The Cost of Stateless Network Functions in 5G

Umakant Kulkarni
Purdue University
ukulkarn@purdue.edu

Amit Sheoran
asheoran@alumni.purdue.edu

Sonia Fahmy
Purdue University
fahmy@purdue.edu

ABSTRACT

The adoption of a cloud-native architecture in 5G networks has facilitated rapid deployment and update of cellular services. An important part of this architecture is the implementation of 5G network functions statelessly. However, statelessness and its associated serialization and de-serialization of data and database interaction significantly increase latency. In this work, we take the first steps towards quantifying the cost of statelessness in a cloud-native 5G system. We compare the cost of different state management paradigms, and propose a number of optimizations to reduce this cost. Our preliminary results indicate that sharing user state among 5G functions reduces the overall cost by on an average of 10% in experiments with 100 to 1000 simultaneous requests. Optimizations such as non-blocking calls and custom database APIs also reduce cost, albeit to a lower extent. We believe that the paradigms proposed in this paper can aid operators and software vendors as they design cloud-native 5G networks.

CCS CONCEPTS

• **Computer systems organization** → **Cellular architectures**; Cloud computing; • **Networks** → **Network management**; *Mobile networks.*

KEYWORDS

Cloud-native architectures; 5G; Cellular networks; Stateless Network Functions

ACM Reference Format:
Umakant Kulkarni, Amit Sheoran, and Sonia Fahmy. 2021. The Cost of Stateless Network Functions in 5G. In *Symposium on Architectures for Networking and Communications Systems (ANCS '21), December 13–16, 2021, Lafayette, IN, USA.* ACM, New York, NY, USA, 7 pages. https://doi.org/10.1145/3493425.3502749

ANCS '21, December 13–16, 2021, Lafayette, IN, USA
© 2021 Copyright held by the owner/author(s).
ACM ISBN 978-1-4503-9168-9/21/12.
https://doi.org/10.1145/3493425.3502749

1 INTRODUCTION

The 5G System (5GS) architecture (section 4 of [18]) recommends employing *stateless* network functions (NFs) or microservices, interacting via service-based interfaces. One way to follow this recommendation is by deploying 5G network functions in a cloud-native fashion with containers [20]. A network function in 5GS can operate statelessly by storing the current state of the end user device (referred to as user equipment, or UE) in a remote database known as the Unstructured Data Store Function (UDSF). However, this statelessness comes at a performance cost.

The performance cost of statelessness corresponds to additional processing and interaction with a remote database. Specifically, in stateful deployments, 3GPP-defined procedures simply store state in the cache or main memory, whereas in stateless deployments, the current state is stored in the database. A network function needs the current state when it receives a trigger to modify the UE context (state). A stateless NF fetches the latest state from the database (the UDSF), processes the triggered request, and then updates the database with the new state. Stateless implementations take a longer time to process the request due to additional processing for data storage and retrieval.

In this paper, we explore the design space of state management for 5G network functions, and quantify the cost of different design choices using prototype experiments with an open-source 5GS implementation. To the best of our knowledge, this is the first work to take an in-depth look at the performance overhead of stateless 5G network functions. We investigate two types of stateless functions, procedurally stateless and transactionally statelessness, and propose a number of optimizations to mitigate the performance cost of transactional statelessness. We find that state sharing among 5G functions reduces the cost of transactional statelessness by an average of 10% in our experiments with 100 to 1000 simultaneous requests. Using non-blocking or custom database APIs also reduces costs, albeit to a lower extent. Optimizations proposed in this paper do not make changes to the 3GPP [1] architecture and follow the design guidelines of the N18 interface, as described in [14].

2 STATELESS 5GS FUNCTIONS

In this section, we partially explore the design space of state management in 5GS, depicted in Figure 1.

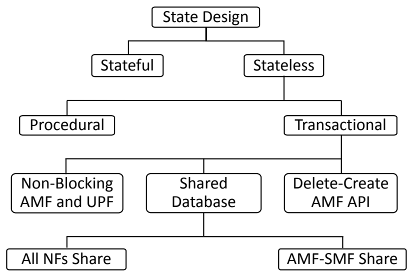


Figure 1: Partial design space of state management approaches in 5GS

Several system *procedures* provide 5G services to an end user [17]. These procedures are executed as a sequence of *transactions* that transfer the state of the UE from one function to the next in a service function chain. A *transaction* is a message interaction between two functions that involves exactly one request and its corresponding response. A transaction is triggered when one function sends a request to another function, either to process the request or to update the state. ~~Transactions must ensure that state is synchronized among all functions.~~

2.1 Procedural and Transactional Statelessness

Statelessness among the 5G functions: Access and Mobility Management Function (AMF), Session Management Function (SMF), and User Plane Function (UPF), can be implemented either at the procedural level or at the transaction level.

Procedural statelessness refers to storing the UE state in the database at the end of a procedure. During the operation of a procedure, the function stores the state into local cache or memory. Once the entire procedure is completed, the function updates the state in the database.

Transactional statelessness is more fine-grained: the function stores the UE state after completing each individual transaction.

2.2 Drawbacks of Procedural Statelessness

Although procedural statelessness can speed up control-plane signaling, it suffers from two important drawbacks. *First*, all transactions in a given procedure must be processed by the same instance. Thus, operators need to implement an additional intermediate node to function as a load balancer. This UE-aware load balancer routes the incoming control-plane messages to the instance of a function containing the UE state from previous transactions. *Second*, procedural statelessness is less resilient to node failures or function restarts since a function may lose the state of the UE. Network functions that precede it in the service chain need to re-transmit the messages to recover. To avoid delays caused by such cascading failures and to take advantage of fully distributed and stateless functions, it may be beneficial to adopt transactional statelessness. Therefore, we focus on better understanding and reducing the cost of transactional stateless in the remainder of this paper.

2.3 Cost of Transactional Statelessness

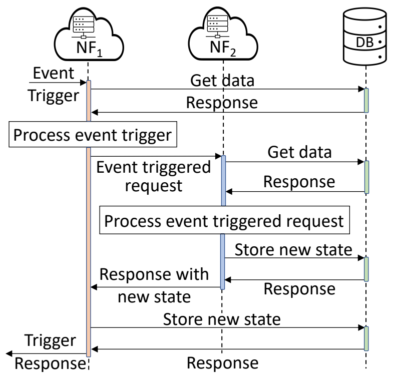
All participating network functions in a stateless transaction need to fetch the latest state from the remote database. When an NF (NF_1 in Figure 2a) receives an event trigger, it requests the latest data required to process the event from the database. After receiving the response from the database, NF_1 processes the request and may trigger a new request to another NF (NF_2) in the service function chain. NF_2 and successive NFs in the service chain follow the same process.

When the last NF (say NF_n) in the service function chain processes the request, it stores the newly processed data in the database and sends back the response to NF_{n-1}. All NFs from NF_{n-1} to N_1 now receive responses, store data, and respond in the reverse order until the very first NF (NF_1) in the service function chain receives the response from NF_2, stores the data in the database, and sends the response back to NF_0. The number of messages exchanged with the database is $2 \times n$ for the read operations plus $2 \times n$ for the write operations, for a total of $4 \times n$ messages for a service function chain of n functions (where n does not include the triggering function NF_0).

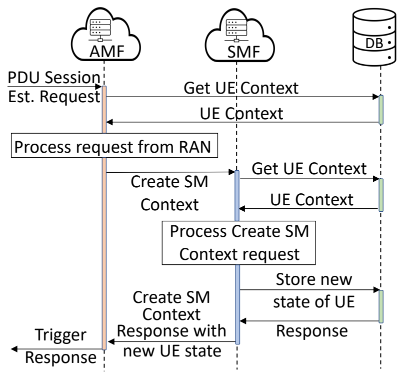
3 REDUCING THE COST OF TRANSACTIONAL STATELESSNESS

Shared database. In our example in Section 2.3, NF_1 and NF_2 independently write the state to the database. If NFs can share the state in the database, it is sufficient for a single NF to write the data to the database. In other words, if NF_2 writes the data on behalf of both itself and NF_1, then the write operation by NF_1 is no longer required. If there are n NFs in the service function chain, then the cost saved will be the cost associated with $n-1$ database write operations.

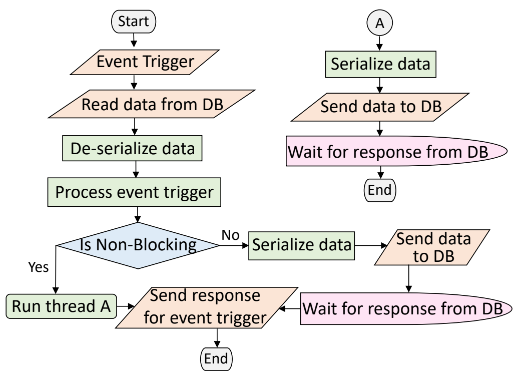
Applying the above idea to cellular network functions, we note that there is always a function in 4G or 5G that maintains the latest and correct UE state. This role is played by the packet gateway in the 4G architecture, and the Session Management Function (SMF) in 5G, as the SMF is the connecting link between the 5G control and user planes. As described in section 6.2 of [18], the SMF receives access network data from the AMF, and configures the UPF. Hence, the state maintained by the AMF or UPF is a subset of the state maintained by the SMF. We take advantage of this fact, and propose transactional statelessness, sharing the state in the database to reduce write operations.



(a) Transactional statelessness



(b) Sharing among AMF and SMF



(c) Non-blocking paradigm

Figure 2: Transactionally-stateless paradigms

Consider the example of the "Create SM Context" transaction, executed as a part of several 5G procedures. This transaction is between the AMF and SMF, where the AMF sends a request to the SMF when it receives a trigger from the gNB (i.e., the cell tower). The SMF processes the request received from the AMF and sends the response back. The processing involves other interactions (e.g., with AUSF, UDM, and PCF) by both the AMF and SMF, but we will omit these for the purposes of this discussion.

Figure 2b depicts the "Create SM Context" transaction. Upon receiving the request from the AMF, the SMF processes it and writes the new UE state to the database. The response to the AMF contains the same data that the SMF has just stored into the database. When the AMF receives the response from the SMF, it writes the same data to the database and sends data back to the gNB. In other words, with transactional statelessness, both functions store the same data at the end of their respective transactions.

We propose that a *single function* store data on behalf of both functions in this case. Specifically, the SMF writes the data to the database and the AMF can later access that same data. Hence, when a new request arrives as a part of a new transaction, the AMF will fetch the data from this shared database, which was written by the SMF in the earlier transaction. This reduces the number of database writes by one for each such transaction.

Non-blocking AMF and UPF. An alternative optimization can be performed through parallelism. As defined in the 3GPP specifications [15, 16], 5G NFs share data or UE state by sending JSON [6] payloads using a REST API over the service-based interfaces for processing calls or events. Additional latency is introduced by serialization and de-serialization of binary and JSON data, and converting it to a database-compatible format, e.g., BSON [3] if the database is implemented using mongoDB [8].

To reduce this processing latency incurred by data structure translations, we propose a paradigm where the AMF and the UPF make non-blocking calls to the database while writing data, whereas the SMF continues to make blocking calls to the database (Figure 2c). This ensures that the SMF still maintains the correct UE state at the transactional level, and all three functions are fully stateless without violating cloud-native principles.

Delete-create AMF API. A final optimization we will explore is the API to update the database. NoSQL databases like MongoDB use WiredTiger storage engine [12] which reads and writes data through in-memory B+ tree data structures [13]. When an update adds new fields, the B+ tree executes search, insert, swap or move operations of key-value pairs across its leaf nodes. This makes the total time complexity $C \times O(log\ n)$ where n is the number of keys and C is the number of new fields. The larger the document and the number of new fields, the longer it takes to modify it. To address this performance issue, we will explore using two separate APIs (delete and then create) to perform the update operation with time complexity $1 + O(log\ n)$.

All transactions between the SMF and other NFs that process calls can be optimized as discussed above. We list these transactions as defined by 3GPP in Table 1.

4 EVALUATION

The goals of our experiments are to quantify the costs of procedural and transactional statelessness (Section 2), and gain a preliminary understanding of the benefits of optimizations to transactional statelessness (Section 3).

4.1 Setup

We deploy the components of 5G cellular system architecture in a cloud-native fashion using `open5gs` [5], an open-source 5GS implementation written in C. To simulate RAN

	NF	Transaction
1	AMF	Create SM Context
2	AMF	Update SM Context
3	AMF	Release SM Context
4	AMF	N1/N2 Message Transfer
5	UPF	PFCP Session Establishment
6	UPF	PFCP Session Modification
7	UPF	PFCP Session Deletion

Table 1: SMF transactions with other NFs

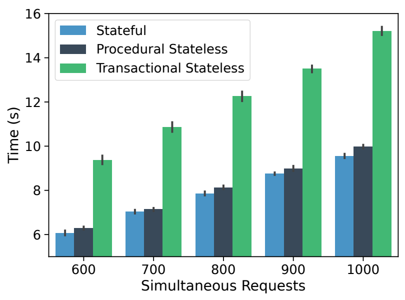


Figure 3: Stateful and stateless paradigms

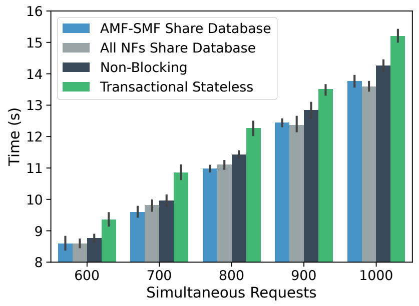


Figure 4: Transactionally-stateless optimizations

interfaces for the UE and gNB, we use the open-source tool `UERANSIM` [4] which is written in C++.

We use CloudLab [29] servers where we create a network with 15 nodes, 11 of which constitute a Kubernetes cluster with one master and ten worker nodes. We deploy `open5gs` v2.3.6 along with mongoDB v5.0.3 on the cluster. The remaining four nodes are used for `UERANSIM` v3.2.2, where two nodes are assigned as two separate gNBs and the remaining two as UEs. All nodes are of type m510 [21], equipped with Intel Xeon D-1548 processor supporting x86_64 architecture consisting of 16 CPUs with maximum speed of 2 GHz. The nodes run on 5.4.0-77-generic kernel with Ubuntu 20.04 and Kubernetes v1.22.4. We use Helm charts from the publicly available repository `opensource-5g-core-service-mesh` [9] to manage these cloud-native `open5gs` functions.

We make the following changes to the three open-source implementations: (1) `open5gs`: Since the default `open5gs` implementation is stateful, we modify `open5gs` to add interactions with the database for the transactions listed in Table 1. (2) `UERANSIM`: Instead of creating a TUN interface for each UE, we modify `UERANSIM` to write the UE IP address to a file to ensure that the system is not slowed down due to 100s of interfaces. (3) `opensource-5g-core`: We modified the Helm charts to match the templates with our Kubernetes cluster environment. All our changes are available at https://github.com/UmakantKulkarni/stateless5g.

4.2 Methodology

We experiment with the UE-initiated PDU session establishment request procedure, defined in section 4.3.2.2 of [17]. We trigger this procedure through "UE registration," which involves communication between almost all functions within the 5G system (UE, RAN, AMF, SMF, UPF, PCF, UDR, UDM, NRF, NSSF and AUSF).

We compare seven alternatives: (1) Stateful NFs, (2) Fully transactionally-stateless NFs, (3) Procedurally-stateless NFs, (4) All NFs share the database, (5) Only AMF and SMF share the database, (6) Non-blocking AMF and UPF APIs, and (7) Delete-Create AMF API. We vary the number of simultaneous requests made between 100 and 1000, in steps of 100. We repeat each set of experiments ten times and compute system and application-level statistics for successful runs.

4.3 Results

4.3.1 Procedural and Transactional Statelessness. As discussed earlier, with procedural statelessness, NFs read from database when they receive a trigger at the start of a procedure. With transactional statelessness, NFs read and write during each transaction listed in Table 1. We compare these two paradigms with stateful NFs based on the time taken by the 5GS to complete a UE-initiated PDU session establishment request procedure for a given number of simultaneous requests. The time taken is the difference between the time when the first message arrives at the AMF from the gNB and the time when the last response arrives at the AMF from the SMF. We choose the AMF for this time computation because it is the first NF in the service function chain responsible for processing the procedure.

Figure 3 plots the mean values and 95% confidence intervals of the time taken for the cases of 600 to 1000 simultaneous requests. Since there is a single read and a single write from/to the database per NF, no significant time difference between the stateful and procedurally-stateless implementations is observed. The mean time difference is only 3%, which can go up to 10% maximum (for 100 to 1000 requests). In contrast, if we compare the stateful and transactionally-stateless implementations, the mean time difference is around 56% which goes up to 71%. This is because there are seven additional read and write interactions with the database by the three NFs (the AMF, SMF and UPF).

4.3.2 Sharing Alternatives. We compare the transactionally-stateless paradigm with the first optimization that we propose in Section 3, i.e., sharing UE state in the database among the NFs. We experiment with two alternatives for sharing (Figure 1): (1) The AMF, SMF and UPF all share the UE state

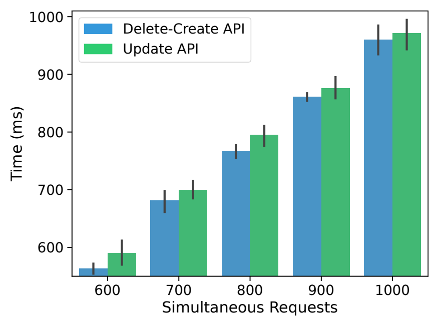


Figure 5: Time spent by mongoDB with update and delete-create

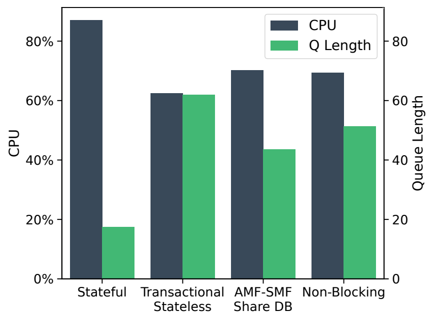


Figure 6: CPU and queue length at 1000 sessions

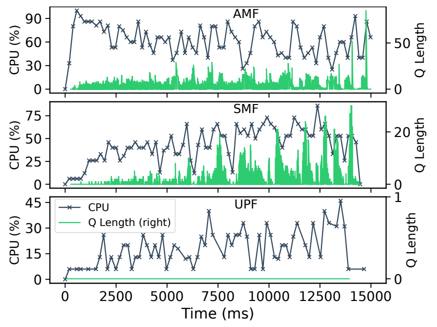


Figure 7: Time series of CPU and queue length at 1000 sessions

stored in the database, and (2) Only the AMF and SMF share the UE state stored in the database.

Although the first approach where all three NFs share the UE state is ideal, it may not be practical from the network operator’s point of view. This is because, as specified in section 5.13 of [18], the UPF should be placed close to the UE to support edge computing. In contrast, the database should be on the same premises or vicinity as other control-plane NFs (AMF and SMF). Thus, we experiment with both types of implementations to explore the latency implications if the UPF is not located close to the database.

From Figure 4, we see that both types of sharing yield gains by reducing the number of write transactions to the database. The improvements over traditional transactional statelessness are 10% on average, and can go up to 21% (for 100 to 1000 requests).

4.3.3 Non-Blocking AMF and UPF. We now experiment with an approach where the AMF and UPF make non-blocking calls to the database for the second, fifth, and sixth transactions listed in Table 1. We create a thread pool during NF initialization, and use a thread from this pool (stack) to serialize and upload the data to the database. Once the upload is complete, the thread is pushed back onto the stack. This approach saves time over creating and destroying a thread during each transaction.

We plot the time taken to complete a procedure in Figure 4 and find that this approach performs better than default transactional statelessness with an average reduction of more than 6%, and a maximum reduction of 17% (for 100 to 1000 requests). We may also benefit from making non-blocking calls with other transactions listed in Table 1.

4.3.4 Database APIs. We explore another optimization by exercising the fourth transaction in Table 1 in two ways: (1) using the update API defined in the mongoDB driver [7], and (2) using a sequence of delete and create APIs. Although the number of interactions with the database increases with the delete-create API, the total (read + write) time that mongoDB spends on these operations is on an average 3.5% lower than the update API (Figure 5). Thus, we see reduction in the procedure completion time by on average 2.5%, and up to 12% (for 100 to 1000 requests). This may be important when the 5G control plane is deployed on a public cloud where service providers are charged based on time consumed on each database compute as described in [2] and [10].

4.3.5 System Metrics. Since additional computing is involved in case of transactional statelessness for serializing and de-serializing data, we expect to see a higher average CPU usage by NFs in this case. We find that the opposite is true. Therefore, we take an in-depth look at system metrics to investigate the reasons. For each event trigger, we use `epoll` to calculate the application queue lengths and compare it to the CPU usage for each NF during the course of a procedure for four paradigms: stateful, transactionally-stateless, non-blocking transactionally-stateless and stateless with shared UE state among the AMF and SMF. As seen in Figure 6, as the cost of statelessness increases, the average NF CPU usage (over 300 ms) decreases. To explain this behavior, we plot the time series for one of our transactionally-stateless experiments to observe the change in queue length and CPU for AMF (top), SMF (middle) and UPF (bottom).

Figure 7 shows that the CPU spikes immediately after an increase in the queue length. This trend starts at the first NF in the service chain (AMF) and propagates to the following NFs (SMF, UPF) in the chain. As SMF and UPF process calls and serialize/de-serialize the data, AMF waits for responses from them. Thus, a stateless AMF spends a longer time waiting, making its average CPU usage per unit time lower than with the stateful paradigm. This can be confirmed from Figure 7, where CPU and queue length spikes move to the right as we traverse the service function chain from the first to last NF. The last spike in the AMF queue length represents processing the responses from SMF and UPF on which the AMF was waiting.

The queue lengths for the non-blocking and transactionally-stateless paradigms in Figure 6 confirm that the total number of messages is the same. The non-blocking paradigm does not wait for responses from the database, and hence it processes the sessions faster than the transactionally-stateless paradigm. This can be confirmed by the higher CPU usage observed in Figure 6 and the lower time taken in Figure 4.

5 DISCUSSION AND FUTURE WORK

5G with stateless NFs: Our analysis indicates that frequent state fetch/update operations performed by stateless NFs significantly contribute to latency. Since a single database instance may store data from multiple NFs within the service function chain, even a small delay for each NF can result in significant latency for the entire procedure. Additionally, since current implementations use JSON-based messages to communicate with the database, NFs incur high overhead during the message serialization and de-serialization process associated with each database interaction.

5GS implementations must therefore minimize database interactions involved in a procedure. This can be achieved by allowing NFs in a service function chain to piggyback user data required to process a user request along with the request. That is, an NF at the beginning of a service function chain can fetch the user state required to complete the transaction and then forward this state to subsequent NFs in the chain. While this may require the network to transmit larger amounts of data, such a solution, coupled with Binary Large Object (BLOB) or in-memory databases and using protocol buffers [11] for serialization, can reduce the impact of database communication latency on performance.

Request-response model with stateless NFs: 5GS uses the standard request-response model in which each request sent receives a response. This entails that both the request and response messages (including failure responses) traverse the entire service function chain. While the request-response model was necessary in systems where the state of a user was maintained (created/updated/deleted) at each NF, this communication model has limited utility in cases in which stateless NFs maintain user state in remote databases. Since stateless NFs must fetch state from the remote database, the NFs can use the database state to infer the success/failure of previous transactions in an ongoing procedure, eliminating the need for propagation of some response messages through the service function chain. This approach would reduce the number of messages required to complete a procedure, but the performance gains, resource utilization, and correctness of network communication in this case needs to be evaluated.

Finally, since the HTTP/2-based REST communication mandates a request-response model, that is, HTTP/2 requires a response to each request, 5G control-plane operations can also benefit from the use of QUIC/gRPC-based communication, e.g., as described in [24].

6 RELATED WORK

State management: Our work follows the principles of stateless network functions described in [22] and [23], which decouple the state from NFs by introducing an external data store. That prior work considers NFs such as NATs, which are simpler in functionality than most 5G network functions. Hence, certain optimizations described in [23], such as avoiding database locking while reading, are not directly applicable to 5G functions.

State management in cellular networks: Our work is inspired by PEPC [27], Contain-ed [30], and DPCM [26], which highlight the benefits of sharing state among network functions. Unlike PEPC which consolidates multiple functions within the 4G control plane into a single slice, we share the UE state stored in the database among different functions. Thus, optimizations described in our work strictly follow 3GPP-recommended design guidelines.

Similar to our work, the lock-based 5G NF design described in [25] uses a global data structure to share the UE state, between AMF threads in their case. As the authors point out, the lockless AMF implementation (described in section 3 of [25]) encounters similar challenges to those described in Section 2.2 since UE-aware routing is required to direct messages to a specific virtualized NF.

Neutrino [19] replicates the UE state from primary to secondary control-plane functions for redundancy in case of NF restarts or failure. However, this non-blocking synchronization takes place at the procedural level, which is less fault tolerant and more resource consuming than the transaction-level optimizations described in this paper.

Parallelism in cellular networks: Similar to our non-blocking optimization, DPCM [26] and logic-based NFs [28] parallelize control-plane procedures in 4G networks. However, our non-blocking approach parallelizes control-plane procedures with database transactions within 5G. Thus, our model does not make changes to inter-NF transactions.

7 CONCLUSIONS

In this paper, we have introduced a number of state management optimizations for 5G networks, and compared their cost to a stateful implementation. Procedural statelessness has a low performance overhead, but has robustness concerns. Transactional statelessness is robust, at the cost of higher performance overhead. This motivates us to propose optimizations to transactional statelessness. Our results indicate that the shared state optimization is the most effective, followed by non-blocking APIs, then delete-create database APIs. Applying these optimizations to the entire call flow would increase the performance gains. We believe that this work serves as a first but important step in exploring stateless paradigms in the 5G system.

REFERENCES

[1] About 3GPP. https://www.3gpp.org/about-3gpp.

[2] Amazon DynamoDB Pricing for Provisioned Capacity. https://aws.amazon.com/dynamodb/pricing/provisioned/.

[3] BSON - Binary JSON Serialization. https://bsonspec.org/.

[4] GitHub - aligungr/UERANSIM: Open source 5G UE and RAN (gNodeB) implementation. https://github.com/aligungr/UERANSIM.

[5] GitHub - open5gs/open5gs: Open5GS is a C-language Open Source implementation for 5G Core and EPC. https://github.com/open5gs/open5gs.

[6] JSON - JavaScript Object Notation. https://www.json.org/json-en.html.

[7] MongoDB C Driver — MongoDB C Driver 1.19.0. http://mongoc.org/.

[8] Mongodb the most popular database for modern apps. https://www.mongodb.com.

[9] opensource-5g-core-service-mesh. https://bitbucket.org/infinitydon/opensource-5g-core-service-mesh/src/main/.

[10] Pricing - Azure Cosmos DB | Microsoft Azure. https://azure.microsoft.com/en-us/pricing/details/cosmos-db/.

[11] Protocol Buffers. https://developers.google.com/protocol-buffers.

[12] Storage Engines — MongoDB Manual. https://docs.mongodb.com/manual/core/storage-engines/.

[13] WiredTiger: Tuning page size and compression. http://source.wiredtiger.com/mongodb-3.4/tune_page_size_and_comp.html.

[14] 3GPP. Study on the Nudsf Service Based Interface. Technical Report (TR) 29.808, 3rd Generation Partnership Project (3GPP), 12 2019. Version 16.0.0.

[15] 3GPP. 5G System; Access and Mobility Management Services; Stage 3. Technical Specification (TS) 29.518, 3rd Generation Partnership Project (3GPP), 6 2021. Version 17.2.0.

[16] 3GPP. 5G System; Session Management Services; Stage 3. Technical Specification (TS) 29.502, 3rd Generation Partnership Project (3GPP), 6 2021. Version 17.1.0.

[17] 3GPP. Procedures for the 5G System (5GS). Technical Specification (TS) 23.502, 3rd Generation Partnership Project (3GPP), 6 2021. Version 17.1.0.

[18] 3GPP. System architecture for the 5G System (5GS). Technical Specification (TS) 23.501, 3rd Generation Partnership Project (3GPP), 6 2021. Version 17.1.1.

[19] Ahmad, M., Jafri, S. U., Ikram, A., Qasmi, W. N. A., Nawazish, M. A., Uzmi, Z. A., and Qazi, Z. A. A low latency and consistent cellular control plane. In *Proceedings of the Annual Conference of the ACM Special Interest Group on Data Communication on the Applications, Technologies, Architectures, and Protocols for Computer Communication* (New York, NY, USA, 2020), SIGCOMM '20, Association for Computing Machinery, p. 648–661.

[20] Chandramouli, D., Chandrashekar, S., Maeder, A., Niemela, T., Theimer, T., and Thiebaut, L. *Next Generation Network Architecture*. John Wiley & Sons, Ltd, 2019, ch. 4, pp. 127–223.

[21] CloudLab. m510 server. https://www.utah.cloudlab.us/portal/show-nodetype.php?type=m510.

[22] Kablan, M., Alsudais, A., Keller, E., and Le, F. Stateless network functions: Breaking the tight coupling of state and processing. In *Proceedings of the 14th USENIX Conference on Networked Systems Design and Implementation* (USA, 2017), NSDI'17, USENIX Association, p. 97–112.

[23] Khalid, J., and Akella, A. Correctness and performance for stateful chained network functions. In *Proceedings of the 16th USENIX Conference on Networked Systems Design and Implementation* (USA, 2019), NSDI'19, USENIX Association, p. 501–515.

[24] Kiran Buyakar, T. V., Agarwal, H., Tamma, B. R., and Franklin, A. A. Prototyping and load balancing the service based architecture of 5g core using nfv. In *2019 IEEE Conference on Network Softwarization (NetSoft)* (2019), pp. 228–232.

[25] Kumar, A., Naik, P., Patki, S., Chaudhary, P., and Vutukuru, M. Evaluating network stacks for the virtualized mobile packet core. In *Proceedings of Asia-Pacific Workshop on Networking (APNet)* (June 2021).

[26] Li, Y., Yuan, Z., and Peng, C. A control-plane perspective on reducing data access latency in lte networks. In *Proceedings of the 23rd Annual International Conference on Mobile Computing and Networking* (New York, NY, USA, 2017), MobiCom '17, Association for Computing Machinery, p. 56–69.

[27] Qazi, Z. A., Walls, M., Panda, A., Sekar, V., Ratnasamy, S., and Shenker, S. A high performance packet core for next generation cellular networks. In *Proceedings of the Conference of the ACM Special Interest Group on Data Communication* (New York, NY, USA, 2017), SIGCOMM '17, ACM, pp. 348–361.

[28] Raza, M. T., Kim, D., Kim, K. H., Lu, S., and Gerla, M. Rethinking LTE network functions virtualization. In *2017 IEEE 25th International Conference on Network Protocols (ICNP)* (Oct 2017), pp. 1–10.

[29] Ricci, R., and Eide, Eric et al. Introducing CloudLab: Scientific infrastructure for advancing cloud architectures and applications. *The magazine of USENIX & SAGE 39*, 6 (2014), 36–38.

[30] Sheoran, A., Sharma, P., Fahmy, S., and Saxena, V. Contain-ed: An NFV micro-service system for containing e2e latency. In *Proceedings of the Workshop on Hot Topics in Container Networking and Networked Systems* (2017), HotConNet '17, pp. 12–17.