AnnaBellaDB: Key-Value Store Made Cloud Native

Mark Szalay*, Peter Matray†, Laszlo Toka*
*MTA-BME Network Softwarization Research Group, †Ericsson Research,
szalay@tmit.bme.hu, peter.matray@ericsson.com, toka@tmit.bme.hu

***Abstract*—The cloud-native paradigm has become a well-known approach to ensure the elasticity and reliability of applications running in the cloud. One recurrent motif is the stateless design of applications, which aims to decouple the life-cycle of application states from the life-cycle of individual application instances. Application data is written to and read from cloud databases, deployed close to the application code to ensure low latency bounds on state access. When applying a stateless design, the performance of the cloud service is often limited by the cloud database. In order not to become a bottleneck, database instances are distributed on multiple hosts, and strive to ensure data locality for all application functions. However, the shared nature of certain states, and the inevitable dynamics of the application workload necessarily lead to inter-host data access. If the service is geographically distributed, this is even across data centers and edge servers resulting in a significant delay. To minimize the service performance loss due to the stateless design of applications, we propose a latency and access pattern aware state storage method, called state-layer, that can be easily applied in any kind of key-value store with the ability of deciding where to store replicas in the cluster and measure networking/computing delay. By adapting our solution to Anna, a key-value store from academia, we show the proposed state-layer is ideal to use as a cloud database for storing application states. To foster further research in this area, we make our proof-of-concept solution open-source.**

I. Introduction

The suitability of monolithic applications or Network Functions (NFs) in the telco field has reduced in recent years. The two most critical problems are i) the specific hardware dependency, and ii) the lack of auxiliary functions, such as load balancing already available for cloud applications. The former causes deployment flexibility issues, while the latter typically increases the development and the operation costs of the applications/NFs. In order to avoid these issues, a modern Network Function Virtualization (NFV) ecosystem must be fundamentally stateless; if the virtual NFs do not maintain persistent state on their own, then scale-in/scale-out, and even fail-over events, are less complex to handle, improving overall elasticity, scalability, resiliency, and upgradability [1]–[6]. This is especially true for the carrier-grade telco NFV ecosystems since they must maintain a level of consistent performance in order to meet strict Service-Level Objectives (SLOs), at the order of 1–10 ms delay requirement [5].

Although the performance overhead of the stateless design depends on various factors (such as an application's state access pattern, the performance of the underlying data store and the deployment characteristics), according to our research [5] we can anticipate that the cost of external state access may easily become the limiting factor in the overall NF performance. NFs might use externalized state either for their control or data plane operations. In the latter case, NFs are to access such state for each incoming packets, which may result in a severe performance deterioration in their PPS throughput. Furthermore, this is especially true for edge computing systems where the NFs are distributed geographically and an external access may result in delays of the order of hundreds of milliseconds to core functionality [7]. That is why considering the network latency between NFs and their states is essential when it comes to state externalization.

The operation lag can be decreased by locating data as close to the application instances as possible in terms of latency. Many public cloud databases, both from academia and from industry, offer in-memory state storage for cloud applications, but they typically do not take into account the underlying network properties [8]–[10]. The placement of application states, taking into account the locations of already deployed database instances and NF applications, is of paramount importance to improve application performance.

II. Our proof of concept prototype

In this section we introduce the cloud and/or edge based system where the stateless cloud-native NF instances might run, and propose our latency and access pattern aware state storage architecture that could be used as a state handling service, called *state-layer*, for stateless applications. Let us consider a general cluster of hosts, e.g., physical servers, virtual machines, Kubernetes pods or Docker containers. Each of these hosts may run the user's NFs, and include a database (DB) server instance of the distributed cloud database, i.e., the *state-layer* over the cluster. In order to make the application elastic and resilient, the NFs' internal states need to be externalized into this state-layer.

Existing key-value stores, such as [10]–[15], either concentrate on a single-node setup or use hash functions to place data. In contrast, we argue that placement needs to be delay-, and access-pattern aware to minimize the access time of the externalized states. To meet this goal, we have extended a key-value store called Anna [10] from academia with a State Placement module, a Replacement initiator, and Infrastructure monitors to create a latency and data-access-pattern aware, distributed cloud database that enables the state-layer we envisioned. We call it AnnaBellaDB.

978-3-903176-31-7 © 2020 IFIP

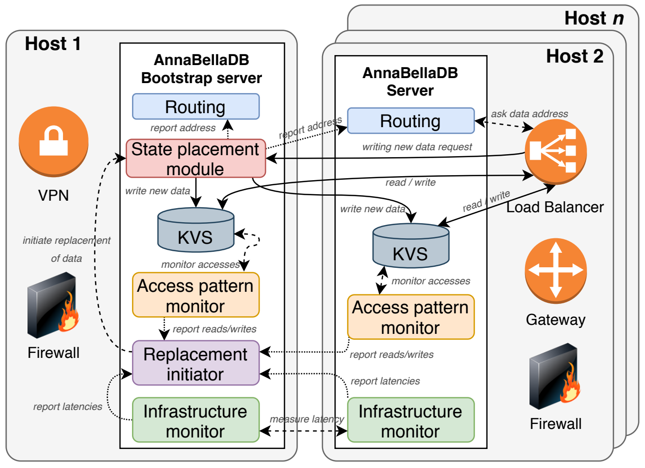

Figure 1. Proposed state-layer architecture

Fig. 1 shows an example cloud-based system. The cluster consists of *n* hosts where stateless NF instances run, e.g., VPN, Firewall, Gateway, and Load Balancer. Besides the applications, each host contains an AnnaBellaDB server instance (white squares). Two types of DB servers can be distinguished in the AnnaBella cluster: a single privileged Bootstrap server and the others, ordinary Key-Value servers. Both types include *Routing*, *KVS* (Key-Value Store), *Access pattern monitor* and *Infrastructure monitor* architecture elements, while the Bootstrap server further contains *State Placement module* and *Replacement initiator*.

The *Infrastructure monitors* ping each other to measure the perceived latency between cluster hosts, and periodically report them to the Bootstrap server. The measured latency includes both network and computational delay between AnnaBellaDB servers, consequently, the effect of high load is also monitored. The *Access pattern monitors* track the number of reads and writes of each data entry stored in the local *KVS* individually, and report these access patterns to the *Replacement initiator*. The report frequency of both monitor components is configurable by the state-layer operator in order to fine-tune the reaction time of AnnaBellaDB for infrastructure and data access changes.

NF instances, e.g., Load Balancer, ask the *Routing* component for the location of the requested value. This module returns the address of the AnnaBellaDB server instance where the certain state can be found. If a key-value store client library is applied in the application, it can take care of caching the address for future requests, and targets the appropriate *KVS* instance to perform the access operation (read or write). If the requested state has not been stored yet (e.g., it is the first write of a data), the *Routing* component returns the address of the Bootstrap server, where the *State Placement module* runs, waiting for new entries to be stored.

The *State placement module* contains the data location optimization logic, detailed in [4], that decides where to store, i.e., in which *KVS* of the cluster, the requested data, based on i) the latency across cluster elements and ii) the historic access pattern of the requested state. For new entries the latter

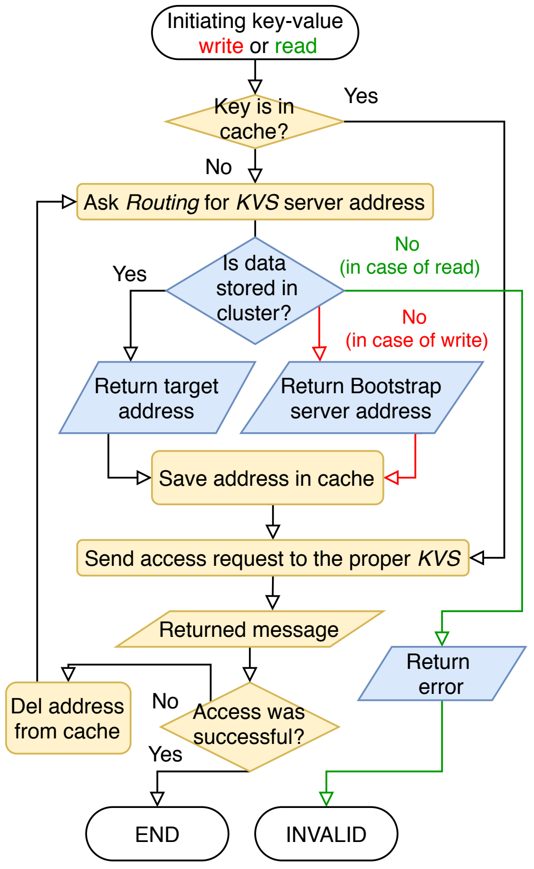

Figure 2. Flow chart of how reads (green) and writes (red) are performed in AnnaBellaDB

information does not exist, but as time passes the *Access pattern monitors* collect this metadata. The access pattern plays an important role in the reoptimization of data locations, when we want to migrate the - typically shared - state within the cluster to minimize its access time. Our applied migrating heuristic method is based on a QP optimization problem, published in [4], which is proven to be NP-hard. The *State placement module* also reports the target *KVS* address(es) to each *Routing* component. It is easy to see the less frequently it reports, the more likely *Routings* hold invalid addresses where the data is located.

The *Replacement initiator* triggers data migration within the DB cluster. When data access pattern changes exceed a predefined threshold (e.g. the access rate of a data is increased or decreased more than 20%), this element initiates the migration of the affected entries. This module ensures the mobility of the states and enables us to move data close to the application instances from where they are accessed the most frequently.

For the ease of understanding how reads and writes are performed in AnnaBellaDB, we present the operational steps in a flow chart in Fig. 2. The red lines represent the execution flow of writes, while the green ones depict the read process. The NF executes the operations in yellow and the *Routing* module is responsible for the blue components.

The first step of accessing, e.g., reading or writing, a data entry is checking if the target *KVS* address, where the data is located, is stored in the local cache or not. If it is not stored, AnnaBella client sends an *address request* message to the local *Routing* component. The Routing checks whether the data already exists in the DB cluster and returns the responsible server address if it does. Otherwise, in case of a write, this is the moment to save new data into the DB cluster, so it returns the Bootstrap server address. In case of a read, it returns an error as the NF tries to read a non-existing entry.

The NF saves the given target address and sends the read/write request to the proper *KVS* instance. At this point, there are three possible events: 1) the request is sent to the Bootstrap server where the *State placement module* stores the data successfully, 2) it is sent to the relevant server which preforms the requested operation, and 3) the read/write request is sent to a server which is no longer responsible for the data, e.g., the data was migrated to another *KVS*. In this last case, the access fails, so the NF deletes the cached address related to the data, and restarts the process by requesting from the local *Routing*. Since the *State placement module* periodically reports the *Routings* about any modification of data placement in the DB cluster, the access will be successfully performed eventually.

III. Performance evaluation

In order to identify corner cases in which performance differences appear between the existing solutions and the proposed AnnaBellaDB, we made some synthetic measurements highlighting the advantages of delay and access-pattern aware data placement over the legacy hash function based solutions. Since our prototype is based on Anna which places data according to a hash function and not to their access pattern, we consider its performance as a proper basis for comparison. Our goal is twofold: i) show the performance of AnnaBellaDB in terms of the access time of a data and ii) examine which type of infrastructure (cloud or edge computing systems) is a good context for our solution to maximize its benefits.

We emulated two types of cluster topologies in terms of network latency: i) a geographically distributed edge/cloud network called *interDC*, and ii) a data-center network named as *intraDC*. Between the hosts we apply 5 ms delay in the former, while for the *intraDC* network, according to [16], 300 μs was used. With these latency characteristics, both cluster setups included four Docker container hosts (*kvs1 - kvs4*), on which the NF instances are launched one by one. Each host runs an AnnaBellaDB instance which altogether form the state-layer service for the NFs.

In order to get the analysis of the proposed state store, we made emulation measurements on a 24-core Intel Xeon E5-2620 @ 2.00GHz server with 64 GB RAM. All containers (with the NFs and AnnaBellaDB components in them) run on this single physical machine and we applied the traffic control on the containers' network interfaces to enable the above described delays between node pairs. We used synthetic measurements and NFs in which we emulated cloud manager-triggered NF scale-out events, resulting in access-pattern changes in time. In the emulations we measured two metrics of the state-layer to quantify its performance: *data access time*, i.e., how much time is necessary to access (read or write) the data, and *throughput* [number of accesses/second], i.e., how many successful accesses happened within a second by the NFs. The shorter the access time, the more suitable our solution is for a state-layer: in case of negligible added latency, stateless NFs reach the performance of stateful ones.

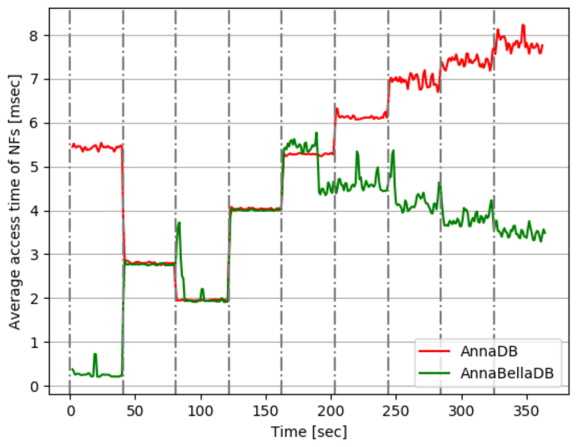

Figure 3. The average access times of NFs

A. Data access time in interDC network

In our first measurement, we used NFs which read ten times and write one time (a certain state) from/to the state-layer. We launched the NF instances one after another: *NF0* on *kvs1*, *NF1* and *NF2* on *kvs2*, and from *NF3* to *NF8* on *kvs3*. The performance results are shown in Fig. 3 where the *x-axis* depicts the time in seconds. The dashed vertical lines represent the moments when a NF has been launched, i.e., the first *NF0* was started at time 0, while the last one *NF8* was launched at the 325th second. Finally, the *y-axis* shows the average of all access times of all NFs in a 2s wide sliding window. The red line represents the Anna access time, and the green one is the result of AnnaBellaDB. As a reminder, the lower the value, the more efficient the database as a state-layer. The measurement can be divided into three phases: 1) between 0 and 40 seconds, 2) between 40 and 200 and 3) from 200 sec. In the first phase, the placement module of AnnaBellaDB decided to save the state locally (in *kvs1*) contrary to Anna: it was stored in *kvs2* resulting in longer access time. In the second phase the two solutions more or less operated similarly. Within this time, the data is stored in *kvs2* (in case of AnnaBellaDB, thanks to the replacement initiator, the data was migrated from *kvs1* resulting in a spike in the access time after the third NF deployment). After the fifth NF instance has launched, our solution migrated the data to *kvs3* from where the majority of accesses (*NF3*, *NF4* and *NF5*) originated, resulting in lower average access time of the NFs.

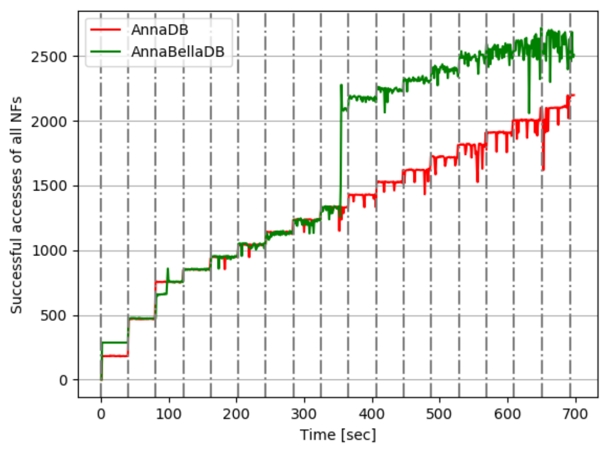

Figure 4. Throughput (maximum number of data accesses per second)

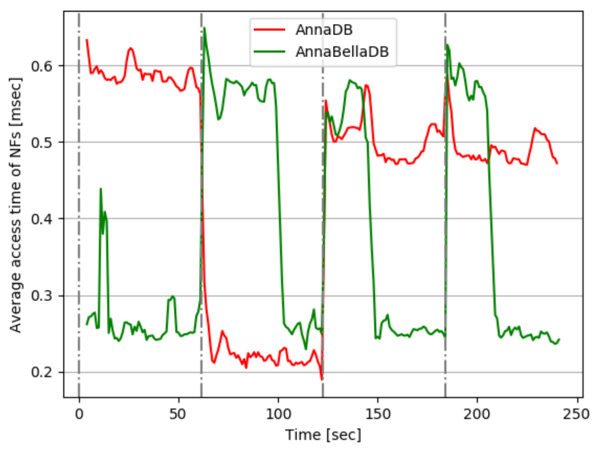

Figure 5. The average access times of NFs

B. Data access throughput in interDC

In the second measurement, we examined the throughput of our proposed DB, i.e., how many state access can be executed during a second by the NF instances. In this case, the NF we used reads a state 260 times and writes it 26 times per second. This type of NF instances are launched periodically: one on *kvs1*, two on *kvs2*, six on *kvs3* and nine on *kvs4*. The obtained results can be seen in Fig. 4, in which this time the *y-axis* shows the sum of successfully completed data accesses by all NF instances within the given second. The higher the DB throughput, the more suitable the DB solution for stateless NFs as a state-layer. Until the 9th launched NF, the examined approaches are similar to each other, but like in the previous measurement, when the *Replacement initiator* of AnnaBellaDB moves the state to the appropriate host, the number of local accesses significantly increases. At this moment (350s), one NF is located on *kvs1*, two on *kvs2* and six on *kvs3*, consequently, when AnnaBellaDB migrated the data from *kvs2* to *kvs3*, six NF are capable of accessing it locally at the same time. This is the reason for the significant performance improvement in our proof-of-concept solution. In contrast, Anna stored the data in *kvs2* resulting in most of the NF instances performing external accesses. However, the remote accesses are limited by the network latency.

C. Data access time in intraDC network

We also examined how the proposed state-layer behaves in the *intraDC* environment. First, we repeated the evaluation described in Sec. III-A, but for now the preliminary network delay was configured for 300 μs. We found, there is no considerable difference in the two databases' performance since the low *intraDC* network delays are comparable to a processing latency of the servers. The obtained statistical metrics of data access times during the entire measurement are summarized in the Table I.

Table I
DATA ACCESS STATISTICS IN μs IN CASE OF *intraDC* NETWORK.

	min	Q1	median	mean	Q3	max
Anna	0.146	0.236	0.517	0.477	0.558	22.99
AnnaBellaDB	0.150	0.226	0.478	0.535	0.537	35.98

However due to the fact that AnnaBellaDB is optimised to efficiently handle the dynamics of NFs, we also examined the corner case when an NF is continuously moving from one host to another. In this scenario, the NF instance was initially deployed to *kvs1* and migrated forward every minute to *kvs2*, *kvs3*, and finally to *kvs4*. We used the same NF as in Sec. III-A that reads ten times and writes only once. The obtained results are presented in Fig. 5. In this evaluation only one NF exists and the vertical lines represent the instances when the application was migrated between the hosts. Anna stored the data in *kvs2* during the measurement, while AnnaBellaDB initially saved it into *kvs1* and migrated it to the appropriate host on every occasion when the NF was migrated. To conclude, in the case of AnnaBellaDB, the NF always has the opportunity, after it was migrated, to access the state locally, while in case of Anna this happens only between 60 and 120 seconds.

To see more details, video and GitHub links are available at https://netsoft.gsuite.tmit.bme.hu/demos/annabelladb.

IV. CONCLUSION

The existing key-value stores do not perform data migration in order to minimize the data access time for the applications. This fact gave us the motivation to create our own latency and access pattern aware database solution that could be ideal for telco use cases: our proof-of-concept prototype forms a state-layer, where the cloud-native stateless NFs are able to externalize their internal states to enable complete elasticity and reliability. Comparing to another key-value store from the academia, our proposed solution can achieve more efficient state access time and state access throughput performance in edge-cloud systems due to the data placement optimization. In single cloud environments the two solutions differ only in the case of moving NFs (e.g. due to a node failure or NF migration initiated by the cloud provider).

REFERENCES

[1] M. Kablan, B. Caldwell, R. Han, H. Jamjoom, and E. Keller, "Stateless network functions," in *Proceedings of the 2015 ACM SIGCOMM Workshop on Hot Topics in Middleboxes and Network Function Virtualization - HotMiddlebox 15*. ACM Press, 2015. [Online]. Available: https://doi.org/10.1145/2785989.2785993

[2] M. Kablan, A. Alsudais, E. Keller, and F. Le, "Stateless network functions: Breaking the tight coupling of state and processing," in *14th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 17)*, 2017, pp. 97–112.

[3] G. Németh, D. Géhberger, and P. Mátray, "{DAL}: A locality-optimizing distributed shared memory system," in *9th {USENIX} Workshop on Hot Topics in Cloud Computing (HotCloud 17)*, 2017.

[4] M. Szalay, P. Mátray, and L. Toka, "Minimizing state access delay for cloud-native network functions," in *2019 IEEE 8th International Conference on Cloud Networking (CloudNet)*. IEEE, 2019, pp. 1–6.

[5] M. Szalay, M. Nagy, D. Géhberger, Z. Kiss, P. Mátray, F. Németh, G. Pongrácz, G. Rétvári, and L. Toka, "Industrial-scale stateless network functions," in *2019 IEEE 12th International Conference on Cloud Computing (CLOUD)*. IEEE, 2019, pp. 383–390.

[6] S. Woo, J. Sherry, S. Han, S. Moon, S. Ratnasamy, and S. Shenker, "Elastic scaling of stateful network functions," in *15th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 18)*, 2018, pp. 299–312.

[7] "Cloudping.info," https://www.cloudping.info/, accessed: 2020-06-04.

[8] M. D. Da Silva and H. L. Tavares, *Redis Essentials*. Packt Publishing Ltd, 2015.

[9] S. Sivasubramanian, "Amazon dynamodb: a seamlessly scalable non-relational database service," in *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data*, 2012, pp. 729–730.

[10] C. Wu, J. Faleiro, Y. Lin, and J. Hellerstein, "Anna: A kvs for any scale," *IEEE Transactions on Knowledge and Data Engineering*, 2019.

[11] L. Lersch, I. Schreter, I. Oukid, and W. Lehner, "Enabling low tail latency on multicore key-value stores," *Proceedings of the VLDB Endowment*, vol. 13, no. 7, pp. 1091–1104, 2020.

[12] J. Ousterhout, A. Gopalan, A. Gupta, A. Kejriwal, C. Lee, B. Montazeri, D. Ongaro, S. J. Park, H. Qin, M. Rosenblum *et al.*, "The ramcloud storage system," *ACM Transactions on Computer Systems (TOCS)*, vol. 33, no. 3, pp. 1–55, 2015.

[13] C. Wu, V. Sreekanti, and J. M. Hellerstein, "Autoscaling tiered cloud storage in anna," *The VLDB Journal*, pp. 1–19, 2020.

[14] B. Chandramouli, G. Prasaad, D. Kossmann, J. Levandoski, J. Hunter, and M. Barnett, "Faster: A concurrent key-value store with in-place updates," in *Proceedings of the 2018 International Conference on Management of Data*, 2018, pp. 275–290.

[15] M. Perron, R. C. Fernandez, D. DeWitt, and S. Madden, "Starling: A scalable query engine on cloud function services," *arXiv preprint arXiv:1911.11727*, 2019.

[16] D. A. Popescu and A. W. Moore, "A first look at data center network condition through the eyes of ptpmesh," in *2018 Network Traffic Measurement and Analysis Conference (TMA)*, 2018, pp. 1–8.