State Management in Real-Time Cloud and NFV Services

Luca Abeni[†], Harald Gustafsson[*], Fredrik Svensson[*], Tommaso Cucinotta[†]
[*]Ericsson Research, Lund, Sweden, *e-mail*: {harald.gustafsson|fredrik.a.svensson}@ericsson.com
[†]Scuola Superiore Sant'Anna, Pisa, Italy, *e-mail*: {first.last}@santannapisa.it

***Abstract*—In the context of distributed, real-time Cloud and NFV services, state management is becoming an increasingly important issue that risks impairing the ability of stateful services to provide timely responses in presence of several types of possible failures that might occur at run-time. This paper presents a simulation-based approach for a comparative exploration of the trade-offs in performance vs fault-tolerance and robustness, achievable in distributed end-to-end real-time applications, arising when using different state management and load-balancing strategies.**

***Index Terms*—Cloud computing, state management, real-time systems, fault tolerance**

I. Introduction and Related Work

The wide availability of broadband connectivity coupled with complex and flexible distributed software architectures, led to the current era of *Cloud Computing* infrastructures [1]. Cloud solutions have been evolving from traditional Cloud-hosted web and storage servers, towards *native* Cloud applications and services, where software is being developed as a composition of *microservices* [2] deployed as VMs or lighter *containers* [3], using a plethora of services available in Cloud providers, like computational, network, security and access control, monitoring and accounting, and reliable data store services. Moreover, *Cloud/Edge* Infrastructures [4], [5] are recently gaining in popularity, where the availability of reduced-size, lower-latency local Edge providers can host services with particularly stringent timing and *latency* requirements.

Therefore, Cloud infrastructures are attracting more and more application domains, becoming the cornerstone of ICT services for a plethora of software-based services we use in our everyday lives. Cloud principles have also been applied to Telecommunication networks, supporting the recent trend towards Network Function Virtualization (NFV) [6]. In this area, network operators have been relying in the past on network functions supported by traditional physical appliances sized for peak-hour operations. Nowadays, they are shifting towards more flexible solutions where softwarized network functions, a.k.a., Virtual Network Functions (VNFs), realized as horizontally scalable Cloud native micro-services, are deployed as VMs or containers [7] onto general-purpose servers within a private Cloud infrastructure, managed through a Cloud/NFV orchestrator. This poses the foundations for flexible network infrastructures that have the ability to dynamically adapt to the continuously changing traffic conditions exhibited by modern, distributed and mobile use-cases.

In this context, an increasingly popular trend is the one to design *Cyber-Physical Systems* with the ability to offload the most computationally demanding parts of their computations to a closeby Cloud/Edge infrastructure, often accessible through a mobile network, especially if supporting advanced ultra-reliable and low-latency capabilities of 5G networks [8], [9]. This enables the use of complex control algorithms or sophisticated data processing pipelines, including Machine Learning and Artificial Intelligence, that would be impossible to run on a local embedded and mobile device but can conveniently be deployed on powerful servers in a remote, but closeby, Cloud/Edge infrastructure. In this new era of *Cyber-Physical Cloud infrastructures*, it becomes paramount to design distributed infrastructures and Cloud-native services, focusing on the diverse range of time-criticality [10] and reliability [11] requirements for the needed services.

A particularly interesting challenge to tackle in this context is the one of *state management* for stateful services [12], [13]. Indeed, *stateless* services can easily be balanced and re-routed across a number of instances that can indistinguishably serve the next request. However, with *stateful* ones, whenever processing a request we may need state information from the last completed one: if that instance just failed, or it is temporarily unreachable, the current request may undergo additional delays, or even need to be failed, unless the state from the last successfully completed request has been replicated. This requires allocating additional resources in a way that makes the service fault-tolerant, and can have an impact on the expected end-to-end latency experienced by clients.

Common solutions for state management in distributed Cloud/NFV services include using a centralized server, like a relational database (e.g., a MySQL instance), a simple key-value data store (e.g., BerkeleyDB [14]) or a more feature-rich NoSQL solution. These approaches suffer from the traditional problem of constituting a single point of failure. Therefore, in Cloud environments, it is commonplace to use distributed data stores employing data sharding and replication across a number of fault-independent zones, using a distributed consensus protocol like Paxos [15], Raft [16] or others [17]. Solutions commonly used in this context include the open-source MongoDB [18] and Cassandra [19], or AWS DynamoDB [20] or Google BigTable [21], accessible in commercial Clouds.

In the research literature, some works explicitly tackled state management. For example, FASTER [12] is a key-value store middleware that can be integrated within an application or service to store reliably state. It uses a concurrent hash-based index, coupled with a persistent append-only record log that caches records in memory, supporting fast read-modify-write transactional updates. Similarly, in [13] a middleware for state management is introduced, where a fast heuristic is proposed to solve the problem of joint function and replica placement throughout a data center, yielding, most of the time, fast access to the state information, locally available.

One remarkable missing element in the existing literature, is the one of a simulator for the comparative evaluation of various state management policies, and their impact on the end-to-end latency of a complex distributed Cloud/NFV service. Existing approaches like CloudSim [22], despite recent efforts for integrating a variety of additional features and modules [23], still lack essential features related to real-time scheduling of tasks, state management and fault tolerance.

A. Contributions and Paper Overview

In this paper, we tackle the challenge of state management in Cyber-Physical Clouds characterized by fault-tolerant, time-critical stateful services by exploring the performance (respecting temporal constraints and fault-tolerance) that can be achieved when designing end-to-end time-critical services. This is done by proposing a model for handling microservices' state (see Section II), and implementing it within a novel simulator (see Section III) that will be used in future work to investigate the impact of various state handling policies on the overall end-to-end latency. In Section IV, we validate the tool and show some examples of the results it can provide, while in Section V we conclude the paper, discussing also possible directions for future research on the topic.

II. System Model

Cyber-Physical Cloud systems (as introduced in Section I) need to respect temporal constraints (modelled as *deadlines*) and tolerate faults. Previous works [11] provided a formal model for these properties by considering Cloud-native applications $\{\mathcal{A}^g\}_{g=1,2,\ldots}$ built by composing multiple *tasks* in a Directed Acyclic Graph (DAG): the i^{th} task of $\mathcal{A}^g$, τ_i^g, is a node of the DAG, and each edge (i,j) represents a communication flow from τ_i^g to τ_j^g. In practice, a task τ_i^g represents a sequential activity that processes input data (input messages), generating output data (output messages). More formally, τ_i^g performs repeatedly the following operations: it waits until it has received a message from all of its inputs (its predecessors in the DAG), then it wakes up and processes the received input data, then it sends the result to all of its outputs (its successors in the DAG). Task τ_1^g receives requests from clients, so it is called *input* task; similarly, the last task of the DAG sends responses to the interested clients (often, the same client that activated τ_1^g), so it is called *output* task.

From a practical point of view, tasks are implemented within a set of distributed *microservices*, composed of pools of multiple containers (also referred as *instances* in this paper). Each one of such containers takes a maximum time C_i to process the input data and generate output data.

From a conceptual point of view[1], a load balancer forwards tasks activations (messages) to one or more of the microservice's containers. In particular, if a task is "critical" then the message is sent to at least two containers that process it in parallel (so that, if a container fails, the other one can still produce a correct result on time), whereas if the task is "not critical" the message is sent only to one *primary container* (selected according to a load balancing algorithm) and will be re-sent to a different *backup container* if the primary container does not provide an output within a *partial deadline*.

[1]Actual implementations might use a different architecture, but conceptually input messages are distributed according to a load-balancing algorithm.

A. Stateful Tasks

The model described above is based on stateless microservices: the containers composing a microservice can generate an output message based only on the input message and do not need any additional data. In considering *stateful* microservices, instead, the j^{th} output message of a task can be generated based on the j^{th} input message and on some internal microservice state: to process a request and generate the corresponding output, the container must use the input data *and the state generated from the previous request*; if such a state is not already available to the container, it must be fetched from somewhere, waiting until the needed state is available. Hence, when a container finishes serving a message, it might need to wait for some time before starting to serve the next message from its input queue. This means that the response time of a container is not given only by the service time C_i and the queuing time, but also by a *blocking time* experienced while waiting for the needed state. Note that the initial state of each container at system start-up is always available.

Clearly, the blocking time experienced by a task depends on the policy used for storing and accessing its state. Using a centralized store, the blocking time is equal to the write and read access latency of the store and can be reduced by locally caching the state in the containers using it. When a container of task τ_i processes the j^{th} message, if the container does not have the needed state in its cache, then it experiences a blocking time equal to the amount of time needed to query the data store, which is available only after it has been written. Then, after processing the message, the container writes the updated state in the global store but also keeps a copy of the state in its local cache. Hence, if the $j+1^{th}$ message is processed on the same container, the needed state is already available, and no blocking time is experienced. This strategy works well if a sticky load balancer is used to distribute messages to the containers if no failures are experienced.

Alternatively, the state can be distributed on the containers instead of being saved in a centralized store. In this approach, every container can store a copy of the state, asking other containers for the needed state if it is not locally available when processing a new request. After processing the request, the container updates its copy of the state, and optionally sends updates to the other containers as a fault tolerance mechanism. Hence, a container can *pull* the state from other containers when it is needed and *push* the state to other containers when updating it. When pushing the state, the container can wait until other containers have received it, or

```
digraph test {
  "Ts" [c="{8:0.5,12:0.5}"];
  "T1" [c="{4:0.8,7:0.2}"];
  "T2" [c="{4:0.8,7:0.2}"];
  "T3" [c="{4:0.8,7:0.2}"];
  "T4" [c="{4:0.8,7:0.2}"];
  "Ts" -> "T1";
  "T1" -> "T2";
  "T1" -> "T3";
  "T2" -> "T4";
  "T3" -> "T4";
}
```

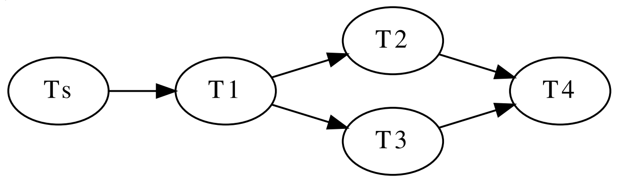

Fig. 1. Example of `.dot` file describing a Cloud-native application (on the top) and corresponding DAG (on the bottom).

send it asynchronously. Using this distributed approach, the blocking time is the minimum of the pull and push times.

The centralized store can tolerate faults by using a replicated architecture, while the distributed state approach tries to tolerate faults by replicating the state on multiple service containers. If, however, all the containers that store the updated state are down due to faults when such a state is needed, then some requests cannot be processed and are dropped. At this point, no messages can be processed anymore until a working state is reconstructed, at the cost of a potentially larger delay.

III. Simulating a Cyber-Physical Cloud

Previous work [11] allowed for guaranteeing that Cloud-native applications running in Cyber-Physical Clouds respect all their temporal constraints by performing a pessimistic analysis, as commonly done in real-time systems. In other words, all the execution times C were considered constant and equal to the worst-case. While this kind of analysis is effective, it can easily result in resource overallocation, reducing the Cloud utilization. To better exploit the Cloud resources without heavy underutilization, a less pessimistic analysis (based on distributions $P(c \leq t)$ of the execution times rather than the worst-case execution times C) is needed. However, performing such a stochastic analysis on complex DAGs with nodes composed of multiple container instances is challenging and this analysis is still under development. Hence, in this paper the Cyber-Physical Cloud performance is evaluated by simulating the model presented in Section II through an event-driven simulator (written in C++ to maximize its performance). An architectual view of the simulator is presented in Figure 2. The events provide the means to simulate e.g. execution times, request inter-arrival times, request timeouts, etc.

At the heart of the simulator is the `Queue` interface, describing a container that can receive requests, enqueue them in a FIFO queue, and serve them one after the other. Each request (described by an instance of the `Packet` class) is served in a time distributed according to a specified random

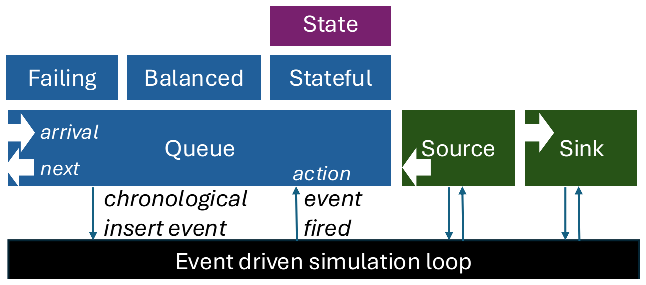

Fig. 2. Simulator architecture, with events that drive the simulation time forward for queues, sources, sinks and state operations.

variable. The `Queue` interface is described by an `arrival()` method (invoked when a new packet arrives at the container), a reference to the next node of the DAG (once a packet is processed, the `arrival()` method of the next node is invoked) and an execution times distribution. Two `Split` and `Join` classes are also provided to model DAGs with parallel branches. The `Split` class provides one `arrival()` method, which creates multiple copies of the incoming packets to be passed to the next nodes (one per outgoing branch), while the `Join` class receives multiple `arrival()` calls and merges their inputs in the packet to be passed to the next node.

The simplest implementation of the `Queue` interface models a standard G/G/1 queue [24], characterized by a generic random variable for the service times and a single server. A special `Source` class is used to generate multiple streams of packets, with inter-arrival times distributed according to a generic random variable, and a `Sink` class receives the packets after they have been processed by all the queues, collecting the end-to-end response times. The `Source` and the `Sink` classes model the clients described in Section II, sending requests to the cloud-native application and receiving back responses. A file in the standard `.dot` format[2] [25] can be used to connect all the queues in a DAG, configure them, and configure and connect to the DAG a `Sink` and one or more `Sources`. Figure 1 shows an example of DAG with the corresponding `.dot` file. In this case, the simulator instantiates a `Source` class (corresponding to node `Ts`, which has no inputs), a `Sink` (not described in the `.dot` file), a `Split` (which receives packets from `T1`, duplicates them, and sends the two copies to `T2` and `T3`), and a `Join` (which receives packets from `T2` and `T3`, merges them, and sends the results to `T4`). Then, it creates `Queue` instances for `T1`, `T2`, `T3`, and `T4`, and connects them according to the DAG.

Other implementations of the `Queue` interface model more complex behaviours. For example, the `FailingQueue` class modifies the simple G/G/1 queue to model different kinds of failures: transient network failures (a packet is randomly discarded at the `arrival()` method, when it arrives at the container), transient container failures (a packet is randomly discarded after it has been processed, consuming execution time), and non-transient container failures simulating a con-

[2]https://graphviz.org/doc/info/lang.html

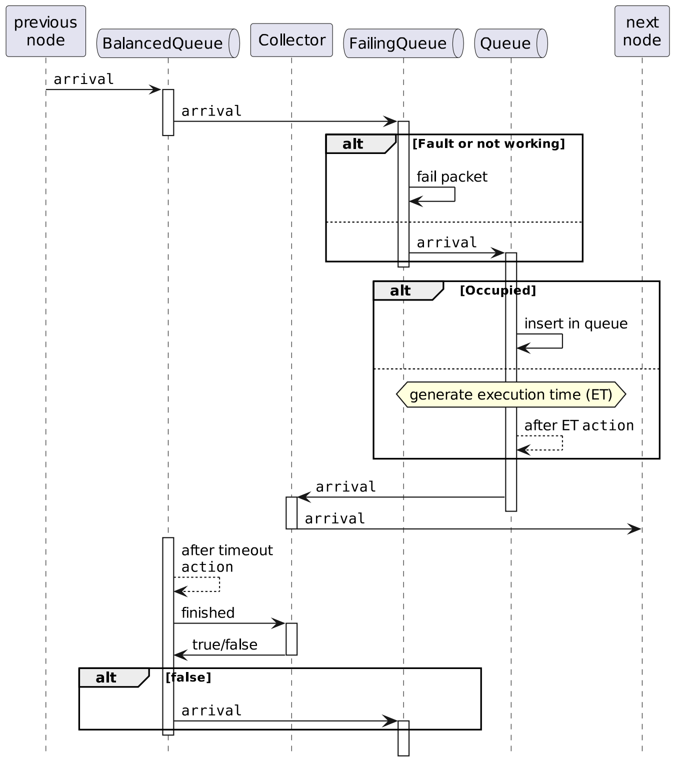

Fig. 3. Sequence of method invocations for serving a packet in a `BalancedQueue` with multiple failing queues.

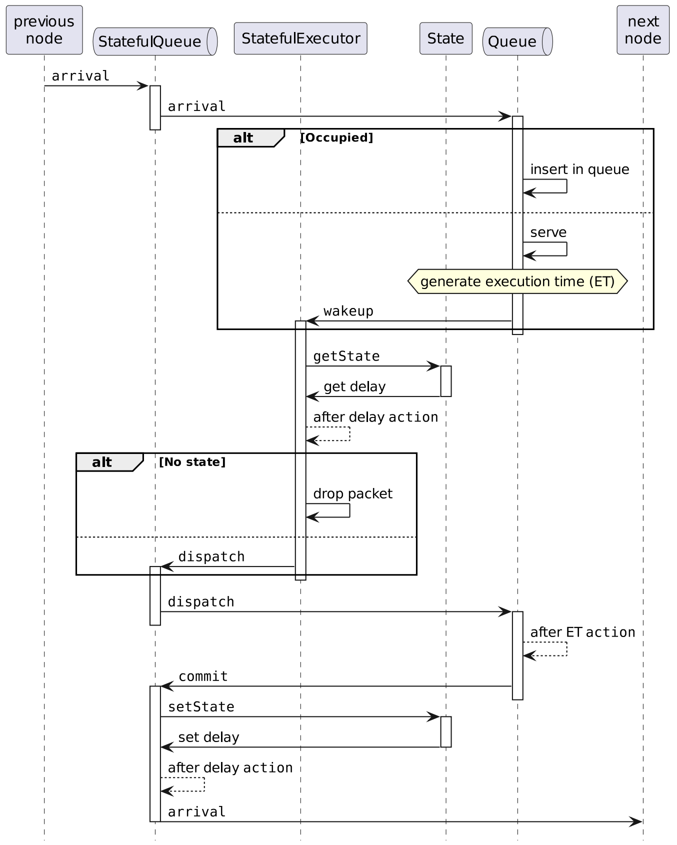

Fig. 4. Sequence of method invocations for serving a packet in a `StatefulQueue`.

tainer taking time to be replaced. The simulator creates instances of `FailingQueue` classes instead of `Queue` based on the attributes of the nodes in the `.dot` file.

The `BalancedQueue` class can be used to model microservices as composed of multiple containers, as it models the `Queue` interface (and can hence replace a "simple" G/G/1 queue) as a load balancer distributing incoming packets among multiple `Queues`. A `BalancedQueue` has references to multiple objects implementing the `Queue` interface, and when the `arrival()` method is invoked, it forwards the incoming packet to one of these queues, according to a sticky (each stream stays on the same queue) or round-robin policy. It also implements fault-tolerance features, as it can duplicate the packet and forward it to another queue based on a replication policy, either directly (when critical), after a timeout or reaching a partial deadline when the first packet has still not finished being processed. Figure 3 shows the sequence of actions performed by a `BalancedQueue` with multiple `FailingQueues` to serve a request (for the sake of simplicity, only network failures are displayed). When the packet arrives at the queue, the `arrival()` method of the `BalancedQueue` class is invoked; this method selects one of the served queues and invokes its `arrival()` method. In this example, the `BalancedQueue` has references to a set of instances of the `FailingQueue` class, so the `arrival()` method of the `FailingQueue` class is invoked. This method checks if the queue is working correctly or if the packet must be discarded (so, the request fails); if the request does not fail, it is forwarded to the `arrival()` method of a G/G/1 queue, which checks if the server is idle or not; if yes, the request is served immediately (and it is processed in a time c, generated by a random variable). Otherwise, it is enqueued in a FIFO queue (when the server becomes idle again, the next packet from the FIFO queue is served – extracting a new value from the random variable for the service time). After a time c is passed, the `action()` method of the queue is invoked; this method invokes the `arrival()` method of a `Collector` class, which records the fact that the request has been correctly processed and passes the request to the next node in the DAG. After a timeout, the `BalancedQueue` class checks if the request has been processed by invoking the `finished()` method of the `Collector` class (remember that an object of this class is able to keep track of the correctly processed requests). If the request has not been processed yet, a different instance of the `FailingQueue` class is selected, and a copy of the request is passed to it.

The blocking time caused by state handling, described in Section II-A is simulated by the `StatefulQueue` class (another implementation of the `Queue` interface). When the server is ready to serve a request, the `StatefulQueue` does not immediately start to serve the first packet of the FIFO queue but sends a request to a class (implementing the `Executor` interface) that will later dispatch the server after a blocking time. An `Executor` receives requests from a queue (when the `Executor`'s `wakeup()` method is invoked), and invokes the queue's `dispatch()` method when the queue's server can start serving the packet. The `StatefulExecutor` class implements the `Executor` interface, providing a `wakeup()` method that checks the state availability and invokes the queue's `dispatch()` only when the state is available. Similarly, after finishing serving a request, the `StatefulQueue` does not immediately forward it to the next task of the DAG but first waits for the amount of time needed to save the state.

The sequence of actions performed by a `StatefulQueue` to serve a request is shown in Figure 4. Notice that the `arrival()` method invokes the `wakeup()` method of an instance of `StatefulExecutor`, which uses an instance of a class implementing the `State` interface to compute the time needed to get the state. This `getState()` method can also fail (if the needed state is not available and cannot be fetched/recovered from anywhere); in this case, the packet is dropped. After the delay needed to get the state, the `dispatch()` method of the `StatefulQueue` is invoked, and the request can be processed as in a standard G/G/1 queue. An additional delay is added to save the state after processing the request; then, the packet is passed to the next node in the DAG by invoking its `arrival()` method.

In this way, the `getState()` and `setState()` methods of a class implementing the `State` interface allow simulating the delays introduced by the state push and pull operations described in Section II-A. Of course, the values returned by these methods depend on the specific state handling policy that is simulated; hence, the simulator provides different implementations of the `State` interface; the two most important are the `RemoteState` class, modelling a centralized data store (with or without local cache) and the `SynchronizedState` class, implementing a distributed state that default to push state to all the containers.

The `StatefulExecutor` uses one of these implementations of the `State` interface to check if the needed state is available and to compute the blocking time that has to be imposed on the queue.

IV. Simulation Results

In this section, the correctness and accuracy of the simulator described in Section III is validated by simulating simple services that can be formally analyzed and verifying that the simulator's results are consistent with the theoretical analysis.

For example, consider the simple DAG shown in Figure 1, where the inter-arrival time for the packets generated by the source `Ts` is 8 with a probability of 0.5 and 12 with a probability of 0.5, and the service time for all the tasks is 4 with a probability of 0.8 and 7 with a probability of 0.2; every task is composed of $M = 2$ containers. All simulations use the discreate time measured in time units.

Figure 5 shows the experimental Cumulative Distribution Function (CDF) of the DAG's response times when a centralized store is used for the tasks' state. The application

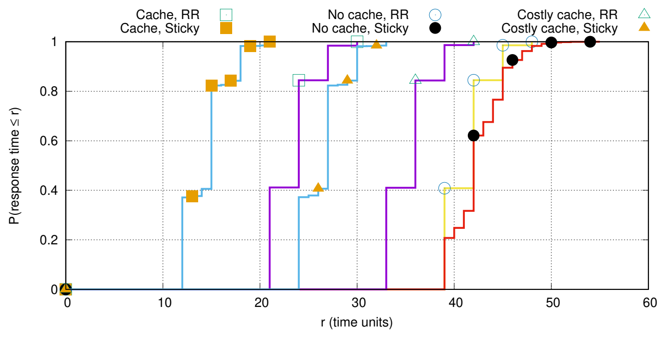

Fig. 5. Experimental CDF of the response times for the DAG of Figure 1 when the tasks' state is stored in a centralized store.

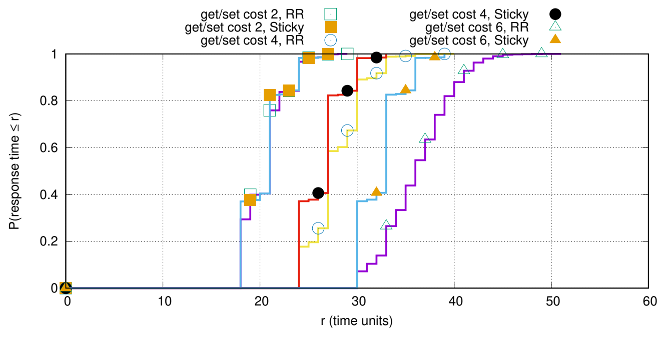

Fig. 6. Experimental CDF of the response times for the DAG of Figure 1 when the tasks' state is distributed.

is simulated using two different load-balancing algorithms (Round-Robin and sticky) for the tasks' containers, with and without local caches for the state. Here, the same setting is used for all the tasks, even though the simulator allows different state configurations and load-balancing algorithms for each task. If no local caching is used ("No Cache" line in the figure), then all the accesses to the state are performed using the centralized store (3 time units are needed to get the state, and 6 time units are needed to set it — the state is written synchronously, as discussed below). If, instead, a local cache is used, then accesses to the state can be very fast (0 time units both for reading the state from the cache and saving it to the cache - marked as "Cache" in the figure) or can require some time (4 time units to save the state to the cache - marked as "Costly cache" in the figure). Write accesses to the store are more expensive than read accesses because the store is assumed to be implemented using some replication algorithm that requires synchronously distributing every write to multiple nodes, while reads can be performed on a single node without issues (nodes are synchronized by the algorithm).

From the figure, it is possible to notice that the local caches with a sticky load balancer work well (because the load balancer makes sure that the $j + 1^{th}$ packet is processed in the same container where the j^{th} packet was processed, so the needed state is in cache). If writing the state in the cache introduces some cost (see the "Costly cache, Sticky" line in the graph), the response times increase (the "Costly cache" lines are shifted right by $12 = 4 * 3$ time units respect to the corresponding "Cache" lines — remember that the cost to write the state to the cache is increased to 4 time units, and the critical path from `Ts` to `T4` traverses 3 tasks), consistently with theoretical expectations. Using a Round-Robin (RR) load balancer invalidates the performance gains provided by the caches when getting the state (the "RR" lines are on the right of the corresponding "Sticky" lines by at least $9 = 3 * 3$ time units — 3 time units is the amount of time needed to get the state from the central storage). This is again consistent with theoretical expectations and can be explained by noticing that the load balancer does not care about processing the $j + 1^{th}$ packet on the same container as the j^th packet. It is also worth noticing that when local caching is not used, the sticky load balancer ("No cache, Sticky" line) results in response times which are slightly worse than the ones provided by the RR load balancer. This result is also consistent with theoretical expectations, as the Round-Robin load balancer can distribute the requests on multiple containers (reducing the queueing time), while the sticky load balancer only uses one container (increasing the load on such a container even if other containers are unloaded). Finally, notice that the smallest response time, obtained with local caching and the sticky load balancer, is equal to 12 time units; this response time is obtained when the request is processed with the minimum execution time (which is 4) in all the tasks (since `T2` and `T3` are processed in parallel the critical path is `T1`$\rightarrow$`T2`$\rightarrow$ `T4` or `T1`$\rightarrow$`T3`$\rightarrow$ `T4`; hence, the DAG's makespan is equal to 3 times the minimum execution time) and no blocking time is experienced because the needed state is found in cache.

Figure 6 shows the results of a similar experiment with the distributed state approach. In this case, the sticky and RR load balancers behave similarly when the get/set delays are low (the sticky load balancer performs slightly better). However, when the communication cost increases, the sticky load balancer starts working better. Again, this behaviour can be explained: with the sticky load balancer the needed state will always be found when needed, while using a RR load balancer the $j+1^{th}$ packet can arrive at a container different from the one where the j^{th} packet has been processed, and the needed state has not arrived to this container yet.

A more complex DAG (as illustrated in Figure 8) has been used to check the simulator's ability to run larger simulations. The tasks have an individually set number of instances, timeouts and service time Probability Mass Functions (PMFs) as shown. The number of instances and timeouts are different for the distributed state (first number) and centralized state (second number), due to the fact that longer response times need to be compensated by more resources. To simulate a more realistic setup with multiple streams of requests, the number of sources is increased to 1000, with an average inter-arrival time of 503 time units concentrated between 500 and 504, and with a uniform probabilistic spread of arrivals between sources. Moreover, in these simulations containers can experience non-transient failures with a fault rate of 0.1% and return to be operating after *recovery latency*. The recovery latency is a random variable distributed according to the PMF shown in Figure 7, having a long minimum recovery latency. This simulates real containers with a non-null setup time after a failure. The simulation runs for 6 million time units and produces almost 12 million requests.

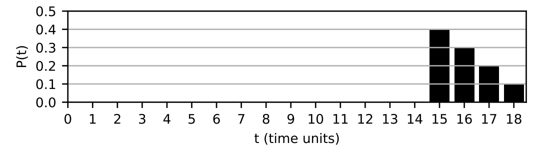

Fig. 7. The recovery PMF for complex DAG in Figure 8.

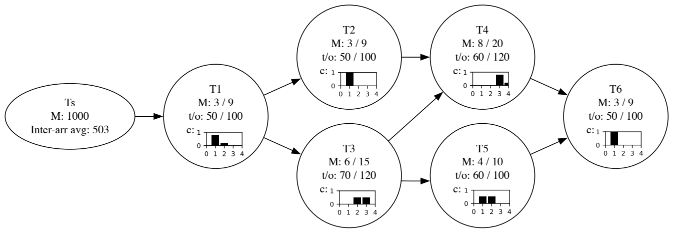

Fig. 8. The DAG used for the large simulations presented in this section. The tasks number of instance (M), timeout (t/o), and service time PMF (c) is shown for each task for distributed and centralized state, respectively.

When simulating this more complex DAG with a larger number of sources and longer recovery times, it is important to correctly tune the number of instances and the timeouts. To that end, we had to tune the centralized and distributed state handling differently to avoid overloading the system or over-dimensioning the computational resources. The centralized state store uses $2.5-3$ times more instances and approximately double the re-send timeout. All the resulting configurations still show a similar served request success rate of about 99.7%. Potentially, with a more advanced re-send policy, stability could be achieved with fewer resources.

Comparing the CDF results of centralized storage, Figure 9, it is clear that the Round Robin Load Balancers manages to achieve shorter response times both for average and tail latency. Even though the RR load balancer is not able to exploit the cache efficiently, it still manages to achieve shorter response times by better distributing the load than the sticky load balancer. This tends to increase the queues lengths, which is not shown, but derived from simulation traces. Another difference between the CDFs is the more flack curve for the sticky load balancer, potentially due to the increasing influence of the queue length vs the state-related latencies. As shown in Figure 10, the distributed state handling approach shows a similar advantage to the RR load balancer. Comparing the CDF results of distributed state and centralized state storage, it can be seen that the distributed state handling approach manages to achieve a shorter response even when using fewer resources.

V. Conclusions

This paper proposed a simulation-based approach for modelling and design of different techniques for state handling of

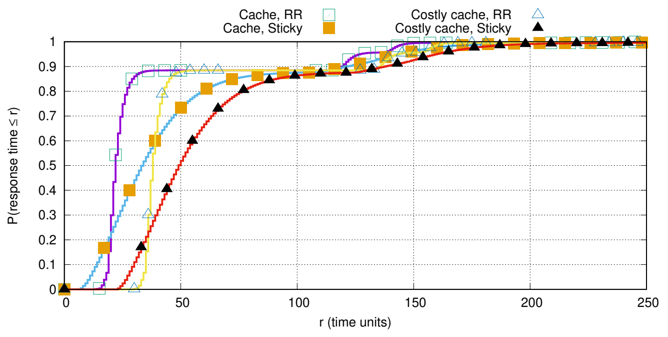

Fig. 9. Experimental CDF of the response times for the more complex experiments for the DAG of Figure 8 with 1000 sources, 0.1% failure rate, re-send and using a centralized storage for the state. Using 2.5-3 times more instances than the distributed state simulation to be stable.

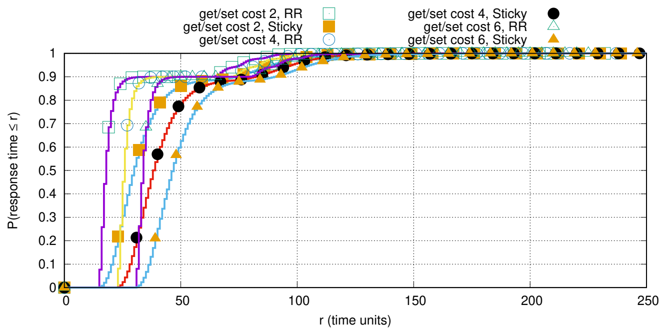

Fig. 10. Experimental CDF of the response times for the more complex experiments for the DAG of Figure 8 with 1000 sources, 0.1% failure rate, re-send and using a distributed state handling.

microservices, and evaluating their impact on the end-to-end latency of real-time Cloud/NFV services. The paper presented our proposed simulator, and showed some simple examples of the results it can generate.

In the future, we will use the simulator to validate the probabilistic analysis we are currently developing, and we will develop a distributed state-handling protocol providing better real-time performance and fault-tolerance properties.

References

[1] R. Buyya, C. Vecchiola, and S. T. Selvi, "Chapter 1 - introduction," in *Mastering Cloud Computing*, R. Buyya, C. Vecchiola, and S. T. Selvi, Eds. Boston: Morgan Kaufmann, 2013, pp. 3–27.

[2] R. Buyya, S. N. Srirama, G. Casale, R. Calheiros, Y. Simmhan, B. Varghese, E. Gelenbe, B. Javadi, L. M. Vaquero, M. A. S. Netto, A. N. Toosi, M. A. Rodriguez, I. M. Llorente, S. D. C. D. Vimercati, P. Samarati, D. Milojicic, C. Varela, R. Bahsoon, M. D. D. Assuncao, O. Rana, W. Zhou, H. Jin, W. Gentzsch, A. Y. Zomaya, and H. Shen, "A manifesto for future generation cloud computing: Research directions for the next decade," *ACM Comput. Surv.*, vol. 51, no. 5, Nov. 2018.

[3] C. Carrión, "Kubernetes scheduling: Taxonomy, ongoing issues and challenges," *ACM Comput. Surv.*, vol. 55, no. 7, Dec. 2022.

[4] R. Moreno-Vozmediano, E. Huedo, R. S. Montero, and I. M. Llorente, "Disaggregated cloud management for edge computing," in *2019 28th International Conference on Computer Communication and Networks (ICCCN)*, July 2019, pp. 1–6.

[5] P. Arthurs, L. Gillam, P. Krause, N. Wang, K. Halder, and A. Mouzakitis, "A taxonomy and survey of edge cloud computing for intelligent transportation systems and connected vehicles," *IEEE Transactions on Intelligent Transportation Systems*, vol. 23, no. 7, pp. 6206–6221, July 2022.

[6] X. Cai, H. Deng, L. Deng, A. Elsawaf, S. Gao, A. M. D. Nicolas, Y. Nakajima, J. Pieczerak, J. Triay, X. Wang, B. Xie, and H. Zafar, *ETSI White Paper No. 54 – Evolving NFV towards the next decade*, May 2023.

[7] F. Wiedner, A. Daichendt, J. Andre, and G. Carle, "Control groups added latency in nfvs: An update needed?" in *2023 IEEE Conference on Network Function Virtualization and Software Defined Networks (NFV-SDN)*, Nov 2023, pp. 40–45.

[8] O. Lhamo, T. V. Doan, E. Tasdemir, M. Attawna, S. Senk, F. H. Fitzek, and G. T. Nguyen, "Improving reliability for cloud-native 5g and beyond using network coding," in *2023 IEEE Conference on Network Function Virtualization and Software Defined Networks (NFV-SDN)*, Nov 2023, pp. 1–7.

[9] X. Limani, A. Troch, C.-C. Chen, C.-Y. Chang, A. Gavrielides, M. Camelo, J. M. Marquez-Barja, and N. Slamnik-Kriještorac, "Optimizing 5g network slicing: An end-to-end approach with isolation principles," in *2024 IEEE Conference on Network Function Virtualization and Software Defined Networks (NFV-SDN)*, Nov 2024, pp. 1–6.

[10] R. Andreoli, R. Mini, P. Skarin, H. Gustafsson, J. Harmatos, L. Abeni, and T. Cucinotta, "A multi-domain survey on time-criticality in cloud computing," *IEEE Transactions on Services Computing*, p. 1–19, 2025.

[11] L. Abeni, R. Andreoli, H. Gustafsson, R. Mini, and T. Cucinotta, "Fault tolerance in real-time cloud computing," in *2023 IEEE 26th International Symposium on Real-Time Distributed Computing (ISORC)*. IEEE, May 2023.

[12] B. Chandramouli, G. Prasaad, D. Kossmann, J. Levandoski, J. Hunter, and M. Barnett, "Faster: an embedded concurrent key-value store for state management," *Proc. VLDB Endow.*, vol. 11, no. 12, p. 1930–1933, Aug. 2018.

[13] M. Szalay, P. Mátray, and L. Toka, "State management for cloud-native applications," *Electronics*, vol. 10, no. 4, 2021. [Online]. Available: https://www.mdpi.com/20A79-9292/10/4/423

[14] M. A. Olson, K. Bostic, and M. I. Seltzer, "Berkeley db," in *USENIX Annual Technical Conference, FREENIX Track*, 1999, pp. 183–191.

[15] L. Lamport, "The part-time parliament," *ACM Trans. Comput. Syst.*, vol. 16, no. 2, p. 133–169, May 1998.

[16] D. Ongaro and J. Ousterhout, "In search of an understandable consensus algorithm," in *2014 USENIX annual technical conference (USENIX ATC 14)*, 2014, pp. 305–319.

[17] M. Barborak, A. Dahbura, and M. Malek, "The consensus problem in fault-tolerant computing," *ACM Comput. Surv.*, vol. 25, no. 2, p. 171–220, Jun. 1993.

[18] W. Schultz, T. Avitabile, and A. Cabral, "Tunable consistency in mongodb," *Proc. VLDB Endow.*, vol. 12, no. 12, p. 2071–2081, Aug. 2019.

[19] N. B. Seghier and O. Kazar, "Performance benchmarking and comparison of nosql databases: Redis vs mongodb vs cassandra using ycsb tool," in *2021 International Conference on Recent Advances in Mathematics and Informatics (ICRAMI)*. IEEE, 2021, pp. 1–6.

[20] S. Perianayagam, A. Vig, D. Terry, S. Sivasubramanian, J. C. S. III, A. Mritunjai, J. Idziorek, N. Gallagher, M. Elhemali, N. Gordon, R. Krog, C. Lazier, E. Mo, T. Rath, and S. Sosothikul, "Amazon DynamoDB: A Scalable, Predictably Performant, and Fully Managed NoSQL Database Service," in *USENIX Annual Technical Conference*, Carlsbad, CA, 2022, pp. 1037–1048. [Online]. Available: https://www.usenix.org/conference/atc22/presentation/vig

[21] F. Chang, J. Dean, S. Ghemawat, W. C. Hsieh, D. A. Wallach, M. Burrows, T. Chandra, A. Fikes, and R. E. Gruber, "Bigtable: A distributed storage system for structured data," *ACM Trans. Comput. Syst.*, vol. 26, no. 2, Jun. 2008.

[22] R. N. Calheiros, R. Ranjan, A. Beloglazov, C. A. F. De Rose, and R. Buyya, "Cloudsim: a toolkit for modeling and simulation of cloud computing environments and evaluation of resource provisioning algorithms," *Software: Practice and Experience*, vol. 41, no. 1, pp. 23–50, 2011.

[23] R. Andreoli, J. Zhao, T. Cucinotta, and R. Buyya, "Cloudsim 7g: An integrated toolkit for modeling and simulation of future generation cloud computing environments," *Software: Practice and Experience*, 2025.

[24] L. Kleinrock, *Queueing Systems, Volume 1: Theory*. USA: Wiley-Interscience, 1975, vol. 2.

[25] J. Ellson, E. Gansner, L. Koutsofios, S. C. North, and G. Woodhull, "Graphviz— open source graph drawing tools," in *Graph Drawing*, P. Mutzel, M. Jünger, and S. Leipert, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2002, pp. 483–484.