Optimising Microservice-based Reliable NFV Management & Orchestration Architectures

Thomas Soenen*, Wouter Tavernier*, Didier Colle* and Mario Pickavet*
*Ghent University - imec: {firstname.lastname}@ugent.be

***Abstract*—A highly reliable set up of Management and Orchestration (MANO) functionality is crucial for telecom operators in order to support demanding NFV-powered telecom services such as elastic emergency or security services, or other applications requiring fast, dynamic and reliable (re-) provisioning processes. NFV MANO functionality controls the entire life cycle of such services, from instantiation and configuration to monitoring, migrating, scaling and terminating them. In this paper, we introduce tunable and scalable mechanisms that provide MANO with high availability and fault recovery, two reliability facets that are barely covered in the data plane dominated state-of-the-art. The mechanisms use state sharing and distributed load balancing in the context of both a centralised and distributed microservice-based architecture. The proposed mechanisms are unique as they are able to quantitatively characterise the trade-offs between both the degree of reliability and the associated cost in terms of bandwidth and computing power. This feature allows us to introduce a cost function to determine which configuration of the mechanism is optimal with respect to an operator's needs.**

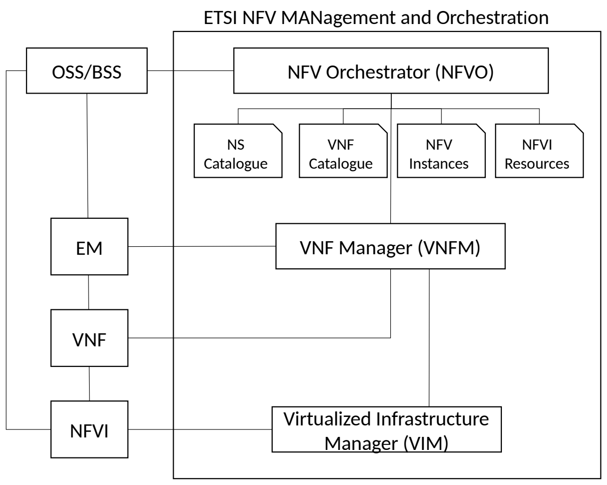


Fig. 1: ETSI MANO framework architecture.

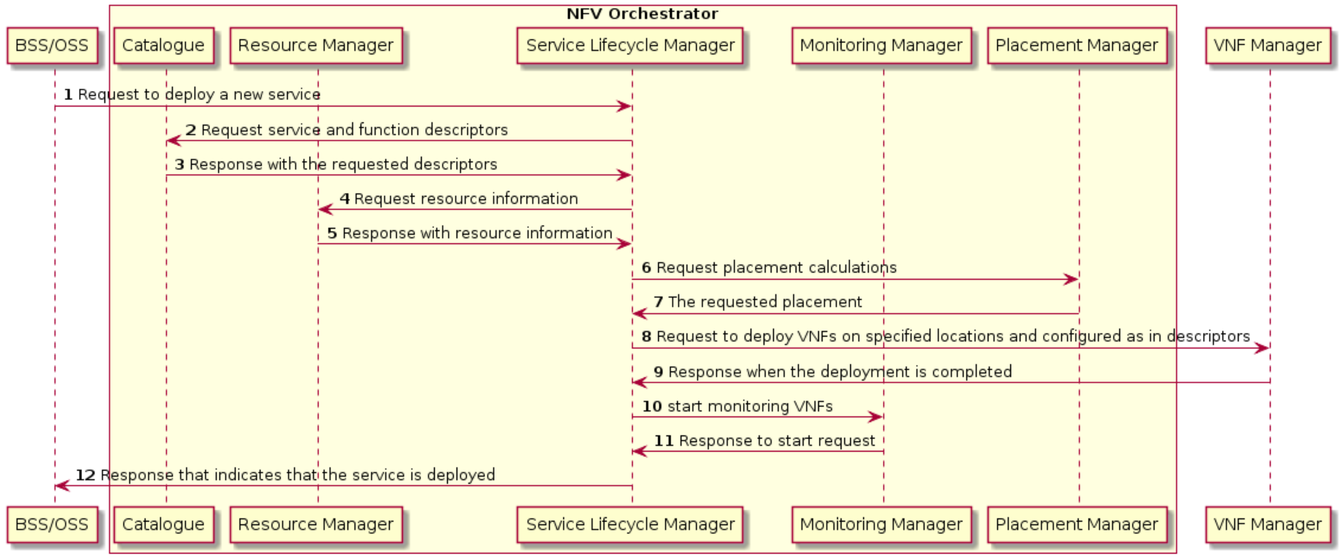
I. Introduction

Through the adoption of Network Function Virtualisation (NFV), telecom operators aim to become more agile in providing services to customers, while simultaneously reducing operational costs. Instead of implementing network functions (e.g load balancers, firewalls and proxy servers) as dedicated hardware devices in a telecom operator's network, they get deployed as software applications in data centers on general purpose hardware. By steering traffic through a chain of such virtual network functions (VNF), operators create network services such as VPNs, Content Delivery Networks and Security Services. The advantages [1] of this approach are numerous: it allows operators to (i) deploy network services on demand, (ii) optimise resource usage, (iii) improve the Quality of Service (QoS), and (iv) reduce maintenance costs as it lowers the number of dedicated hardware in their network.

To capitalise on these potential advantages, operators need to automate and optimise their NFV processes. Satisfying a customer's request for a new network service requires calculating where in the network the VNFs are best deployed. VNFs should be monitored throughout their life cycles, so that they can be migrated or scaled in case more or other resources (e.g. compute power, storage and memory) are required. By automating these and other NFV processes, services become available on demand and the customer's QoS improves. This explains the current emphasis on NFV Management and Orchestration (MANO) frameworks [2], as they automatically manage and orchestrate the life cycles of services, VNFs and the infrastructure on which they run. Multiple MANO framework implementations, such as OSM [3] and SONATA [4], are being developed. To prevent a wild grow of frameworks with incompatible APIs, the European Telecommunications Standards Institute (ETSI) streamlined the MANO framework design by proposing a functional software architecture [5] (Fig. 1) that has been accepted throughout the sector.

The ETSI MANO framework architecture has three main components: the NFV orchestrator (NFVO), the VNF manager (VNFM) and the Virtual Infrastructure Manager (VIM). The NFVO manages all tasks on the network service level. It interacts with the operator's operations and business support system (OSS/BSS), through which customers make requests (e.g. instantiate, update and terminate a service). The NFVO processes a request through a set of tasks, such as on-boarding a VNF and steering traffic. The NFVO receives monitoring feedback which can trigger it to scale the service. The VNFM operates on the level of the VNF. It receives management (e.g. deploy, configure and terminate) instructions for VNFs from the NFVO, which it executes through its interface with the VNFs. The third major component, the VIM, manages the bare infrastructure (NFVI) that is used to run the VNFs on.

Operators require their NFV platforms to be carrier grade [6], i.e. highly reliable and well tested. Existing reliability research in NFV focuses on the data plane by ensuring the deployed network services run without hiccups. Monitoring probes [4] [7] deployed near or inside VNFs quickly detect issues with resources or connectivity, and services are mapped so they are protected against resource failure [8]. In this paper we focus as one of the first on the control plane (i.e. MANO framework) aspect of reliability in NFV platforms. Section



NFV Orchestrator
BSS/OSS
Catalogue
Resource Manager
Service Lifecycle Manager
Monitoring Manager
Placement Manager
VNF Manager
1 Request to deploy a new service
2 Request service and function descriptors
3 Response with the requested descriptors
4 Request resource information
5 Response with resource information
6 Request placement calculations
7 The requested placement
8 Request to deploy VNFs on specified locations and configured as in descriptors
9 Response when the deployment is completed
10 start monitoring VNFs
11 Response to start request
12 Response that indicates that the service is deployed

Fig. 2: NFVO design for the network service instantiation workflow with a centralised manager.

II explains why our work focuses on MANO frameworks implemented as microservices. In Section V, we propose two mechanisms to add reliability, in terms of high availability and fast fault recovery, to such MANO frameworks. Section VI demonstrates these mechanisms and aids in configuring them for optimal reliability.

II. MANO Framework and Microservices

To explain why we focus on MANO frameworks implemented as microservices, we first take a closer look at the NFVO. It needs to support a range of workflows such as instantiating, scaling and migrating network services. The instantiation workflow creates a new service, while the scaling workflow changes how a running service is constructed to increase or decrease its processing power. Such a change can include adding more VNFs, replacing a running VNF with a new and more performant VNF or granting the running VNFs more resources. A migration workflow relocates running VNFs to different infrastructure.

With such workflows in mind, the NFVO design can be refined by introducing a set of modules which execute well-defined subsets of the workflows. Fig. 2 shows how the instantiation workflow can be performed by 5 such modules. The Service Life Cycle Manager (SLM) receives a service instantiation request from a customer trough the BSS. The SLM fetches descriptors containing deployment and management information for the service from the Catalogue. Next, the SLM requests the Resource Manager, the interface between the NFVO and the VIM, for infrastructure information (e.g. topology and usage). The SLM forwards this to the Placement Manager, which calculates the optimal placement for the service. The obtained information is used to provide the VNFM with instructions to deploy and connect the different VNFs. After the SLM instructs the Monitoring Manager to start monitoring the VNFs, it informs the BSS that the service is deployed.

Fig. 2 is an example of an NFVO design for the instantiation workflow with 5 modules: one SLM invokes 4 other modules in the correct order and provides them with the data they require to perform their task. Fig. 3 shows a different NFVO design for the same workflow, without a centralised module managing it. Here, each module invokes the next module in the chain directly. The information that is being gathered and shared by the modules is the state of the request. Throughout the workflow, it accumulates data such as service and VNF descriptors and infrastructure topologies. Due to the complexity and size of the topology, the size of the state can run high. As (parts of) this state are being transmitted, one should make sure that the network is not overloaded. One way of dealing with this is using abstractions of the topology [9] which are considerably smaller.

To obtain the complete set of modules in the NFVO design, charts such as shown on Figs. 2 and 3 should be made for every workflow. Modules can off course be reused by multiple workflows. Which type of design to follow for the NFVO is up for discussion, whether it is the one with the centralised component in Fig. 2 or the more distributed one in Fig. 3. An advantage of the centralised case is that the SLM only sends a subset of the (large) state to each module, while in the distributed case each module forwards the entire state. An advantage of the distributed design is that it doesn't require the implementation of a complex centralised manager. This paper does not argue which design is better. We describe reliability mechanisms for both designs and propose a cost function to determine which design provides the best reliability with respect to the resources it consumes.

State-of-the-art MANO frameworks [4] [10] implement the ETSI MANO architecture as microservices. The term microservice emerged in the context of cloud computing and De-

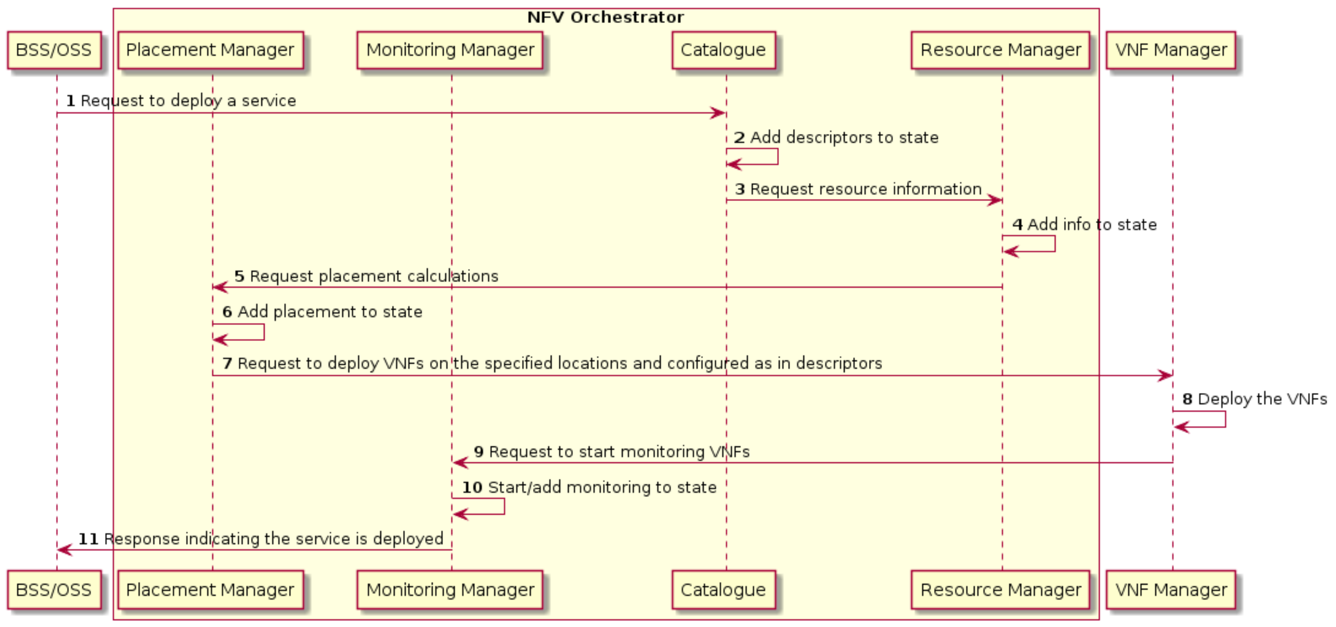

NFV Orchestrator
BSS/OSS
Placement Manager
Monitoring Manager
Catalogue
Resource Manager
VNF Manager
1 Request to deploy a service
2 Add descriptors to state
3 Request resource information
4 Add info to state
5 Request placement calculations
6 Add placement to state
7 Request to deploy VNFs on the specified locations and configured as in descriptors
8 Deploy the VNFs
9 Request to start monitoring VNFs
10 Start/add monitoring to state
11 Response indicating the service is deployed

Fig. 3: distributed NFVO design for the network service instantiation workflow.

vOps [11] to indicate a software architecture that structures an application as a collection of loosely coupled services. These fine grained services are developed and deployed as standalone processes, i.e. microservices [12], and interact through remote calls over a network. Designing software through microservices enables incremental and iterative releases that rapidly change the application. The decomposed nature of the architecture allows autonomous teams to develop and run the modules in parallel without the need to shut down and restart the entire application. NFV is heavily supported and stimulated by open source initiatives, so the microservice-based architecture allows multiple organisations to work together autonomously to build MANO framework functionalities.

For this reason, we focus our reliability work on NFVOs and MANO frameworks where each module in the design is implemented as a separate microservice which communicates with the other microservices over a network.

III. Problem description

Telecom operators require their NFV platform to be carrier grade, implying it is highly reliable and well tested. In terms of reliability, it should provide a high availability and a fast fault recovery. This paper focuses on reliability in the MANO framework part of NFV platforms. In terms of high availability, the MANO framework should be processing as many of the requests as possible in a timely manner, especially time-critical requests such as the deployment of emergency services or monitoring messages reporting resource scarcity for running services. In terms of fault recovery, the MANO framework is vulnerable to a range of faults. It is subject to low-level hardware failures of the infrastructure hosting a microservice. Natural disasters like earthquakes and power outages can cause the hosting resources to fail, as can malicious attacks. These faults can cause the loss of state during a workflow, making the MANO framework unable to finish it. Looking at Figs. 2 and 3, such a loss of state occurs if one of the microservices crashes when it is processing the request.

In this paper, we provide mechanisms that make MANO frameworks implemented as microservices highly available and tolerant against faults that cause the loss of state. To the best of our knowledge, this has not been described before in the state-of-the-art. To ensure high availability, we will deploy additional instances of the microservices and a state preservation mechanism between these microservices prevents that workflows are restarted from the beginning if a fault occurred. We will show that the responsiveness of both the fault recovery and the availability is related to the characteristics of the mechanisms and their resource usage. This allows us to quantify and optimise the trade-offs between the availability, in terms of the added time ratio (R_a), fast fault tolerance, as the recovery ratio (R_r), and resource usage, with

$$R_a = \frac{t_c}{t_m}, R_r = \frac{t_r}{t_c} \tag{1}$$

where t_c is the time it takes to finish a workflow without a fault occurring, t_r is the worst case time interval it takes to finish a workflow in case one fault occurs and t_m is the time it takes to manage a workflow by a monolithic implemented framework without a reliability mechanism. A lower value for R_a indicates that the reliability mechanism doesn't add much to the processing time of a workflow, while a lower value for R_r indicates a faster fault recovery, and thus a better predictability of how long a workflow is going to take.

IV. Related Work

Reliability in MANO frameworks is barely covered in scientific work. [13] mentions it as a challenge. Looking at

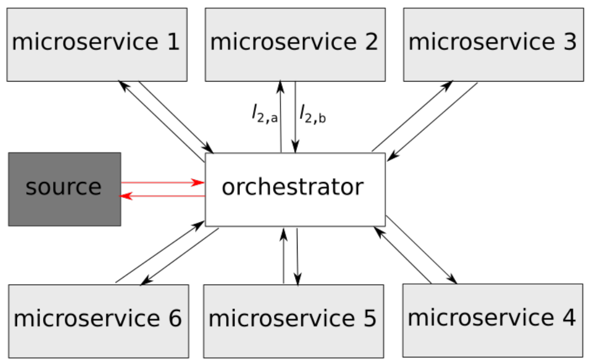


Fig. 4: Abstracted centralised network of microservices.

microservice research, some work has discussed reliability in terms of recovering from faults. The available solutions for fault tolerance are based on redundancy. Multiple instances of the same microservice are running, ensuring that if one crashes, another takes over its tasks. Examples of such solutions are Serfnode [14], Kubernetes [15] and Akka [16]. What they lack compared to our solution is state preservation. If a microservice goes down, our mechanism preserves parts of the already accumulated state for a workflow, ensuring a faster recovery from the fault, while their mechanisms just restart the workflow. Our mechanisms also have the advantage of being able to minimise the recovery time, in terms of resource consumption.

[17] proposes a microservice architecture that enables scalable and resilient self-management of microservices. This architecture selects a leader that is responsible to restart failed microservices. Although this architecture looks familiar to our centralised architecture shown in Fig. 4, it lacks any concern for the built-up state in a microservice or for the duration of the recovery.

V. Solution

In this section, we describe our scalable mechanisms that add fault recovery and high availability to a both a centralised and distributed microservice-based implementation of the MANO framework architecture.

A. Centralised MANO framework

Fig. 4 shows an abstraction of the design in Fig. 2. The orchestrator is the microservice with the central role, like the SLM in Fig. 2. It receives the requests and calls the other microservices in a specific order to go through the workflow.

The mechanism to make this design reliable starts by deploying additional instances of the orchestrator. There should be at least one additional instance, more can be deployed when the mechanism scales up. In what follows, we describe the mechanism for the case with two instances. Extrapolating to a scenario with more instances is straightforward. The SLMs send heartbeat messages to each other at a fixed rate h_r, indicating they are alive. If an SLM does not receive a heartbeat from the other orchestrator for a period longer than time-out interval t_h, it assumes it encountered a fault.

The source, the BSS in case of an instantiation workflow, sends the request to both SLMs. To prevent both orchestrators from processing the same requests, they share the load in a distributed way. When initialised, each orchestrator generates a Universally Unique Identifier (UUID) [18] and includes it in the heartbeat message. By comparing these orchestrator UUIDs, each SLM gets a rank. By matching this rank to the parity of the request UUID, we ensure that only one SLM processes a request.

Each SLM logs the requests that the other orchestrator is processing. If it stops receiving heartbeat messages from the other instance, it uses this log to process the requests that the other instance did not finish. This assures that no requests remain unfinished if an orchestrator encounters a fault. To reduce the fault recovery time, both SLMs share the state that is built up throughout the workflow. The orchestrator that processes a request sends the accumulated state to the other orchestrator after a microservice completed a task in the workflow. If the processing orchestrator fails, the other SLM continues the workflow onward from the last task it received an updated of. This reduces the orchestrator failure recovery time, as the recovering orchestrator does not need to restart the workflow from the beginning. For example, if the SLMs in Fig. 2 share the state after the Resource Manager responds, and if the processing SLM crashes while the Monitoring Manager is called, the recovering SLM can continue this workflow from the request to the Placement Manager onward. The frequency of these state sharing events is a parameter of the mechanism.

The state sharing mechanism adds fault recovery to the MANO framework. High availability is reached due to the scalability of the mechanism, as we can increase the number of SLMs without adding complexity. If the number of orchestrators is higher than 2, the orchestrators perform a modulo calculation on the request UUID to decide which one processes it, instead of using the parity. To ensure availability from the non-orchestrator microservices, they can be duplicated as well and also send a heartbeat message to the orchestrators.

Based on this description, we can derive expressions for t_c and t_r, for a design with an SLM delegating to n microservices. Each microservice m_i requires t_i time to finish. t_o is the time the orchestrator needs between tasks to process the input and to prepare the data for the next task. $l_{i,a}$ is the latency for messages from the orchestrator to m_i, $l_{i,b}$ from m_i to the orchestrator. t_c can be written as the sum of the time it takes each microservice to complete its task, the time the SLM intervenes and the latency between them:

$$t_c = \sum_i t_i + (n+1)t_o + \sum_i (l_{i,a} + l_{i,b}) \tag{2}$$

t_r can be written as $t_c + t_l$, with

$$t_l = l_{hb} + t_h + \max\left\{ \forall j \in [s_p, n] : \sum_{k=j-s_p-1}^{j} t_k + l_{k,a} + l_{k,b} + t_o \right\} \tag{3}$$

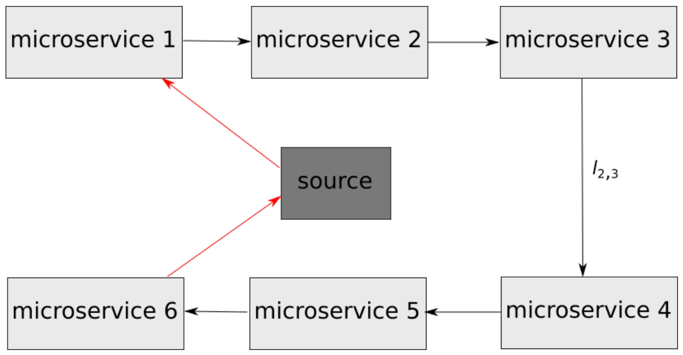


Fig. 5: Abstracted distributed network of microservices.

where l_{hb} is the latency of the heartbeat message. The processing SLM shares the state every s_p tasks. We can now write the worst case time interval that is lost due to an orchestrator fault t_l as Eq. 3. If the SLM crashes directly after sending out a heartbeat, the other orchestrator will take t_h time to notice the fault. If the state is shared every s_p tasks, we need to derive which combination of s_p adjacent tasks adds the most time to t_l. This is done in the last term of Eq. 3. Its easy to see that the lost time due to a crashed non-orchestrator microservice will not increase t_r if its heartbeat rate and time-out are identical to h_r and t_h, as long as there is always an instance of the microservice running. This factor is therefore not considered in Eq. 3.

To measure the impact of this mechanism on resource usage, we can evaluate the consumed bandwidth. The total bandwidth B_{tot} is the sum of the bandwidth of the normal microservice-based design B_{op} and the bandwidth added by the reliability mechanism B_{rel}, with

$$B_{rel} = h_m + h_s h_r + \frac{s_z}{t_c}(1 + \text{trunc}((n - s_o)/s_p)) \quad (4)$$

$$B_{op} = \frac{1}{t_c}\sum_i (P_{i,a} + P_{i,b}), \quad (5)$$

where h_s is the size of the heartbeat message, s_z the size of the shared state, $trunc$ the truncation operator and s_o the state sharing offset that indicates after how many tasks the first state sharing event takes place. $P_{i,a}$ is the size of the message from the orchestrator to m_i and $P_{i,b}$ the size from m_i to the orchestrator. h_m is the bandwidth by heartbeat messages from the non-orchestrator microservices.

These equations can be used to tune the reliability mechanism. By adjusting parameters like the number of microservices, the heartbeat rate and the state sharing period, we can look for optimal values for R_a, R_r and B_{tot}. Such an evaluation is demonstrated in Section VI.

B. Distributed MANO framework

To add reliability to the distributed design shown in Fig. 5, we deploy two instances of each microservice. Both instances share a heartbeat to indicate they are alive. Both receive every request and share the load similar to the centralised case. The instance that is not processing the request logs it, to continue the workflow when it no longer receives a heartbeat. Once a microservice is done with its task, it informs the other one.

t_c for this design can be written as

$$t_c = \sum_i t_i + \sum_{i=0}^{n-1} l_{i,i+1} \quad (6)$$

where $l_{i,i+1}$ is the latency for messages sent from microservice m_i to m_{i+1}. The worst case lost time due to a microservice encountering a fault is

$$t_l = \max\{\forall i \in [1, n] : l_{hb,i} + t_{h,i} + t_i\} \quad (7)$$

where $l_{hb,i}$ is the latency of the heartbeat and $t_{h,i}$ is the heartbeat time-out interval for m_i.

As each microservice needs to receive the entire state, we can express B_{rel} and B_{op} as

$$B_{rel} = \sum_i h_s h_{r,i} + \frac{1}{t_c}\left(\sum_{i=0}^{n-1} P_{i,i+1} + \sum_i p_c\right) \quad (8)$$

$$B_{op} = \frac{1}{t_c}\sum_{i=0}^{n-1} P_{i,i+1} \quad (9)$$

where $h_{r,i}$ is the heartbeat rate for m_i, p_c is the message size that a microservice sends to indicate it finished. $P_{i,i+1}$ is the size of the state sent from m_i to m_{i+1}.

VI. Evaluation

Varying the characteristics of the proposed reliability mechanisms and the MANO framework (e.g. the number of tasks and the heartbeat rate) results in different values for added time ratio R_a, recovery ratio R_r, bandwidth B_{tot} and compute power usage Q_r. As operators want to minimise each of these metrics, they can find the optimal configuration for their MANO framework by minimising the following cost function:

$$C(z) = \alpha R_a(z) + \beta R_r(z) + \gamma B_{tot}(z) + \delta Q_r(z) \quad (10)$$

z represents one configuration of all the parameters that define the system. R_a and R_r can be obtained using the equations in Eq. 1 and Section V. Q_r represents the additional compute power compared to the monolithic implementation. This is proportional to the number of microservices, i.e. $n+2-1$ for the centralised case and $2n-1$ for the distributed one. α, β, γ and δ are parameters that allow the operator to tweak the importance of each metric in the cost function. For example, if the quality of the reliability greatly out weights the resource consumption for an operator, α and β should be chosen high, while γ and δ are chosen low.

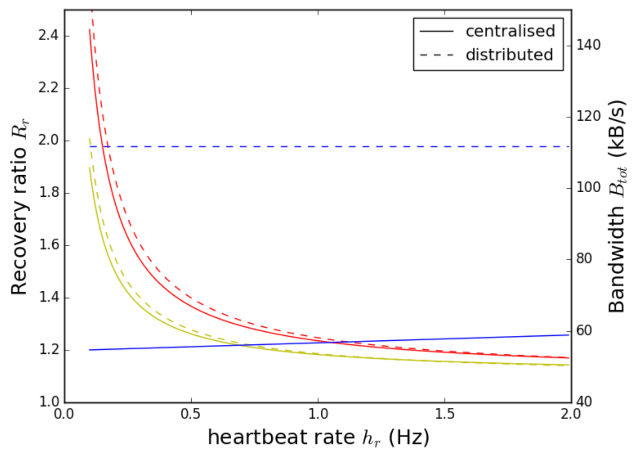


Fig. 6: R_r (red for $t_h = 5h_i$, yellow for $t_h = 3h_i$) and B_{tot} (blue) for varying h_r.

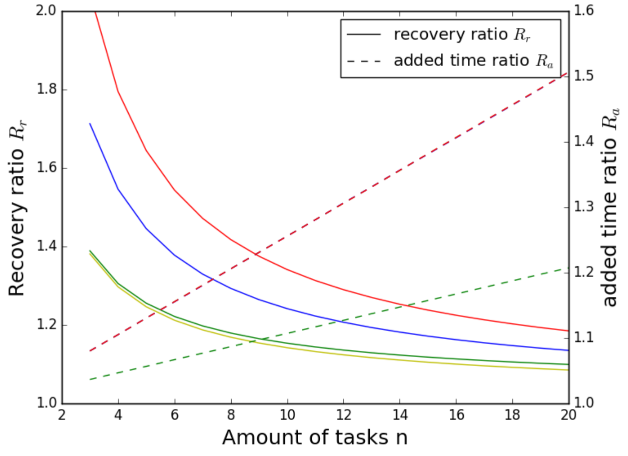


Fig. 7: R_r and R_a for n. Red, blue and yellow lines show centralised case ($s_p = 3, 2, 1$), green lines distributed case.

A. Ranges

We start by defining some ranges for z. Looking at Fig. 2, the time t_i that it takes m_i to finish depends on the task. Some tasks finish quickly, e.g. fetching the descriptors ($< 1s$), while others take more time, e.g. calculating the optimal placement ($> 60s$) which is NP-hard and installing monitoring probes ($> 20s$). Therefore, the range for t_i is $[1s, 60s]$. The workflows in the SONATA [4] show a range of $[5, 15]$ for n. t_m, the processing time of a monolithic implemented workflow, is the sum of t_i of each microservice in the workflow. Since the tasks vary for each workflow, the range for t_m is $[10s, 120s]$.

A heartbeat message only contains a UUID and is very small, i.e. $h_s = 0.1kB$. Since microservices will mostly run in data centers which have fast ISP connections, we set a low range (depending on the distance) for the latency of the heartbeat at $[0.01s, 0.2s]$. Since the size of data sent between microservices is still small compared to the available bandwidth in data centers, its delay is dominated by the latency between the microservices, which is the same as the latency range of the heartbeat.

The size s_z of the state greatly depends on the size of the topology. The size of a large topology, including usage information, can run into the mBs, while abstractions of it contain less detail with a size of $\sim 100kBs$. Based on the SONATA project, the range for s_z is $[100kB, 500kB]$. The size of the data sent between the orchestrators and other microservices is, based on which part they need, $[10kB, 300kB]$.

In between tasks, the orchestrator processes the received content and selects the section of the state for the next task. Based on profiling in SONATA, we set the duration for t_o as $0.15s$. The heartbeat rate h_r should be deducted by the operator by evaluating its influence on Eq 10. The time-out t_h of the heartbeat should be at least three times the heartbeat interval $h_i = 1/h_r$, to prevent false negatives due to latency.

B. Analysis

Fig. 6 shows the influence of the heartbeat rate h_r on R_r and B_{tot}. Red lines show R_r in function of h_r for $t_h = 5h_i$, the yellow lines for $t_h = 3h_i$. The blue lines show B_{tot}, which is independent from t_h. Other parameters are within their ranges defined in Section VI-A. A full line shows the centralised case ($s_p = 1$), a dashed line the distributed one. The figure shows that R_r drops logarithmic with increasing heartbeat rate for every scenario. Once $h_r > 1$, the gain in R_r diminishes, i.e. the recovery time doesn't imrpove anymore with increasing h_r. Although an increase in h_r has a bandwidth cost, Fig. 6 shows that the heartbeat part of B_{tot} is minimal for both the centralised and distributed case. B_{tot} is significantly higher for the distributed case.

Fig. 7 shows how R_r (full lines) and R_a (dashed lines) vary with the number of tasks n. The red ($s_p = 3$), blue ($s_p = 2$) and yellow ($s_p = 1$) lines show the centralised case, the green lines show the distributed case. R_a is independent of s_p, so only the red line is shown for the centralised case. Based on Fig. 6 a value of 2 was chosen for h_r. Fig. 7 shows that increasing the frequency of state share events in the centralised case by lowering s_p decreases R_r, implying that more share events result in a faster fault recovery. While R_r decreases with increasing n for every scenario, this increases the overall time the workflow takes, as indicated by an increasing R_a. Fig. 7 also shows that R_r for the distributed case and for the centralised case with $s_p = 1$ are almost similar in this scenario.

Fig. 8 shows the cost function from Eq. 10 for changing n and various combinations of α, β, γ and δ. Other parameters are the same as in the previous experiments. For the green line, the cost function was tweaked to favour high availability through R_a and recovery time through R_r over resource minimisation. It shows that for both the centralised (full) and distributed case (dashed) the optimal number of tasks lays between 5 and 8. For the red line, the cost function was tweaked to only favour R_r, showing an optimal n of 20 for both systems. This implies that if you only care for a fast recovery time, you divide the workflow in as much tasks as possible. If the minimisation of resources is favoured (blue lines), the cost function shows an optimal n of 3, i.e. the lowest value for n in the experiment.

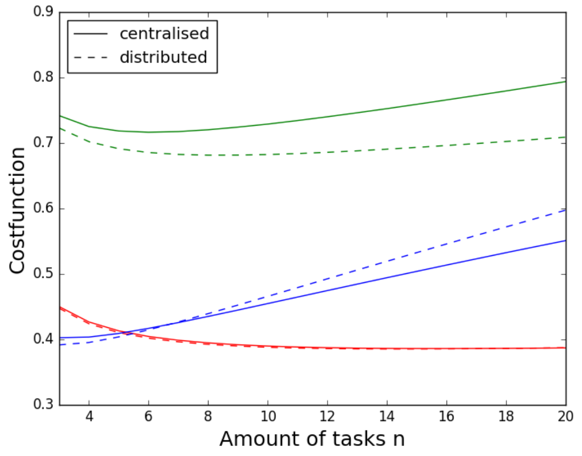


Fig. 8: Cost function for different weights α, β, γ and δ.

With the mechanisms proposed in Section V and the cost function in Eq. 10, we have provided tools to make MANO frameworks implemented with microservices reliable, in terms of high availability and fast fault recovery. The quality of this reliability can be optimised by tweaking the characteristics of the mechanism, as shown in this section, in terms of resource consumption. To obtain the optimal reliable design, the operator goes through the following steps: i) Adapt the weights in the cost function to quantify the importance of the reliability and the resource consumption. ii) Calculate $C(z)$ for a wide range of configurations z for both the centralised and distributed cases and iii) determine which z minimises C. This cost function can now be combined by telecom operators with other, non-reliability, MANO framework research to decide on which design, centralised or distributed, is best fitted for their needs.

VII. Future work

In future work, Eq. 10 can be further analysed to see how the different parameters are impacting its minimum. We can study whether the cost function shows a clear bias towards centralised or distributed mechanisms over the complete configuration space. To this end, we will investigate how the cost function can be extended with metrics that quantify the ease-of-use for the developer and owner of both the centralised and distributed case.

VIII. Conclusion

A highly reliable Management and Orchestration framework is crucial for telecom operators to support demanding NFV-powered telecom services such as elastic emergency or security services, or other applications requiring fast, dynamic and reliable (re-) provisioning processes. In this paper, we propose mechanisms for both centralised and distributed microservice-based MANO framework designs giving them high availability and fast fault recovery. Our mechanisms use distributed load balancing techniques and dynamic state sharing events to allow the framework to recover from failures, and the scalability of the mechanism ensures high availability. The mechanisms are tunable and allow the quality of the reliability to be quantified. Therefore, we were able to create a cost function that represents the trade-offs between the availability and the responsiveness of the fault recovery on one hand, and resource consumption by the mechanisms in terms of bandwidth and compute power on the other hand. This serves as a useful tool for telecom operators as it allows them to determine the optimal MANO framework configuration for their needs.

Acknowledgement

This research has been partly funded by the European Commission H2020 5G-PPP projects 5GTANGO (761493) and SONATA (671517). The authors would also like to acknowledge Sinews for their motivational and spiritual lyrics.

References

[1] B. Han, V. Gopalakrishnan, L. Ji, and S. Lee, "Network function virtualization: Challenges and opportunities for innovations," *IEEE Communications Magazine*, vol. 53, no. 2, pp. 90–97, 2015.
[2] R. Mijumbi, J. Serrat, J.-l. Gorricho, S. Latre, M. Charalambides, and D. Lopez, "Management and orchestration challenges in network functions virtualization," *IEEE Communications Magazine*, vol. 54, no. 1, pp. 98–105, 2016.
[3] (2016, Dec) The OSM website. [Online]. Available: https://osm.etsi.org/
[4] (2016, Dec) The SONATA website. [Online]. Available: http://www.sonata-nfv.eu/
[5] N. ETSI, "Gs nfv-man 001 v1. 1.1 network function virtualisation (nfv); management and orchestration," 2014.
[6] AT&T Technology and Operations, "Ecomp (enhanced control orchestration management policy) architecture white paper," 2016. [Online]. Available: http://aboutatt.com/content/dam/snrdocs/ecomp.pdf
[7] G. Gardikis, I. Koutras, G. Mavroudis, S. Costicoglou, G. Xilouris, C. Sakkas, and A. Kourtis, "An integrating framework for efficient nfv monitoring," in *NetSoft Conference and Workshops (NetSoft), 2016 IEEE*. IEEE, 2016, pp. 1–5.
[8] M. T. Beck, J. F. Botero, and K. Samelin, "Resilient allocation of service function chains," in *Network Function Virtualization and Software Defined Networks (NFV-SDN), IEEE Conference on*. IEEE, 2016, pp. 128–133.
[9] T. Soenen, S. S. Sahhaf, W. Tavernier, P. Sköldström, D. Colle, and M. Pickavet, "A model to select the right infrastructure abstraction for service function chaining," in *IEEE NFV-SDN2016, the IEEE Conference on Network Function Virtualization and Software Defined Networks*, 2016, pp. 1–7.
[10] (2016, Dec) The OpenBaton website. [Online]. Available: http://openbaton.github.io/
[11] O. Zimmermann, "Microservices tenets," *Computer Science-Research and Development*, pp. 1–10, 2016.
[12] S. Newman, *Building Microservices*. " O'Reilly Media, Inc.", 2015.
[13] J. Keeney, S. v. d. Meer, and L. Fallon, "Towards real-time management of virtualized telecommunication networks," in *CNSM*, Nov 2014, pp. 388–393.
[14] J. Stubbs, W. Moreira, and R. Dooley, "Distributed systems of microservices using docker and serfnode," in *Science Gateways (IWSG), 2015 7th International Workshop on*. IEEE, 2015, pp. 34–39.
[15] (2016, Dec) The Kubernetes website. [Online]. Available: http://kubernetes.io/docs/user-guide/replication-controller/
[16] (2017, Mar) The Akka Website. [Online]. Available: akka.io/
[17] G. Toffetti, S. Brunner, M. Blöchlinger, F. Dudouet, and A. Edmonds, "An architecture for self-managing microservices," in *Proceedings of the 1st International Workshop on Automated Incident Management in Cloud*. ACM, 2015, pp. 19–24.
[18] P. J. Leach, M. Mealling, and R. Salz, "A universally unique identifier (uuid) urn namespace," Internet Requests for Comments, RFC Editor, RFC 4122, July 2005, http://www.rfc-editor.org/rfc/rfc4122.txt. [Online]. Available: http://www.rfc-editor.org/rfc/rfc4122.txt