Elastic Provisioning of Stateful Telco Services in Mobile Cloud Networking

Elastic Provisioning of Stateful Telco Services in Mobile Cloud Networking

Paolo Bellavista, *Senior Member, IEEE*, Antonio Corradi, *Member*, *IEEE,*
Andy Edmonds, *Member*, *IEEE*, Luca Foschini, *Member, IEEE*,
Alessandro Zanni, *Student Member, IEEE*, Thomas Michael Bohnert, *Member, IEEE*

Abstract — Several relevant research and innovation activities have recently investigated the technical and economic advantages of cloud computing for the provisioning of telco service infrastructures, in particular towards all-IP next generation 5G networks. In fact, the evolution of telco service infrastructures traditionally requires a significant upfront investment (and a long adoption process). Conversely, cloud exploitation significantly lowers investment risks by potentially providing elasticity in service provisioning via flexible Virtual Network Functions (VNFs) on top of a Network Functions Virtualization (NFV) Infrastructure. In this context, the paper presents novel solutions that we have designed, implemented, and evaluated within the EU FP7 Mobile Cloud Networking project (MCN). Their aim is to achieve cost-effective elastic provisioning of telco services over heterogeneous and federated cloud providers, with the specific focus of supporting the extreme quality levels that are demanded by traditional, non-virtualized, and dedicated telco infrastructures. In particular, we concentrate on how to effectively and efficiently automate service state migration for coarse-grained telco service (cloudified) components by leveraging industry-mature orchestration technologies and cloud management frameworks. While our proposed state migration model and procedure are general, its implementation is experimented for MCN's Rating, Charging, and Billing as a Service (RCBaaS). This MCN functionality has been chosen by purpose due to its challenging reliability and uptime requirements. The reported experimental and simulation results show the technical feasibility of the proposed solution under different and realistic load conditions for next-generation and cloudified 5G services.

Index Terms — Cloud computing; Service State Migration; Virtual Machine and Containers; Service Provisioning; Service Composition; Mobile Cloud Networking; Network Function Virtualization; Resource Management.

1 Introduction

CLOUD computing is widely accepted as the dominant paradigm to enable elastic resource provisioning and dynamic configuration. This is primarily motivated by its advantages in saving management costs and in maximizing resource usage efficiency, by dynamically enabling on-demand services tailored to end-user requirements. Virtualization allows partitioning a cloud data center into multiple virtual data centers and hardware resources into Virtual Machines (VMs) or containers, with the purpose of providing dynamically customized resource allocation and of isolating, consolidating, and migrating the received workload at runtime.

Boosted by the opportunity to cut operational costs and to gain wider flexibility, and pushed also by the relevant standardization efforts in the field, such as the Open Cloud Computing Interface (OCCI) and ETSI Network Function Virtualization (NFV) initiatives [1][2], the telecommunications industry is going through a major technological shift. The vision is to take telco services and their entire infrastructures, which once ran on specialized hardware, and their softwarized analogous functions to cloud computing environments. The key driver is efficient execution and reliability on commodity hardware, as well as the cost-effective exploitation of elastic resources similarly to Internet-oriented Over-The-Top (OTT) services. In addition, note that this softwarization can significantly help telco providers to be more dynamic in rapidly offering new services to their customers. Moreover, this allows them to ultimately exploit their market positioning as the owners of the infrastructure closest to end users. The related methodologies for cloudifying virtualized network functions are typically grouped under the term of either cloud-optimized or cloud-native services. These telco services typically require to manage complex virtualized infrastructure deployments including multiple VMs, virtual networks, and Virtualized Network Functions (VNFs) [3].

Cloudification of telco-oriented functions is complicated by the tight performance requirements and constraints that telco software stacks traditionally have. For example, consider the availability and latency requirements that are regulated by standard specifications for the IP Multimedia Subsystem (IMS) and the Evolved Packet Core (EPC). Note that latency requirements are not typically strict in general-purpose cloudified services provided by general-purpose public cloud providers such as MS Azure or Amazon AWS, while they become a central con-

- *P. Bellavista, A. Corradi, L. Foschini, and A. Zanni are with DISI, Univ. Bologna, Viale Risorgimento 2, 40136 Bologna, Italy. E-mail: {paolo.bellavista, antonio.corradi, luca.foschini, alessandro.zanni3}@unibo.it*
- *A. Edmonds and T.M. Bohnert are with the InIT Cloud Computing Lab, ZHAW School of Engineering, Obere Kirchgasse 2, 8400 Winterthur, Switzerland. Email: {andrew.edmonds, thomasmichael.bohnert}@zhaw.ch*

straint for the support to (and acceptance of) next generation networking infrastructures built on top of cloudified resources.

Given those infrastructure and end-user quality requirements, several technical cloudification challenges still remain hard and partially unsolved. This spans from optimized placement of service components in the datacenter (at both creation and runtime) to optimum scaling in/out based on current and predicted load, from proactive, adaptive, and quality-aware service composition to integration with internal and external business/operation support services.

Within this challenging context, this paper describes our novel service state migration solution that we have designed, implemented, and evaluated in the Mobile Cloud Networking (MCN) initiative, a large scale EU project that had involved several leading 5G and cloud companies, research centers, and universities. MCN has the ambitious goal of provisioning quality-constrained, carrier-grade, and cloudified telco services in an efficient way[1] [4]. This goal is pursued via the creation of an ecosystem of service management mechanisms, tools, and frameworks for self-management, self-maintenance, on-premises design, and operations control functions.

More specifically, our service state migration allows moving all the VMs that compose an end-to-end telco service, along with all the data collected and associated to that service. The target service is then reactivated at the new destination with full transparency for the final users. The proposed state migration service allows the MCN framework, for example, to transparently move cumulated application/session state from a service instance to a dynamically deployed replica of it. This allows for enhanced elasticity and for reliable continuous provisioning of services with no user-perceivable downtime. It can easily facilitate dynamic modifications in resource placement for deployment optimization and relieving of congested links.

General-purpose VM migration is not a new research issue and has been widely explored in the past, with the two main variants of: *pre-copy* [5], which pushes most of the data to destination host before stopping and migrating the VM; *post-copy* [6], which pulls most of the data from source host after resuming VM at the destination host. In our original solution for 5G cloudified service provisioning, service state is efficiently migrated by:

i) anticipating state migration through traffic patterns prediction based on gray-box techniques requiring very limited *a-priori* knowledge [7] and,
ii) creating disjoint epochs that enable simultaneous service provisioning by both old and new virtualized instances over disjoint subsets of final users, according to the pre-copy base approach.

The solution is completely integrated with state-of-the-art open-source cloud and monitoring infrastructures, e.g., OpenStack and Zabbix. In addition, we exploit OpenStack Heat templating[2] as the basis to enable i) the runtime creation and management of targeted VM replicas, ii) the connection of new replicas to old state instances and iii) the state migration process. The employment of standard widespread solutions not only makes our proposal highly portable on different cloud providers and underlying hosts, but also provides a valuable implementation for further experimentation, refinement, extension, and utilization by the community[3].

The remainder of the paper is structured as follows. After related work about VM management and service state migration, the paper gives an overview of MCN, needed to fully understand our original proposal. The design guidelines and architectural organization of our solution for service state migration are in Section IV, while Section V goes into in-depth technical insights for the efficient implementation of our proposal. Performance results (from both in-the-field experimentation and CloudSim simulations), conclusive remarks, and directions of ongoing related work end the paper.

2 Related Work

Management of cloudified services has been widely investigated in recent years. Here, with no ambition of being exhaustive, we concentrate on two main research directions that are central for our proposal: i) resource/VM management and orchestration, and ii) service state migration.

On the one hand, OpenStack Heat, Nirmata, and Hurtle have emerged as effective and widespread solutions for service and VM management and orchestration. OpenStack Heat has been one of the pioneers in the management of VMs and their lifecycle, working on top of OpenStack clouds. It is also one of the first solutions to introduce the idea of management templating [8]. Nirmata has adopted a novel micro-services architecture [9]. It provides seamless service discovery, registration, load-balancing and customizable routing for micro-services. It provides functionality that makes it easy to automate the entire DevOps lifecycle. Hurtle is an orchestration framework, adopted in the MCN project, that allows to automate the service life-cycle , from the deployment of cloud resources to configuration and runtime management [10]. It ensures the creation and management, not only of the foundational resources required to operate the target service logic, but also of the so-called external requirements, i.e., the needed external service dependencies.

Some other recent proposals are based on the Topology and Orchestration Specification for Cloud Applications (TOSCA) [11]. TOSCA is a model for topology and orchestration specification in the form of a service template. The adoption of TOSCA enables interoperability of application descriptions, of cloud service infrastructures, and of the operational behavior of these services, e.g., deploy, patch, shutdown, independently from the service provider. [12] presents how TOSCA enables the portable

[1] MCN Website at http://www.mobile-cloud-networking.eu
[2] https://docs.openstack.org/developer/heat/Template_guide/hot_guide.html
[3] Additional information, tools, experimental results, and the state migration prototype code are available at: http://lia.disi.unibo.it/Research/MCN

and standardized management of cloud services. Cloudify also exploits TOSCA and includes functions to enabling deployment on any cloud or data center environment, monitoring/detecting issues and failures, and manually/automatically handling maintenance tasks [13].

Several migration and replication mechanisms have been proposed in the related literature. Seminal research efforts focused on VM migration and on its impact on network/service performance [14, 15]. To alleviate these impacts, various VM management plans based on live VM migrations have been widely explored. Pre-copy is the most common approach used for live migration and has been successfully optimized into some widespread and commercial hypervisors, e.g. Xen, KVM, VirtualBox, and VMware [6]. [14, 15] adopts a typical pre-copy strategy as a base, which combines a push phase, during which the VM memory is transferred in subsequent rounds while the VM is still running, and a stop-and-copy phase, during which the VM is stopped and just a residual part of the data is transferred.

Many works have investigated performance in relation to the migration efficiency of the cloud data center, with different purposes. S-CORE is a scalable live VM migration scheme, based on a distributed migration solution with multiple distinct policies, to dynamically reallocate VMs in order to minimize communication cost [16]. [17] attempts to enhance data center performance and scalability, by minimizing the overall network cost with the introduction of an algorithm to improve VM placement management. [18] can migrate groups of VMs depending on network indicators (e.g., cost of migration, available bandwidth), by detecting the most overloaded links to alleviate network congestion. [19] proposes hierarchical placement of VMs for wide-scale problems on IaaS cloud. Sandpiper [20] automates resource allocation and migration of virtual servers in a data center to avoid machine overload by relocating VMs via Xen's migration mechanisms. They adopt black-box and gray-box strategies to automate monitoring of system resource usage, hotspot detection, resource allocation, and triggering of proper migrations. Finally, live migration of multiple joint VMs is a hot topic and requires more efforts for improvement and optimization given that only a few works have started proposing solutions for it [15].

Regarding session and application state migration, Albatross [21] proposes a technique for live migration in multitenant databases in a shared storage architecture that initially creates a snapshot of the database on the destination host and then uses several iterations copying the state incrementally to minimize the unavailability window. Zephyr [22] efficiently migrates a live database in a shared nothing transactional database architecture. Differently from Albatross, Zephr's virtual disks are locally attached to every node and, to minimize service unavailability, the destination of a VM migration starts serving new transactions while the "old" VM source completes the still active old transactions. This is partially similar to our original proposal (see later), but in Zephyr, during migration, requests at the destination force a pull on the data page from the source. Furthermore, any transaction at the source accessing a migrated page must restart at the destination.

Slacker [23] is an end-to-end database migration system that optimizes the impact of migration changing the throttling rate with which persistent state is migrated from the source to destination. Moreover, it uses recovery mechanisms to stream updates from the source to the destination. To avoid straining the other tenants at migrating nodes, a Proportional-Integral-Derivative (PID) controller monitors average transaction latency to adjust throttling.

ProRea [24] represents a live database migration approach that combines proactive and reactive measures, in order to reduce page faults and improve buffer pool handling compared to purely reactive approaches. To prepare migration, the source sets up local data structures and migration infrastructure and sends an initial message to the destination to create an empty database and sets up its local migration infrastructure. ProRea proactively migrates hot pages (i.e., recently accessed and in the buffer pool)/ New transactions start at the destination and pull pages on-demand. To complete the migration, the source additionally pushes pages which have not been transferred during previous phase or as response of a pull request from the destination, similarly to our proposal.

Dolly [25] exploits VM cloning to spawn database replicas. In particular, it clones the entire VM of an existing replica, comprehensive of operating environment, database engine, and all settings and data; the cloned VM then synchronizes state with other replicas prior to processing application requests. Since creating a new database replica is a time consuming process (with overhead linearly dependent on size of replicated database), Dolly incorporates a model to estimate latency and uses this model to trigger replication in advance [26].

Going beyond the state of the art, *we propose a live and lightweight state migration mechanism integrated in the MCN architecture*. Our proposal aims at using existing management components and avoiding adding new support components (e.g., migration controllers and routers) to grant the widest possible interoperability and integration with existing cloud management frameworks. Therefore, to benefit from the OpenStack features, we leverage the standard Heat service to create VM replicas that are able to start and manage the connection to the old instance including the database migration by proper configuration of the managed Heat template files.

Finally, although many activities have proposed various database migration mechanisms to move data in the most efficient way, *very little attention* has been given to the challenging task of live migration in a timely way [26]. Hence, *our proposal goes further* in relation to this aspect by including novel resource monitoring techniques combined with congestion prediction models.

3 MCN Complete Background and Architecture

To facilitate the full understanding of our original proposal and to make this paper self-contained as much as possible, this section gives an overview of the MCN archi-

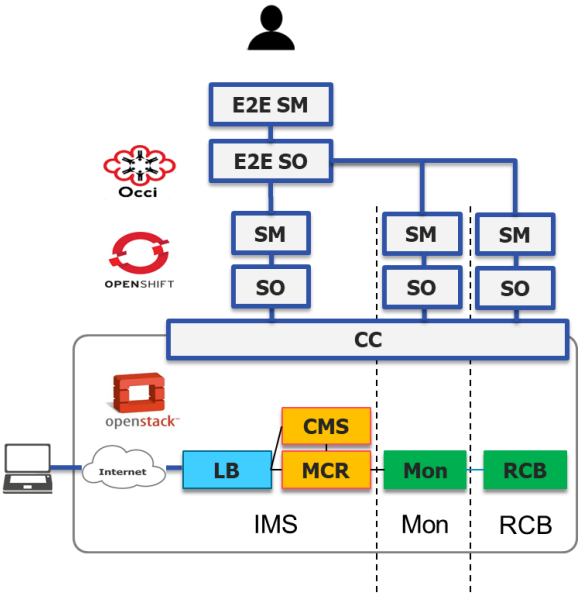

Fig. 1. MCN Architecture Instantiation.

tecture. We focus specifically on the services and functionality that are most central to our proposal. For additional details about the general MCN architecture, interested readers can refer to [27].

The main objective of MCN is to develop a novel architecture and a set of cloud-related technologies to provision carrier-grade virtualized telco services to 5G mobile users. Cloud resources (not just networking-related ones) are used to support on-demand and elastic provisioning of mobile end-to-end services, thus exploiting the opportunities of processing/storage near end-points through virtualized edge nodes (edge computing).

The MCN architecture, supported by the Hurtle orchestration framework, is very modular. Its key concept is to make possible the different services to create other more complex end-to-end (E2E) composed services. To this purpose, its Service Management Framework enables and defines the tools to compose and orchestrate the MCN operations across multiple domains and service types. Notably MCN went beyond NFV in some key areas such as E2E service composition as well as scalability objectives [28].

The architecture has two key aspects, its lifecycle management and architectural entities. The technical phase of the lifecycle includes all activities from technical design all the way through to technical disposal of a service:

- Design of the architecture, implementation, deployment, provisioning and operation solutions. Supports Service Owner to "design" their service.
- Implementation of the designed architecture, functions, interfaces, controllers, APIs, etc.
- Deployment of the implemented elements, e.g., data centers, cloud controllers, etc.
- Provisioning of the service environment (e.g., network functions, interfaces, etc.).
- Operation and Runtime Management. At this stage the service instance is ready and running. Activities such as scaling, reconfiguration of service instance components (SICs) are carried out here.
- Discarding of the service with the release of SICs and the service instance itself is carried out here.

Fig. 1 illustrates an instance of the MCN architecture, which is based on some key components used by all the services of the architecture:

- The Service Manager (SM) exposes an external interface to the Enterprise End-User (EEU) and is responsible for managing SOs to request the creation of services instances. The SM programmatic interface (northbound interface, NBI) is designed so it can provide either a CLI and/or a UI. Through the NBI, the SM gives either EEU or SO, both classed as tenant, capabilities to create, list, detail, update and delete (EEU) tenant service instance(s). Its Service Catalogue contains a list of the available services offered by the provider. Its Service Repository is the component that provides the functionality to access the Service Catalogue.
- The Cloud Controller (CC) supports the SO requirements and service life-cycle management, providing the management interfaces used by SM and SO, abstracting from specific technologies that are used in the technical reference implementation.
- The Service Orchestrator (SO) creates, configures, orchestrates and manages every service instance in order to access functionality provided by the specific service. The SO component has the task of receiving requests from the SM and overseeing the deployment and provisioning of the service instance. Once the instantiation of a service is complete, the SO can oversee tasks related to runtime of the service instance and also disposal of the service instance. Service instances are tracked in the SO Registry component. The SO is a self-contained tenant process that runs within a container, managed by the CC whose primary responsibility is to manage, resources and external services required to deliver the tenant's service instance.

The SO is implemented as a collection of imperative and declarative code: this is named as the SO bundle. The imperative code is responsible for creating and managing (e.g. scaling in and out) the resource and service instances. The declarative code is embodied by representations of two types of related graphs:

- Service Template Graph (STG) defines services requirements and how they could and should be composed together. These requirements are hence represented as dependencies. The STG interface can be queried through the Service Managers NBI.
- Infrastructure Template Graph (ITG) defines how resources should be composed to be able to host Service Instance Components (SICs). For example, an Analytics service requires two virtual machines: one to handle compute execution and one to handle the storage backend, both of which are connected through a network. ITGs are handled by template documents, which can be placed upon different infrastructure service providers such as CloudSigma, Amazon EC2, as well as OpenStack and Joyent Triton. Finally, the SO

enables multi-region/zone deployments: it can either hold multiple ITGs in the SO bundle or compute them on the fly.

In particular, the SO in its turn consists of three key components: the SO Execution (SOE) that is responsible for procedural phases, the SO Decision (SOD) that is responsible for runtime decisions related to scaling out or in and SO Resolver that is responsible for composition. In order to operate, the Resolver component requires understanding the service dependencies a specific service requires. These requirements are logically described by the SO bundle STG.

In this paper we specifically propose elastic provisioning targeted to two MCN services: Monitoring as a service (MaaS) and Rating, Charging and Billing as a service (RCBaaS). Within the set of MCN implemented services, we have decided to focus on these two because they are widely used during normal system operations, as important support services that communicate with all the other services of the system. Thus, MaaS and RCBaaS must handle high workload conditions preserving scalability and must ensure high availability requirement for the correct functioning of the rest of the system.

MaaS enables the design, implementation, and test of monitoring mechanisms across the four different domains: radio access network, mobile core network, cloud data center and applications. MaaS is considered as a full-stack monitoring system equipped with the capabilities to provide monitor and metering functionalities in a large scope of telecommunication systems. Service stability of MaaS is achieved by making use, as its monitoring mechanism basis, of the solid and established Zabbix open-source project [29].

Zabbix is a software toolkit that provides an effective, scalable and reliable monitoring, with a wide range of monitoring performance indicators and metrics; it offers a distributed infrastructure where Zabbix agents can collect data locally on behalf of a centralized Zabbix server and can report the data to the server. Zabbix provides agents for a wide range of operating systems and supports both active and passive checks to monitor data and CRUD operations via JSON-RPC based API interface. MaaS retrieves information in polling mode using the Zabbix APIs and exchanging data with Advanced Message Queuing Protocol (AMQP) protocol based on the publish/subscribe model. MaaS provides an interface to retrieve at runtime the monitoring information from agents associated with the services to monitor; this interface allows services to dynamically subscribe and retrieve asynchronous data from the IaaS. Each monitored service using MaaS implements the interaction interface with the MaaS service and has to integrate the configuration of its Zabbix agents per node to be monitored in its provisioning or deployment. The wrapper objects support methods for retrieval of metrics without being locked-in to a specific realization of the MaaS.

Complementing MaaS in the composition, the RCBaaS (i.e., Cyclops[4]) is a service that collects information for accounting and billing purposes. RCBaaS is employed in MCN as a support service that takes as input the service consumption metrics, processes them, calculates the price to be charged to the user, and generates the invoice for payment. It allows to charge both End User (EU) and Enterprise End User (EEU) or service operator itself in a cloud, as-a service way. It also takes into account information about anomalous events (e.g., service failures, resource consumption over given thresholds, etc.) in order to correctly enforce different types of charging models. RCBaaS is integrated with other MCN mechanisms and components, such as the Rabbit Message Queue (RabbitMQ) used to collect asynchronous messages generated from different entities.

Once instantiated, RCBaaS subscribes to MaaS to receive monitoring data through the RabbitMQ server (which collects information from RabbitMQ clients), dynamically instantiated on MaaS to mediate the interaction towards RCBaaS. RabbitMQ clients receive monitoring values from Zabbix, translate them in the RCB format, and deliver messages to the RabbitMQ server used by the RCBaaS.

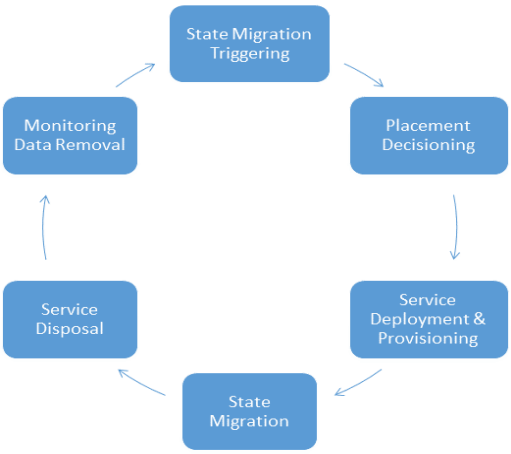

Fig. 2. High-level overview of our service state migration process.

4 Design Guidelines and Architecture of our Service Instance Migration Solution

This section describes the design principles behind our novel service state migration process, detailing the main phases of it and their technical differences if compared with the existing and related migration solutions in the field. We start by presenting the model behind our state migration and then illustrate in detail our primary design decisions, as well as their rationale.

Our goal is to enable dynamic, transparent migration of the whole state of an MCN telco service (typically represented by either persistent data in a database or simply objects stored in persistent mass memory). That state migration is key for manifold reasons and allows to really enhance system Development and Operations (DevOps) under many points of view. To enable maximum flexibility, it is proper to maintain the implementation of state migration mechanisms independent of the currently enforced migration policy, as widely accepted in the related literature. In addition, the state migration process should

[4] https://icclab.github.io/cyclops/

work independently of its trigger, e.g., motivated by Continuous Integration (CI)/Continuous Delivery (CD) operations, or generated by internal runtime resource monitoring, e.g., due to a sufficiently long overload peak). For instance, thanks to the support of our service instance migration solution, it is possible to modify a STG dynamically to transparently migrate it from one location to another and to maintain the state from the previous equivalent service instance. From the CI/CD perspective this works as i) modified STG and associated SO code pass unit and/or integration tests (CI) and ii) new service deployment executed transparently, including state migration (CD).

State migration operations are key for high-performance and reliable continuous provisioning of services that cannot suffer from downtime. The performance objectives of various services, such as scalability and responsiveness, may be significantly improved by optimizing the placements of involved VMs; thus the role of resource allocation is central to avoid bottlenecks due to congested links. In terms of reliability, fault tolerance is another major concern, in particular for critical services. State migration can significantly help in addressing fault-tolerance requirements by minimizing the impact of failures on service execution and by anticipatedly moving interested services in the case of failure predictions.

In our solution, the service state migration process is composed by multiple steps, as depicted in Fig. 2. The whole life-cycle of the physical host is observed by a monitoring system that can activate a trigger when the host is likely to go congested under an overload of incoming requests. When a migration trigger is activated (see below), our service state migration starts and our SO aims at finding the most suitable destination region (among the possibly heterogeneous ones offered by different cloud providers – original support to migration in heterogeneous cloud providers). Let us quickly note that placement decisions are taken at the service orchestration level and are out of the scope of this specific paper (internal management goal of MCN CC, please see the example in the following part of the section).

On migration trigger, an empty copy of the same service is deployed and provisioned on the new and less loaded destination, (re-)binding the new service with the same references earlier attached to the origin service. After this, our original core migration phase executes, by pushing all the service state toward the destination in order to replicate exactly the same situation of the origin host. Finally, after all the service state has reconstituted in the target destination, the old service instance is unbound and disposed in order to release the previously utilized resources promptly. Before restarting monitoring the newly created service instance, we reset the collected monitoring data in order to refresh the instance state by deleting old measurements that are related to the previous service location and no longer valid.

The overall goal of our infrastructure support to state migration is to coordinate a set of mechanisms that allow moving the whole internal state of a service instance to another instance created from scratch at runtime, with null or very limited service interruption (also these continuity requirements are rarely addressed by the existing state migration literature and represent an original technical aspect of our contribution). In addition to the placement decision, we have designed and implemented all the previously outlined orchestration and migration activities, commonly used during service state migration, by adopting a pre-copy approach. Initially, the service is monitored using the Zabbix-based MaaS functionality as the main potential trigger for migrations

In our solution, to properly manage the gathered monitoring data, also in the perspective of further analysis when overloading is detected, we have also added an ad-hoc functionality between Zabbix and the selected trigger. This functionality realizes either black-box and gray-box analysis techniques, in a completely modular and transparent way for the applications. In particular, in the following parts of the paper, we will consider our gray-box model implementation to realize behaviour predictions capable of smoothing peaks and fluctuations, by at the same time amplifying the monitored resource metric growth [30]. This is motivated by the fact that gray-box modelling is recognized in the literature as a simple baseline solution for multi-parametric systems control, which acts as a low-pass filter to detect usage resources in order to trigger events related to resource scarcity or specific service placement problems. It tends to offer both high hit-rate and good overall performance, even when the model information is partial or incomplete [31]. In addition, the gray model is particularly suitable as it can analyse a system with only few discrete data points. This permits to generate forecasts of next values even when decision makers only have a limited set of historical data. In particular, in our solution, these characteristics of gray-box modelling allow us to start host observation just after the VMs are created. In fact, in our prototype of the proposed MCN infrastructure, we derive the one-step ahead prediction via the gray model approach for the utilization percentage of each monitored resource (Ures) as:

$$Ures_{t+1} = f_{GM}(Ures_t, Ures_{t-1}, \dots, Ures_{t-k}), \quad (1)$$

where $f_{GM}()$ is the gray model function and k is the number of historical values that we consider for the prediction, thus influencing model accuracy.

Hence, we combine all the one-step ahead predictions (1) into a polynomial linear formulation with all the P parameters considering both the host resources, comprehensive of CPU, memory, network and disk operations, and application requirements present in the system SLA criteria, including Round Trip Time (RTT) and throughput in order to define the overall resource usage ($Uhost$) into the equations (2):

$$\begin{aligned} Uhost_{t+1} = {} & x_1 * Ucpu_{t+1} + x_2 * Umem_{t+1} + x_3 * Unet_{t+1} \\ & + x_4 * Udisk_{t+1} \\ & + x_5 * Urtt_{t+1} + x_6 * Uthr_{t+1}, \end{aligned} \quad (2)$$

where: $Uhost_{t+1}$ is the overall resource usage on the host one-step ahead; $Ucpu_{t+1}$ is the CPU usage one-step ahead; $Umem_{t+1}$ is the memory usage one-step ahead; $Unet_{t+1}$ is the network bandwidth usage one-step ahead; $Udisk_{t+1}$ is the read/write disk operations usage one-step ahead; $Urtt_{t+1}$ is the deviation from the minimal RTT

value one-step ahead; $Uthr_{t+1}$ is the deviation from the maximum throughput value one-step ahead; x_i is the weight for each resource and is used to augment or weaken the relevance of specific resources to be able to adjust the monitoring system in relation to the importance of the single resource to monitor, depending on the specific type of service (CPU-/memory-/storage-intensive). For an example of proper x_i selection, please refer to the experimental result section. Of course, for x_i the following equation holds:

$$\sum_{i=1}^{P} x_i = 1 \,, \qquad (3)$$

Finally, we define a threshold T that we compare with the $Uhost_{t+1}$ to detect if the host is overloaded and, hence, to activate our trigger and the consequent migration procedure. In particular, in relation to $Uhost_{t+1}$ we are able to detect either to move just one single VM instance or to migrate multiple VM instances due to high forthcoming congestion:

$$(Uhost_{t+1} - T) \propto N_{VM} \,, \qquad (4)$$

where N_{VM} is the number of VM instances to migrate.

Let us stress that we keep out of the scope of our implementation description here the decision of the target place where to migrate the targeted service instance because it may depend on the internal management goal of the MCN CC. Once the target destination place is identified, the state migration procedure can start. In particular, we split service state migration time into epochs and every epoch is related to a particular service state; note that, even if not original per se from the algorithmic/methodological point of view, the epoch exploitation approach is completely new in state migration for elastic cloud provisioning of telco infrastructures. We define the following epochs:

- E_1: the interval to re-create the same environment onto the new selected host;
- E_2: the interval to perform the first data migration phase (push phase);
- E_3: the interval to perform the residual second data migration phase (stop-and-copy phase);
- E_4: the interval to release the unused resources.

According to this epoch definition, our solution exploits the following algorithm for service state migration:

- **Step 1** (E_1): create the new state skeleton, consisting of M_{VM} (independent from N_{VM}) VM instances and prepare the target place to receive the state to be moved (operated by the SO).
  - **1.1** VM instances creation and startup
  - **1.2** VMs configuration to recreate the same environment
- **Step 2** (E_2): freeze the database, with the associated service state, and migrate the whole service state towards the newly created VMs, while the old VMs continue to collect data and to be able to serve requests. Thus, we adopt a modification of the pre-copy based approach to migration. This is the initial "push phase" where all the state for the captured time instant coincident with the start of the E_2 epoch is moved from the origin service to the pre-created destination service.
  - **2.1** push all the data collected in VMs (at the time E_1 starts)
  - **2.2** change service references to retrieve the "new" service from the new VMs
  - **2.3** delete the data moved to be fully aware of the data collected during Step 2 (not moved yet)
- **Step 3** (E_3): this is the "stop-and-copy phase" and pushes the residual data collected during Step 2 to the new VMs in order to update the service state with the latest information
- **Step 4** (E_4): dispose the old overloaded VMs to release the associated resources and lighten the workload on the overloaded host.

We designed the above state migration procedure to achieve several goals: (i) to minimize data losses and service unavailability time, independently from the amount of data and time of the migration; (ii) to integrate different algorithms and techniques to manage the specific temporal epochs in a more adaptive and distributed way among different service instances, for example by introducing priority settings (high-priority sessions) or fault-tolerance requirements.

In our specific use case, we decided not to further overcomplicate the existing MCN framework and, for simplicity, we opted to manage requests for only one instance ($M_{VM} = N_{VM} = 1$), thus without distributed management. However, the proposed design and implementation have been made to be easily extensible towards distributed instance management. When the push phase is terminated, the service is active on the new VMs and the newly created instance is the target service instance that serves all the incoming requests from clients. The service is re-established on the new host and the origin service instance does not receive more incoming data.

As soon as the origin service instance is no longer operational, we perform the stop-and-copy phase, pushing all the residual data inserted on the old instance previously and belonging to the temporal epoch, towards the destination service instance. In this way, we run out all the old epochs data in a timely fashion in order to converge and realign the service state and complete state migration consistently. After the whole state has been moved, our implementation deletes the stack resource of the old service instance to release both migration-related internal and application-related resources. Finally, before restoring the monitoring component, we reset the buffered monitoring values read from Zabbix to enable new future triggering actions without old spurious data.

Let us highlight that we prototyped and experimented the above solution by considering the migration of the real MCN RCBaaS: during its whole service instance migration procedure, RCBaaS has demonstrated to be responsive, with complete transparency for both system operations and end-users and with limited performance degradation (see Section 6).

Fig. 3 details the steps of our service state migration process. In addition to starting the typical SM/SO activities to deploy and provision a new RCBaaS service (1, 2.1, 2.2, 2.3, and 2.4), the first step regarding the state migration is the triggering of the whole migration. The resource usage values provided by Zabbix (3.1 and 3.2) are stored into a sliding window array with a fixed buffer length that can be easily configured programmatically. The buffering of monitoring time-series of data samples enables the exploitation of many algorithms for resource usage analysis and prediction. In our case, we have implemented a lightweight first-order gray model filtering module (4) as an external triggering decision algorithm. When this trigger has been activated, the steps already explained in Fig. 2 start: preparation of the target place to receive data (5); data migration towards the target VMs (6); stack disposals (7); reset of the buffered monitoring (8). Finally, we return to store monitoring data from the considered targets and restart the loop.

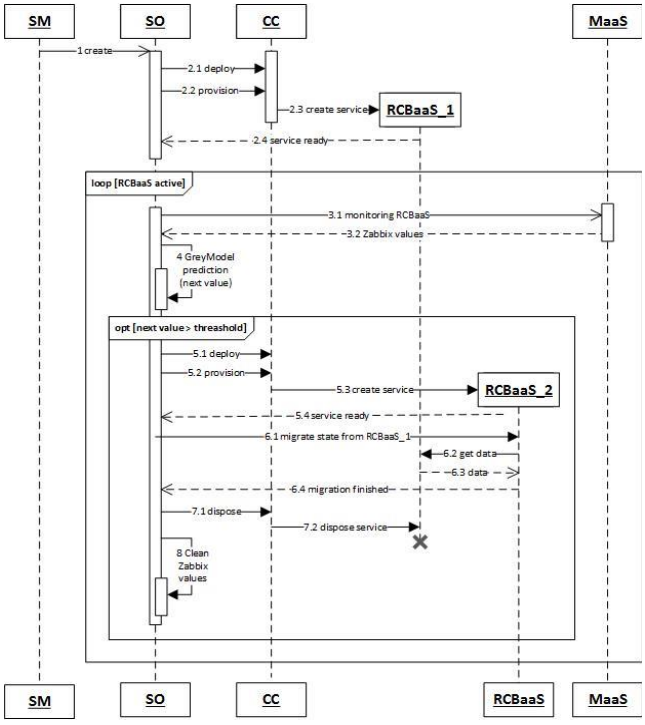

Fig. 3. State migration process

5 Most Relevant Implementation Insights

Here we go in-depth into the technical details of how exactly we have implemented our original stateful service migration solution by concentrating on a few most challenging and original aspects. We start from the RCBaaS division into smaller and more specific VM instances in order to monitor accurately and effectively resource consumption; this opportunity of VM separation may exhibit in several use cases and application scenarios. Then, we focus on how to efficiently implement resource monitoring in MCN. Finally, we concentrate on efficient data migration with no impact on service continuity.

5.1 Preliminary Work on Service VM Separation

RCBaaS is composed by two main components that interact very frequently: Cyclops [32] and InfluxDB [33].

Cyclops is the core component of RCBaaS, responsible for the service logic for accounting and billing purposes. Cyclops is divided into three micro-services: User Data Records (UDR) that collects the usage data from a source, such as OpenStack, CloudStack, SaaS, PaaS, etc., and stores it in the database; Rating and Charging (RC) that uses the records generated by the UDR to calculate, in relation to the cloud resource rate, the charge data records; Billing that generates an invoice.

InfluxDB is an open-source time-series database (particularly suitable to keep track of large amounts of sensor data), widely used in particular in the context of real-time analytics. In MCN it is the backend of RCBaaS as the monitoring metrics repository to keep the history of all service measurements.

To apply our service state migration procedure and mechanisms in a practical valuable case, we have focused on the RCBaaS monitoring service as a monolithic VM for the sake of maximum separation. As a first step, to enable the state transfer migration process, we split it into disjoint and dynamically bound VMs. The RCBaaS-VM performs the core operation of the monolithic service and contains only the Cyclops components. The InfluxDB-VM contains the backend component where data are stored. These two VMs are introduced to act on the state datastore (that is our original component that actually stores the service state), in our case the InfluxDB instance, without requiring any change on the RCBaaS, thus actually behaving as a stateless component in this uncoupled version.

This specific case can be generalized in several application domains and for several practical services. To effectively and efficiently use our solution for stateful service migration, it is recommendable to identify a few coarse-grained macro-blocks to be the migration targets and possibly to isolate service state into a single (or very few) of them, thus benefitting from stateless migration in all other cases.

```
class MyList(list):
    def append(self, item):
        list.append(self, item)
        if len(self) > myparameters.WINDOW_SIZE: self[:1]=[]
class SOD(service_orchestrator.Decision, threading.Thread):
    def getGrayModelValues(gateway, composedList):
        values = []
        for list_py in composedList:
            list_java = ListConverter().convert(list_py, gateway._gateway_client)
            nextValue = gateway.entry_point.nextValue(list_java)
            values.append(float("{0:.4f}".format(nextValue)))
            return values
    def monitoring(self):
        self.monitor = RCBaaSMonitor(myparameters.MAAS_DEFAULT_IP)
        self.hosts_cpu_load = []
        self.hosts_cpu_load.append(MyList())
        metrics = self.monitor.get(myparameters.ZABBIX_INFLUXDB)
        self.hosts_cpu_load[0].append(metrics[0])
        if len(self.hosts_cpu_load[0]) >= myparameters.ZABBIX_MIN_READING:
            cpu_load_GM = getGrayModelValues(self.gateway, self.hosts_cpu_load)
        if cpu_load_GM > myparameters.TRIGGER_VALUE:
            print "Trigger activated. I'm going to move the VM state."
            ...
                        time.sleep(myparameters.ZABBIX_UPDATE_TIME)
```

Fig. 6. SO Monitoring Code Snippet.

We cleanly split the RCBaaS service by using OpenStack Heat, which implements an orchestration engine that allows to launch multiple composite cloud applications based on Heat template files. In particular, we properly configured the Heat template to create two different VMs for Cyclops and InfluxDB when launched and to execute two script files located on the Cyclops-VM after the creation, in order to automatically configure the IP address of the InfluxDB-VM to which RCBaaS sends its data for storage purposes. Fig. 4 outlines an excerpt of the script code used on the RCBaaS VM, which edits 3 configuration files, one for each Cyclops micro-service.

5.2 Monitoring

The monitoring system is mainly based on two main components: MaaS and our gray model. MaaS runs a Zabbix server that communicates with the distributed monitoring agents within the VMs that are instantiated during service provisioning. MaaS aggregates the resource information retrieved from the agents. Each monitoring agent is designed to collect networking statistics, to normalize and process the raw monitoring data, and to send its processing results to the Zabbix server of MaaS. Every deployed service that needs to integrate with MaaS for resource monitoring purposes requires the installation and configuration of a Zabbix agent, as shown in Fig. 5. This is automated through a proper Heat template file to allow active resource monitoring.

In the SO implementation, the monitoring information required is set to allow the exploitation of our gray model. The parameters to activate the trigger are set here as well and they direct the activation of state migration. Fig. 6 shows a snippet of the SO implementation for the monitoring part. We check the CPU load as the value to monitor for determining host overload. Other metrics may be used with simple proper configuration, always via Heat files. The CPU load considers the queue length of processes waiting to be processed, i.e., a common and widespread parameter to detect host workload. We set 10 as the CPU load threshold for the InfluxDB-VM and, given that the VM has 2 virtual CPUs, it means the trigger is activated when at least 5 processes per single core are waiting to be processed. The CPU load value to be considered is returned by the gray model using the last five values read from MaaS and is stored into a sliding window array. At startup, we consider 3 minimum readings to invoke our gray model, in order to avoid false positives and thus to prevent from trigger activation caused by few anomalous readings. Finally, the monitoring values from MaaS are retrieved every 1 minute, which is the maximum sensitivity configurable in Zabbix (lower bound on the period for refreshing monitoring data). This setting has been selected to have a relatively fine-grained periodicity and consequently good responsiveness.

```
ip=$1
echo "-> Cyclops-udr configuration file"
python string_substitution.py /home/ubuntu/cyclops-udr/src/main/webapp/WEB-
INF/configuration.txt InfluxDBURL= http://$ip:8086
echo "-> Cyclops-rc configuration file"
python string_substitution.py /home/ubuntu/cyclops-rc/src/main/webapp/WEB-
INF/configuration.txt InfluxDBURL= http://$ip:8086
echo "-> Cyclops-billing configuration file"
 python string_substitution_js.py /home/ubuntu/cyclops-udr/install/openstack/config/config.js
            url: ' "http://'$myip':8086/db/udr_service"," ' "http://'$ip':8086/db/grafana",'0
```

Fig. 4. Cyclops configuration.

```
apt-get install -y zabbix-agent
sed -i -e 's/ServerActive=127.0.0.1/ServerActive=160.85.4.28:10051/g' –e
's/Server=127.0.0.1/Server= 160.85.4.28/g' -e 's/Hostname=Zabbix server/#Hostname=/g'
/etc/zabbix/zabbix_agentd.conf
service zabbix-agent restart
```

Fig. 5. Zabbix Agent Configuration in an InfluxDB-VM heat template.

5.3 Service Instance Migration

The service instance migration step consists of two phases in the presented RCBaaS case: i) a complete InfluxDB dump from the old to the new instance; ii) storage and migration of new data inserted on the old database instance while the migration occurs and after the dump operation. Regarding the database dump, it is the core operation of the state migration to move the data of the old InfluxDB instance to the new instance. To reduce the duration of this phase, we adopt compression techniques for the old InfluxDB that, after migration, is extracted and put into the target InfluxDB data folders.

The dump operation is the most critical one under different perspectives. It could have a non-negligible duration, even tens of seconds, in relation to the amount of database instances and records to transfer, due to both external operations (e.g., data compression, movement, extraction, and database restart) and internal database operations to synchronize the new state.

Therefore, in order to minimize database unavailability and to preserve overall service continuity, we have originally designed and implemented the second phase. As soon as the old data have been copied into the archive, all dumped data in the old databases are dropped, to be sure that successively inserted data have not been transferred during migration and, as a consequence, to further relieve the old database performance.

When the database dump has completed and the target InfluxDB has been configured and made available, we select all the new data at the old instance and move them to the target InfluxDB, merging with the data already migrated during the dump. Delving into some finer implementation details, this mechanism required to save these "during-migration" entries as a JSON file, and then to convert them into a LineProtocol format file used by InfluxDB to insert data on-the-fly[5]. This copy of the new data to the target InfluxDB instance completes the data migration step.

Fig. 7 graphically summarizes all the operations performed during our data migration process, by distinguishing the actions executed on the old and those run on the new InfluxDB-VM instances. In particular, the first three blocks from the left refers to the database dump operation (phase i) and the last two blocks to the storage of new data inserted during the migration process (phase ii).

[5] https://docs.influxdata.com/influxdb/v0.9/write_protocols/line/

Fig. 8 shows an excerpt of the code used inside the SO component to invoke the migration script and to move data between the two instances. The command sent to the remote host is implemented through a Python library: it is possible to access the new instance and execute, from the SO implementation, the script already prepared on the newly created InfluxDB-VM instance, by passing the IP address of the old instance to migrate as a parameter.

```
try:
    ssh = paramiko.SSHClient()
    ssh.set_missing_host_key_policy(paramiko.AutoAddPolicy())
    key = BUNDLE_DIR + myparameters.MIGRATION_KEY
    ssh.connect(self.so_e.influxdb_ip,  username=myparameters.MIGRATION_USERNAME,
     key_filename=key)
    command1='cd '+myparameters.MIGRATION_SCRIPT_FOLDER+';'
    command2='  bash  '+myparameters.MIGRATION_SCRIPT_NAME+'
     '+self.so_e.influxdb_ip_old
    command=command1+command2
    stdin, stdout, stderr = ssh.exec_command(command)
    ssh.close()
    break
  except paramiko.ssh_exception.NoValidConnectionsError:
    print "VM not ready"
    time.sleep(2)
```

Fig. 8. SO Data Migration.

6 In-the-field Experimental Evaluation and Simulation Work

As already stated, one of the key contributions of the research work presented here is that our service state migration solution has been implemented and completely integrated in the articulated MCN architecture and infrastructure prototype. As a valuable side-effect, differently from what available in the existing literature, we are able to report in-the-field experimental results about how our solution prototype behaves over a real deployment environment (Section 6.1).

However, several technical elements make hard to report here a technically sound performance evaluation comparison with other similar solutions in the literature and based on in-the-field experimentation. In particular, the primary technical barriers and complexity are i) the impossibility/difficulty of integrating existing solutions into the MCN architecture or into a similar orchestration framework for carrier-grade virtualized services (e.g., because of lack of released source codes), ii) the need to compare the E2E performance and thus to consider "similar techniques" when supporting RCBaaS, which is unfeasible, and iii) the impossibility/difficulty to be deployed over the same cloud support stack (not frequently based on OpenStack).

Anyway, by considering the most relevant related literature in terms of reported performance results, some existing proposals for live stateful migration consider either pre-copy or post-copy approaches, but only a very few of them propose a hybrid solution, similarly to us. Generally speaking, pre-copy and post-copy approaches have demonstrated worst performance in terms of latency, unavailability time, and errors (abortion of all the active and incoming transactions at the moment of the migration) [34, 35]. Hybrid solutions can provide better performance and downtime because they are more flexible and allow continuing to keep the service active while the infrastructure performs migration management. For instance, Zephyr [35] has shown to achieve a total time to migrate a tenant (with one db of 10^6 rows) of around 10-18s; to this, an additional latency has to be considered as perceived by final users. [36] proposes a live VM migration based on checkpointing/recovery and trace/replay techniques. The migration overhead and delay are much higher than Zephyr and, in WAN networks, the solution presented in [36] introduces a non-negligible latency (magnitude order of seconds for all the benchmarks used) to serve requests during migration management (if compared with so-called "regular operations", in absence of migration occurrence). Finally, [37] evaluates the performance of live VM migration, in particular to quantify the slowdown/downtime experienced by end user applications during migrations. With 600 concurrent users, the downtime is in the order of some seconds and, in particular, the collected performance indicators show a strong dependency on the number of users to serve. In our solution, migration performance indicators are not related to the users number because the RCBaaS implementation is organized into different VMs with different tasks, thus clearly decoupling service management for users (Cyclops VM) and for data (InfluxDB VM). Note that all the above solutions do not deal with high workloads that are typical of carrier-grade telco applications, differently from our MCN-integrated solution.

Because of the technically barriers mentioned above, in order to improve anyway the performance evaluation reported here, we also include in this section the results of the simulation work accomplished to compare the performance of our MCN state migration with "more traditional" reactive-only or proactive-only approaches to state migration (Section 6.2).

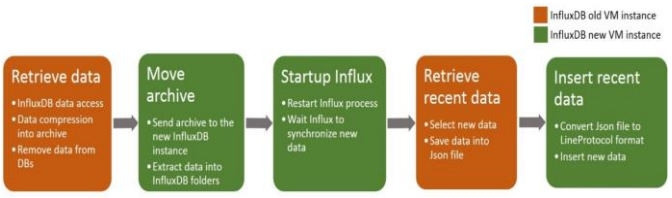

Fig. 7. Essential steps of the proposed data migration process.

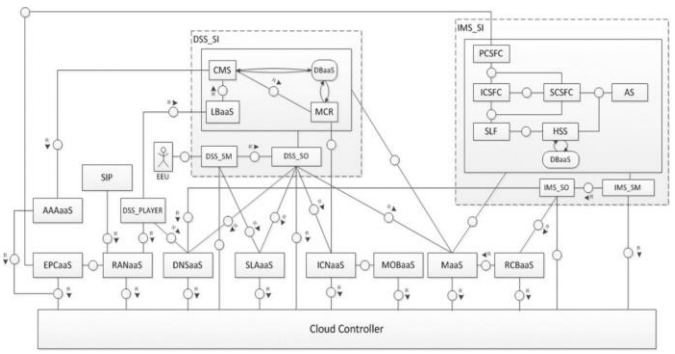

Fig. 9. MCN E2E Deployment Scenario.

TABLE I

	Operation	Time (s)
At Startup	MaaS monitoring setup (Zabbix)	9.5
	Setup Gray Model Java classes	3.5
Monitoring	Read Zabbix values	0.5
	Calculate Gray Model value	< 0.1
Stack	Create a new stack	45
RCBaaS VM	Cyclops-UDR deploy	25-26
	Cyclops-RC deploy	15
	Cyclops-Billing deploy	30-32
	Total time to setup the VM	90-100

6.1 In-the-field Experimental Measurements of Performance Indicators

To better understand the complete MCN testbed and to have a precise idea of the overall complexity of typical MCN deployment environments, we quickly introduce an End-to-End (E2E) MCN deployment scenario and omit the details about all MCN services, for the sake of brevity and focus on our novel approach. The integrated E2E MCN deployment scenario, shown in Fig. 9, creates a very heterogeneous and broad-based cloud platform composed of a high number of services and components, which have been evaluated following several different tests. It should be noted that a common methodology [38] has been used to evaluate a complex of scenario including: Digital Signage System (DSS) to evaluate the over-the-top applications for playing content through digital signature services; IP Multimedia System (IMS) to evaluate the support of video and voice for mobile users; Follow-Me-Cloud (FMC) to better demonstrate the capabilities of mobility prediction and dynamic content placement, particularly important to show how information-centric networking technologies can be fostered by prediction and how the FMC concept is supported.

In these very wide and complex scenarios, the importance of each component greatly varies in relation to the specific use case considered. In the following we will focus only on the RCBaaS service. RCBaaS is the component for state migration management, thus, we will show only the experimental results that strictly highlight in isolation the behavior of our state migration functionality among all the functionalities present in MCN.

We performed several tests to cover all the steps and phases discussed in the previous sections, by deploying stacks on the OpenStack platform and using RegionOne, as the default region. OpenStack "Bart" is a testbed provided by MCN consortium that runs the OpenStack services, based on Kilo version. In particular, for sake of performance evaluation, we desired to stress the RCBaaS utilization with a wide range of different workloads in order to observe the performance of the newly introduced function. Important was the perceived limited unavailability time, notwithstanding dynamic state migration and synchronization. All performance tests reported in this subsection refer to average values that we measure across multiple runs, each exhibiting an overall low variance (<5%). In addition, based on preliminary experimentations and evaluations, for the reported case of RCBaaS, we have set the weights of equation 2 as x1=x4=1/3, and x2=x3=x5=x6=1/10. In fact, in this specific case, CPU usage and disk utilization have shown to be the most effective indicators to trigger our migration facility: on the one hand, CPU has shown to be the primary motivation of overall system slow-down in overloaded RCBaaS; on the other hand, disk performance is crucial in the targeted E2E scenario because the database and persistence support are the second bottlenecks after CPU, with their overload propagating negative effects on the whole RCBaaS, as also highlighted by the following results. As a general consideration, we suggest a methodological approach where weights are preliminary determined in a first approximation based on identified bottlenecks and then counterchecked/refined with a few experimental runsto determine local optima; the determination of their globally optimal values is out of the scope of this paper.

Table I shows the performance related to: service initialization, Zabbix monitoring, the gray model usage, the new target stack creation, and the RCBaaS-VM creation. Let us note that only the monitoring performance, in our case negligible, may potentially cause performance issues because they are repeated continuously during the service life-cycle. The other reported operations are only performed at startup, thus they do not introduce any latency during system operations at runtime.

Fig. 10 shows the performance of the data migration for different amounts of data. We report average values measured on multiple runs because the overall performance varies slightly from test to test mainly depending to the network conditions and the load on the physical host where the VM is running.

We divide the overall latency into several contributions that allow us to distinguish the different phases; in particular, we measure and define main timing as follows:

- T_{vmconn}: time the SO takes to connect to the InfluxDB-VM, or in other words, the latency time between when the trigger becomes active and the data migration starts;
- $T_{compress\_move}$: time to compress data into a tar.gz archive and move to the new VM instance;
- T_{delete}: time to delete all the data from the old InfluxDB-VM instance, directly proportional to the number of database to delete;
- $T_{extract}$: time to extract the archive into the InfluxDB folders of the new VM instance;
- $T_{restart}$: time to restart the InfluxDB service in order to get the update about the new data;
- T_{sync}: time used by Influx process for internal synchronization after the dump.

Other time latencies are related to the storage and insertion of the new data during the migration, that is the time necessary to: get all measurements, retrieve data inserted into a JSON file, convert the JSON file into the LineProtocol format and insert the data into the databases. We do not report these latencies in the chart, Fig. 10, because we assume the amount of data inserted during the migration is limited and, thus, the associated time can be migrated (in the order of 0.1-0.2s to move a dozen

of records).

Let us stress that during the overall state migration procedure the database unavailability, considering the latest assumption that ignores the new data time retrieval, is limited to the process related to the measurements deletions (T_{delete}), proportional to the number of databases presents but always very low and, for typical execution and average migration, below 1s, that grants relatively negligible unavailability, thus proving the effectiveness of the proposed state migration function and its wide applicability to stateful services service state migration.

Summing up, depending on the dimension of the state to be migrated, the overall service migration process time can go from 112 seconds for up to two millions of records (namely, 100 seconds to setup the target VM and 12 seconds for data migration) to 590 seconds (for 100 millions of records). In any case, we are able to achieve a fully scalable behavior with good overall performance, mainly limited by the InfluxDB internal operations (T_{sync}), that is the real bottleneck of the solution, but it does not affect the unavailability time but only the duration time of the migration.

6.2 Simulation Results about Migration Latency and Data Loss Compared with Baseline

For additional quantitative evaluations and comparisons, we employed CloudSim [39], an extensible and widely adopted simulation toolkit that enables the modelling and simulation of cloud computing environments. In particular, the CloudSim simulation framework supports (and is recognized as suitable for) the modelling and creation of infrastructures and application environments for distributed multiple clouds, as in our testbed of Section 6.1. In particular, we map the RCBaaS service infrastructure into the simulator by creating:

- two datacenters, used to migrate our VMs to;
- one host per datacenter, with 2GB RAM and 250GB storage each;
- two VMs for each host, with 512MB RAM, 100GB storage, and 1 CPU each;
- one process per VM representing Cyclops and InfluxDB components.

In this simulated environment, we extensively compared our hybrid solution with two baseline approaches, i.e., reactive-only and proactive-only. The reactive baseline adopts the approach of migrating all data at once when the host is already overloaded. Thus, it is characterized by small migration time (because it does not include data reconciliation), but also may cause significant data loss in case of high amount of data received during the migration process. The proactive approach, instead and dually, moves the data in advance before the service migration takes place and then reconciles the new inserted data after service migration. This generates small data loss but higher migration time if data variability is high.

Fig. 11 and 12 show, respectively, the results about the time required to complete migration with the proactive baseline, and the data loss during migration with the reactive baseline. The migration time for the reactive approach and the data loss in the proactive one are not reported in the figures because they are almost invariant

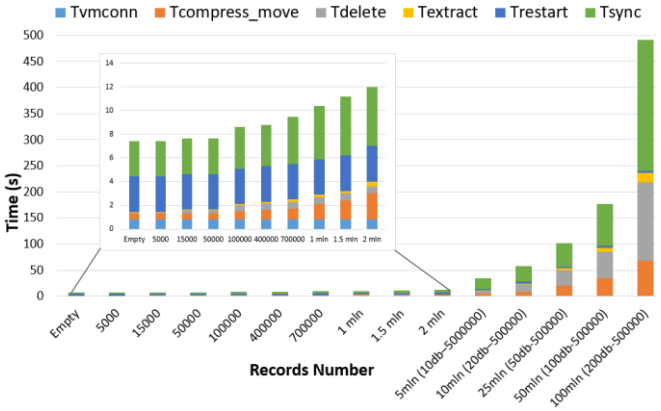

Fig. 10. Performance Evaluation of RCBaaS State Migration.

when modifying data variability and equal to the best cases of the other baseline approach.

Let us note that the results previously reported in Fig. 10 have been obtained by considering an average data variability, i.e., 1B/s, in our real testbed. Our hybrid migration solution works more proactively than the reactive baseline, by keeping at the same time a negligible data loss. In fact, as shown in Fig. 11 and 12: under 1kB/s of data variation rate, if the time constraints of the supported services allow them, proactive approaches show to grant low migration time and data loss; above 1kB/s, the migration time distance between the two baselines becomes relevant and a hybrid solution should consider the best tradeoff to adopt depending on the specific characteristics and requirements of the application scenario.

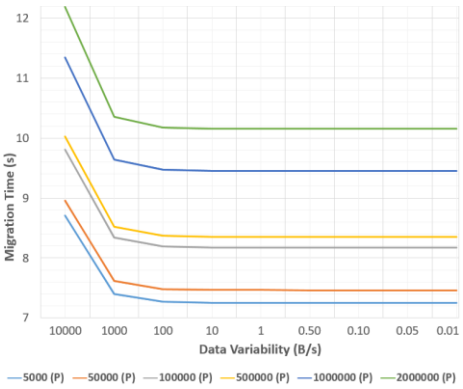

Fig. 11: Migration time for Proactive (P) baselines.

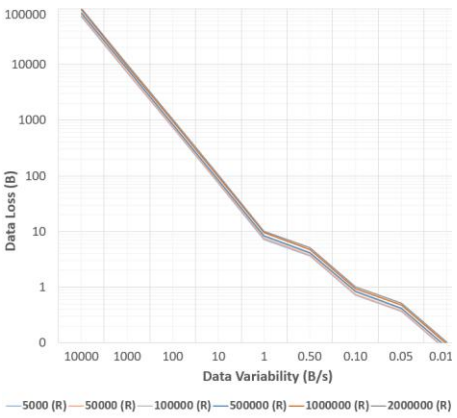

Fig. 12. Data loss for Reactive (R) baselines.

In other words, the results in Fig. 11 and 12 highlight the relevance of being able to dynamically adapt the migration behavior to expected data variability, as in our proposed prototype where we use the delta number of records inserted into the database between two consecutive time intervals.

Finally, Fig. 13 reports about how we have modeled the workload in the targeted datacenter in our simulations, by showing how the workload changes in relation to the number of users and the amount of data stored into the database, and ii) under which circumstances the datacenter is overloaded and our migration support is likely to be triggered. In fact, by considering the characteristics of our RCBaaS scenario and also based on our in-the-field experimentation of it, in our simulations the datacenter workload shows strong dependence on database storage (percentage of overall host storage) and is limitedly affected by the number of users who concurrently invoke the service.

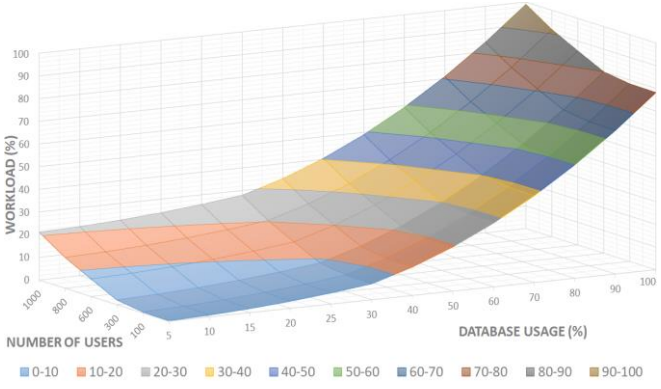

Fig. 13. Datacenter workload for our RCBaaS simulations.

7 Conclusions and Ongoing Work

The implementation and integration work accomplished for service state migration within the MCN project has demonstrated that it is possible to achieve cost-effective prototyping with full compliance with industry-mature standards and frameworks in the field. We have already validated our approach experimentally and performed a large experimental campaign of performance indicators measurements. The reported results confirm that the proactive triggering mechanism adopted in our solution can significantly minimize the service down time (crucial key performance indicator for migration of critical telco services) in the cases of very unexpected and abrupt traffic peaks, by imposing a very limited overhead on the overall support infrastructure.

Fueled by these significant results, we are working on two main ongoing research directions. On the one hand, we are deploying the realized solution, already widely tested in the geographically-distributed MCN cloud testbed, in a federated cloud environment with heterogeneous orchestrators that need to interoperate across different domains. On the other hand, we are running extensive experiments to thoroughly assess the impact of abrupt external load peaks, out of the direct control of our monitoring infrastructure, to mitigate the potentially disruptive effect on ongoing service state migration sessions.

Acknowledgements

We want to thank the European Commission for co-founding the FP7 Large-scale Integrating Project (IP) Mobile Cloud Networking (MCN) project under the 7th Framework Programme, grant agreement no. 318109.

References

[1] A. Edmonds, T. Metsch, A. Papaspyrou, A. Richardson, “Toward an Open Cloud Standard,” in *IEEE Internet Computing*, vol. 16, no. 4, pp. 15-25, Jul. 2012.

[2] ETSI, “Network Functions Virtualisation (NFV); Architectural Framework”, in *GS NFV 002 V1.1.1*, Oct. 2013.

[3] M. Stine, “Migrating to Cloud-Native Application Architectures”, O’Reilly, Mar. 2015.

[4] B. Sousa, et al., “Toward a Fully Cloudified Mobile Network Infrastructure”, in *IEEE Transactions on Network and Service Management*, vol. 13, no. 3, pp. 547-563, Sept. 2016.

[5] C. Clark, et al., “Live Migration of Virtual Machines”, in *Proceedings of 2nd USENIX Symp. Networked Systems Design & Implementation (NSDI)*, 2005.

[6] M. R. Hines, U. Deshpande, K. Gopalan, “Post-copy live migration of virtual machines”, in *ACM SIGOPS Operat Syst* Rev 43(3): 14–26, 2009.

[7] A. C. Arpaci-Dusseau, R. H. Arpaci-Dusseau, “Information and control in gray-box systems”, in *Proceedings of the 18th ACM Symp. Operating systems principles (SOSP)*, pp. 43–56, 2001.

[8] OpenStack Heat. Available online at: https://wiki.openstack.org/wiki/Heat.

[9] Nirmata. Available online at: http://nirmata.com.

[10] Hurtle. Available online at: http://hurtle.it.

[11] OASIS (Topology and Orchestration Specification for Cloud Applications). Available online at: http://docs.oasis-open.org/tosca/TOSCA/v1.0/csd03/TOSCA-v1.0-csd03.html.

[12] T. Binz, G. Breiter, F. Leymann, T. Spatzier, “Portable Cloud Services Using TOSCA”, in *IEEE Internet Computing*, vol. 16, no. 3, pp. 80-85, 2012.

[13] Cloudify. Available online at: http://getcloudify.org.

[14] M. F. Bari, et al., "CQNCR: Optimal VM Migration Planning in Cloud Data Centers", in *Proceedings of the IFIP Networking Conf.*, 2014.

[15] F. Callegati, W. Cerroni, “Live Migration of Virtual Network Functions in Cloud-Based Edge Networks”, in *Proceedings of IEEE SDN for Future Networks and Services (SDN4FNS)*, pp. 1-6, 2013.

[16] F. P. Tso, G. Hamilton, K. Oikonomou, D.P. Pezaros, “Implementing Scalable, Network-Aware Virtual Machine Migration for Cloud Data Centers”, in *Proceedings of IEEE 6th Int. Conf. Cloud Computing*, 2013.

[17] X. Meng, V. Pappas, and L. Zhang, “Improving the Scalability of Data Center Networks with Traffic-aware Virtual Machine Placement”, in *Proceedings IEEE INFOCOM*, pp. 1–9, 2010.

[18] V. Mann, et al., “Remedy: Network-aware Steady State VM Management for Data Centers”, in *Proceedings of the 11th Int. IFIP TC6 Conf. Networking*, vol. 7289, pp. 190–204, 2012.

[19] D. Jayasinghe, et al., “Improving Performance and Availability of Services Hosted on IaaS Clouds with Structural Constraint-aware Virtual Machine Placement”, in *Proceedings of the IEEE Int. Conf. on Services Computing*, pp. 72–79, 2011.

[20] T. Wood, et al., “Sandpiper: Black-box and Gray-box Resource Management for Virtual Machines”, in *Computer Networks*, vol. 53, issue 17, pp. 2923–2938, 2009.

[21] S. Das, S. Nishimura, D. Agrawal, and A. El Abbadi, “Albatross: Lightweight Elasticity in Shared Storage Databases for the Cloud Using Live Data Migration.”, in *Proceedings of the VLDB Endowment*, vol. 4, issue 8, pp. 494-505, 2011.

[22] A. J. Elmore, S. Das, D. Agrawal, A. El Abbadi, “Zephyr: Live Migration in Shared Nothing Databases for Elastic Cloud Platforms”, in *Proceedings of the ACM SIGMOD Int. Conf. Management of Data*, pp. 301-312, 2011.

[23] S. K. Barker, Y. Chi, H. J. Moon, H. Hacigümüs, P. J. Shenoy. "Cut Me some Slack: Latency-aware Live Migration for Databases”, in *Proceedings of the 15th Int. Conf. Extending Database Technology*, pp. 432-443, 2012.

[24] O. Schiller, N. Cipriani, and B. Mitschang, “ProRea: Live Database

Migration for Multi-Tenant RDBMS with Snapshot Isolation", in *Proceedings of the 16th Int. Conf. Extending Database Technology*, pp. 53-64, 2013.
[25] E. Cecchet, R. Singh, U. Sharma, P.J. Shenoy, "Dolly: Virtualization-driven Database Provisioning for the Cloud", in *Proceedings of the 7th ACM SIGPLAN/SIGOPS Int. Conf. Virtual Execution Environments (VEE)*, 2011.
[26] S. Sakr, "Cloud-hosted Databases: Technologies, Challenges and Opportunities", in *Cluster Computing*, vol. 17, issue 2, pp. 487–502, 2014.
[27] T.M. Bohnert and A. Edmonds, "MCN D2.2: Overall Architecture Definition, Release 1", mobile-cloudnetworking.eu/site/index.php?process=download&id=124&code=93b79f8f5b99f67a6cdc28369c05b65f624cfee7, Oct 2013.
[28] A. Edmonds and T.M. Bohnert, "MCN D2.5: Final Overall Architecture Definition, Release 2," mobile-cloudnetworking.eu/site/index.php?process=download&id=263&code=aa37ba15e0479f1ecb7a696876ab498d7f3ff0ef, April 2015.
[29] "Zabbix: An Enterprise-Class Open Source Distributed Monitoring Solution", available online at: http://www.zabbix.com, 2015.
[30] H.L. Wang, "Gray Cloud Model and Its Application in Intelligent Decision Support System Supporting Complex Decision", in *Proceedings of the Int. Colloquium on Computing, Communication, Control, and Management*, vol. 1, pp. 542–546, 2008.
[31] P. Bellavista, A. Corradi, C. Giannelli, "Evaluating Filtering Strategies for Decentralized Handover Prediction in the Wireless Internet", in *Proceedings 11th IEEE Symp. Computers and Communications*, 2006.
[32] Cyclops. Available online at: http://icclab.github.io/cyclops.
[33] InfluxDB. Available online at: https://influxdata.com.
[34] T. Hirofuchi, et al., "Reactive Consolidation of Virtual Machines enabled by Postcopy Live Migration", in *Proceedings of the 5th ACM Int. Workshop on Virtualization Technologies in Distributed Computing*, 2011.
[35] A.J. Elmore, et al., "Zephyr: Live Migration in Shared Nothing Databases for Elastic Cloud Platforms", in *Proceedings of the 2011 ACM SIGMOD Int. Conf. Management of Data*, 2011.
[36] H. Liu, et al., "Live Virtual Machine Migration via Asynchronous Replication and State Synchronization", *IEEE Transactions on Parallel and Distributed Systems*, vol. 22, issue 12, pp. 1986-1999, 2011.
[37] W. Voorsluys, et al., "Cost of Virtual Machine Live Migration in Clouds: A Performance Evaluation", in *Proceedings of the IEEE Int. Conf. Cloud Computing*, 2009.
[38] B. Sousa, L. Cordeiro, P. Simoes, A. Edmonds, S. Ruiz, G. Carella, M. Corici, N. Nikaein, A. S. Gomes, E. Schiller, T. Braun, T.M. Bohnert, "Towards a Fully Cloudified Mobile Network Infrastructure," *IEEE Trans. Network Service Management*, Aug. 2016.
[39] R.N Calheiros, et al. "CloudSim: a Toolkit for Modeling and Simulation of Cloud Computing Environments and Evaluation of Resource Provisioning Algorithms", in *Software: Practice and Experience*, pp. 23-50, 2011.

Paolo Bellavista (SM'06) graduated from the University of Bologna, Italy, where he received a Ph.D. degree in computer science engineering in 2001. He is now an associate professor at the University of Bologna, Italy. His research activities span from mobile agent-based middleware solutions and pervasive wireless computing to location/context-aware services and management of cloud systems. He serves on the Editorial Boards of IEEE TNSM, IEEE TSC, Elsevier PMC, Springer WINET, and Springer JNSM.

Antonio Corradi (M'77) graduated from University of Bologna, Italy, and received MS in electrical engineering from Cornell University, USA. He is a full professor of computer engineering at the University of Bologna. His research interests include distributed systems, middleware for pervasive and heterogeneous computing, infrastructure for services and network management.

Andy Edmonds received the M.Sc. degree in distributed systems from Trinity College Dublin. He is a Senior Researcher with the Zürich University for Applied Sciences. He was in industrial and academic positions in Siemens, Infineon, Intel, and the Distributed Systems Group, Trinity College in Dublin. He has been involved in several European FP6 and FP7 projects. He's is been on the FP7 MCN project, as Deputy Technical Coordinator. His research interests include distributed system architectures, service-oriented architectures, and cloud computing.

Luca Foschini (M'04) graduated from University of Bologna, Italy, where he received PhD degree in computer science engineering in 2007. He is now an assistant professor of computer engineering at University of Bologna. His interests include pervasive wireless computing environments, system and service management, context-aware services, and management of Cloud computing systems.

Alessandro Zanni graduated from University of Modena and Reggio Emilia, Italy, in 2011. He is currently a candidate for a PhD degree in computer science engineering at University of Bologna, Italy. His research interests include pervasive computing, middleware for cloud-sensors system, Internet of Things and management of cloud computing systems.

Thomas Michael Bohnert is a Professor with the Zürich University of Applied Sciences and the Head of the ICC Laboratory. His interests are focused on enabling ICT infrastructures, ranging across mobile/cloud computing, service-oriented infrastructure, and carrier-grade service delivery (Telco + IT). He was with SAP Research, SIEMENS Corporate Technology. He serves as an Regional Correspondent (Europe) for the IEEE Communication Magazine News section (GCN). He is the founder of the IEEE Broadband Wireless Access Workshop and holds several project and conference chairs. He acts as MCN's Technical Coordinator.