling mechanism between E-UTRAN and EPC that consists of functions such as bearer management, initial context transfer function, reset functionality, NAS function, etc. According to 3GPP, the communication between ENodeB and MME is on SCTP protocol as a single channel, but our prototype had multiple TCP connections from EnodeB to MME server. To make our LTE-EPC close to standard we made a single TCP link between RAN and MME. As each worker thread at RAN sends data to the main thread through tits respective domain socket, main thread writes all the incoming data to a single channel. Internally, packets combine at the TCP level and reach the other end as a single packet to application. The other end reads data from the buffer, it contains multiple packets corresponding to each UE.

3.1.3 Load Balancing at RAN

Load balancing is one of the important tasks in distributed systems. Distribute the traffic among multiple servers is the main task of a load-balancer. We used a layer-4 load balancer (embedded in RAN itself) which has visibility on some of the network information such as IP address, TCP port number, the protocol used, etc. The load balancer uses some of this information to distribute data among multiple back-end servers.
Our load balancer maintains a map of UE to MME connection by which it can determine which UE will go to which MME server. Requests related to one UE from registration to de-registration should go to the same MME server until there is some scale-up event occurred. Load balancer continuously checks for the hotspot, changes in traffic. Based on some parameters, it should generate some scaling event and redistribute the UE connections. One of the other tasks of the load balancer is to re-distribute the connections with minimum interruption. We have used simple hashing and consistent hashing for state partitioning between multiple MME replicas. Algorithms like consistent hashing provide the least disruption. Here simple hashing mechanism is explained.

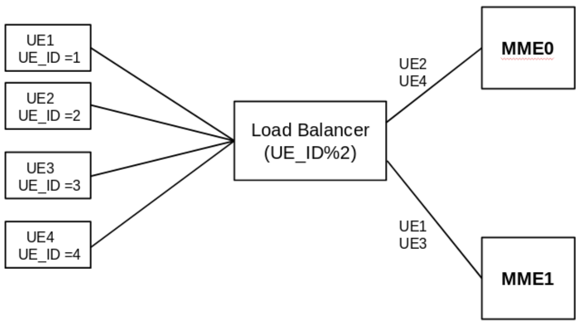


Figure 3.2: Simple Hashing

By using simple hashing mechanism UE connections are equally distributed between MME replicas. If there are only 2 MME replicas and we need to distribute the UE connections between these two, we use modulo based distribution. With each UE socket FD, we calculate a number (socket FD%2) and if this number is 0 then the UE connects to MME1 otherwise MME2. There are other load balancing techniques available like consistent hashing, round-robin distribution, etc. which we choose based on the application requirement and performance. We implemented both simple hashing and consistent hashing. Consistent hashing is explained in the later part of the report.

3.2 Scale-up and failure handling without persistent data storage

In this section, scaling and failure handling without any data synchronization is explained. We just used load balancer to distribute the UEs among multiple MME replicas. We are able to scale MME when it gets overloaded and if one MME fails all connections are diverted to the active set of MMEs. All the redirected UEs will start from initial attach request after scaling because the new MME will not have any prior knowledge about UEs. In case of failures also , UEs have to restart from initial attach request when one of MME fails.

3.2.1 Scale-Up

In this section, we explained scaling of MME when it becomes a bottleneck. RAN contains IP and the port number corresponding to the MME replicas. We always start with a single MME and RAN continuously sends it data. By continuously increasing traffic, latency increases and when latency exceeds the threshold, scaleup function is triggered. Each UE is mapped to a particular instance of MME in the main thread of RAN. When the scale-up procedure is called, some of the UEs have to be transferred to the new MME. For the distribution of UEs, we used layer-4 load balancer at RAN. After scaleup, the UEs with in-between attach request is still forwarded to old MME since the new MME will not have the necessary UE context to complete the ongoing request.

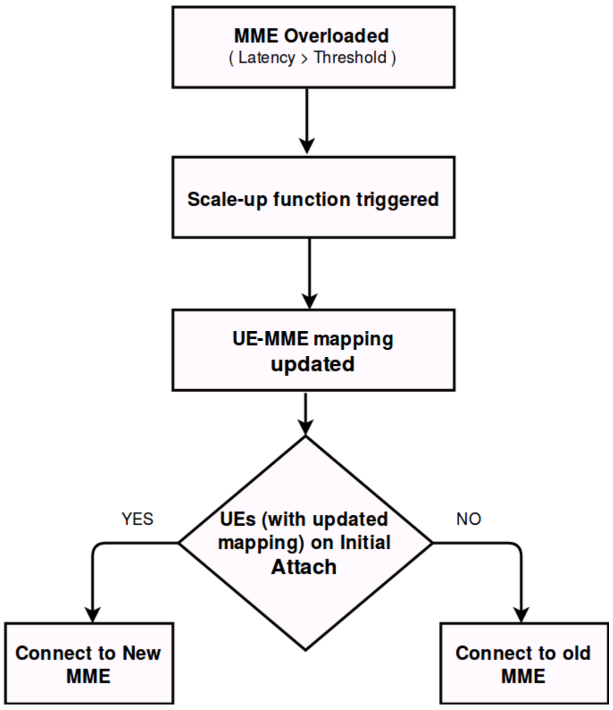


Figure 3.3: Scale-up

To simulate the scale-up, we continuously increase the traffic on MME. When Latency exceeds some threashold, new MME replica spawned and load balancer redistribute the UEs among both the replicas. After some time the system stabilizes.

3.2.2 Failure Handling

We handle the failure of MME by diverting connections to the active pool of MMEs. As soon as one the MMEs from active pool fails, RAN (Load Balancer) detects this failure and sends a reset request to all the worker threads which are connected to the failed MME. If UE-MME TCP connection is broken, RAN consider it as a machine failure(MME server). It also redistributes the UE connections to the remaining MME replicas.
The Load balancer distributes UE connections to the active set of MMEs by changing the mappings. Now all the UE threads restart their attach procedure and send a request to one of the MMEs of the active pool. Thus, we are able to handle the failure of MME by sending reset request to all the users. To explain failure handling more clearly we take an example. Suppose 2 UE threads are in between attach procedure (one at security setup and other at location update) and one of the MME fails, how our design handles this case, is explained in the diagram below.

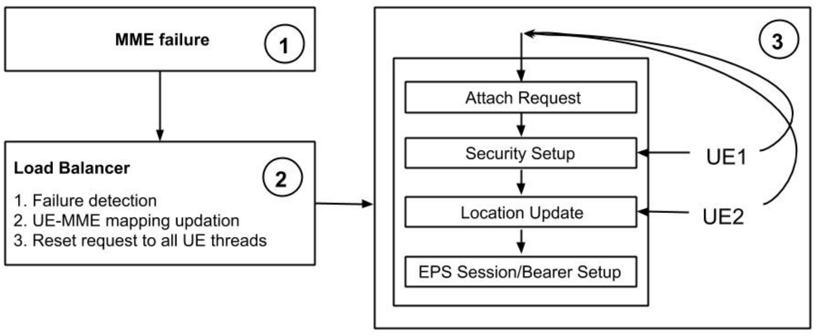


Figure 3.4: Failure Handling

To simulate the failure in our design, we stopped one of the MME processes. Load-balancer considers it as an MME failure and behaves as explained in the above diagram.

3.3 Scale-up and failure handling with persistent data storage

In scaling, without data store, we need to check if the UE is in initial attach or not. In case of failure handling also, UE threads need to resend the attach request. If persistent data storage is used then this problem can be resolved.

We extend our design to support both state replication and state synchronization. We used Redis key-value data storage based on analysis done by Dave J. [?] to save UE context.

3.3.1 Data synchronization at Redis

Redis is an open source (BSD licensed), in-memory data structure store, used as a database, cache and message broker. In our design, Redis-cluster is being used. Redis-cluster is a better choice in terms of distributed systems where we always deal with load imbalance and failures. Redis-cluster provides the users with a way in which data is automatically sharded among multiple Redis nodes. Also, it is able to work continuously in case of failures of some of the Redis nodes. We have maintained a cluster of size 12 for our design.
Other than Redis data-store, there is another entity which is a Redis client. Users communicate with Redis-client to store and retrieve their data from Redis. we have used **Hiredis-VIP**: a C client library for the Redis database. It also supports Redis-cluster.

3.3.2 Design and Implementation of Redis client: Hiredis-VIP

We have used asynchronous event-driven Hiredis API to communicate with both MME and Redis. The event-driven mechanism is based on epoll. Hiredis sends connection request to Redis using a function call **redisClusterAsync-Connect()**. After connection establishment, Hiredis waits in epoll loop for the events coming from MME.

Figure 3.5 explains the overall design, how MME, RAN, Redis, Hiredis-VIP are connected with each other.

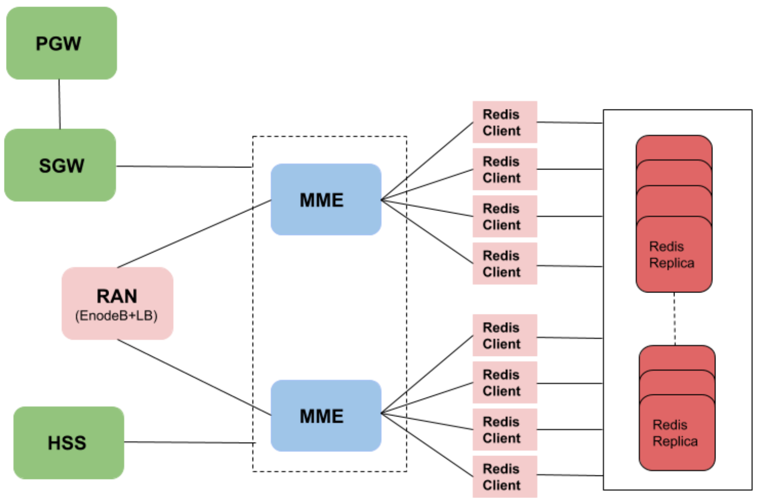


Figure 3.5: System design with state synchronization

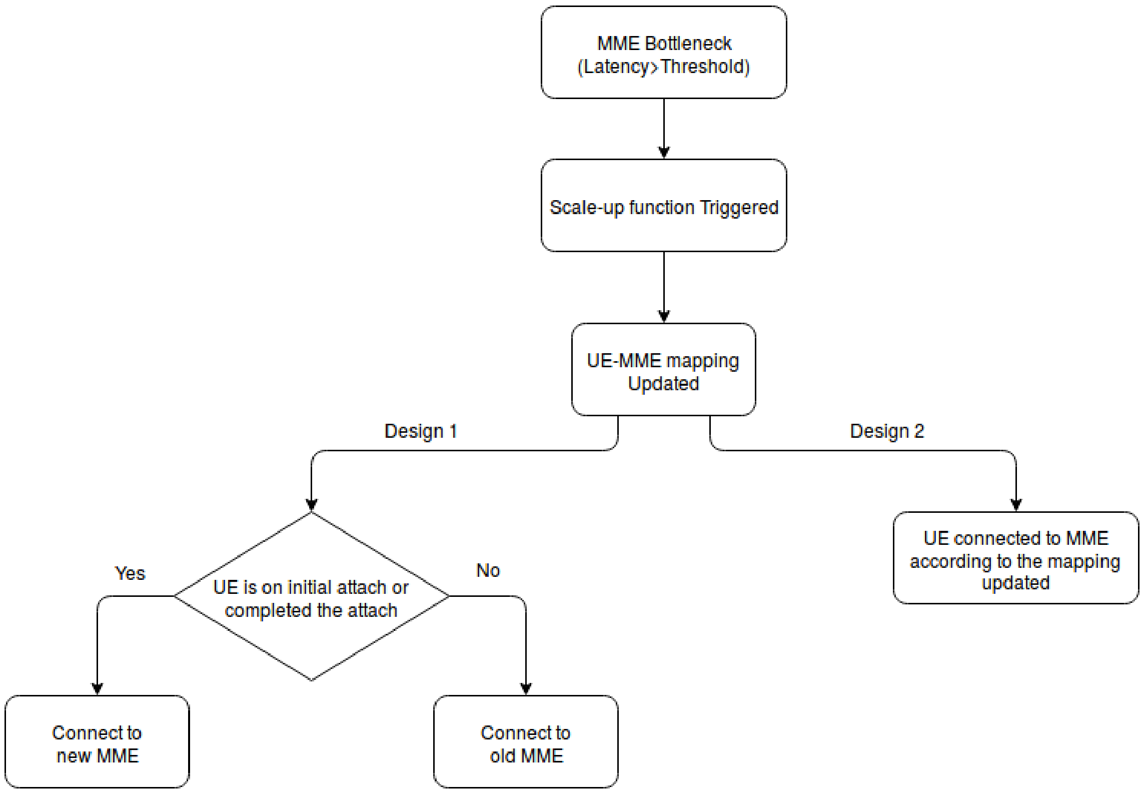
Whenever MME sends some request to Redis, it first reaches to Hiredis client and an epoll event is triggered.

Based on the MME request, a callback is also registered. Hiredis send the MME request to Redis by function call **redisClusterAsyncCommand()**. Reply from the Redis is trapped in the callback function registered previously. In our design, we use 4 hiredis client with one MME server so that hiredis does not become a bottleneck. MME opens 4 TCP connections to different hiredis clients. MME use all the four connections in round robin sequence.

3.3.3 Scaling MME

The scaling procedure will be the same as explained in above section. In place of locally storing information regarding each user, MME stores the user context at redis-cluster. Based on when to store the user context in Redis, we chose 2 design options:

- Scaling MME with synchronizing UE context after whole attach procedure (before detach).
- Scaling MME with synchronizing UE context after every sub procedure.



MME Bottleneck (Latency>Threshold)
Scale-up function Triggered
UE-MME mapping Updated
Design 1
Design 2
UE is on initial attach or completed the attach
Yes
No
Connect to new MME
Connect to old MME
UE connected to MME according to the mapping updated

Figure 3.6: Scale-up

In the first design option, MME store UE-context after the registration is complete. And when RAN sends the detach request, before executing it MME retrieve the user context from the data store. By this mechanism, user can attach to one MME but at the time of detach, it can send the request to other MME replica.

The second design option is pure stateless. After every sub-procedure, MME store the UE context to Redis. When RAN send some request to MME, MME first retrieves the context from Redis then only process the request. This design option is most suitable for both scaling and failure handling but the drawback of this design is the overhead of Redis. Due to frequent set-get request to Redis, the throughput of MME reduced and latency increased. By using Redis to store user-context, time to stabilize the system during scale-up will be less. This is because user threads in RAN will wait less amount of time (as compared to the previous design(without Datastore)) before migrating to the new MME replica.

3.3.4 Failure Handling

Same as scaling, failure handling is also easy if we store context at Redis-cluster. In case of failure of one MME, the new MME replica take the context from data store and processes the ue requests. If the UE is in-between of some request, it has to re-initiate the request after redirection. As explained above, failure handling also has 2 design options.

- Failure Handling while storing UE context after attach procedure
- Failure Handling while storing UE context after every sub-procedure

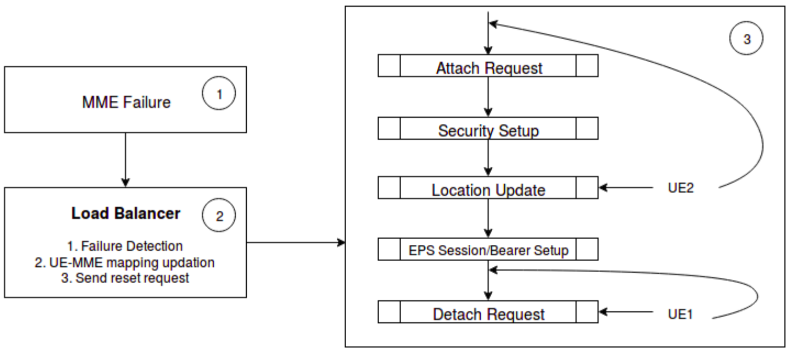


Figure 3.7: Design1: Failure Handling

In first design when one of the two MME fails, associated UEs which are in-between the attach procedure have to restart the attach after redirection to the new MME. All UEs who have completed their attach, they can directly migrate to new MME replica because the new MME can retrieve their context from datastore and UEs just need to resend the current request to new MME. In the second design when MME fails, associated UEs can directly attach

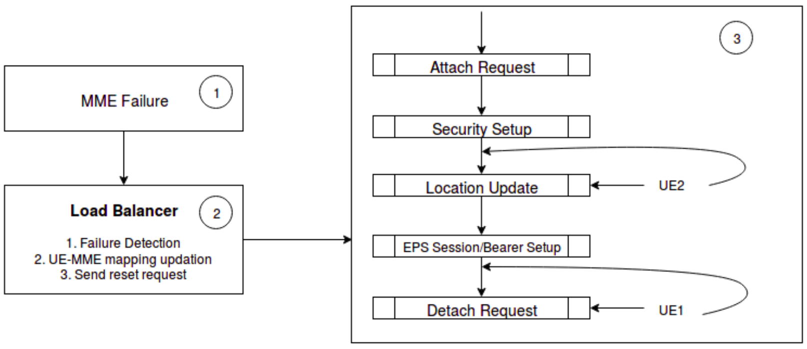


Figure 3.8: Design2: Failure Handling

to new MME replica because after every subprocedure user context is being stored at Redis. So UE just need to re-send the current request to the new MME. Procedure for failure handling is already explained. Using redis for state synchronization, UEs do not need to re-start from attach request again. Below diagram explains how in both the designs UEs will react in case of failure.

In Figure 3.7 where we are storing context after whole procedure, UE1 can directly go to new MME and restart the current detach request. UE2 is at location update, has to restart from initial attach request because the new MME will not have the context for it.

In Figure 3.8 where we are storing context after every sub-procedure, both the UEs can directly go to new MME and restart the current request.

Chapter 4

Design and Implementation of Scalable 5G Architecture

We already have a 5G-core testbed, running as a project at IIT Bombay. As the title suggests, we are making this 5G-testbed distributed so that it becomes scalable and handle failures. We are storing UE-context at a new NF called UDSF which was previously stored at AMF. By storing context at UDSF, it will be easy to apply scaling and failure handling logic at AMF. As AMF is the main entity in the 5G core network, It is obvious that after a certain load, AMF will become a hotspot in the network. To remove that hotspot, it is essential to divert some traffic to other AMF replica.

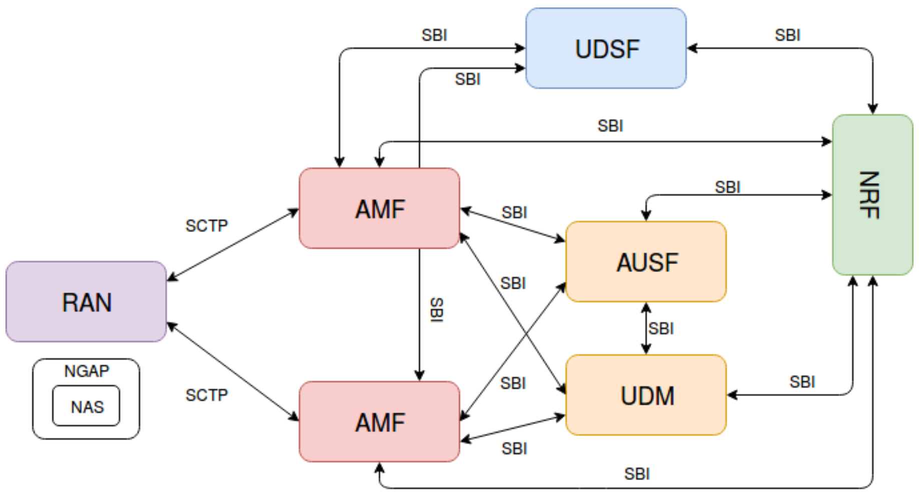


Figure 4.1: Distributed 5G Core Network

We have an entity called RAN which is emulating user-equipments and 5G RAN. RAN also contain a load balancing logic which is required to distribute UEs to AMF replicas. Previous AMF design had no way to store context at UDSF, we added logic in AMF to store/retrieve context to/from UDSF. Below is the design of distributed 5G core network and RAN. We added a new component called UDSF. The network also has AUSF, UDM, NRF and multiple AMF replica.

4.0.1 RAN design and Implementation

Same as 4G, this design also contains RAN. Currently, we do not have physical UEs so we have simulated UE processing in the RAN itself. Every UE-thread follow a fixed set of procedures, starting from NGsetUp to Registration and at the end de-registration. In a while loop, every thread continuously sends/receive packets from MME. There is a single SCTP connection between RAN and AMF. Every thread sends the packet to this single SCTP connection. AMF is an SCTP server for RAN.

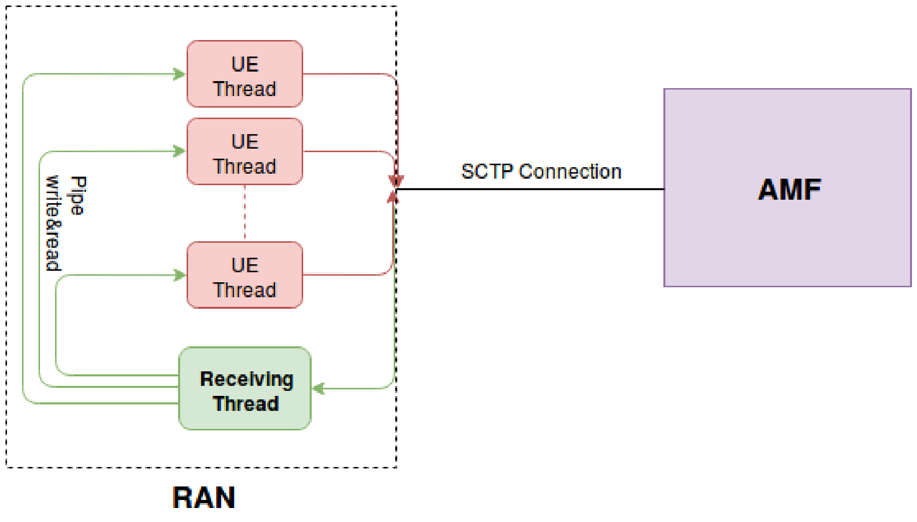


Figure 4.2: RAN Design

Whenever AMF send some packet to RAN, it is received by another thread(receiving thread) which is not among UE threads. Communication between receiving thread and all UE-threads is through pipes. Based on ranUeNgapId (this ID is allocated by the RAN and used as a unique identification for that UE in the RAN) receiving threads direct the packet to the particular UE. Receiving thread writes to one of the pipes, which is read at the other end by user-thread.
At RAN, every packet is encoded 2 times before sending to AMF. Every UE apply NAS header and NGAP header to the packet before sending to AMF. AMF also apply NGAP header(encoding) before sending packets to RAN, which is removed(decoding) by the receiving thread before redirecting to the particular UE.

This call flow between RAN and AMF maintains two types of the message stream. The first pair of messages which is setup request/response is NON_UE_MESSAGE_STREAM because it uses non-UE associated signaling. All other messages are of UE_MESSAGE_STREAM type.

4.0.1.1 Load Balancing at RAN

As we are talking about scaling/failure of AMF, there will be multiple AMF replica present. To deal with multiple AMF replicas, some logic has to be added in RAN. Also, some algorithm has to applied at RAN to distribute the UEs between AMF replicas. We did all experiments using 2 AMF replica.
In starting, RAN maintain NGsetup with both the AMF servers. It opens two sctp connections for two AMF servers respectively. After that, both the AMF replica are added to the consistent hashing algorithm to know the UE to AMF mapping. To store the mapping, a local map(ue_to_amf) is also maintained at RAN. /Both the AMF servers are mapped to some integer using a hash function. Input to the hash function is (RAN to AMF replica socket file descriptor + replica number). To reduce the load imbalance, we can have virtual nodes corresponding to each replica. Currently, we are using 10 virtual nodes correspoing to each physical node. Now hash value corresponding to each UE is also calculated, we call the hash output as key. Each UE mapped to one AMF server whose key is less than the hash value of the server that we previously calculated. Now each UE is mapped to one of the two AMF. Every UE thread before sending the packet, check in this map for corresponding AMF replica.

4.0.2 UDSF design and Implementation

The 5G system architecture allows any NF to store/retrieve unstructured data to/from UDSF. UDSF is a network function called Unstructured data storage function. In our design, we use UDSF to store/retrieve user contexts of RAN, which were previously stored at AMF(locally). 3GPP specifies any NF-UDSF interface as N18/Nudsf (Service based interface exhibited by UDSF). In our design, UDSF internally contains redis-cluster same as 4G. Some of the services that UDSF can provide insert, query, delete, update, etc. We have not implemented all the services in the current design.
Like any other NF, the first step of UDSF also is to register its profile at NRF. NFProfile consists of nfInstanceId, nfType, nfStatus, ip, plmn, amf-region-id, amf-set-id, etc. UDSF sends its profile using the PUT method. Now UDSF entered into register state. At UDSF, http_listener handler continously listening to all incoming http-request. Listener handler call appropriate methods(handle_get, handle_post, handle_put, handle_delete) based on the http request and a Casablanca thread is instantiated to handle that method.
When AMF send a request to store/query user context, handle_post() method is called at UDSF. Based on the type of HTTP request UDSF calls different Redis functions. If the request type is nudsf-insert then store() function is called. When the request type is nudsf-query then query() function is called. As there are multiple Casablanca threads requesting for set/get context from Redis. It will be error-prone because Redis is not thread-safe. To handle

this situation we maintained multiple redis context and stored them in a queue. When some thread wants to perform some operation on redis, a context is dequeued to perform that operation. After the operation is completed, the context is stored back to the queue. Below is the diagram explaining the working of UDSF and interaction with AMF, NRF.

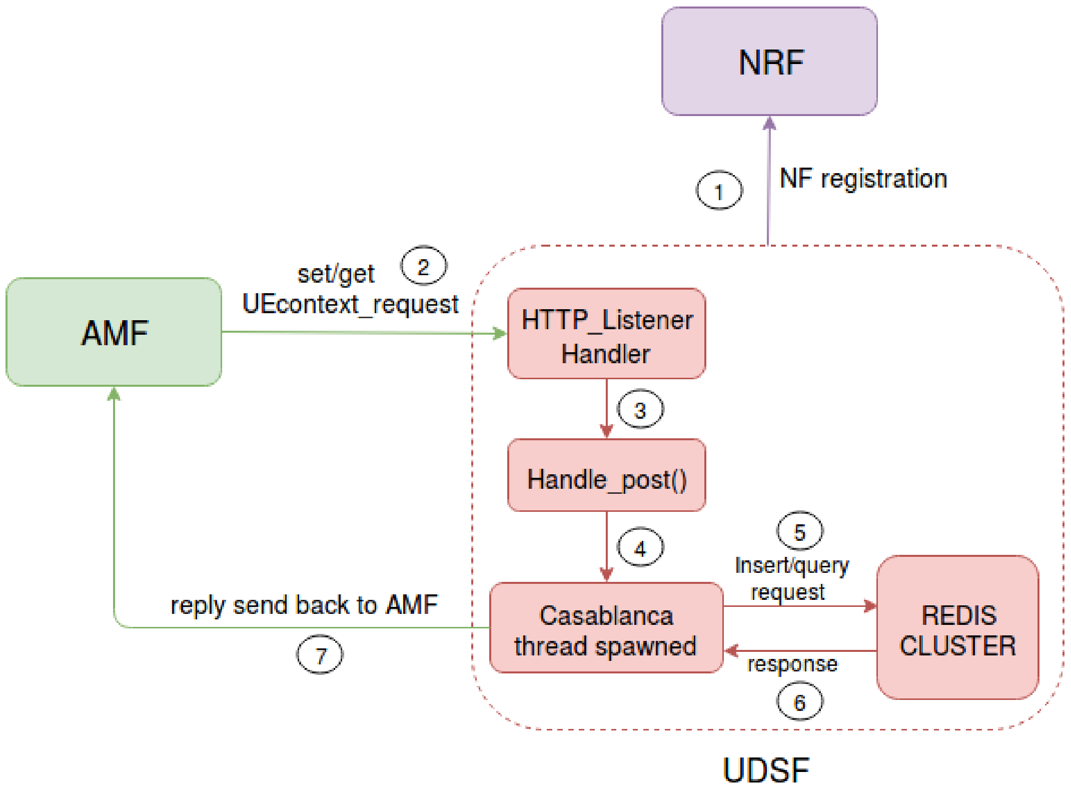


Figure 4.3: UDSF design and implementation

4.0.3 AMF design and Implementation

AMF is the core entity of 5G core network. It interacts with NGRAN and User Contexts using N1 and N2 interface respectively. Inside the core network, AMF communicates with other 5G NFs using service-based interface. AMF supports registration management, connection management, mobility management, access authentication and authorization, security context management, etc. AMF behaves as mobile management entity from LTE-EPC architecture.

4.0.3.1 Basic architecture of AMF

AMF communicate with NGRAN using NGAP protocol, with UE using the NAS protocol. AMF uses REST API / HTTP protocol to communicate with other network function in the 5G-core like AUSF, UDM, UDSF, NRF, etc.

AMF behaves as an SCTP server for RAN and web server for other 5G-core components. All UEs send packets to RAN and RAN sends packets to a single SCTP link connected to AMF. AMF works asynchronously and contains a single event based loop.

Front-end implementation of AMF is explained in the below diagram, which activates when RAN sends UE packets.

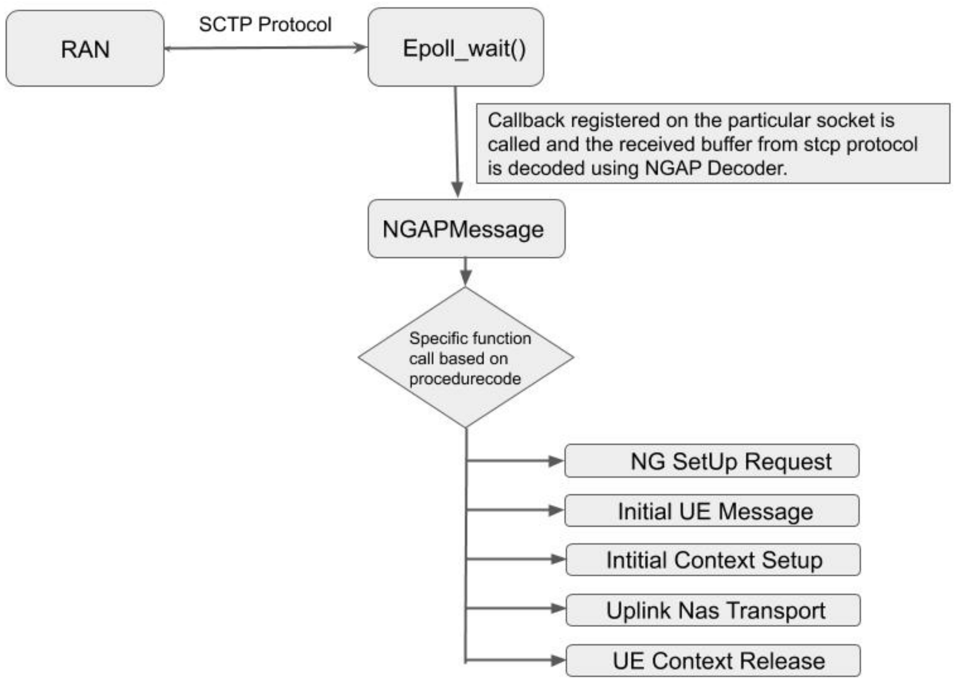

RAN
SCTP Protocol
Epoll_wait()
Callback registered on the particular socket is called and the received buffer from stcp protocol is decoded using NGAP Decoder.
NGAPMessage
Specific function call based on procedurecode
NG SetUp Request
Initial UE Message
Intitial Context Setup
Uplink Nas Transport
UE Context Release

Figure 4.4: Front-end AMF

As shown in the figure, RAN sends NAS and NGAP encoded packets to AMF. As soon as the packets reach AMF, an event is triggered using EPOLL API. For that particular event, a callback function is registered at AMF. When the event triggered, functions are called: using NGAP decoder, packet buffer is decoded and NGAP message is received. Then, based on the procedure code, a specific function is called.

4.0.3.2 AMF SBI Interactions

AMF uses Casablanca framework to communicate with other network functions over HTTP. The framework contains multiple Casablanca thread pool (by-default 40). An http-request is send by AMF(http-client) to other NFs using request() method. Now the function returns a task of http-response. The returned task is triggered when the reply from the destination NF comes. A'.then() function is associated with the task object. Reply from the NFs is captured in .then() function which is executed by one of the threads in the thread pool. The threads receive the response from

NF and send the updated packet to RAN.

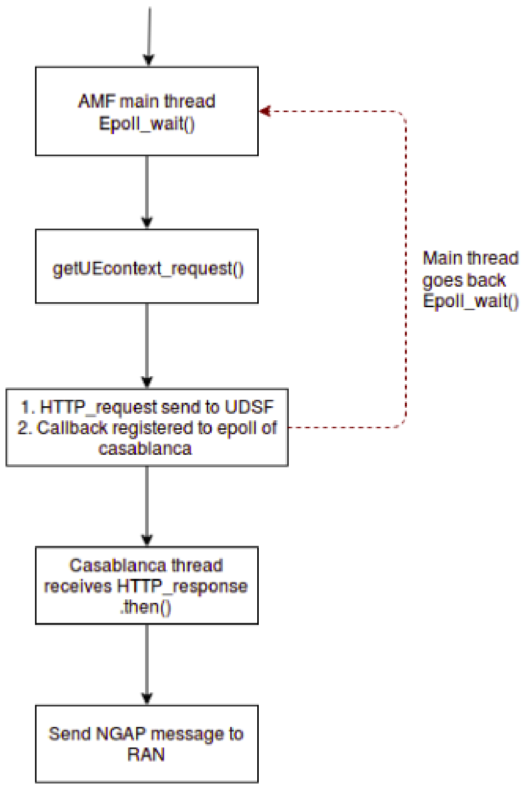

AMF main thread
Epoll_wait()
getUEcontext_request()
1. HTTP_request send to UDSF
2. Callback registered to epoll of casablanca
Casablanca thread receives HTTP_response .then()
Send NGAP message to RAN
Main thread goes back Epoll_wait()

Figure 4.5: AMF SBI Interaction with UDSF

4.0.3.3 NRF Registration and NF Discovery

When the AMF network function starts, it registers NFProfile to NRF. NFProfile consists of nfInstanceId, nfType, nfStatus, ip, plmn, amf-region-id, amf-set-id, etc. AMF sends its profile using the PUT method. In response, NRF sends heartbeat messages to AMF. When AMF receives a heartbeat message, it sends keepalive messages to NRF. Whenever AMF needs to communicate with any other NF in the network, the URL of that server will be required. AMF has a cache containing NFprofile of all NFs, which is periodically updated. Like AMF, all other NFs also registers their profile at NRF. When AMF wants to use the services provided by other NFs, first it checks in its local cache. “DiscoverNFINstance()” is a function which is used to find the profile of a particular NF. This procedure requires NFname and NFService-type. If not found in local cache then an HTTP request is sent to NRF but before that, the remaining work has to be stored somewhere because the response from NRF reaches asynchronously. For that AMF has a concept of the work queue, in which the pending job is enqueued. When the NRF sends the information related to the desired NF, that job is dequeued from the work-queue and remaining work is completed.

4.0.3.4 Changes in AMF to make it scalable

Same as 4G, here we are trying to make 5G-core scalable by using multiple AMF replicas. When one AMF replica becomes the bottleneck, some of the connections redirected from one AMF replica to newly spawned AMF. The new replica should know the context(user information) of all the redirected UEs beforehand. we used UDSF to store the context of all the user equipment. First, we will explain the registration call flow between RAN and AMF.

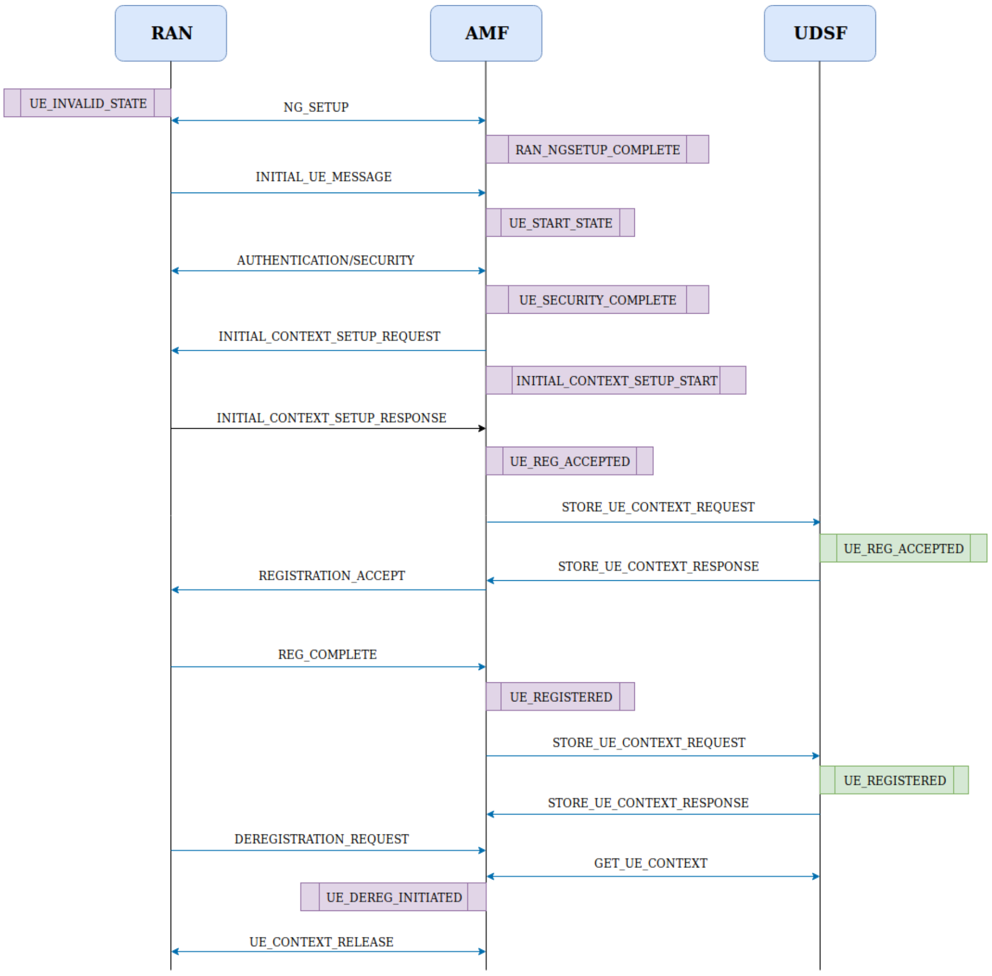


Figure 4.6: Call flow between RAN and AMF

Currently, we are storing user context before sending UERegistrationAccept message to RAN and after UERegistration complete phase. Before deregistration, AMF retrieves the context from UDSF. AMF retrieves context from

UDSF too when it is newly spawned due to scaling process.

To store the context, AMF calls setUEcontext_request() procedure. Until reply of set-context comes from UDSF, AMF can not send the data back to RAN. Therefore it enqueues the remaining job in a work queue with the job named as E_WORK_UDSF_INSERT. When the response from UDSF receives, AMF sends the reply packet to RAN. Figure 4.7 explains how setUEcontext_request() procedure called and context is stored at UDSF.

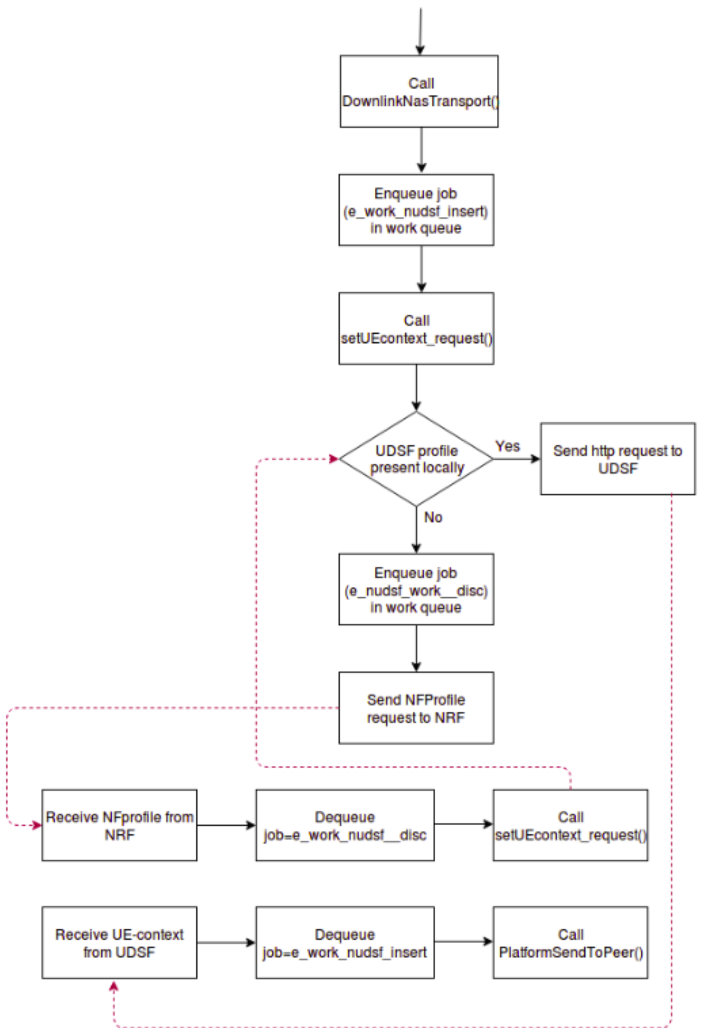

Call DownlinkNasTransport()
Enqueue job (e_work_nudsf_insert) in work queue
Call setUEcontext_request()
UDSF profile present locally
Yes
Send http request to UDSF
No
Enqueue job (e_nudsf_work__disc) in work queue
Send NFProfile request to NRF
Receive NFprofile from NRF
Dequeue job=e_work_nudsf__disc
Call setUEcontext_request()
Receive UE-context from UDSF
Dequeue job=e_work_nudsf_insert
Call PlatformSendToPeer()

Figure 4.7: Storing Context to UDSF

Before deregistration phase, AMF calls getUEcontext_request() function. Until the context for particular UE received by AMF, the remaining procedure is enqueued in work queue with a job named as E_WORK_UDSF_QUERY.

When UE-context from UDSF receives, the job is dequeued from the work queue. In both the functions before sending a request to UDSF, AMF first checks for the UDSF profile. If present then it takes from the local cache, otherwise send HTTP-request to NRF. Figure 4.8 explains how getUEcontext_request() procedure called and context is retrieved from UDSF.

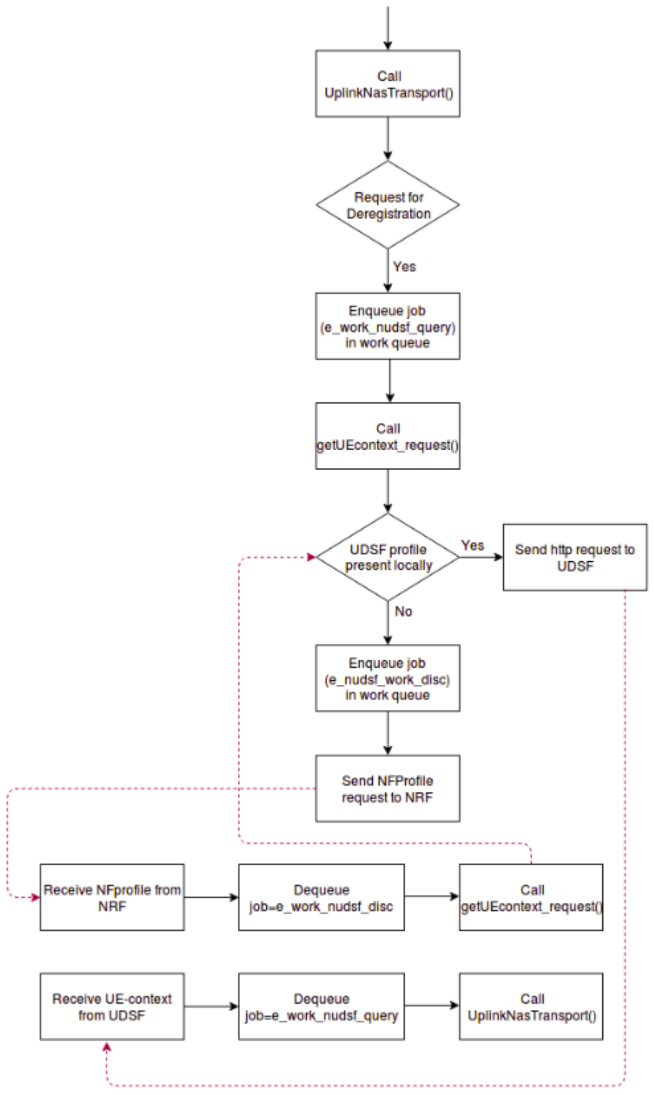

Call UplinkNasTransport()
Request for Deregistration
Yes
Enqueue job (e_work_nudsf_query) in work queue
Call getUEcontext_request()
UDSF profile present locally
Yes
Send http request to UDSF
No
Enqueue job (e_nudsf_work_disc) in work queue
Send NFProfile request to NRF
Receive NFprofile from NRF
Dequeue job=e_work_nudsf_disc
Call getUEcontext_request()
Receive UE-context from UDSF
Dequeue job=e_work_nudsf_query
Call UplinkNasTransport()

Figure 4.8: Retrieve Context from UDSF

Chapter 5

Evaluation

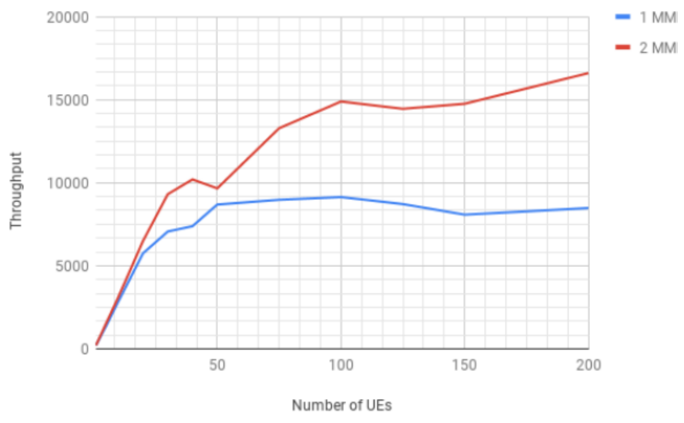
5.1 Experiments for 4G LTE-EPC

This section covers experimental results that we have taken in case of scaling and failure handling of 4G. We have taken experiments for 2 design options: state partition, state partition + state synchronization. In the below table our setup is explained, we placed MME, SGW, PGW on a virtual machine and RAN on the host machine.

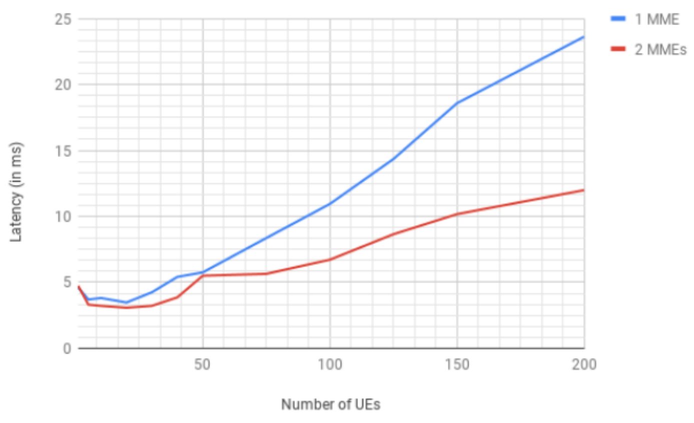
COMPONENT	CPU CORE	RAM	OPERATING SYSTEM
RAN	4	8GB	Ubuntu 14.04
MME	1	2GB	Ubuntu 14.04
HSS	1	2GB	Ubuntu 14.04
SGW	1	2GB	Ubuntu 14.04
PGW	1	2GB	Ubuntu 14.04
REDIS	1	8GB	Ubuntu 14.04

Table 5.1: Individual Component Specification

For each experiment, we run 1 thread of MME, 50 threads of HSS/SGW/PGW. We run all the experiments for control plane data traffic of EPC.



(a) Throughput



(b) Latency

Figure 5.1: Scaling from 1MME to 2MME

5.1.1 Experiments for state partition based LTE-EPC design

In this section, we have considered all the experiments where the load balancer is used to distribute the UEs across multiple MME replica. We examine throughput and latency for 1MME and 2MME. Later, we diverted some of the UEs to other MME replica to perform scaling also calculated the convergence time. To simulate the failure, we stopped one of the MME and re-distribute the UEs again to the active MME pool. In case of failure, recovery time is calculated.

5.1.1.1 Scaling from 1MME to 2MME

We calculated the average throughput of MME, the number of requests processed by MME in one second. Also calculated average latency which is time to complete a UE request(attach+detach). As we expect, throughput increased until saturation is reached and then become constant because MME is fully utilized. Latency increase as we increase the load at MME. We also experimented the same scenario with 2MME. All the UE are divided between both the MMEs. We saw an increment in throughput and less latency in case of 2MME. Thus, scalability is achieved. It is shown in the Figure: 4.6.

Next experiment is to scale 1MME to 2MME at run time when the MME becomes the bottleneck due to high traffic. To overload, the MME, RAN continuously sends attach/detach request concurrently for the constant number of UE threads. We measured throughput and latency while scaleup and calculated the time the system takes to stabilize. As worker threads are running in a closed loop, after completion of a UE-attach request all threads wait (sleep) for some amount of time and then again send an attach request. After each run sleep time is decreased for each thread. By continuously decreasing the sleeptime, frequency of sending request increases. The load is increased by decreasing the sleep time and when the latency of the overall system exceeds from the threshold, latency scaling is performed and convergence time of the system is measured. Experiments gave an indication of at what latency we should trigger the scaleup event.

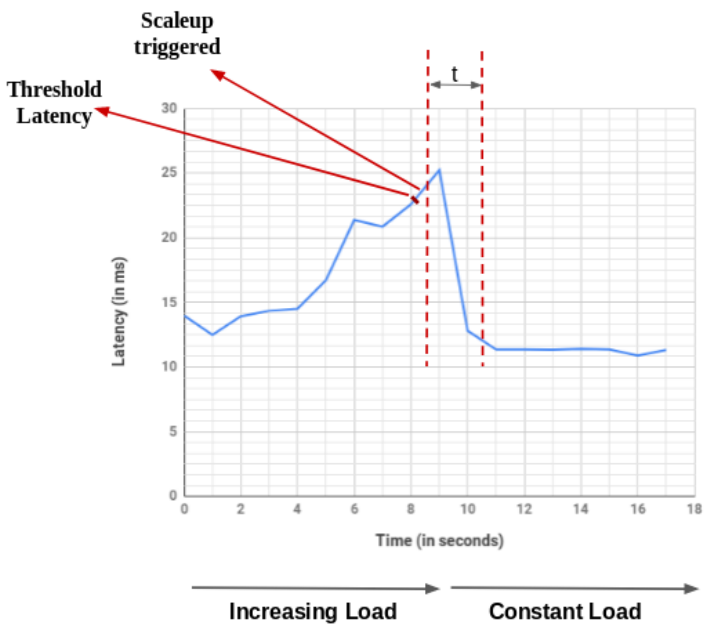


Figure 5.2: Latency while scaling

The total duration of our experiment is 18 seconds for the above graph (Figure 5.2) and we measured the latency every second. The x-axis in the above graph is the time in seconds and Y-axis is the measured latency (ms). As we can see in Figure 5.2 latency of the system increases as we increase the load and at threshold latency (22ms) scaleup procedure is called and after that, the latency becomes constant around 12ms. After scaleup, we kept the sleep-time constant resulting in constant load. We observed that system stabilizes from overloading situation within 11213us.

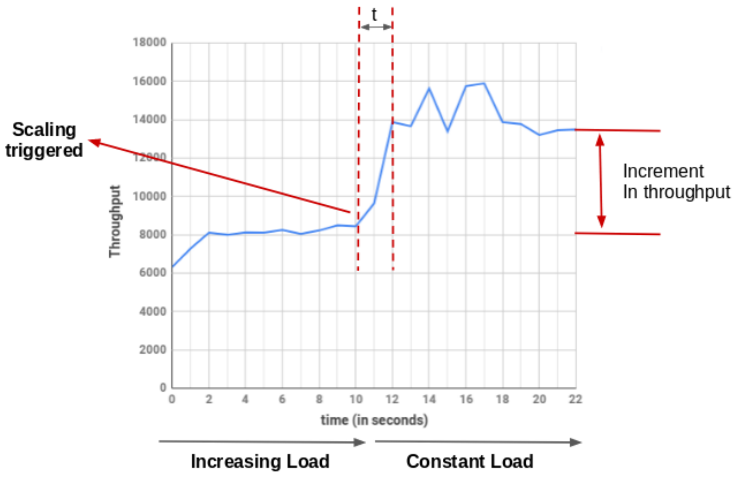


Figure 5.3: Throughput while scaling

The total duration of our experiment is 22 seconds for the above graph (Figure 5.3) and we measured the throughput(requests/sec) every second. The x-axis in the above graph is time in seconds and Y-axis is the measured throughput. As we increased the traffic, throughput increased till saturation and then became constant. As expected, the throughput approximately doubles (8000 to 14000) after scaleup.

5.1.1.2 Failure Recovery

In this experiment, we measure recovery time when one of the MME fails. 2 MMEs run after scaleup event and all the connections are divided between these two. To get results in case of failure, we stop the MME process in one of the VM. Now all connections are diverted to the active MME. Here we considered the time taken by all UEs to re-initialise the attach request. With a different number of UE threads, we measured this recovery time. As shown in the graph, when the number of UE threads is 200 and one MME fails, threads connected to failed MME collectively take around 0.004 sec to send their attach request to the active MME. As the number of UE threads increases at RAN, the recovery time of the system also increases. In the below graph X-axis is the number of UE threads running at a time, Y-axis is the recovery time. Each run of this experiment is of 300 seconds.

5.1.2 Experiments for state partition + state synchronization based LTE-EPC design

We have performed all the experiments explained above with state partition + state synchronization. To synchronize the state we use Redis key-value data store. We stored user context at Redis which was previously locally stored at AMF. We expect that due to state synchronization, convergence time and recovery time will reduce. As explained in chapter 3, we consider 2 design options. We perform experiments for both the designs which are explained in the

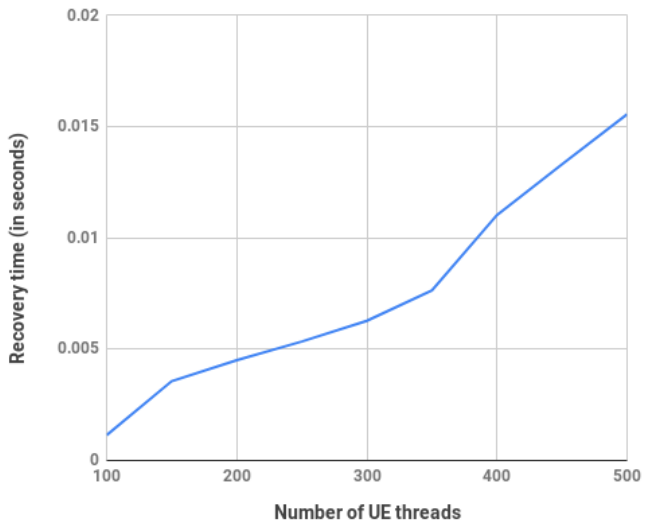


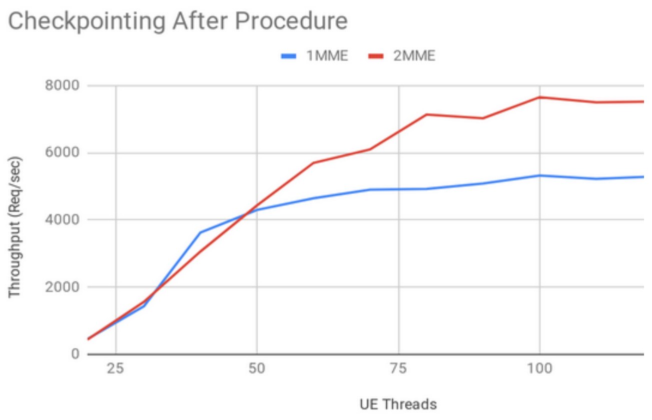
Figure 5.4: Recovery time while MME failure

below sections.

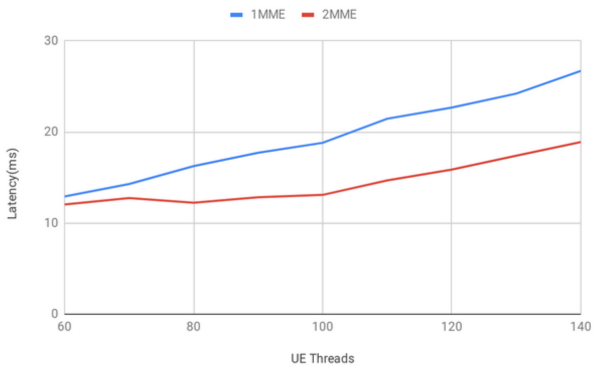
5.1.2.1 Scaling MME with synchronizing UE context after whole attach procedure

In this design, after the attach/before detaching procedure we store the UE context in the datastore. If the UE thread is in the starting of the attach procedure/before detaching sub-procedure and scaleup function called then those threads can directly migrate to the other MME replica because we are storing states of every UE before the detach sub-procedure. First, we analyzed latency and throughput in the case of 1 MME and 2 MME. We observed higher throughput and lower latency in case of 2 MME replica, as expected. The x-axis for both the graphs (Figure: 5.5a and Figure: 5.5b) is the number of UE threads at RAN and Y-axis is Throughput and Latency(ms) respectively.

To spawn new MME replica when current MME becomes a hotspot (Figure: 5.6a and Figure: 5.6b), we did the experiment for a constant load (300 threads) and the load on MME increased by decreasing the time between two attach requests in every thread. When the latency increases and exceeds the threshold, scaleup() function called and some of the connections are diverted to new MME replica. Convergence time for this design is 10211 us. Convergence Time is the time to stabilize the system after scaling procedure is called. In our design, as soon as scaleup() called, the time elapsed between redirected threads to get connected from one MME to other MME replica is the convergence time. For a particular load, we considered a threshold latency (maximum latency). After that latency, if the load increases then the current system latency will be more than the threshold and the scaling procedure will be called. By doing the scaling, latency decreased from 60ms to 40ms and throughput increased from 5000 to 8000 requests per second. For Figure: 5.6a and Figure: 5.6b, x-axis is time in seconds and Y-axis is Latency(ms) and Throughput

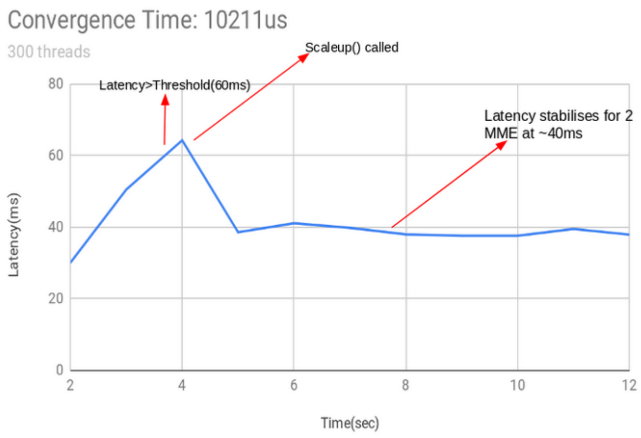


(a) Throughput

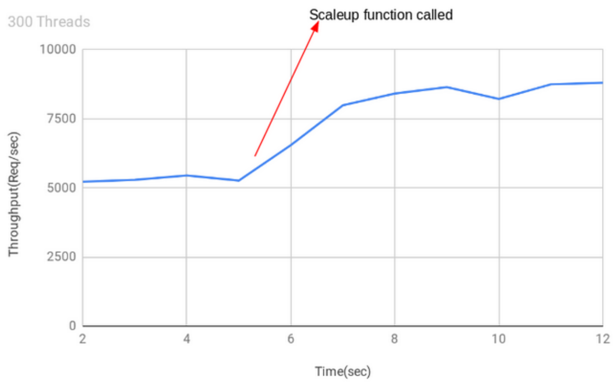


(b) Latency

Figure 5.5: Scaling MME with synchronizing UE context after whole attach procedure

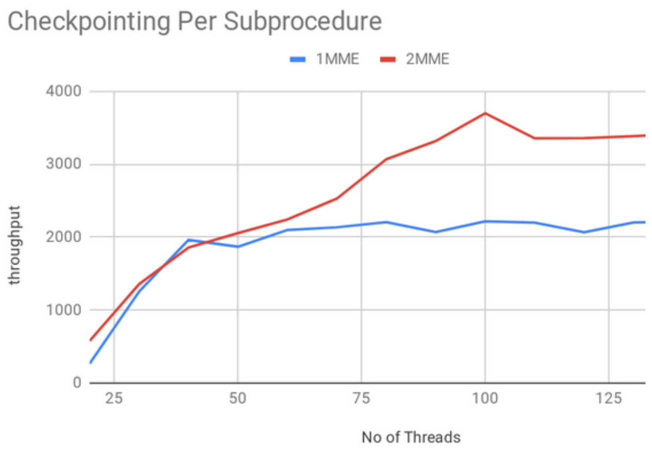


(a) Change in latency while Scaling

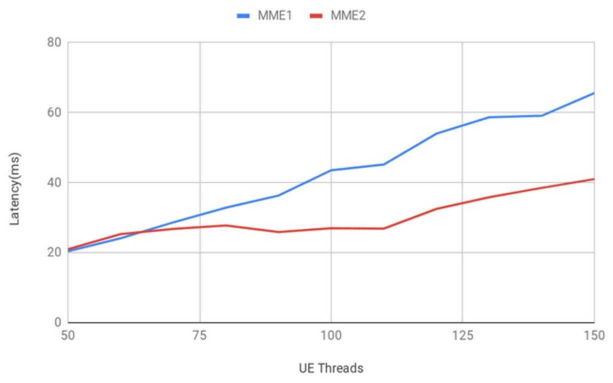


(b) Change in throughput while Scaling

Figure 5.6: Convergence Time



(a) Throughput



(b) Latency

Figure 5.7: Scaling MME with synchronizing UE context after every subprocedure

respectively.

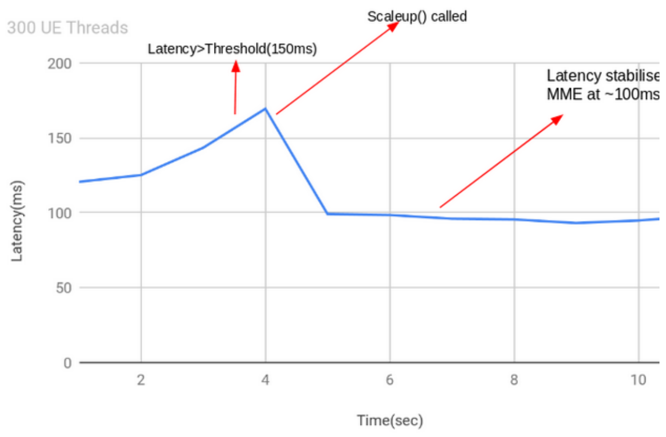
5.1.2.2 Scaling MME with synchronizing UE context after every subprocedure

In this design, we store the UE context after every sub-procedure in the datastore. When scaleup function called then all UE threads can directly migrate to the other MME replica because we are storing states of every UE before every sub-procedure. First, we analyzed latency and throughput in the case of 1 MME and 2 MME. For Figure 5.7a and Figure 5.7b the x-axis for both the graphs is the number of UE threads at RAN and Y-axis is Throughput and Latency(ms) respectively.

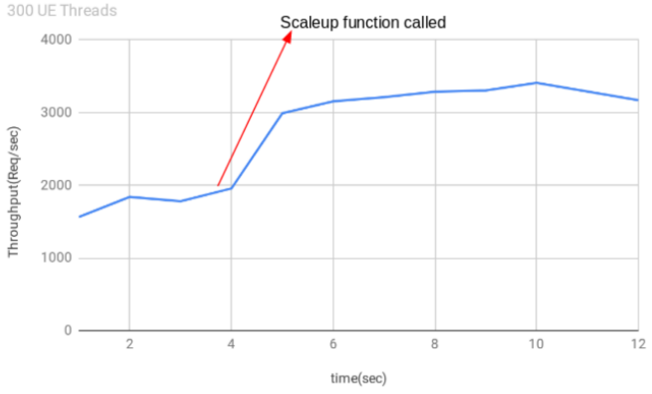
For Figures 5.8a and 5.8b, we did the experiment for a constant load (300 threads) and the load on MME increased by decreasing the time between two attach requests in every thread(at RAN). When the latency increases and exceeds the threshold, scaleup() function called and some of the connections are diverted to new MME replica. Convergence time for this design is 2660 us. Latency decreasing from 150ms to 100ms and throughput increased from 1800 to 3300 requests per second. The x-axis is the time in seconds and Y-axis is Latency(ms) and Throughput respectively.

5.1.2.3 Failure Handling with synchronizing UE context after complete procedure

In these experiments, we killed one MME from the active MME pools and observed change in throughput and latency. Fixed number of UEs send attach-detach requests continuously in loop. Experiment started with 2MME/3MME and throughput/latency measured every one second. After some time one MME process killed manually and throughput/latency measured again. We found some deviation in both throughput and latency. Throughput decreased and latency increased as expected. X axis for both Figures 5.9a and 5.9b is time in seconds and Y-axis is throughput



(a) Change in latency while Scaling



(b) Change in throughput while Scaling

Figure 5.8: Convergence time

and latency respectively.

5.1.2.4 Failure Handling with synchronizing UE context per-procedure

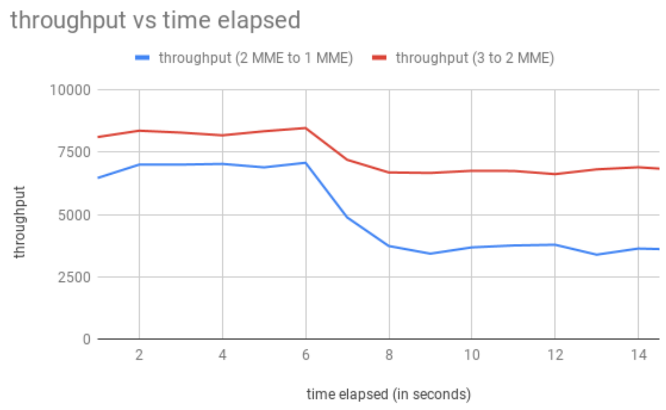
Instead of storing context after whole attach, here we are storing context after every per-procedure. We did the same experiment as above section for both throughput and latency. We found some deviation in both throughput and latency. Throughput is less and latency is high as compared to previous experiment due to datastore communication overhead. The deviation in this design is less as compared to previous design as we stored context frequently and UEs do not need to resend already completed requests.

For both the failure handling scenarios, we also measured total number of registration completed in the time intervel of 100ms. We performed this experiment with 4MMEs and 110 UEs. Initially UEs are divided among 3 MMEs only, 1 MME sits in ideal condition. UEs distributed among all the 3MME. Then we killed one MME and UEs again distributed among rest 3 MMEs. When failure occured during that duration of 100ms, registration completed were less as compared to other time intervels. Average number of registration completed in 100ms for:

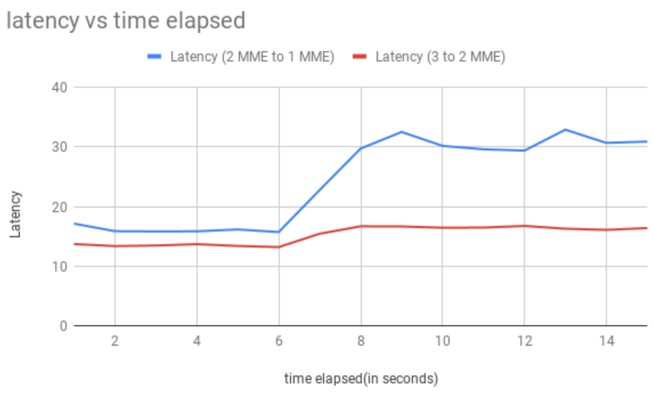
- Design1(Checkpointing per sub-procedure): 239
- Design2(Checkpointing per procedure): 441

Average number of registration completed during failure for:

- Design1(Checkpointing per sub-procedure): 223

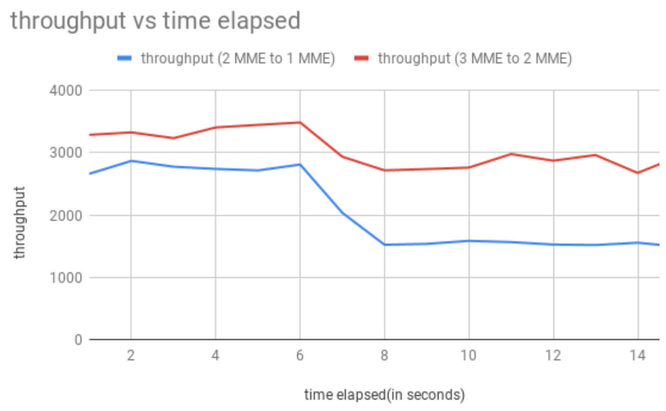


(a) Throughput

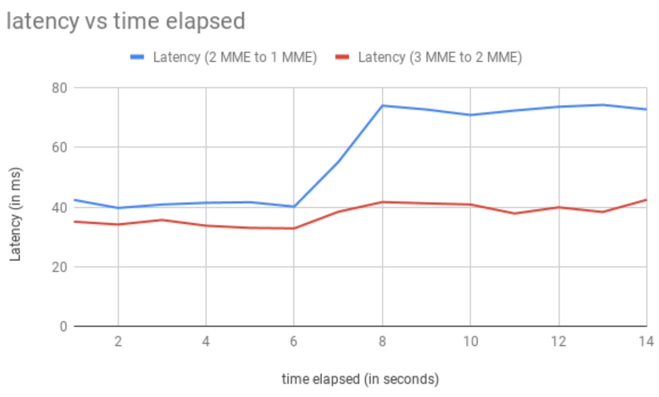


(b) Latency

Figure 5.9: Failure Handling with synchronizing UE context after complete procedure

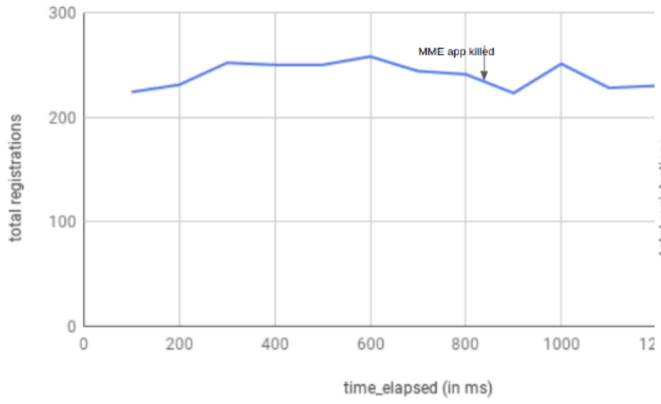


(a) Throughput

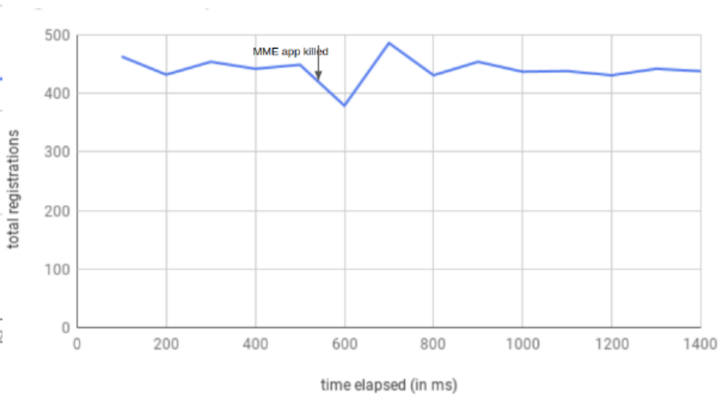


(b) Latency

Figure 5.10: Failure Handling with synchronizing UE context per-procedure



(a) State synchronization after every sub-procedure



(b) State synchronization after every procedure

Figure 5.11: Total number of registrations

- Design2(Checkpointing per procedure): 379

In case of synchronization per procedure when MME failed, UEs when in-between attach procedure so they had to restart from initial attach. That is why there is a large difference between number of registration completed at failure(379) and average number of registration completed(441). In case of synchronization per sub-procedure when MME failed, UEs did not need to start from initial attach request. They directly able to sent the current request.That is why there is less difference between number of registration completed at failure(223) and average number of registration completed(239).

For Figure 5.11a and Figure 5.11b, X-axis is time in ms and Y-axis is Number of registration completed.

5.1.3 Experiments for 5G core architecture

This section covers experimental results that we have taken in case of scaling of 5G core. In the below table our setup is explained, we placed AMF, AUSF, NRF, UDM, UDSF, RAN on virtual machine.

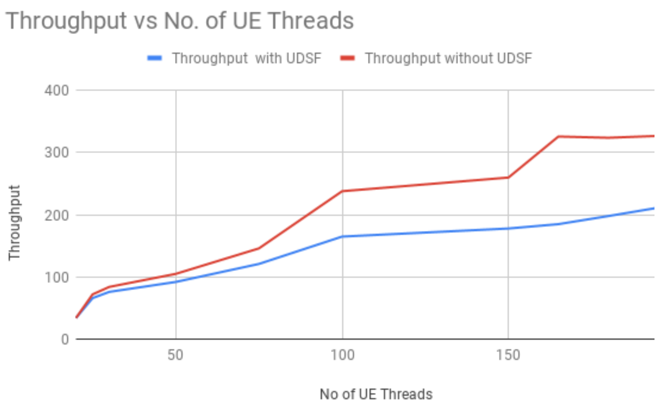
For each experiment, number of casablanca threads that we initialized in starting of every process are 200. As we explained in chapter 3, we are storing UE-state at UDSF after registration complete and at the end of registration accept request.

First experiment is to observe the overhead of UDSF and change in throughput and latency. For different number of UEs, throughput/latency in measured in both the conditions. For Figures 5.12a and 5.12b, X-axis is Number of user threads and Y-axis is Throughput and latency respectively.

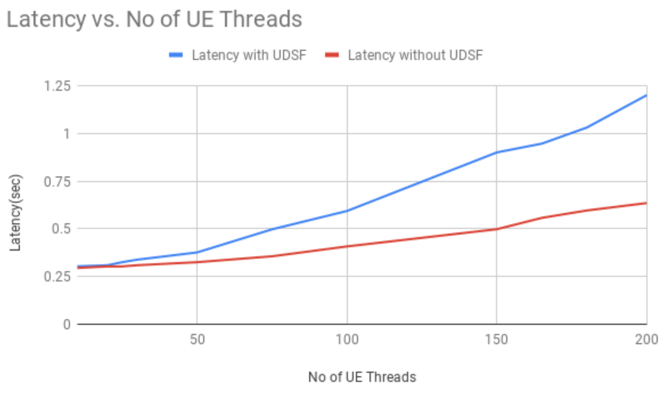
At saturation, throughput without UDSF is around 330 which reduces to almost 200 in case of state synchronization at UDSF. If UDSF is accessed in every procedure, latency will increase. As expected, latency with UDSF

COMPONENT	CPU CORE	RAM	OPERATING SYSTEM
RAN	4	8GB	Ubuntu 18.04
AMF	1	4GB	Ubuntu 18.04
NRF	1	4GB	Ubuntu 18.04
AUSF	1	4GB	Ubuntu 18.04
UDM	1	4GB	Ubuntu 18.04
UDSF	1	8GB	Ubuntu 18.04

Table 5.2: Individual Component Specification



(a) Throughput



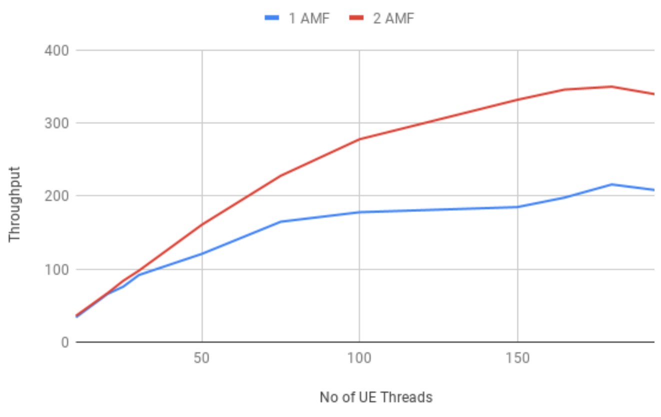
(b) Latency

Figure 5.12: Performance with and without UDSF

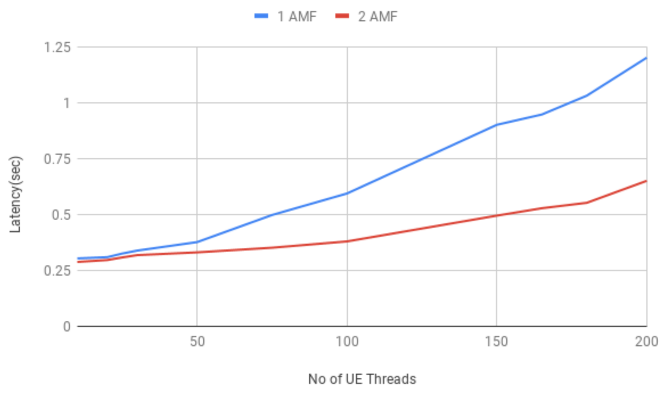
is more than without UDSF design.

5.1.3.1 Scaling from 1AMF to 2AMF

In this experiment we observe that if we have 2AMF replica in place of one, we can achieve higher throughput and lower latency which is expected. For different number of threads, throughput/latency is measured with 1AMF and 2AMF. In case of 2AMF replica, UEs are distributed using consistent hashing between both the replicas. At saturation, throughput with 1AMF is around 200 which increases to 335 with 2AMF. For Figures 5.13a and 5.13b, X-axis is number of UE-threads and y-axis is throughput.



(a) Throughput



(b) Latency

Figure 5.13: Scaling from 1AMF to 2AMF

Chapter 6

Conclusion and Future Work

In this report we explained complete design and implementation of distributed 4G LTE-EPC. There is an on going project on 5G-testbed in IIT Bombay. We also applied our design choices of distributed EPC to 5G core architecture. We successfully completed scaling and recovery from failure for NFV based LTE-EPC architecture. We compared different load balancing techniques like consistent hashing , simple hashing. We also added Redis data store in 4G core for state synchronization which is very effective for both scaling and failure handling scenarios.
We analyzed that due to a surge in control traffic, the latency of the system increases and MME/AMF gets overloaded and becomes a bottleneck. To decrease the latency (response time for user) we add a new MME/AMF replica and distributed the traffic. We did failure handling for 4G EPC architecture and observed that due to state synchronization at Redis, it is easy to recover from MME failure. We also calculated recovery time in case of failure of one of the MMEs. As the number of UE threads increases, the recovery time of the system also increases.
Below is the conclusion of our work based on all the experiments:

- For distributed 4G design without state synchronization, we observed change in throughput from 8000(1MME) to 16000(2MME). Average response time also reduced in case of 2MME. Convergence time for this design is 11213us.

- For distributed 4G design without state synchronization, failure handling is not that easy because UEs have to retry whole procedure. As the number of UE threads increases at RAN, the recovery time of the system in case of MME failure also increases.

- For distributed 4G design, by doing the data synchronization after every subprocedure, we are able to converge the system fast. With synchronization once per procedure, convergence time is 10211us which decreased further to 2660us if we are doing check-pointing after every sub-procedure. In the first design(checkpoint after whole attach), in-between threads have to wait before redirection to other MME because the context is only saved after attach procedure. But in the second design(check-point after per procedure), UE threads are not waiting and can directly connect to other MME as we are storing context after every subprocedure. So convergence time for second design is less than the first design. Saturation throughput for 2MME in design1 (7800) is more and latency is less as compared to design2(3500). If we need more throughput from our system then Design 1 will be good, but the time to stabilize after scaling will be more.

- For distributed 4G design, by comparing both the design option of state synchronization, in case of failure handling we observed that when one of the MME fails, deviation in throughput and latency is less in case of state synchronization per-procedure. This is because UEs does not retry from start, they only resend the current request.
- In 5G core architecture, we added another NF called as UDSF which is used to store user context. We first experimented the effect of UDSF on both latency and throughput. As expected throughput decreased when we added UDSF in the system. While scaling from 1AMF to 2AMF, we found the change in throughput from around 200 to 330 approximately.

Future work of this project is to apply failure handling strategies in 5G core architecture. Also to automate failure handling and scaling scenarios. AMF/MME can directly send some heartbeat to load balancer to show that they are alive. After 2-3 retries by UE if AMF/MME does not send the heartbeat, it means that AMF/MME failed. Other strategy is to use an orchestrator which always observe hotspot in the system and if CPU utilization or Memory used increased above threshold then automatically spawn another replica.