MobileStream: A Scalable, Programmable and Evolvable Mobile Core Control Plane Platform*

Junguk Cho
University of Utah
junguk.cho@utah.edu

Ryan Stutsman
University of Utah
stutsman@cs.utah.edu

Jacobus Van der Merwe
University of Utah
kobus@cs.utah.edu

ABSTRACT

Control planes in future mobile core networks face two new challenges. First, they must scale to process the growing control traffic generated by an ever increasing number of mobile devices. Second, they must be flexible and evolvable to support the range of emerging service abstractions and to realize customized network slices to meet the broad range of requirements of these networks. To address these challenges, we propose *MobileStream*, a scalable, programmable, and evolvable mobile core control plane platform. MobileStream provides a set of refactored basic building blocks, functionally decomposed from existing monolithic control plane components. It leverages realtime streaming frameworks to assemble, execute, and scale these blocks as streaming control plane applications. Moreover, it allows users to add their own functions to customize and optimize streaming control plane applications. We present several streaming control plane applications to showcase the flexibility and generality of MobileStream. We describe our extensive functional testing, with a variety of mobile devices and base stations, to validate the MobileStream prototype, and present the results of large-scale experiments demonstrating its scalability.

CCS CONCEPTS

• **Networks** → **Mobile networks**; *Programmable networks*;

KEYWORDS

Mobile Core Network; Control Plane; Realtime Streaming Framework

1 INTRODUCTION

Control planes in mobile core networks face two key growing concerns: exploding traffic volume and number of connected devices, and growing network service diversity. A large number of mobile devices will be connected to future mobile networks (e.g., 11.6 billion mobile-connected devices by 2021 according to some estimates [34]). These devices will generate an unprecedented volume of control traffic, creating the potential for signaling storms [30]. Cost-effective handling of heavy and bursty control traffic will correspondingly require highly scalable control planes.

These scalability concerns are leading, in part, to the second issue: mobile networks must be programmable to support flexible service abstractions and instantiate customized network slices to meet the broad range of requirements of these networks. Recently, several new mobile network architectures have been proposed, for example, Multi-Access Edge Computing (MEC) [36] for latency sensitive services; Cellular IoT (CIoT) Serving Gateway Node (C-SGN) [24] for IoT and M2M; and Control and User Plane Separation of EPC nodes (CUPS) [23] using SDN for better scalability and flexibility. In addition to new architectures, customized logical networks (i.e., network slices) can be dynamically created to meet the specific service requirements [17, 67]. In 5G networks, Network Slicing Selection Function (NSSF) [21] is introduced as a new control plane function to natively support network slicing. This requires easy and rapid adoption of new optimizations and control plane protocols as well as composition and instantiation of control plane functions in a variety of ways.

The fundamental limitations of inflexibility and scalability due to the current EPC architecture design are well understood [39, 45, 49–51, 53, 54, 59, 61, 63, 64, 66]. First, monolithic mobile network components make it difficult to customize networks (i.e., network slicing) and scale independently. For example, the MME performs most control plane operations (i.e., authentication, mobility and session management) and S/P-GWs perform both control and data plane operations. Second, distributed control plane components, which generate unique user equipment (UE) states, result in frequent state synchronization with sequential control procedures for state consistency. These procedures cause long delays.

Several research efforts have proposed a redesign of existing EPC networks [39, 45, 49, 51, 53, 54, 59, 61, 63, 64, 66] to address the limitations using NFV and SDN. They clearly separate control and data plane and take approaches to consolidate control and data components in different ways. Some of these efforts merge control and data plane components into one new component [53, 54, 63] and others merge control plane components based on mobile events [51, 64], or different service requirements [61] using modular components.

While existing works show better performance in terms of scalability than LTE/EPC networks, we argue that there is a need for a general and programmable control plane platform to address concerns related to inflexibility and scalability. Such a general platform will allow users to build various control plane architectures and easily run them in a scalable manner. However, to design a new mobile core control plane platform, there are several questions: (i) what are the key and reusable functions from control planes, (ii)

*Instructions for accessing the MobileStream code and using it in the powder testbed are available here: https://gitlab.flux.utah.edu/powder-profiles/mobilestream

CoNEXT '18, December 4–7, 2018, Heraklion, Greece

ACM ISBN 978-1-4503-6080-7/18/12…$15.00
https://doi.org/10.1145/3281411.3281442

how to manage complex UE states, (iii) how to provide programmability, and (iv) what is a right runtime environment to build and manage the control plane applications?

To answer these questions, we propose *MobileStream*, a scalable, programmable, and evolvable mobile control plane platform. MobileStream starts by decomposing the functions of the mobile network control plane into a set of refactored basic building blocks. As a result, MobileStream is **highly modular** which allows creating a pipeline to function as a control plane application by assembling basic building blocks. In addition, these pipelines can be rearranged to support new services. MobileStream is also **scalable** since individual blocks can be scaled independently to avoid pipeline bottlenecks. Key components that makes MobileStream easy to work with are its *user state management building blocks*, which assign globally unique UE states in a single point and externalize UE states in a remote key-value store. This allows control plane pipelines to remain mostly stateless.

As a runtime environment, MobileStream uses a stream processing framework which provides **programmability**. With the standard realtime stream processing framework APIs, users can (i) define pipelines based on basic building blocks, (ii) consolidate building blocks in one node in the pipeline, or distribute them across multiple nodes, (iii) specify data flows using different routing policies and data format between nodes in the pipeline and parallelism on a per-block level, and (iv) add custom functions on top of basic building blocks to optimize performance or support new services. MobileStream executes these stream control plane pipelines on a realtime stream processing framework in a cloud environment. Finally, streaming frameworks provide built-in redundancy and fault tolerance to transparently handle failures; something essential in moving critically reliable mobile control plane components to less reliable cloud platforms.

However, designing a mobile control plane over a streaming platform is not straightforward. MobileStream has to address a number of key challenges. First, to be effective and fault-tolerant, the stream control plane pipeline must be stateless, but it also must avoid frequent access to a remote store for UE states that would stall the pipeline and destroy its scalability. A second related challenge is that the control plane faces tight deadlines: the stream control plane pipeline processes control plane messages as soon as possible since control plane operations directly impact data access latency [50, 64].

MobileStream solves these issues following two design principles: *centralized and smart state management*, and *distributed computations*. It uses a single synchronization point with a remote store at the start of the pipeline, after which all operations in the pipeline are non-blocking. It maintains only one remote key-value store, regardless of radio connectivity mode of a UE, by leveraging novel state allocation. In addition, MobileStream uses state partitioning, a right parallelism of each building block, and different routing policies between basic building blocks for fair load distribution. Finally, it customizes building blocks to reduce the processing time of specific mobile events.

While designing 5G core networks is still ongoing [20, 42, 55, 58], for example, 3GPP proposes a Service Based Architecture (SBA) [21, 22]. The main concepts of the architectural design are similar to MobileStream: clean separation of control and data plane, functional decompositions of current monolithic components and stateless network functions (NFs) [21]. MobileStream can be a platform to realize the 5G next generation core architectures enabling the service diversity, scalability and flexibility envisioned by 5G.

To prototype MobileStream, we have implemented basic building blocks as well-encapsulated C++ and Java classes, which support most 4G network mobile events. We use the Apache Storm [69] realtime stream processing framework and programming APIs from Storm to assemble basic building blocks and run stream control plane applications on clusters of machines. In addition, we develop control plane protocol-specific routing classes based on Storm routing interfaces to build stream control plane applications. A Redis [16] in-memory key-value store holds user states, serialized as protocol buffers [15]. We demonstrate the generality and flexibility of MobileStream by implementing several mobile architectures which extend and refactor mobile control planes (ACACIA [33] and Fat-Proxy [64]) as Storm stream applications.

To validate the MobileStream framework, we evaluate our prototype with extensive functional tests including multiple UEs and eNBs (i.e., commercial smartphones and ip.access as eNB and Software-Defined Radio (SDR) based UEs and eNBs). Large-scale experiments demonstrate its scalability.

To the best of our knowledge, MobileStream is the first control plane platform that provides scalability and programmability by leveraging a realtime stream processing framework to build mobile control applications. We make the following specific contributions in this work.

- We design MobileStream, which provides a set of refactored basic building blocks and a remote key-value store to build stream control plane applications and executes them on a realtime stream processing framework.
- We introduce key design principles and use them to design several control plane applications, optimized for performance and consistent user state management in the face of a range of mobile events.
- We implement and evaluate a MobileStream prototype.

2 RELATED WORK

Modular design for mobile networks: Approaches to refactor mobile networks based on modular components have been proposed [51, 61]. Both works strategically locate multiple modules in the same machine to process a specific mobile event [51] or to meet different service requirements [61] to reduce the control messages exchanged in mobile networks. While MobileStream is also based on a modular design, MobileStream provides finer-grained modularity and a way of assembling the modules. In addition, both works leave the prototype of their approaches as future work and MobileStream can be a platform to realize their approach.

Fat-Proxy [64] optimizes control plane execution by combining logic from multiple EPC components into one Network Function (NF) running on a virtual machine to process a specific event. Its decomposition and reassembly of logic are similar to MobileStream, but MobileStream generalizes it as a full platform for existing functionality and for extending control plane functionality.

EPC-in-a-box design: CleanG [53] consolidates refactored control plane functions in EPC into one component. PEPC [63] and

SoftBox [54] propose EPC-in-a-box on a per UE basis by consolidating both control and data plane into one component. The proposed architectures [53, 54, 63] avoid distributed states management and long delay to synchronize them. MobileStream consolidates state management to avoid the complex state management, but distributes control plane computations by providing stateless building blocks and exploiting computation parallelism for them.

Distributed MME architectures: Recent works propose distributed MME architectures to address control plane scalability [26, 30, 62, 68]. DMME [26] proposes geographically distributed MMEs with remote storage. SCALE [30] proposes distributed MMEs consisting of front-end load balancers, MME processing entities and state replication. Stateless distributed MMEs with load balancers, MME processing entities, and remote storage have been proposed [62, 68]. One motivation of MobileStream is addressing signaling storms using distributed stream control plane applications, but its approach more generally supports function-level scalability beyond MME instance-level scalability. In addition, unlike [26, 30, 62, 68], which are a drop-in replacement for the MME with NFV, MobileStream goes beyond MME functionality and supports other control plane functionality (e.g., S/PGW-C and AuC).

NF frameworks: Click [47] provides packet processing modules and configuration abstractions (i.e., a directed graph) to assemble them as various network functions. NetBricks [60] proposes a small set of customizable network processing elements to build network functions and guarantees memory and packet isolation with very low overheads. StreamNF [46] proposes an NFV framework which leverages a streaming framework for network functions and a remote key-value store to externalize the internal state of NFs. It provides a simple and flexible API for operators to hide low level details of managing NFs and offers high performance and correctness. MobileStream takes similar approaches [47, 60] such as providing small building blocks and assembling them as a pipeline and has a similar architecture (e.g., a stream framework with a remote key-value store) to StreamNF [46], but MobileStream differs from them in the sense that it is a specialized NF framework to support mobile control plane network functions.

3 BACKGROUND

3.1 Mobile Networks

LTE/EPC architecture: The LTE/EPC network architecture consists of the Radio Access Network (RAN) and the Evolved Packet Core (EPC). RAN (i.e., eNBs) offers radio connectivity to user equipments (UEs) and communicates with the EPC. The EPC performs both control and data plane operations. MME (Mobility Management Entity) is the main control plane component; it performs user authentication, mobility and session management. MMEs interact with the HSS (Home Subscriber Server), which is a database of user profile information (e.g., subscription information). The data plane components (the S/P-GWs) interact with the control plane to set up data paths to enable forwarding of user traffic.

UE states: One of main functions in the EPC is to read, generate and update UE states based on mobile events. Table 1 shows the classification of important UE states managed by the EPC (a UE's complete state is much larger). They are classified as *static*, *dynamic* and *global* states. *Static* states, shown with bold text in Table 1, are permanently stored in UE and HSS; they do not change during mobile events. The rest excluding *static* states are *dynamic* states, shown with italic text.

Categories	Variables
UE IDs	**IMSI**, **IMEI**, *GUTI**, *UE IP address**
Security	*Authentication vectors (AVs) (i.e., auth, rand, xres, k_asme), NAS (i.e., encryption, integrity keys, nas_count_ul/dl), Radio security key*
Control plane	*S1AP ids* (i.e. mme/enb_ue_s1ap_id)*
Data plane	*S1 UL/DL TEIDs*, S5 UL/DL TEIDs**
Subscription info	**Subscribed profile (e.g.,QCI, ARP, etc.), APN conf (e.g., PDN type, DNS)**
State machines	*EMM, ECM states, and EMM procedure states*

Table 1: Classification of UE states[1]

Dynamic states are generated from different EPC components and eNB; for example, mme_ue_s1ap_id and UE IP address are generated by the MME and PGW respectively. Some dynamic states are shared between EPC components via control messages. As the main control plane component, the MME maintains all of the *dynamic* states. Among *dynamic* states, the GUTI, UE IP address, AVs, S1 UL TEID and S5 UL/DL TEIDs are only generated when the UE first attaches to the mobile network (during an initial attachment event) and are *semi-permanent*. Some states are changed per control message or mobile event. For example, nas_count_ul/dl values change per control message when the control message is security protected. EMM and EMM procedure states are mainly changed per control message to keep track of the current progress of mobile events for the UE. S1AP ids, S1 TEID DL in the data plane, and radio security key are changed based on the UE radio connectivity mode. When a UE goes into idle mode, from active mode, due to data inactivity (upon an S1 release event), they are released and when the UE becomes active (upon a service request event), they are re-generated by the eNB and MME.

Global states are a subset of the *dynamic* states and marked with asterisk. They should be globally unique per UE in each MME and S/P-GWs. They are used as identities in EPC components to perform UE-specific actions (e.g., processing control and data traffic in the MME and the S/P-GWs respectively). So, uniqueness is critical; if the same values for the global states are assigned to multiple UEs, the EPC malfunctions for the UEs. GUTI and mme_ue_s1ap_id in the MME, TEIDs of the data plane in the S/P-GWs and IP address in the PGW are *global* states.

3.2 RealTime Stream Frameworks

There are many open-source stream processing frameworks [1, 3, 11, 31, 48, 56, 69, 70] to support a variety of use cases (e.g., big data processing, realtime business intelligence [4, 7, 9], and network analytics [43, 44]). Each stream framework has slightly different characteristics and architecture, but Figure 1 shows a generic streaming framework architecture. The *Streaming manager* manages and schedules the execution of streaming applications

[1]Static and Dynamic variables are shown in bold and italic text respectively. Global variables are marked with an asterisk among dynamic variables.

submitted to the framework. *Worker agents* running on each machine in a cluster launch scheduled streaming applications from the streaming manager. *Workers*[2] running on the worker agents perform application-specific computation and routing logic specified in submitted streaming applications. The *Central coordinator* coordinates the communication among the components.

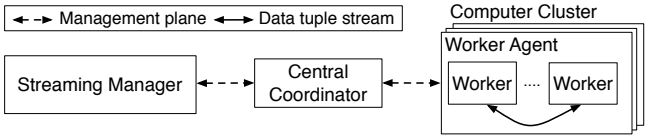


Figure 1: Stream framework architecture

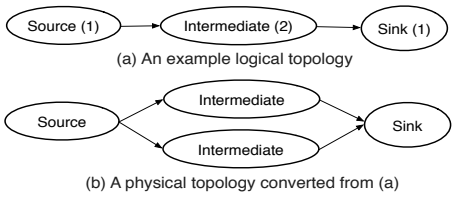


Figure 2: Example topology

Streaming applications: A streaming application in realtime streaming frameworks is represented as a topology. A logical topology shown in Figure 2(a) defines how input data tuples generated at the source node(s) are transformed into final output data tuples at the sink node(s) through the topology. *Data tuple* is a stream communication data model for node-to-node communications and a named list of values. Each node in a topology defines application-specific functions that converts incoming tuples to output tuples at the node, a routing policy that decides the next node(s) for the output tuples, and the degree of parallelism for the node. The logical topology is implemented by the user using the stream framework APIs. Given a logical topology, the scheduler converts it into a *physical topology* shown in Figure 2(b) based on the parallelism of nodes. The physical topology is deployed on available worker agents based on scheduling policies and resource availability of current worker agents. Individual nodes in the topology are launched as *workers* by *worker agents* and interconnect with one another based on assigned TCP connections.

Data tuple communication: A data tuple consists of the output values defined by the stream framework APIs from a worker and metadata. The metadata including source/destination node IDs, output length, and stream type is added to the output values by the stream framework. The metadata is used for parsing, forwarding and multiplexing in destination workers.

To decide the destination of data tuples, a stream processing framework can provide several types of routing policies [18] for individual workers. For example, *Key-based routing* and *Round-Robin routing* are used to guarantee the same data tuples go to the same next-hop worker by using a hash of data tuples and give fair load distributions respectively. In addition, a stream processing framework provides a routing policy programming interface to enable users to define *custom routing policies*.

[2]In the remainder of this paper, we interchangeably use *node* and *worker*.

4 MOBILESTREAM DESIGN

4.1 Design Goals

In architecting a new control plane platform for future mobile networks, we identify several key design goals. First, the platform must enable users to easily build distributed control plane architectures. It should provide a set of modular building blocks disassembled from existing control plane functionality and APIs to compose them. After assembling building blocks, the platform must be easy to execute and should manage the distributed control plane application without extensive configurations. Second, these distributed control plane applications must achieve horizontal scalability, high throughput and low latency even under massive control plane traffic by transparently exploiting parallelism and different composition of building blocks. In addition, it must always guarantee UE state consistency despite distributed operations. Finally, the resulting framework must be evolvable in the sense that users can create new building blocks and incorporate them to support new services.

4.2 MobileStream Overview

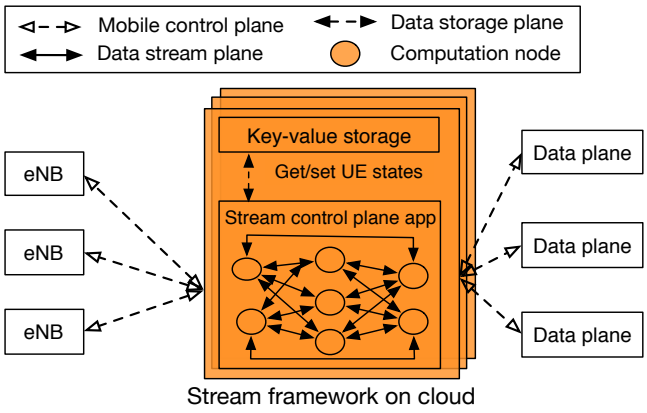


Figure 3: MobileStream overview

Figure 3 shows an overview of the MobileStream platform, which consists of a realtime stream processing framework and a remote key-value store. A MobileStream user builds a stream control plane application by defining a logical topology and submitting it to MobileStream. After transforming the logical topology to a physical topology, MobileStream runs it atop a realtime stream processing framework. After deployment, a stream control plane application interacts with eNBs and data plane components, and performs control plane operations. The first nodes in the stream control plane application get control request messages from multiple eNBs. The control request messages are processed and their corresponding control response messages are generated through the entire topology. Each node in the topology performs a small part of the control plane functionality and forwards an output to a next node. The generated control response message is finally forwarded to the first

node that originally received the control request message. This first node transfers the control response message back to the eNB that sent the original request. As the control plane operates, it occasionally must exchange control message with data plane components in order to set up data paths. Whenever a node must read or update UE states, it contacts the remote key-value store.

4.3 MobileStream Primitives

MobileStream provides three primitives: *Control plane building blocks and MobileStream programming APIs,* to define a logical topology, and *UE state management* to manage UE states in stream control plane applications. Given these three primitives, MobileStream enables a user to build a stream control plane application. In the following, we explain each primitive in detail.

4.3.1 Control Plane Building Blocks. MobileStream provides a set of basic building blocks, which perform a small part of control plane functionality, as C++ or Java classes, to build distributed mobile control plane architectures. The highly modular approach in MobileStream has multiple advantages. First, it enables users to compose building blocks in a different way according to their use cases. Second, each building block can scale independently based on its usage and resource requirements.

To decompose control plane components (i.e., MME, S/P-GWs, AuC) in 4G core network into a set of basic building blocks, we take three steps. First, we consolidate all control plane components into one logical control plane component to remove redundant state management. Then, we divide the logical control plane component into *computation* and *state management* layers. Finally, we decompose the two layers into refactored basic building blocks which minimize the dependency among them. The basic building blocks consist of (i) message transport layers, (ii) S1AP, mobility and session management, (iii) UE states management, (iv) security, and (v) utility. This classification makes them easy to parallelize, modify and replace.

Based on the classification, MobileStream provides transport layer building blocks (e.g., SCTP, UDP, TCP) to manage connections with other components (e.g., eNBs, S/P-GWs) and exchange control messages. The *SCTP* block manages SCTP connections and is used to send/receive control messages with multiple eNBs.

To perform control plane operations with eNB and UE, MobileStream provides three building blocks. *S1AP* block supports all procedures with eNB (e.g., S1 data and control bearer management and handover management). *Mobility Management (MM)* and *Session Management (SM)* blocks are responsible for procedures with a UE. *MM* block supports (un)registration, location management, and authentication of a UE and *SM* block handles EPS bearer and PDN connection management for a UE. Their first function is to convert byte streams to structured protocol data and vice versa (i.e., (un)pack control messages). After unpacking control request messages, they update UE states from structured protocol data. Then, they execute their own specific functions (e.g., generating dynamic variables shown in Table 1) and update UE states. Finally, they prepare structured protocol data with updated UE states and pack it to byte streams.

For UE states management functions, MobileStream provides three building blocks. The *UEstates* block is a data structure that holds all variables shown in Table 1. The *Global variable allocator* block manages globally unique variables per UE defined as global variables in Table 1. *UE states manager* block is responsible for managing *UEstates* in a local cache and a remote store. The *Global variable allocator* and *UE states manager* make UE state management simple and avoid distributed and duplicated states problems in existing EPC architecture. In addition, they allow other blocks to remain stateless.

For security functions, MobileStream provides the *authentication center (AuC)* building block, which generates security keys for mutual authentication between UE and networks (EPS Authentication and Key Agreement (AKA) in 4G networks). The *Integrity* block computes a message authentication code (MAC) for data integrity and the *Cipher* block encrypts and decrypts control messages.

Besides core control plane functions, MobileStream provides several utility functions like multiple protocol header decoder building blocks that return message types of control messages. They are used with other building blocks to assist in event-aware routing in stream control plane applications. For example, the *S1AP decoder* block returns a procedure code (e.g., initial UE message, handover, etc.) in the S1AP message. The *MM decoder* and the *Security decoder* blocks return mobility management type (e.g., attach request/accept, service request, etc.) and security type (e.g., plain text, integrity protected, ciphered, etc.) respectively.

Note that since a control message is (un)packed based on 3GPP standard, each building block only parses a required part of the control message to avoid frequent packing and unpacking the control message. For example, S1AP block does not (un)pack NAS message from the control message.

4.3.2 MobileStream Programming APIs. MobileStream provides two programming APIs to build distributed control plane applications. The first programming API is to use control building blocks. The other is streaming framework APIs used to define a topology of a stream control plane application.

```
/* Common programming interfaces */
public interface ControlMessageHandler {
StreamValue ProcessReqMsg (byte[] PDU,byte[] ueStates)
StreamValue ProcessResMSg (byte[] PDU,byte[] ueStates)
}

/* Return value of the functions */
class StreamValue{
    NEXT_TASK nextTask;
    byte[] PDU;
    byte[] ueStates;
}

public enum NEXT_TASK {
        NEXT_TASK_FORWARD_MSG,
        NEXT_TASK_REPLY_MSG,
        NEXT_TASK_REPLY_AND_UE_UPDATE,
        NEXT_TASK_UE_UPDATE,
        NEXT_TASK_S1AP_SRC_HO_REPLY_AND_UE_UPDATE,
        NEXT_TASK_S1AP_DST_HO_REPLY_AND_UE_UPDATE,
        NEXT_TASK_ERROR
};
```

Listing 1: Interface for basic building blocks.

(i) Programming APIs for basic building blocks: We define a common programming interface for control building blocks since it helps users to easily use and extend them. Basically, the control plane processes control request messages, updates UE states and sends control response messages based on updated UE states. Therefore, inputs and outputs of control plane building blocks are a control message payload and current UE states.

Based on this observation, we define a programming interface shown in Listing 1. *ControlMessageHandler* has two functions which process control request and response messages. Both use a control message payload and UE states as function parameters and return *StreamValue*. The *StreamValue* is a data structure including a *nextTask* value, a payload and updated UE states. The payload and updated UE states will be inputs to the next building block.

```
/* S1AP building block*/
S1AP s1ap = new S1AP();
/* Process S1AP and NAS control messages */
StreamValue streamValue = s1ap.ProcessReqMsg(s1apPDU,ueStates);
StreamValue streamValue = s1ap.ProcessResMsg(nasPDU,ueStates);
```

Listing 2: Basic building block APIs.

Most control plane building blocks implement *ControlMessageHandler*. Listing 2 shows how to process S1AP control messages with an S1AP block. S1AP block is created using the S1AP class, which implements *ControlMessageHandler*. The S1AP block calls *ProcessReqMsg* with an S1AP payload and UE states to process an S1AP control request message. *ProcessReqMsg* returns a payload, which can be a *NAS* request, a *S1AP* response, or no payload, and updated UE states. The *nextTask* value in *StreamValue* decides a type of payload and a next action. For example, when *NEXT_TASK_FORWARD_MSG* is a *nextTask* value, a *NAS* request message and updated UE states are forwarded to a next node. To process S1AP control response message, *ProcessResMsg* is called with a *NAS* response message and UE states.

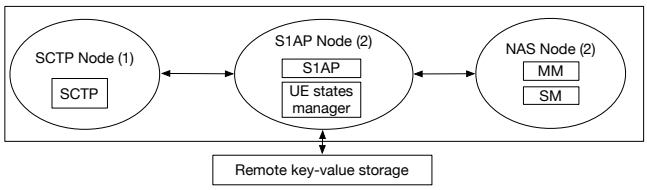


Figure 4: An example logical topology

(ii) Programming APIs for defining a topology: We need to define a topology based on building blocks to run a distributed control plane application as a streaming application on a realtime stream processing framework. This is done via the streaming framework APIs. With streaming framework APIs, users can configure building blocks by consolidating or distributing them in a node and specifying the parallelism of each block. Also, users can program the data flow between nodes by defining specific routing policies and data format. Lastly, on top of basic building blocks, users can add custom functions in nodes to optimize performance and support new services. Well-known stream framework operations (e.g., join, aggregations, filter, windows, etc.) can be used.

The logical topology shown in Figure 4 processes control messages from and to eNBs. The SCTP node, S1AP node, and NAS node internally create instances of building blocks represented as rectangle boxes in the nodes and use them to process control messages. Note that one or multiple building block(s) can be used in a node. For example, S1AP node uses *UE states manager* and *S1AP* blocks in the topology. When the topology is deployed on a stream framework, one SCTP node and two S1AP and NAS nodes are physically deployed and data flows through the topology.

4.3.3 UE State Management. As a distributed control plane application on the cloud, MobileStream needs a remote store. So, nodes can share UE states. Additionally, since any nodes in a stream control plane application running on cloud can fail, a remote store is required to avoid losing UE states. A failed node can recover lost UE states from the remote key-value store.

To store and fetch UE states in a remote key-value store while handling mobile events (e.g., receiving control messages from eNB), a unique key that is included in all control messages is needed. So, after unpacking a control message, a node including UE states manager block can fetch UE states based on a key from a remote key-value store.

In 4G networks, eNBs and MME identifies a UE using S1AP-ids (i.e., *enb_ue_s1ap_id* and *mme_ue_s1ap_id*). We decide to use *mme_ue_s1ap_id* as a key to store UE states in a remote key-value store. Since *enb_ue_s1ap_id* is only unique within an eNB and a MME can connect multiple eNBs, it is not unique. Instead, *mme_ue_s1ap_id* is unique within a MME and does not change even during intra X2 and S1 handovers, since a new eNB still uses the same MME.

However, S1AP-ids are released when a UE goes to idle mode from active mode due to data inactivity. When the UE has a mobile event (e.g., service request and tracking area update), the first control message for the mobile event does not have the *mme_ue_s1ap_id*. Instead, the control message includes an SAE-temporary mobile subscriber identity (S-TMSI), which consists of a MME code and MME-temporary mobile subscriber identity (M-TMSI). The S-TMSI is a part of GUTI assigned during an initial attachment procedure. Thus, the *mme_ue_s1ap_id* cannot be used as a key in idle mode. A simple solution is to maintain two key-value stores with different keys according to idle and active modes. However, considering the frequent transition between idle and active modes [50, 64] and more accesses to two key-value stores to transfer UE states between them, this naive solution is not efficient.

Instead, to avoid managing two key-value stores for a UE, in MobileStream the *Global variable allocator* block assigns the same value for *mme_ue_s1ap_id* and *M-TMSI* during an initial attachment procedure. This does not break EPC functionality since both are 32-bit, which is the maximum UE capacity of a MME. Due to this smart variable allocation, the *UE states manager* block can find UE states with *M-TMSI* from a key-value store when a control message is from an idle UE. In addition, the same *mme_ue_s1ap_id* assigned before is reused for the same UE within a MME by reserving *mme_ue_s1ap_id* in UE states. Note that a MME code is added to a key to differentiate between UEs associated to different MMEs.

5 MOBILESTREAM APPLICATION DESIGN

In this section, we introduce the design of a stream control plane application with MobileStream primitives based on two design principles: *centralized and smart state management*, and *distributed computations*. Specifically, we introduce three optimizations to (i) reduce the number of accesses to a remote key-value store, (ii) evenly distribute load to nodes, and (iii) avoid a bottleneck in a topology We then show how a designed stream control plane application performs common mobile events. Note, however, that MobileStream allows users to follow different approaches to building control plane applications.

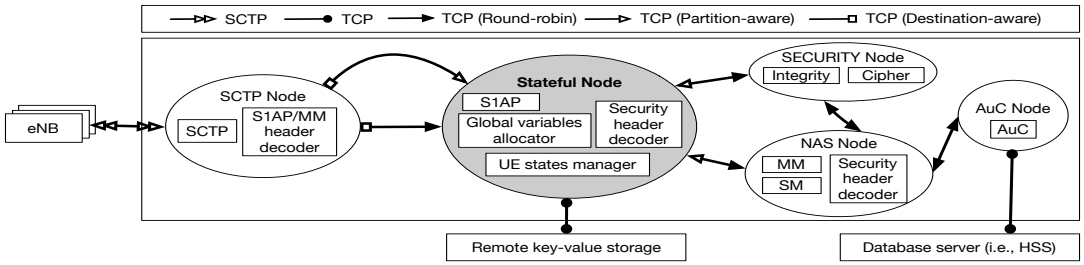


Figure 5: Stream EPC control plane application

5.1 Challenges in Designing a Stream Application

While MobileStream provides basic primitives to build stream control plane applications, designing a performant, scalable and robust mobile control plane over a streaming platform is not straightforward. There are two main challenges. First, since MobileStream splits computation and states management of the control plane into multiple building blocks, it requires efficient management of user states to utilize stateless computation building blocks. However, the design of a stateless stream control plane application requires frequent access to a remote store that would stall the pipeline, saturate the throughput of a remote key-value store, and destroy its scalability. In addition, it increases the latency to complete control plane operations due to round trips to a remote key-value store. Second, a stream control plane application, which consists of multiple nodes, should evenly distribute the management of user states and computations to nodes and avoid computation bottlenecks in a topology since that will slow down the entire pipeline. This is important in the sense that the stream control plane application should complete control plane operations as soon as possible since control plane operations directly impact data access latency [50, 64].

5.2 Optimizations for a Stream Application

Figure 5 shows a *logical topology* which is designed as a refactored EPC control plane architecture based on MobileStream primitives. The logical topology performs control plane operations of MME, AuC, and S/P-GWs in 4G networks based on three optimizations: *(i) as many stateless nodes as possible and a synchronization node at entry point in a topology*, *(ii) smart topology partition*, and *(iii) independent scaling of individual blocks*. This makes stream control plane applications performant, scalable and robust. Figure 6 shows the data tuple format exchanged between nodes in the topology. Detailed values in the data tuple are explained when they are used in the rest of this section.

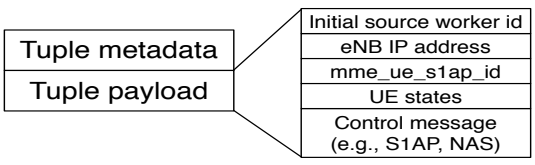


Figure 6: Data tuple format used between nodes

5.2.1 More Stateless Nodes and One Stateful Node. The first design choice is that a stream control plane application be as *stateless* as possible. The definition of *stateless* is that nodes in a topology do not store UE states locally and access a remote key-value store. This design improves performance linearly as more computation resources are allocated to stateless nodes. It makes scale-up/down operations easy and a topology robust in case of node failure. In addition, this design evenly distributes computations to stateless nodes with round-robin routing policy. This is depicted in Figure 5. All nodes are stateless except for one node depicted as gray color and round-robin routing policy is used for all data tuples heading to stateless nodes.

Since all nodes in a topology cannot be stateless, some nodes are *stateful*. *Stateful* nodes have a local cache and are *synchronized* with a remote key-value store. Thus, a stateful node should include the *UE states manager* block. Since stateless nodes in a topology also need UE states to process control messages, a stateful node is located at the entry of a topology, includes UE states in the data tuple shown in Figure 6, and sends it to a next stateless node. With this data flow model, the UE states can flow through the entire topology. One stateful node (i.e., one synchronization point) in the topology has several advantages. First, it improves performance due to infrequent access to a remote key-value store from multiple nodes, which eliminates network latency and makes stateless nodes non-blocking. In addition, it avoids saturating the throughput of a remote key-value store. Second, it helps the consistent management of UE states by avoiding maintaining UE states in multiple nodes and updating UE states from multiple nodes at the same time.

5.2.2 Smart Topology Partition. Since one mobile event completes after successfully exchanging multiple control messages (e.g., 11 messages in an initial attachment and 9 messages in S1 handover between eNB and MME in our testbed), using a local cache in a stateful node helps reduce the number of accesses to a remote key-value store. The challenge of this optimization is that the data tuple from stateless nodes should be forwarded to the same stateful node which has UE states in a local cache.

Smart topology partition using global state partitioning and partition-aware routing with *mme_ue_s1ap_id*, which is a key in a remote key-value store explained in Section 4.3.3, solves this challenge. When a stream control plane application runs, each stateful node is assigned to the same number of *global variables* based on the number of stateful nodes and global variables (i.e., the number of partitioned states = $\frac{\text{the maximum number of states}}{\text{the number of stateful nodes}}$). For example, a pool of *mme_ue_s1ap_id*, which is one of the global variables, is assigned to one stateful node

using maximum number of *mme_ue_s1ap_id* (i.e., 2^{32}) divided by the number of stateful nodes. *Global variables allocator* block in a stateful node manages the assigned global variables. When a UE attaches to mobile networks (i.e., an initial attachment event), the *Global variables allocator* in a stateful node assigns global variables to the UE. The stateful node stores UE states in a local cache.

To make use of a local cache in a stateful node, partition-aware routing running on stateless nodes guarantees all control messages including the same *mme_ue_s1ap_id* value are forwarded to the same stateful node. The partition-aware routing finds the index of stateful node with a simple calculation (i.e., index of stateful node = $\left\lceil \frac{\text{value of mme\_ue\_s1ap\_id}}{\text{max number of mme\_ue\_s1ap\_ids per stateful node}} \right\rceil - 1$) and sends data to the right stateful node using an internally maintained table <node ID, TCP connection> in the stateless nodes. Since all required information to execute partition-aware routing in the stateless nodes are obtained when the stateless nodes are initialized, this functionality does not lead to extra overhead (e.g., higher memory usage for a mapping table). The SCTP node shown in Figure 5 includes the *mme_ue_s1ap_id* in a data tuple shown in Figure 6. The *mme_ue_s1ap_id* flows through the entire topology.

Since one stateful node is responsible for initially assigned UEs due to global state partitioning and partition-aware routing, even distribution of UEs to stateful nodes is important. Header decoder blocks and different routing policies in the first node (i.e., SCTP node in this topology shown in Figure 5) achieve this. When the SCTP node knows that a control message is the first control message in an initial attachment event using the header decoder blocks, the first control message is routed to a stateful node with round-robin routing policy and the subsequent messages are forwarded to the stateful node with partition-aware routing.

While smart topology partition enables each stateful node to manage user states in a local cache, the stateful node updates the user states to a remote key-value store after completing a mobile event due to two reasons. First, it avoids performing heavy control plane operations again to generate the user states (i.e., security keys, sequence numbers, GUTI, tunnel Ids), when the stateful worker is dead. Second, up-to-date states in a remote key-value store can be used in different nodes to process a specific mobile event.

5.2.3 Scale Independently. A right degree of parallelism per node is important to improve performance of a topology. Since each building block requires different computation resources, we profile each building blocks using Linux *perf* tool [35]. Due to heavy mathematical computations in *Integrity* and *Cipher* blocks, their execution time takes longer than other blocks. So, using high parallelism for a node including *Integrity* and *Cipher* blocks avoids a bottleneck in a topology.

5.2.4 SCTP Management. Besides performance optimization, a designed stream EPC application should ensure correct SCTP packet delivery to eNBs. Since SCTP is a connection-oriented protocol at the transport layer, a response control message should be forwarded to a correct SCTP node, which gets a request control message from eNB.

To guarantee this requirement, SCTP node includes an *initial source worker ID* (i.e., its own node ID) and *eNB IP address* in a data tuple shown in Figure 6, when it receives a control request message. The two values are forwarded through an entire topology. When a control response message is ready in a node just before the SCTP node, the node can find the SCTP node using destination-aware routing with the initial source worker ID. The destination-aware routing uses a table <node ID, TCP connection> which is internally maintained in a node. Thus, in this case, the data tuple is forwarded to the SCTP node including the initial source worker ID in the data tuple before. Since one SCTP node can have multiple SCTP connections with multiple eNBs, it finds a right connection from a map table <eNB IP address, socket> using *eNB IP address* in the tuple. SCTP node updates the map when a new eNB is connected.

5.3 MobileStream Application Operations

In the following sections, we show how the topology shown in Figure 5 performs common mobile events.

5.3.1 Initial Attachment. When SCTP node receives a control message from eNB, it uses S1AP and MM header decoders to know an event type of the control message. In case of an initial attachment control message, the SCTP node forwards a data tuple including the control message, *initial source worker ID* and *eNB IP address* shown in Figure 6 to a stateful node based on round-robin routing policy. This guarantees even distributions of initial attachment events to stateful nodes. The stateful node creates new UE states from the request message (e.g., *enb_ue_s1ap_id*, IMSI, etc.) and assigns global variables (e.g., *mme_ue_s1ap_id*, *GUTI*, *TEID*, and *IP address*) using *Global variables allocator* block. After finishing S1AP protocol specific computations in the stateful node, the data tuple goes through NAS, AuC and NAS nodes sequentially with round-robin routing policy. This also guarantees even computation distributions to stateless nodes. When NAS node returns a control response message to the stateful node creating UE states, partition-aware routing with assigned *mme_ue_s1ap_id* is used. Since all next control messages in the initial attachment event have *mme_ue_s1ap_id*, SCTP node can forward the control messages to the stateful node which has UE states in a local cache using partition-aware routing with *mme_ue_s1ap_id*. After completing the initial attachment event procedure, the stateful node updates UE states in a remote key-value store. Since stateful and NAS nodes know which security options (e.g., plain text, integrity protected, ciphered) are used for the control messages due to the Security header decoder block, they forward the data tuple to the SECURITY node if needed.

5.3.2 S1 Release and Service Request. In LTE/EPC networks, if a UE does not send data for about 10∼12 seconds [41, 65] in active mode, it goes to idle mode after completing S1 release procedure. The UE starts a service request procedure to re-establish a connection in idle mode if needed.

In case of S1 release event, all messages are forwarded to a stateful node which has UE states in a local cache using partition-aware routing policy with *mme_ue_s1ap_id*. After completing the S1 release procedure, some UE states (i.e., S1AP ids, S1 TEID DL in data plane, and radio security key) are released and the stateful node updates UE states in a remote key-value store.

When a UE starts the service request procedure, the first message does not include *mme_ue_s1ap_id* and includes *M-TMSI*, which is assigned as the same value of *mme_ue_s1ap_id* during an initial

attachment procedure for the UE explained in Section 4.3.3. SCTP node detects the service request event using S1AP and MM header decoders and extracts *M-TMSI* from the control message. The control message is forwarded to a stateful node using partition-aware routing policy with M-TMSI. The stateful node has the UE states in a local cache and re-assigns UE states released before. Note that the same *mme_ue_s1ap_id* value is assigned for the UE again. Thus, the next control messages which always include *mme_ue_s1ap_id* are forwarded to the same stateful node. After completing the service request procedure, the stateful node updates UE states in a remote key-value store.

5.3.3 Intra S1 and X2 Handovers. Handover happens when a UE moves from a source eNB to a target eNB due to better radio signal from the target eNB. The first control message (i.e., *handover required* message) from the source eNB is forwarded to a stateful node which has UE states in a local cache. The stateful node includes the currently used *mme_ue_s1ap_id* for the UE and sends the *handover request* message to the target eNB. Since the *handover request* message should be sent to the SCTP node which has an SCTP connection with the target eNB, the stateful node cannot use *initial source worker ID* to deliver a response control message. Instead, the stateful node needs to know the SCTP ID and the target eNB IP address. The stateful node maintains a map table <eNB ID, List of <SCTP ID, eNB IP address>>, when eNB is first connected (i.e., *S1-Setup* procedure). The stateful node can find SCTP ID and eNB IP address from the map table since *handover required* message includes *target-eNB ID*, and uses destination-aware routing with the SCTP ID. Thus, the response control message is forwarded to the right SCTP node.

A control message from the target eNB is forwarded to the stateful node since it has the same *mme_ue_s1ap_id*. The stateful node needs to send a response control message to the right SCTP node which has an SCTP connection with the source eNB. Since currently serving eNB ID information (i.e., source eNB) is stored from the *initial attachment procedure* in UE states, the stateful node can find the source eNB information with the same approach. The stateful node updates UE states in the remote key-value store after completing intra S1 handover.

Note that the designed stream control plane application also supports intra X2 handover. A source eNB sends currently assigned *mme_ue_s1ap_id* to a target eNB during X2 handover procedure. When target eNB sends a control message (i.e., *S1AP Path Switch Request* to a core network component (i.e., MME in 4G networks)), it uses received *mme_ue_s1ap_id* as S1AP-id. Therefore, intra X2 handover works like other events.

6 MOBILESTREAM USE CASES

In this section, we introduce several stream control plane applications "derived from" the basic stream EPC control plane application shown in Figure 5. This illustrates how MobileStream enables users to build different use cases without significant modifications of the stream EPC control plane application.

6.1 Event-based Node

Fat-proxy [64] proposes a logic-based Network Function (NF) segregation instead of an instance-based NF segregation to optimize

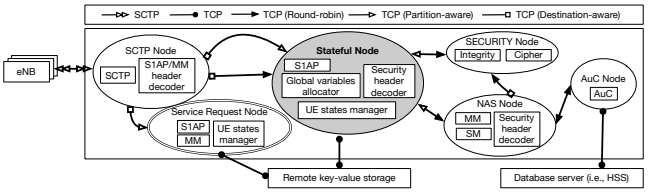


Figure 7: Stream Fat-Proxy-like control plane application

three mission critical mobile events (i.e., service request, handover, and paging). It combines the logic from multiple control plane components (i.e., MME, S/P-GWs) and creates one instance to handle only one critical mobile event. With MobileStream, users can build a stream control plane application which is functionally similar to Fat-proxy. Event-based node can be created to process a specific mobile event by consolidating multiple building blocks in one node. The control messages for a specific mobile event can be forwarded to the event-based node using header decoder blocks.

The topology shown in Figure 7 adds a new *Service Request* node including *S1AP* and *MM* building blocks to the stream EPC control plane application shown in Figure 5 to process service request events. Since SCTP node can know that the request control message is a service request by using S1AP and MM header decoder blocks, the service request message is forwarded to *Service Request* node which is responsible for the completion of the service request procedure.

6.2 Join Streams

Fat-proxy [64] parallelizes control message execution for different protocols (e.g., S1AP and GTPv2-C messages) in mobile event procedures to reduce event completion time. While a stream EPC control plane application shown in Figure 5 does not take the same approach due to an architectural difference (i.e., Fat-proxy keeps all 3GPP protocols, but stream EPC is a refactored architecture), MobileStream can leverage a similar optimization using multiple data tuple transmissions at the same time and join streams to speed-up mobile event procedures.

In case a NAS control message is integrity protected, it is simultaneously forwarded to SECURITY and NAS nodes from the stateful node. When the stateful node sends a data tuple to SECURITY node, it includes the ID of NAS node to which the stateful node sends the data tuple. After SECURITY node finishes its computation, it sends the results of the integrity check to the NAS node which receives the same NAS message using destination-aware routing with the ID of NAS node shown in Figure 7, instead of round-robin routing. In the NAS node, two outputs from both nodes are merged with *mme_ue_s1ap_id*. The same approach can be used to parallelize integrity and cipher computations for control messages.

6.3 Mobile Edge Networks

ACACIA [33] proposes a new mobile edge network architecture in 4G networks. It supports selective traffic offloading for specific services (e.g., VR, AR) to mobile edge computing (MEC) servers close to a UE by leveraging QoS bearer framework (i.e., a dedicated bearer procedure) and split versions of S/P-GWs. A UE sends a

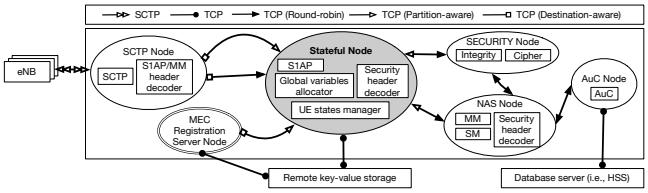


Figure 8: Stream MEC control plane application

MEC connectivity request to mobile networks with its IP address and a service type. LTE/EPC sets up a dedicated bearer for the UE based on the IP address and the service type. After finishing the procedures, the UE can start talking to the MEC server for the service through the dedicated bearer.

MobileStream enables users to build a stream control plane application shown in Figure 8 which is functionally the same as the ACACIA mobile edge network architecture. Two components are added to the stream EPC control plane application shown in Figure 5. First, MEC Registration Server (MSR) node is added to get UE IP address and a service type from a UE and manage the parameters (e.g., Traffic Flow Templates (TFT), QoS, etc.) for the requested service. To find the states of requested UE with the IP address in the MSR node, a stream control plane application needs to manage another key-value store (i.e., key : *UE IP address*, value : *mme_ue_s1ap_id*). To do it, *UE state manager* block in a stateful node is updated to manage the new key-value store.

The stateful node including updated *UE state manager* block needs to maintain the new key-value store when an initial attachment procedure happens. When the MSR node gets a MEC connectivity request from a UE, it can find *mme_ue_s1ap_id* with IP address from the new key-value store. Then, it forwards parameters for the service to the stateful node which has UE states in a local cache using partition-aware routing. The rest of the procedures are the same as a dedicated bearer procedure in LTE/EPC networks.

7 MOBILESTREAM PROTOTYPE

Basic building blocks: We have implemented basic building blocks as C++ and Java classes. Currently, we implemented S1AP, MM, SM, AuC and Integrity and Cipher blocks based on defined MobileStream interfaces described in Section 4. We reuse ASN.1 encoding and decoding control message logic and security functions from srsLTE [37] written in C++. Since Storm natively supports Java/Clojure, we have implemented some building blocks as Java classes to avoid expensive JNI calls. As Java classes, we implemented SCTP transport channel to communicate with eNB using the Netty framework [14], S1AP, MM and Security header decoder blocks to know a type of mobile event and security from control messages.

UE states: We use language-neutral Google protocol buffers [15] to define UE states shown in Table 1. After defining UE state objects with Google protocol buffers's script, Google protocol buffers generates the right data structures (e.g., class) for each language. It also provides small, fast and simple (de)/serialization mechanism for the defined data structure. Since UE states are needed for building blocks written in C++ and Java, we generate C++ and Java classes for UE states using Google protocol buffers. It makes UE state exchange easy between C++ and Java layer as JNI function parameters after serialization of UE states. In addition, since UE states are serialized as byte data to store them in a remote key-value store, a light serialization overhead from Google protocol buffers is beneficial.

UE states management: We use Redis [16] which is an in-memory data structure store as a remote key-value store. We implement ueStateManager as a Java class to manage a local cache and Redis key-value store. We use Jedis [10], which is a Java client library for Redis, to manage UE states. UE states are stored in Redis after serializing them as byte data with Google protocol buffer's serialization.

Stream control plane application: We use Apache Storm [69] as a realtime stream processing framework to run stream control plane applications. It provides common runtime facilities (e.g., centralized job scheduler and coordinator, per-host daemon) and programming APIs which are highly flexible for users to specify routing policy, degree of parallelism, and tuple formats for defining a topology. In addition, it allows users to build custom routing policies by implementing a *CustomStreamGrouping* interface [18]. We implement two routing policies (i.e., destination-aware and partition-aware routing) by realizing *CustomStreamGrouping* interface. Since Storm supports the clojure and Java languages, we implement wrapper functions for each basic building block written in C++ using Java Native Interface (JNI). We implement computation nodes using *BaseBasicBolt* class in Storm for control plane applications and they use basic building block libraries. We implement three stream control plane applications described in Section 5 and Section 6.

Traffic generator: To evaluate stream control plane applications for mobile events, a traffic generator is required to test large scale control messages. We use an eNB emulator in OpenEPC [19] which is LTE/EPC software implementation based on 3GPP standard. Since the OpenEPC which we use does not support S1 release, service request, intra S1 handover, we extend the eNB emulator to support them. We also refactor the commandline interface of the eNB emulator to systematically trigger mobile events.

8 EVALUATION

In this section, we describe our functional and performance evaluations with several stream control plane applications. We run Storm on five physical machines in the PhantomNet testbed [29], each with 32 cores, 64GB of memory and 10GB NICs. We use one machine for running the streaming manager and central coordinator, and one machine for running the Redis key-value store. The rest of the machines are used for running stream control plane applications. In all experiments, we use Storm's default configurations except for the memory allocation per node (i.e., 2GB) with a round-robin topology scheduler which evenly deploys nodes in a physical topology into three physical machines.

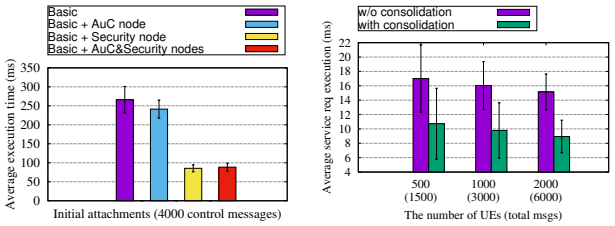
8.1 Mirco-benchmarks

8.1.1 Validating Standards Compliance. We verify standards compliance of the stream control plane application shown in Figure 5 by testing its inter-operability with several open source projects and commercial hardware. It is tested with commercial closed source and standards based UEs (i.e., Nexus 5, One+ One) and eNB (i.e., ip.access small cell [13]), using several mobile events (i.e.,

initial attachment, detach, S1 release, service request, tracking area update (TAU), dedicated bearer setup and S1 intra handover). The MobileStream implementation worked correctly in all these tests. We also test OpenAirInterface (OAI) [57] and srsLTE [37] open source projects using SDR as UE and eNB except for a dedicated bearer setup, S1 intra handover and TAU since these stacks do not support the events yet. In addition to commercial and SDR-based UEs and eNBs, we also test the stream control plane application with OAI and OpenEPC emulators. Our results show the standards compliant operations of the stream control plane applications based on MobileStream primitives.

8.2 Performance Evaluation

For performance evaluations, we use the stream control plane application shown in Figure 5 with different configurations and measure time to complete an event. In addition, due to limited availability of real UEs and eNBs to evaluate large scale experiments, we use emulated eNBs in OpenEPC as traffic generators.



(a) Node parallelism (b) Node consolidation

Figure 9: Node configuration impact

8.2.1 Node Configuration Evaluation. We conduct experiments to show the impact of scaling building blocks independently. In Figure 9(a), *Basic* means no parallelism for all nodes (i.e., one sctp, s1ap, nas, auc, and security node). We compare *Basic* against the topologies which add more workers to *Basic* when one eNB generates 500 initial attachment events (i.e., 4,000 messages) for one second. Figure 9(a) shows the average initial attachment execution time. Assigning one more *Security* node to *Basic* topology (denoted as *Basic + Security node* in Figure 9(a)) shows 3.1x execution time reduction compared to *Basic* topology since it avoids a bottleneck from heavy and frequent mathematical computations (i.e., integrity check) in the topology. However, increasing *AuC* node does not show high benefit even though the AuC node also performs heavy mathematical computations to generate an authentication vector since it is executed once per UE.

Figure 9(b) shows the impact of logic consolidation (explained in Section 6.1 Event-based Node) to handle service request events according to different number of UEs. In *w/o consolidation*, S1AP and MM blocks are physically separated, which means that they run on different JVM processes to process service request events. In *with consolidation*, S1AP and MM blocks run on the same JVM process. The consolidation case shows short execution time since it does not require extra computations (e.g., data tuple serialization) and network traverse. The consolidation case reduces service request event execution time by 70% compared to the w/o consolidation case in case of 2000 UEs.

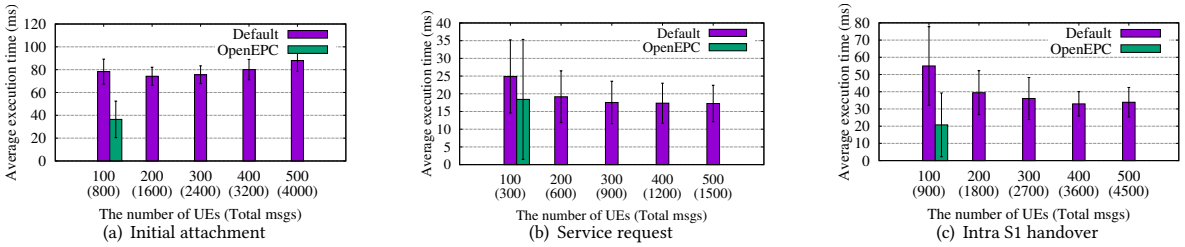
8.2.2 Comparison with OpenEPC. We compare *Default* topology consisting of 6 nodes (i.e., one sctp, s1ap, nas and auc node, and 2 security nodes) against OpenEPC. To allow for a fair comparison, we measure execution time to perform the control plane part only, excluding setting up data path in data plane components (i.e., S/P-GWUs) from control plane components (i.e., S/P-GWCs). For this experiment, we use the split version of OpenEPC which separates S/PGWs into GWCs and GWUs. We set up one MME, S/P-GWCs and HSS in each physical machine for OpenEPC configuration. We use one eNB to generate from 100 up to 500 events for one second. With 100 UEs, OpenEPC shows better performance than the *Default* topology, especially, initial attachment and intra S1 handover. However, since P-GWc in OpenEPC is limited over 100 UEs, we cannot proceed with the rest of the experiments. The advantage of the *Default* topology is that it decreases execution time for service request and intra s1 handover as the number of UEs increases. For initial attachment events, it shows stable execution time even though the number of messages increase 5 times. Note that while OpenEPC shows better performance compared to *Default* topology in case of 100 UEs, our node consolidation optimization shown in Figure 9(b) shows better results for service request event than OpenEPC. It shows that MobileStream can optimize performance by customizing a topology for specific events.

8.2.3 MobileStream Optimization Evaluation. We conduct experiments to show the scalability of the stream control plane applications with multiple eNBs. We use 3 eNBs as a traffic generator and each generates 500 events (i.e., 12,000 (8x1500) for initial attach events and 4,500 (3x1500) for service request events) for one second. We configure different amounts of parallelism in the topologies based on *Default* topology (i.e., one sctp, s1ap, nas and auc node, and 2 security nodes). *Double* and *Triple* mean the number of multiplication of nodes in the *Default* topology except for the SCTP node. Figure 11(a) shows the result for initial attachment events. While the *Default* topology takes average 85 ms to complete an initial attachment event in case of 500 UEs shown in Figure 10(a), it takes over 700 ms on average in case of 1500 UEs shown in Figure 11(a). It shows that the performance significantly drops as traffic load increases. *Double* and *Triple* shows 6.2x and 9.4x execution reduction compared to *Default* in the best case due to optimizations (i.e., parallelism, even load distributions to stateful and stateless nodes) described in Section 5. Figure 11(b) shows the result for service request events. Both *Double* and *Triple* show 13% execution time reduction compared to *Default*. Since service request events are relatively light compared to initial attachment events, both *Double* and *Triple* shows similar improvement.

9 DISCUSSION AND FUTURE WORK

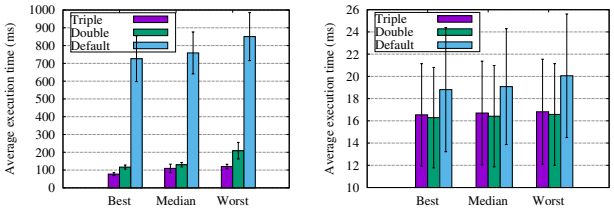
We presented the design and implementation of a mobile core control plane platform. There are several interesting aspects for further investigation from the proposed platform.

Dynamic reconfiguration: Since control traffic in mobile networks is time-varying traffic [12], dynamic scaling for running stream applications is important. As shown in Figure 11, the performance is highly related to the number of nodes assigned to a stream application. In addition, adding or removing building blocks on the fly helps support new services. Since MobileStream uses



(a) Initial attachment (b) Service request (c) Intra S1 handover

Figure 10: Comparison against OpenEPC



(a) Initial attach events (b) Service request events

Figure 11: MobileStream optimization impact

Storm which does not support a dynamic reconfiguration (e.g., dynamic scaling, adding or removing building blocks), it also does not support the dynamic reconfiguration. However, since some stream frameworks like Apache Apex [6] already support the dynamic reconfiguration and our design does not rely on specific functions from Storm, the limitation is not fundamental. In addition, our design (e.g., one stateful node in a topology and externalizing states to a remote key-value store to make most of the nodes in a topology stateless) makes the dynamic reconfiguration easy. To make a dynamic scaling decision for running stream applications, MobileStream can leverage the detailed metric information from stream frameworks (e.g., Storm provides the number of sending and receiving tuples, various tuple processing time, etc. [27]).

User plane extension: We do not currently support data plane functionality *within* the MobileStream framework. However, designing control and data planes as a stream application based on MobileStream will be ideal to realize network slicing. Like a modular approach to design a control plane in MobileStream, multiple data plane operations (e.g., packet forwarding, QoS, policy, etc.) in mobile networks can be designed as a pipeline and the data plane nodes in the pipeline are managed by control plane nodes (e.g., a stateful node). Since the time requirement to process data plane traffic is different from control plane traffic, MobileStream requires high performance packet processing frameworks (e.g., DPDK [5]) for data plane extension. However, the massive machine type communications (mMTC) for IoT and M2M in 5G networks can be readily supported in MobileStream because devices for mMTC sparsely send a very small amount of data with relaxed latency requirements. Especially, control plane CIoT EPS optimization [24], which encapsulates the data from IoT devices in NAS control messages, can be realized to augment MM block or create a new node in MobileStream.

Stream framework tuning: Storm is designed based on multiple threads in one worker and queues to transfer data between threads and other workers through networks. Several parameters (e.g., buffer size, batch size, flush frequency, etc.) to manage the queues are configurable and are related to latency and throughput of a stream application [28]. We will explore performance impact according to different parameter configurations as future work.

MobileStream design in other platforms: Stream frameworks (e.g., Storm [69], Flink [31], Spark [31], and Apex [1]) have different characteristics in terms of latency and throughput [32], internal state management [31], fault-tolerant, and dynamic reconfigurations based on their main design principles. Since the design of MobileStream does not depend on specific functions in Storm, it is interesting to adapt and evaluate MobileStream with other stream frameworks. Serverless computing [2, 8, 25, 38, 40, 52] has emerged as a new cloud-computing execution model. It is an event-driven computing based on stateless functions running on a container. Considering that control messages are generated based on UE events and time-varying traffic [12], and the application design between serverless computing and MobileStream is similar, it is an interesting direction to explore and adapt the MobileStream design to serverless platform.

10 CONCLUSION

We identify scalability and programmability challenges for future control planes in mobile networks. To address these challenges, we propose MobileStream, a scalable, programmable and evolvable mobile control plane platform. MobileStream enables users to build distributed control plane architectures as stream control plane applications. To build stream control plane applications, MobileStream provides a set of refactored building blocks and leverages a real-time stream framework to assemble and execute them in a scalable manner. In addition, we introduce key design principles to optimize stream control plane applications and several stream control plane applications given the programmability of MobileStream.

Acknowledgements: We would like to thank our shepherd and the anonymous reviewers for their feedback on earlier versions of this paper. This work is supported in part by the National Science Foundation under grant numbers 1305384 and 1827940.

REFERENCES

[1] Apache Apex. https://apex.apache.org.
[2] Apache OpenWhisk is a serverless, open source cloud platform. http://openwhisk.apache.org/.
[3] Apache Samza. http://samza.apache.org.

[4] Bullet at Yahoo. https://yahoo.github.io/bullet-docs.
[5] Data Plane Development Kit. http://dpdk.org.
[6] Dynamic Application Modifications. https://apex.apache.org/docs/apex/application_development/#scalability-and-partitioning.
[7] Evolution of the Netflix Data Pipeline. https://medium.com/netflix-techblog/evolution-of-the-netflix-data-pipeline-da246ca36905.
[8] Functions as a Service (FaaS). https://blog.alexellis.io/functions-as-a-service/.
[9] How Spotify Scales Apache Storm. https://labs.spotify.com/2015/01/05/how-spotify-scales-apache-storm.
[10] Jedis. https://github.com/xetorthio/jedis.
[11] JSTORM. http://jstorm.io/.
[12] Managing LTE Core Network Signaling Traffic. https://www.nokia.com/en_int/blog/managing-lte-core-network-signaling-traffic.
[13] nanoLTE Access Points (E-40 Access Point). http://www.ipaccess.com/en/lte.
[14] Netty. https://netty.io/.
[15] Protocol Buffers. https://developers.google.com/protocol-buffers/.
[16] Redis. https://redis.io/.
[17] Service-Based Architecture for 5G Core Networks. https://www.3g4g.co.uk/5G/5Gtech_6004_2017_11_Service-Based-Architecture-for-5G-Core-Networks_HR_Huawei.pdf.
[18] Storm grouping. http://storm.apache.org/releases/current/Concepts.html.
[19] OpenEPC. http://www.openepc.com/, 2015.
[20] White Paper, Designing 5G-Ready mobile core networks. https://tinyurl.com/yaok5lbo, 2016.
[21] 3GPP Ref #: 23.501. System architecture for the 5g system, stage 2, 2018.
[22] 3GPP Ref #: 23.502. Procedures for the 5g system, 2018.
[23] 3GPP Ref #: 23.714. Study on control and user plane separation of EPC nodes, 2015.
[24] 3GPP Ref #: 23.720. Study on architecture enhancements for Cellular Internet of Things, 2015.
[25] AMAZON. AWS Lambda - Serverless Compute. https://aws.amazon.com/lambda/.
[26] X. An, F. Pianese, I. Widjaja, and U. Günay Acer. Dmme: A distributed lte mobility management entity. *Bell Labs Technical Journal*, 17(2):97–120, 2012.
[27] Apache Storm. Metrics.md. https://github.com/apache/storm/blob/master/docs/Metrics.md.
[28] Apache Storm. Performance.md. https://github.com/apache/storm/blob/master/docs/Performance.md.
[29] A. Banerjee, J. Cho, E. Eide, J. Duerig, B. Nguyen, R. Ricci, J. Van der Merwe, K. Webb, and G. Wong. Phantomnet: Research infrastructure for mobile networking, cloud computing and software-defined networking. *GetMobile: Mobile Computing and Communications*, 19(2):28–33, 2015.
[30] A. Banerjee, R. Mahindra, K. Sundaresan, S. Kasera, K. Van der Merwe, and S. Rangarajan. Scaling the lte control-plane for future mobile access. In *Proceedings of the 11th ACM Conference on Emerging Networking Experiments and Technologies*, page 19. ACM, 2015.
[31] P. Carbone, S. Ewen, G. Fóra, S. Haridi, S. Richter, and K. Tzoumas. State management in apache flink®: consistent stateful distributed stream processing. *Proceedings of the VLDB Endowment*, 10(12):1718–1729, 2017.
[32] S. Chintapalli, D. Dagit, B. Evans, R. Farivar, T. Graves, M. Holderbaugh, Z. Liu, K. Nusbaum, K. Patil, B. J. Peng, et al. Benchmarking streaming computation engines: Storm, flink and spark streaming. In *Parallel and Distributed Processing Symposium Workshops, 2016 IEEE International*, pages 1789–1792. IEEE, 2016.
[33] J. Cho, K. Sundaresan, R. Mahindra, J. E. van der Merwe, and S. Rangarajan. Acacia: Context-aware edge computing for continuous interactive applications over mobile networks. In *CoNEXT*, 2016.
[34] Cisco. Cisco Visual Networking Index: Global Mobile Data Traffic Forecast Update, 2016-2021 White Paper. https://www.cisco.com/c/en/us/solutions/collateral/service-provider/visual-networking-index-vni/mobile-white-paper-c11-520862.html.
[35] A. C. De Melo. The new linux perf tools. In *Slides from Linux Kongress*, volume 18, 2010.
[36] G. Fabio et al. Mec deployments in 4g and evolution towards 5g. *ETSI White Paper No. 24*, 2018.
[37] I. Gomez-Miguelez, A. Garcia-Saavedra, P. D. Sutton, P. Serrano, C. Cano, and D. J. Leith. srslte: an open-source platform for lte evolution and experimentation. In *Proceedings of the Tenth ACM International Workshop on Wireless Network Testbeds, Experimental Evaluation, and Characterization*, pages 25–32. ACM, 2016.
[38] Google. Google Cloud Functions. https://cloud.google.com/functions/.
[39] G. Hampel, M. Steiner, and T. Bu. Applying software-defined networking to the telecom domain. In *INFOCOM, 2013 Proceedings IEEE*, pages 3339–3344. IEEE, 2013.
[40] S. Hendrickson, S. Sturdevant, T. Harter, V. Venkataramani, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau. Serverless computation with openlambda. *Elastic*, 60:80, 2016.
[41] J. Huang, F. Qian, A. Gerber, Z. M. Mao, S. Sen, and O. Spatscheck. A close examination of performance and power characteristics of 4g lte networks. In *Proceedings of the 10th international conference on Mobile systems, applications, and services*, pages 225–238. ACM, 2012.
[42] HUAWEI. White Paper, Service-Oriented 5G Core Networks. https://tinyurl.com/y73y5d4z, 2017.
[43] A. P. Iyer, L. E. Li, and I. Stoica. Celliq: Real-time cellular network analytics at scale. In *NSDI*, pages 309–322, 2015.
[44] A. P. Iyer, I. Stoica, M. Chowdhury, and L. E. Li. Fast and accurate performance analysis of lte radio access networks. *arXiv preprint arXiv:1605.04652*, 2016.
[45] J. Kempf, B. Johansson, S. Pettersson, H. Lüning, and T. Nilsson. Moving the mobile evolved packet core to the cloud. In *Wireless and Mobile Computing, Networking and Communications (WiMob), 2012 IEEE 8th International Conference on*, pages 784–791. IEEE, 2012.
[46] J. Khalid and A. Akella. Streamnf: Performance and correctness for stateful chained nfs. *arXiv preprint arXiv:1612.01497*, 2016.
[47] E. Kohler, R. Morris, B. Chen, J. Jannotti, and M. F. Kaashoek. The click modular router. *ACM Transactions on Computer Systems (TOCS)*, 18(3):263–297, 2000.
[48] S. Kulkarni, N. Bhagat, M. Fu, V. Kedigehalli, C. Kellogg, S. Mittal, J. M. Patel, K. Ramasamy, and S. Taneja. Twitter heron: Stream processing at scale. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*, pages 239–250. ACM, 2015.
[49] L. E. Li, Z. M. Mao, and J. Rexford. Toward software-defined cellular networks. In *Software Defined Networking (EWSDN), 2012 European Workshop on*, pages 7–12. IEEE, 2012.
[50] Y. Li, Z. Yuan, and C. Peng. A control-plane perspective on reducing data access latency in lte networks. In *ACM MobiCom*, 2017.
[51] H. Lindholm, L. Osmani, H. Flinck, S. Tarkoma, and A. Rao. State space analysis to refactor the mobile core. In *Proceedings of the 5th Workshop on All Things Cellular: Operations, Applications and Challenges*, pages 31–36. ACM, 2015.
[52] Microsoft. Azure Functions. https://azure.microsoft.com/en-us/services/functions/.
[53] A. Mohammadkhan, K. Ramakrishnan, A. S. Rajan, and C. Maciocco. Cleang: A clean-slate epc architecture and controlplane protocol for next generation cellular networks. In *Proceedings of the 2016 ACM Workshop on Cloud-Assisted Networking*, pages 31–36. ACM, 2016.
[54] M. Moradi, Y. Lin, Z. M. Mao, S. Sen, and O. Spatscheck. Softbox: A customizable, low-latency, and scalable 5g core network architecture. *IEEE Journal on Selected Areas in Communications*, 36(3):438–456, 2018.
[55] NEC. White Paper, Making 5G a Reality. https://tinyurl.com/yd5p74wt, 2018.
[56] L. Neumeyer, B. Robbins, A. Nair, and A. Kesari. S4: Distributed Stream Computing Platform. In *Proc. IEEE International Conference on Data Mining Workshops*, 2010.
[57] N. Nikaein, M. K. Marina, S. Manickam, A. Dawson, R. Knopp, and C. Bonnet. Openairinterface: A flexible platform for 5g research. *ACM SIGCOMM Computer Communication Review*, 44(5):33–38, 2014.
[58] NOKIA. White Paper, Designing Cloud-Native 5G Core Networks, 2017.
[59] L. Osmani, H. Lindholm, B. Chemmagate, A. Rao, S. Tarkoma, J. Heinonen, and H. Flinck. Building blocks for an elastic mobile core. In *Proceedings of the 2014 CoNEXT on Student Workshop*, pages 43–45. ACM, 2014.
[60] A. Panda, S. Han, K. Jang, M. Walls, S. Ratnasamy, and S. Shenker. Netbricks: Taking the v out of nfv. In *OSDI*, pages 203–216, 2016.
[61] M. Pozza, A. Rao, A. Bujari, H. Flinck, C. E. Palazzi, and S. Tarkoma. A refactoring approach for optimizing mobile networks. In *Communications (ICC), 2017 IEEE International Conference on*, pages 1–6. IEEE, 2017.
[62] G. Premsankar, K. Ahokas, and S. Luukkainen. Design and implementation of a distributed mobility management entity on openstack. In *Cloud Computing Technology and Science (CloudCom), 2015 IEEE 7th International Conference on*, pages 487–490. IEEE, 2015.
[63] Z. A. Qazi, M. Walls, A. Panda, V. Sekar, S. Ratnasamy, and S. Shenker. A high performance packet core for next generation cellular networks. In *Proceedings of the Conference of the ACM Special Interest Group on Data Communication*, pages 348–361. ACM, 2017.
[64] M. T. Raza, D. Kim, K.-H. Kim, S. Lu, and M. Gerla. Rethinking lte network function virtualization. In *IEEE ICNP*, 2017.
[65] S. Rosen, H. Luo, Q. A. Chen, Z. M. Mao, J. Hui, A. Drake, and K. Lau. Discovering fine-grained rrc state dynamics and performance impacts in cellular networks. In *Proceedings of the 20th annual international conference on Mobile computing and networking*, pages 177–188. ACM, 2014.
[66] S. B. H. Said, M. R. Sama, K. Guillouard, L. Suciu, G. Simon, X. Lagrange, and J.-M. Bonnin. New control plane in 3gpp lte/epc architecture for on-demand connectivity service. In *Cloud Networking (CloudNet), 2013 IEEE 2nd International Conference on*, pages 205–209. IEEE, 2013.
[67] M. R. Sama, S. Beker, W. Kiess, and S. Thakolsri. Service-based slice selection function for 5g. In *Global Communications Conference (GLOBECOM), 2016 IEEE*, pages 1–6. IEEE, 2016.
[68] Y. Takano, A. Khan, M. Tamura, S. Iwashina, and T. Shimizu. Virtualization-based scaling methods for stateful cellular network nodes using elastic core architecture. In *Cloud Computing Technology and Science (CloudCom), 2014 IEEE 6th International Conference on*, pages 204–209. IEEE, 2014.
[69] A. Toshniwal, S. Taneja, A. Shukla, K. Ramasamy, J. M. Patel, S. Kulkarni, J. Jackson, K. Gade, M. Fu, J. Donham, et al. Storm@ twitter. In *Proceedings of the 2014*

ACM SIGMOD international conference on Management of data, pages 147–156. ACM, 2014.

[70] M. Zaharia, M. Chowdhury, T. Das, A. Dave, J. Ma, M. McCauley, M. J. Franklin, S. Shenker, and I. Stoica. Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing. In *Proceedings of the 9th USENIX conference on Networked Systems Design and Implementation*, pages 2–2. USENIX Association, 2012.