Monitoring Service Level Workload of Highly Available Applications

Mehran N. A. H. Khan

A Thesis

in

The Department

of

Electrical and Computer Engineering

Presented in Partial Fulfillment of the Requirements

for the Degree of Master of Science (Electrical & Computer Engineering) at

Concordia University

Montreal, Quebec, Canada

November 2015

CONCORDIA UNIVERSITY
SCHOOL OF GRADUATE STUDIES

This is to certify that the thesis prepared

By: Mehran Noor Al Haq Khan

Entitled: "Monitoring Service Level Workload of Highly Available Application"

and submitted in partial fulfillment of the requirements for the degree of

Master of Applied Science

Complies with the regulations of this University and meets the accepted standards with respect to originality and quality.

Signed by the final examining committee:

______________________ Dr. M. Mehmet-Ali	Chair
______________________ Dr. R. Glitho (CIISE)	Examiner, External To the Program
______________________ Dr. S. Abdi	Examiner
______________________ Dr. M. Toeroe	Examiner
______________________ Dr. Y. Liu	Supervisor
______________________ Dr. F. Khendek	Supervisor

Approved by: ______________________
Dr. W. E. Lynch, Chair
Department of Electrical and Computer Engineering

____________20_____

____
Dr. Amir Asif, Dean
Faculty of Engineering and Computer Science

ABSTRACT

Monitoring Service Level Workload of Highly Available Applications

Mehran N. A. H. Khan

Elasticity is a key feature of cloud computation and is a major contributor to its popularity. Elasticity is defined as automatic provisioning/de-provisioning of resources to match workload changes over time. Service High Availability (HA) is among one of cloud computing's big challenges. High Availability (HA) is defined as providing a minimum of 99.999% service availability. Maintaining service HA while scaling in/out is even more challenging. Recently, an architecture has been proposed for managing HA. Following the proposed architecture, an Elasticity Engine has been introduced that is capable of managing resources based on application level provisioning or de-provisioning alerts while preserving HA. In contrast to the prevailing monitoring solutions where Virtual Machine (VM) level workload is provided, the Elasticity Engine requires a monitoring solution that monitors service-level workload and triggers alerts accordingly. In this thesis, we propose an approach and an architecture for the monitoring of HA applications at the service level. Accordingly, the monitoring approach starts with monitoring the application components in traditional manner. Workload of the components are mapped to each component's respective service assignment. The resource usages of all the components providing services is aggregated and mapped to the service level workload using a distributed client-server architecture. This approach allows for distinguishing between the different HA states, active or standby that a component can be assigned at runtime and it (the approach) adapts to the situations where switchovers happen under the control of the SA Forum middleware due to failures for example. The proposed monitoring architecture has been implemented and integrated with the Elasticity Engine to test its effectiveness and overhead. It has been shown that the implemented and integrated prototypes achieve elasticity in a cluster based on service level workload while keeping the monitoring overhead within 5% of its total resource.

Acknowledgements

I would like to thank my supervisors, Prof. Ferhat Khendek and Prof. Yan Liu for their patience and belief in me and for giving me the opportunity to pursue my thesis under their supervision. The thesis would not have been possible without their support and encouragement.

I am immensely grateful to Dr. Maria Toeroe (Ericsson Canada) for her support, knowledge, expertise and guidance that helped me overcome the challenges in completing this thesis. I would also like to thank her for her patience and support that helped me get through some trying times.

I am grateful to my friends Arshi Islam and Salman Moazzem for their support and friendship.

I would also like to offer my gratitude to all my colleagues in the MAGIC team for their friendship and for creating a pleasant work atmosphere.

I am grateful to Concordia University, Ericsson Canada, NSERC and the PERSWADE Program for offering their facilities and resources.

This work has been partially supported by Natural Sciences and Engineering Research Council of Canada (NSERC), Ericsson Software Research and Concordia University as part of the Industrial Research Chair in Model Based Software Management.

Finally, I would like to pay earnest gratitude to my family for their love, support, and encouragement. I owe all my accomplishments to them. I dedicate this thesis to them.

Table of Contents

List of figures

List of tables

List of acronyms

AIS	Application Interface Specification
AMF	Availability Management Framework
API	Application Programming Interface
CCB	Configuration Change Bundle
CSI	Component Service Instance
CLM	Cluster Membership Service
DN	Distinguished Name
HA	High Availability
HPI	Hardware Platform Interface
HTTP	Hypertext Transfer Protocol
IM	Information Model
IMM	Information Model Management
LDAP	Lightweight Directory Access Protocol
LTTng	Linux Tracing Toolkit next generation
OI	Object Implementer
OM	Object Modifier
PLM	Platform Management Service
PSUtil	Python System and process Utilities
RDN	Relative Distinguished Name
RM	Redundancy Model
SA Forum	Service Availability Forum
SG	Service Group
SI	Service Instance
SU	Service Unit

TCP	Transmission Control Protocol
UML	Unified Modeling Language
UST	User Space Tracing
VM	Virtual Machine

1. Introduction

This chapter introduces the research domain, the motivations and the contributions of this work.

1.1. Research domain

With our increasing dependency on computer-based systems, the need to ensure that services are always provided to the end-users has become more important than ever. Service Availability is an important characteristic of service excellence in a number of domains such as telecommunication, cloud computing, etc.

Service Availability for a service is defined as the percentage of time a service is provided [1]. High Availability (HA) is defined as providing a minimum of 99.999% service availability, which translates to at most 5.26 minutes of downtime in a year [1].

Cloud computing is a popular paradigm that refers to priced, on-demand delivery of services of applications and other remote resources over a network [2]. Service Availability is one of the big challenges of cloud computing [3]. Elasticity, on the other hand, is a key feature of the cloud that is contributing to its popularity. Elasticity is defined as automatic provisioning/de-provisioning of resources to match workload changes over time [4].

In a typical cloud system elasticity is managed based on the system resource usage of the virtual machines (VM) running the application. I.e. the resource usage of the VM is equated to the resource usage of the application hosted in the VM [5] [6]. Therefore, the smallest resource provisioned to or de-provisioned from any service in this context is a VM. These VMs are assumed to be stateless, allowing each VM to participate in the service from the point they are added to a cloud cluster without any kind of state propagation/synchronization. This design includes a number of assumptions that are not necessarily true for telecom applications that provide state-full HA services and run according to some redundancy model. In such HA systems, the resources providing the application services are defined at a finer detail according to a configuration where each service provider has an active or a standby role. The state of each active service provider is synchronized with its associated standby service provider(s) so that it (they) can assume the active role at any time it becomes necessary. A middleware is responsible for managing the life cycle of these application resources according to the configuration as well as assigning the active and standby roles, in particular, assigning the active role to a standby provider whenever the active provider fails. In such a dynamic system a simple association of a workload with a set of VMs may not be effective since different service providers may be collocated in the same VM, some VMs may only be partially or not at all associated with a given application service.

1.2. Thesis motivations

The Service Availability (SA) Forum [7] middleware is capable of managing HA services [1] in the cloud. The SA Forum [8] middleware's Availability Management Framework (AMF) [9] manages the availability of application services based on the application configuration. Such a configuration can be divided into two conceptual parts: the service provider and the service parts. The service provider part represents the resources and it is made up of sets of interrelated components. The service is described in terms of Service Instances (SIs) that are made up of one or more Component Service Instances (CSIs). The SIs represent the services provided by the application managed by AMF. At runtime AMF assigns the CSIs of the SI to the service provider entities – the application components [9].

An Elasticity Engine [8] [10] has been proposed recently for AMF managed applications. The Elasticity Engine manages the resource usage of AMF applications by changing their AMF configurations, which in turn triggers AMF to redistribute the CSI/SI assignments. To take any such action, the Elasticity Engine needs an input, for example, from a Monitoring Engine that can measure the system load imposed by the SI.

Most existing monitoring solutions are either:

a) Capable of providing workload in terms of resource like CPU, RAM, memory, etc. usage at the platform level per VM or

b) Too platform specific to apply to AMF managed applications.

In this thesis, we are interested in measuring the workload imposed by HA services (SIs) and address three main related challenges:

a) Retrieving the distribution of CSI assignments in the system at runtime

b) Retrieving the system usage that is relatable to the CSI assignments to the nodes and

c) Aggregating the system usage according to distribution of CSI assignments to express the usage in terms of AMF services

In this thesis, we propose an approach and a monitoring architecture to relate platform level workload to workload in terms of SIs so that the existing Elasticity Engine [10] can react and adjust the configuration.

1.3. Thesis contributions

In this thesis, we address the problems mentioned in the previous subsection by introducing a monitoring approach/architecture. The main contributions of this thesis are summarized as follows:

- o A method to instrument AMF components automatically to detect and map CSI assignments to components.
  - ✓ The components in a cluster interact with AMF using an interface based on API defined by the SA Forum. For each new service assignment, service assignment change or service assignment removal, AMF dispatches call-backs to the components using this interface. By instrumenting the AMF-component-interface AMF call-backs to the component can be detected at runtime. It allows us to map the services to their corresponding components.
- o A method to aggregate workload of components into workload of the SIs running in the system.
  - ✓ The mapping of CSIs to the components and the system usage per processes running the components in each of the Monitoring Client are transmitted to the Monitoring Server for

aggregation. The Monitoring Server then creates a tree-like data structure for the component-CSI assignments, keeping the system usage as the leaves of the tree. The workload per SI is then calculated by bottom-up aggregation of system usage metrics along the tree-paths, which provides us with workload in terms of SIs.

- A monitoring architecture to measure workload in terms of potentially collocated SA Forum middleware services.
  - ✓ The architecture follows a client-server architecture.
  - ✓ In a cluster, each node hosting application components is considered as a Monitoring Client, one of the nodes in the cluster is designated the role of Monitoring Server. Hence, one of the nodes in the cluster plays a dual role of both Monitoring Client and Monitoring Server. The Monitoring Server must be reachable from all Monitoring Clients over the network.
- A prototype to perform experiments and evaluate its performance.
  - ✓ The prototype is capable of adapting to change of workload distribution. It can measure and detect significant workload change of any SI and alert the Elasticity Engine.
  - ✓ Two test beds have been used to test the monitoring architecture effectiveness. Each test bed has been used for a unique test case.
  - ✓ The Monitoring overhead has been measured for each of the aforementioned test cases.

1.4. Thesis organization

The thesis is organized in seven chapters. In Chapter 2, the background knowledge related to availability, SA Forum middleware, monitoring tools and related work are discussed. In Chapter 3, an overview of the monitoring approach, its architecture and the steps taken to measure service level workload are discussed. In Chapter 4, the procedure to instrument AMF components and its automation are discussed with an example. In Chapter 5, the algorithms to map and aggregate component-workload to SI-workload are discussed. In Chapter 6, the Monitoring prototype details, its effectiveness in the test cases and its overhead evaluation are discussed. Finally, in Chapter 7, we summarize our contributions and discuss future work.

2. Background on Availability, Cloud, SA Forum Middleware and Related Work

In this chapter, the general definition for service availability is presented, which is followed by discussions on the relevant sections of SA Forum middleware related to this work. We also review some related work in the cloud and service availability.

2.1. Service Availability

Service Availability for a service is defined as the percentage of time a service is provided [1]. The two factors that determine the availability of a service are: Mean Time to Repair (MTTR) and Mean Time Between Failures (MTBF) [1].

- MTBF is the statistical mean time between two consecutive failures of a system, and
- MTTR is the statistical mean time to repair the system.

Keeping these two factors in mind, service availability of a system can be defined using Eq. 2-1 [1]:

$$\text{Availability} = \frac{MTBF}{MTBF+MTTR} \tag{2-1}$$

The service availability of a system relies on the availability of individual components of that system.

2.2. Service Availability Forum (SA Forum)

The objective of the SA Forum is to define standard interfaces that facilitate the development of carrier-grade and mission critical applications and systems [7]. It is a consortium [7] of companies from the telecommunication and computing industries working together to develop and publish high availability middleware service specifications. OpenSAF middleware [11] is an open source implementation of the SA Forum specifications [12].

The SA Forum services are categorized into two main specifications: the Hardware Platform Interface (HPI) and the Application Interface Specification (AIS) [12]. Typically, the services specified by the HPI are implemented as libraries in the hardware platform. These services serve up the hardware information in a standard way so that a user application does not have to be aware of the specifics of the underlying hardware [13]. The main objective of AIS is to provide standardized APIs for middleware functions typically required by HA applications. These specifications consist of different middleware services among which we will be focusing on the AMF [9] service, which is the most relevant to this thesis. Figure 2-1 gives an overview of the SA Forum services.

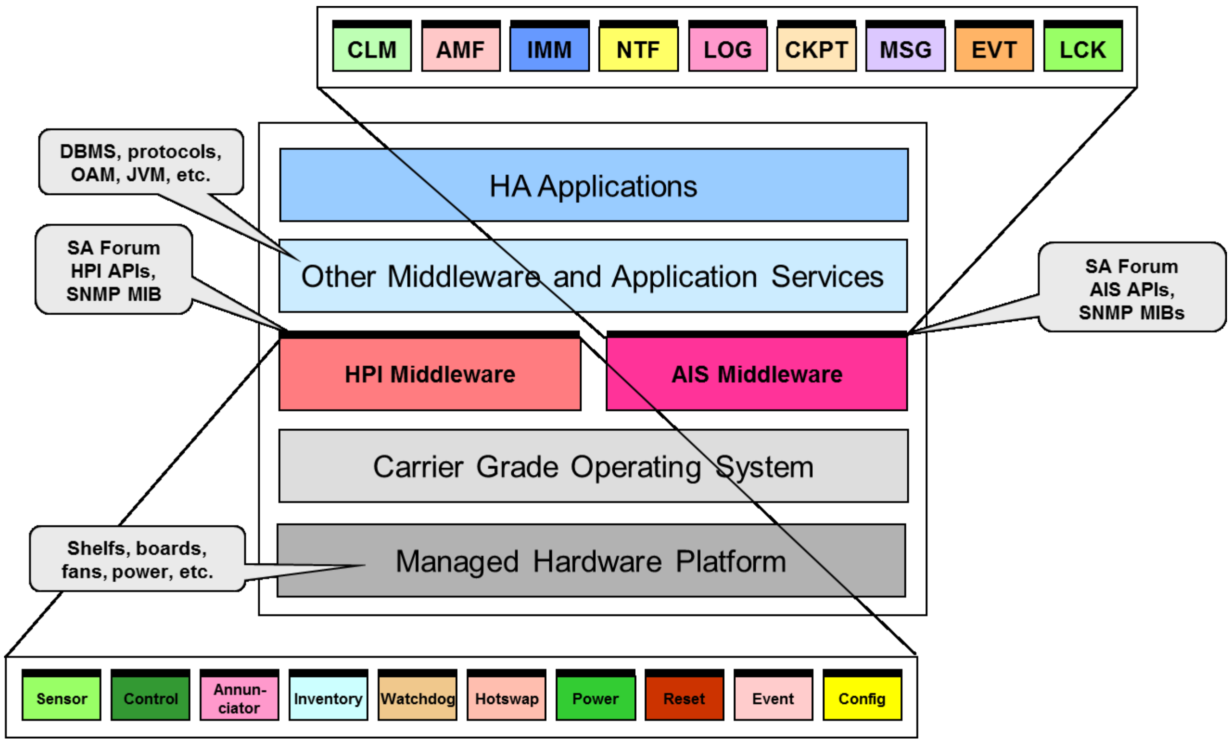

CLM
AMF
IMM
NTF
LOG
CKPT
MSG
EVT
LCK
DBMS, protocols, OAM, JVM, etc.
HA Applications
SA Forum HPI APIs, SNMP MIB
Other Middleware and Application Services
SA Forum AIS APIs, SNMP MIBs
HPI Middleware
AIS Middleware
Carrier Grade Operating System
Shelfs, boards, fans, power, etc.
Managed Hardware Platform
Sensor
Control
Annun-ciator
Inventory
Watchdog
Hotswap
Power
Reset
Event
Config

Figure 2-1: An SA Forum middleware architecture [14]

2.3. Availability Management Framework

AMF is the SA Forum middleware service in charge of managing the service availability of an HA application. AMF is responsible for 1) assigning the workload to the application components, 2) managing the life-cycle of the resources under its control (e.g. software components), 3) reassigning the workload of a faulty component to a standby (and healthy) component, and 4) repairing the faulty component [9].

2.3.1. Logical entities

The AMF uses an abstract system model to represent the resources under its control. This abstract model consists of various logical entities [9]. Fig. 2-2 shows the logical entities of AMF and the relationships among them.

2.3.1.1. Component

A component is the smallest logical entity in the system on which AMF performs error detection, isolation and repair. It represents a specific resource such as a process, which is capable of providing a set of functionalities [9].

2.3.1.2. Component Categories

Components are categorized based on their capability in terms of service availability - awareness. Components of different component categories behave differently based on their different properties [9]. In the context of this work, the component categories based on their HA Awareness and Life Cycle Management have been considered. Based on HA Awareness, components can be categorized into the following:

- **SA Aware Component:** Components that are under the direct control of the AMF can have a high level of integration with this framework, which enables fast workload assignment, error detection, isolation and repair [9]. Such components are called SA-aware components.

  - **Proxy Components:** The proxy component is an SA-aware component that is responsible for conveying requests made by the AMF to its proxied components.
  - **Proxied Components:** The AMF determines the proxied components for which a proxy component is responsible when the proxy component registers with the framework, based on configuration and other factors like availability of components in the cluster.

- **Non-SA-Aware Component:** Components that do not register directly with the AMF are called non-SA-aware components [9].
  - **Non-Proxied, Non-SA-Aware Component:** For non-proxied, non-SA-aware local components, the role of the AMF is limited to the management of the component life cycle. The AMF instantiates a non-proxied, non-SA-aware component when the component needs to provide a service and terminates this component when the component must stop providing the service [9].

Based on Life Cycle Management, components can be categorized into the following:

- **Pre-instantiable Component:** Components that can remain in an idle state without being assigned any service after being instantiated by the AMF. All SA-aware components are pre-instantiable components [9].
- **Non-pre-instantiable Component:** Components that start providing service as soon as they are instantiated, are called non-pre-instantiable components [9].

2.3.1.3. Service Unit (SU)

A service unit (SU) is a logical assembly of several components that, when given an active assignment to provide a service, combine their individual functionalities to provide that particular service [9].

2.3.1.4. Service Unit Type

A service unit (SU) type defines the common characteristics that the SUs of a given type share. It specifies a list of component types that can be aggregated in the SU type. It also determines the number of components of each component type that an SU of a type can accommodate [9].

2.3.1.5. Component Service Instance (CSI)

A component service instance (CSI) represents the workload that AMF can dynamically assign to a component. High availability (HA) states are assigned to a component on behalf of the CSI currently assigned to it [9]. The AMF chooses the HA state of a component for each particular CSI [9].

- **HA State of a Component per CSI:** AMF assigns an HA state to each component on behalf of its assigned CSI. The HA state of a component for a particular CSI can be one of the following: active, standby, quiescing and quiesced [9]. For a given CSI, the HA states of a component are described below:
  - **Active:** A component at this state is responsible for providing the service characterized by this CSI assignment [9].
  - **Standby:** A component at this state acts as a standby for the service characterized by this CSI assignment [9].
  - **Quiescing:** The component that had previously been in an active HA state for this CSI is in the process of quiescing its activity [9]. At this state, a component continues

providing the service it had been providing but rejects any new request from the service characterized by the CSI for which it is in quiescing state [1].

- **Quisced:** The component that had previously the active or quiescing HA state for this CSI has now quiesced its activity related to this CSI, and the AMF can safely assign the active HA state for this CSI to another component [9].

2.3.1.6. Component Service Type (CST)

The component service type (CST) is the generalization of similar CSIs that are seen by the AMF as equivalent and handled in the same manner. The configuration of a component indicates the CST it supports [9].

2.3.1.7. Service Instance (SI)

The AMF supports assembling multiple CSIs into a logical entity called Service Instance (SI) the same way it supports assembling multiple components into one SU. An SI aggregates all CSIs to be assigned to the individual components of an SU in order for them to provide a particular service [9]. It is possible for the same SI to be assigned to multiple SUs.

2.3.1.8. Service Type

The Service Type defines a list of CSTs. An SI is composed of the CSIs that are of the CSTs defined in the service type of that SI. For each CST the service type also defines the number of CSIs that an SI of the given type may aggregate [9].

2.3.1.9. Service Group (SG)

A Service Group (SG) is a logical entity that groups one or more SUs in order to provide service availability for a particular set of SIs. Any SU of the SG must be able to take an assignment for any SI of this set [9]. Each SG has a redundancy model that defines how the SUs in the SG should protect its SIs [9].

2.3.1.10. Service Group Type

The service group type is a generalization of similar service groups that follow the same redundancy model, provide similar availability, and are composed of units of the same service unit types [9].

- **Service Group Redundancy Model:** Each SG has a redundancy model associated to it by configuration. The redundancy model(s) supported by an SG is specified in its SG type. The SUs in an SG provide service availability to the SIs according to the redundancy model supported by the SG. The redundancy models and their characteristics are described in the subsection 2.3.2.

2.3.1.11. Application

An application is a logical entity that contains one or more SGs and SIs protected by those SGs [9].

2.3.1.12. Application Type

An application type defines a list of SG types that an application of its type can be composed of. All applications of the same type share attribute values defined by their application type [9].

2.3.1.13. AMF node

AMF node is the VM or node where SUs are deployed.

- **Mapping of SUs/SGs to Nodes:** SGs and SUs have an optional node group configuration attribute in their configuration. A node group contains a list of nodes. Using this attribute, it can be specified for an SU to be instantiated on a specific node (if a node is specified in the attribute) or on one of the nodes of the specified node group (if a node group is specified) [9].

2.3.1.14. AMF cluster

A number of AMF nodes are grouped together to form an AMF cluster [9].

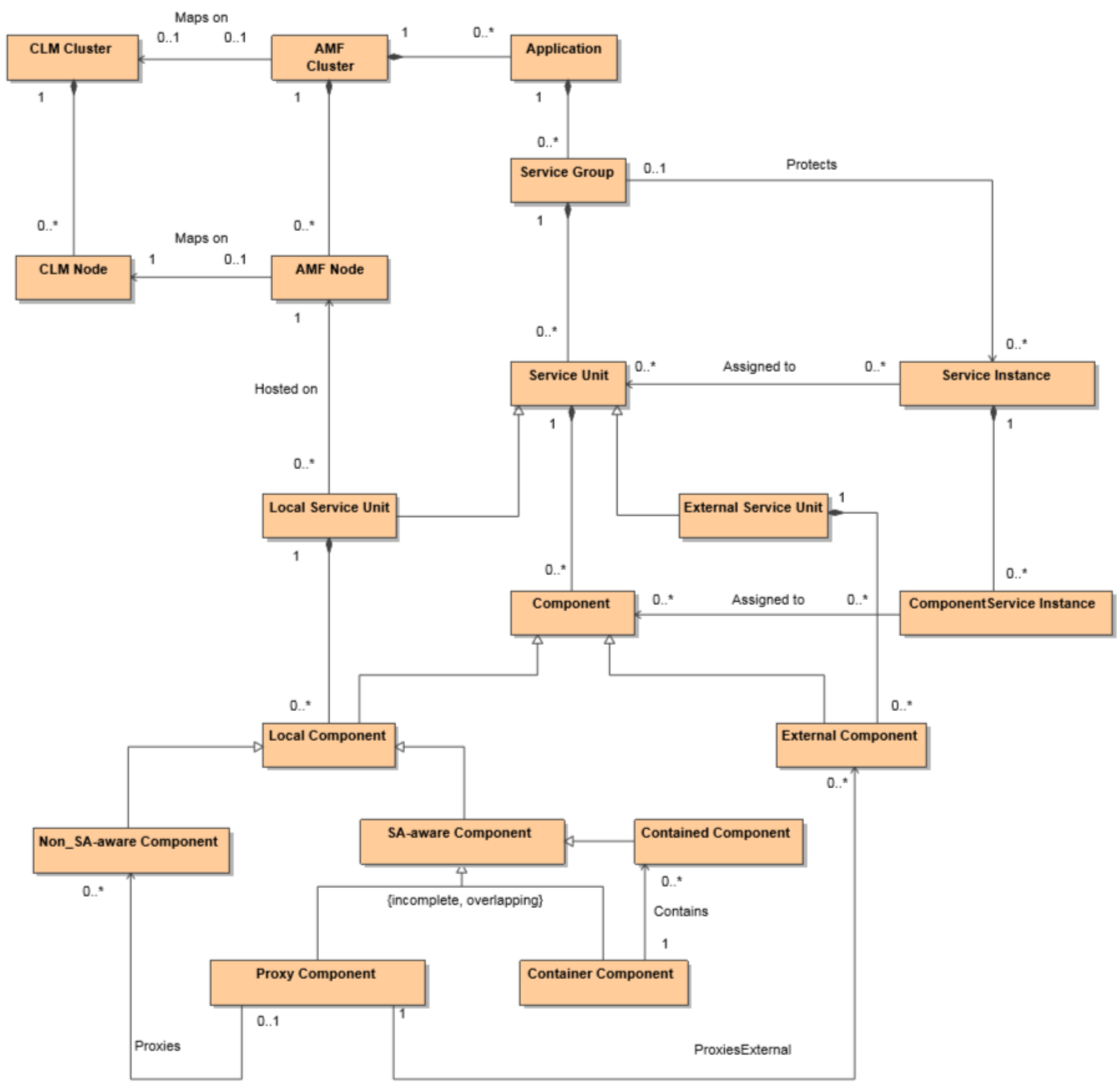


Figure 2-2: AMF logical entities and their relationships [9]

2.3.2. Redundancy Models

There are five different redundancy models: 2N, N+M, NWay, NWay-Active, and No-Redundancy redundancy model [1]. Based on the redundancy model's characteristics, an SI may have a number of active and standby assignments. The distribution of active and standby assignments is also determined by the redundancy model of the SG [9] [1]. An SU may have active or standby HA state for an SI. If an SU is in active HA state for an SI, it means that the SU is providing the service. Similarly, an SU being in standby HA state for an SI means that the SU is synchronizing with the active SU and stays ready to take over whenever the active SU becomes unable to provide the service. If an SU has neither active nor standby assignment, that SU is called a spare SU [9]. The redundancy models are described in the following sub-sections:

2.3.2.1. 2N Redundancy Model

This is the most intuitive redundancy model. In an SG with 2N redundancy model, for all SIs, at most one SU can have an active assignment and at most one SU can have a standby assignment. The SU with the active assignment is called the active SU and the SU with the standby assignment is called the standby SU. In this redundancy model, the SG can have at most one active SU and one standby SU for a given SI [9]. Fig. 2-3 shows an example of an SG with 2N redundancy model and two SIs assignments to three SUs. In this example, there are two SIs and each of them is composed of two CSIs. Provided that the SG protecting the SIs has at least two operational SUs, for any SI, it is only possible to have one active and one standby SU at the same time with this redundancy model. Hence SU3 has no SI assignment to it in the example.

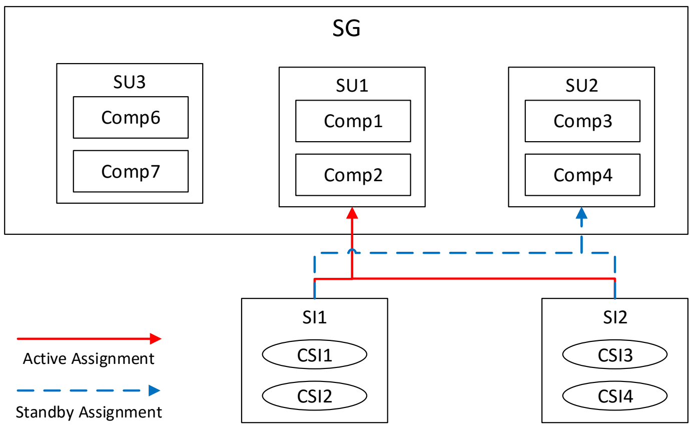


Figure 2-3: SG with 2N redundancy model

2.3.2.2. N+M Redundancy Model

In this redundancy model, N SUs can be assigned as active and M SUs can be assigned as standbys for the SIs being protected by the SG. Each SU of the SG can only have one of the following HA states: active or standby. An SI can have only one active and one standby assignment [9]. An SG comprising of 4 SUs with N+M redundancy model is illustrated in Fig. 2-4 (2+2). Two of the SUs are assigned active assignment and the other two are assigned standby assignment for the SIs being protected by them. As shown, with this redundancy model, an SU cannot have active and standby assignments at the same time. Unlike 2N redundancy model, the SG can have multiple SUs with active or standby assignments.

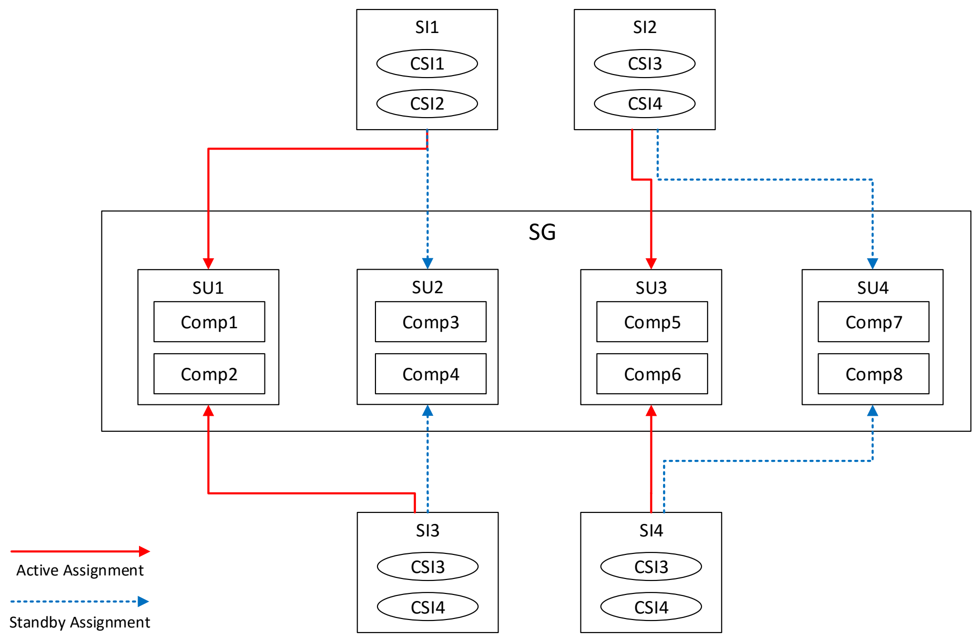


Figure 2-4: SG with N+M redundancy model

2.3.2.3. NWay Redundancy Model

In this redundancy model, for an SG protecting a given set of SIs, SUs can simultaneously have active assignments for some SIs and have standby assignment for other SIs. For each SI, at most one of the SUs in the SG can have active assignment and at most all of the other SUs can have standby assignments. No SU can take active and standby assignments simultaneously for the same SI. An SG with NWay redundancy model is illustrated in Fig. 2-5. In this example, three SUs in the SG get assignments for two SIs protected by the SG. Each SI has one active and two standby assignments. SU1 and SU2 gets one active assignment for SI1 and SI2 respectively and standby assignments for the other two SIs. SU3 gets only standby assignments.

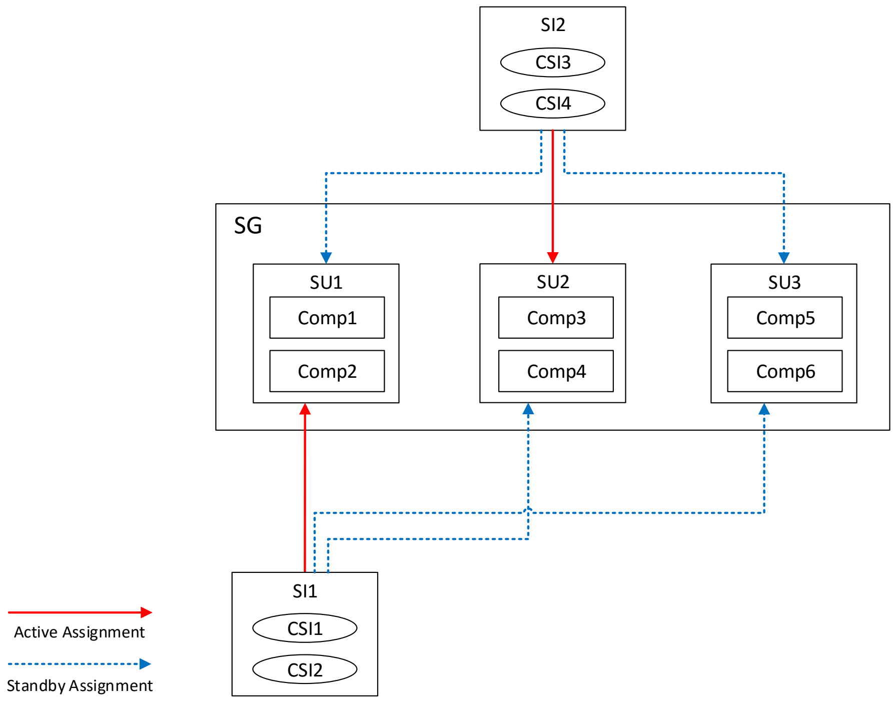


Figure 2-5: SG with NWay redundancy model

2.3.2.4. NWay-Active Redundancy Model

Unlike the redundancy models discussed earlier, NWay-Active redundancy model does not allow for standby assignments for an SI. This redundancy model allows for an SI to be assigned as active to multiple SUs, meaning each SI can have one or many active assignments [1]. An SG with NWay-Active redundancy model is illustrated in Fig. 2-6. The SG protecting three SIs in the illustration is comprised of three SUs. As illustrated, each SI has two active assignments to two different SUs.

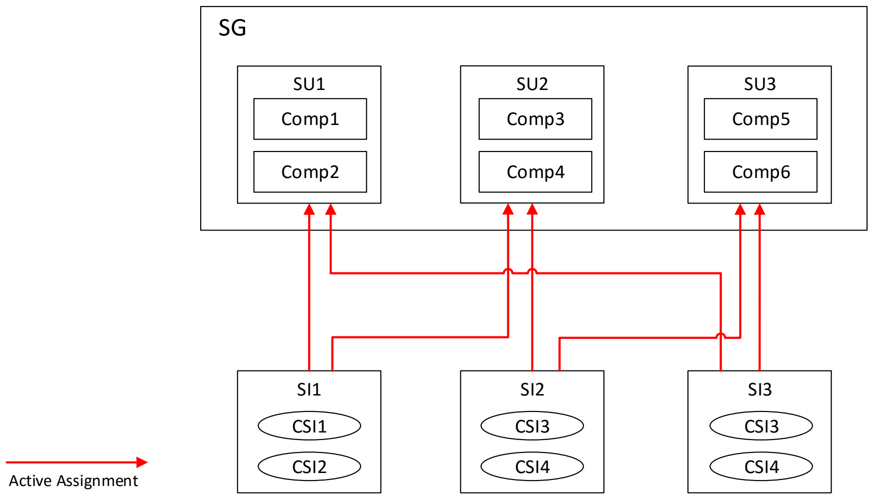


Figure 2-6: SG with NWay-Active redundancy model

2.3.2.5. No-Redundancy Redundancy Model

In this redundancy model, each SU can have at most one active assignment for at most one SI and no two SUs get active assignment for the same SI. No SU is assigned as a standby in this redundancy model [1]. An SG with No-Redundancy Redundancy model comprising three SUs is illustrated in Fig. 2-7. Each SU except SU4 in the illustration gets one active assignment. SU4 is a spare SU as it does not have any assignment.

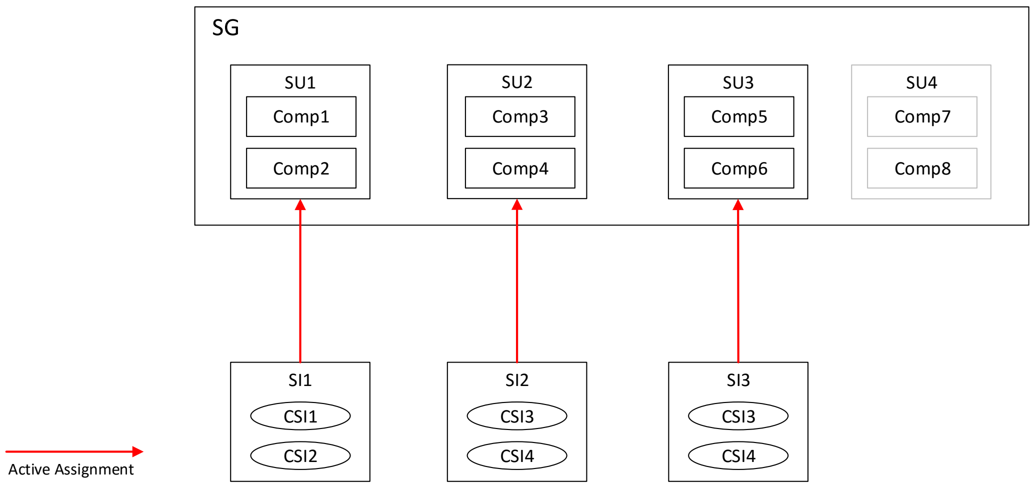


Figure 2-7: SG with No-Redundancy redundancy model

2.3.3. AMF configuration

The AMF manages applications deployed in the cluster according to a configuration, An AMF configuration consists of the description of logical entities such as components, SUs, SGs, SIs, CSIs with their respective types, nodes and the relations among them. The logical entities are described in the AMF configuration by objects and their attributes of the classes defined in the AMF Specifications [9]. Such attributes can consist of configuration that can be either read-only or writable [9]. AMF reacts to any change in the configuration attributes first by evaluating the system state and then implementing the new changes while maintaining service availability.

2.4. Information Model Management (IMM)

The different entities of an SA Forum cluster such as AMF managed components, checkpoints provided by the Checkpoint Service, or message queues provided by the Message Service are represented by various objects of the SA Forum Information Model (IM) [15]. The SA Forum Information Model is specified in UML and managed by the Information Model Management (IMM) Service [15]. The IM can be considered as a cluster wide database for SA Forum compliant systems. The IMM service manages all objects of the SA Forum IM and provides APIs that allows its users to

- Configure SA Forum entities
- Obtain information about objects and runtime status of the system and
- Perform administrative operations [15].

The SA Forum IM also specifies the attributes and the kind of administrative operations that can be performed on the objects managed by it. Fig. 2-8 shows the interfaces provided by the IMM Service.

The users of IMM API are referred to as Object Managers (OM). An OM has the privilege to create, access, manipulate and manage the configuration objects. The IMM notifies the configuration changes made by the OM to the applications that are responsible for implementing the objects to which changes have been made. The applications responsible for implementing these objects are referred to as Object Implementers (OI) [1] [15].

IMM objects and attributes can be classified into two categories:

1. **Configuration objects and configuration attributes:** Configuration objects and attributes carry configuration information. The system administrators manage the cluster by manipulating configuration objects and their attributes [15].
2. **Runtime objects and runtime attributes:** The OIs reflect the current state of their implemented entities via runtime objects and attributes [15].

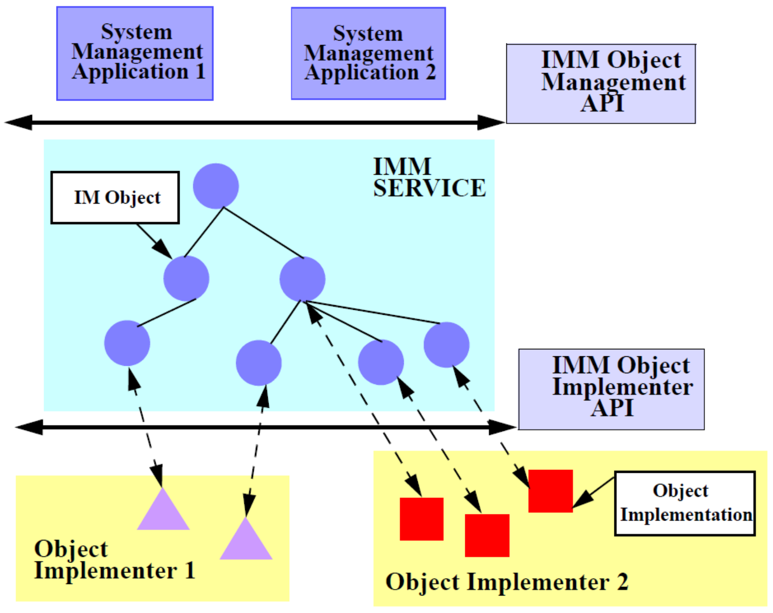

System Management Application 1
System Management Application 2
IMM Object Management API
IMM SERVICE
IM Object
IMM Object Implementer API
Object Implementation
Object Implementer 1
Object Implementer 2

Figure 2-8: IMM service interfaces [15]

2.4.1. Information Model organization

AMF configuration is accessed through the IMM Service [15]. The configuration information in IM is represented as a tree where the object naming scheme is similar to the Lightweight Directory Access Protocol (LDAP) [16]. Each object in a tree is therefore named after the path from its position to the root of the tree. Each object has a unique Distinguished Name (DN). An object also has a Relative Distinguished Name (RDN), which is essentially part of its DN.

Each object in the IM has its RDN value as an attribute. For example, in Fig. 2-9, the RDN value for AmfSU_1 is 'safSu=AmfSU_1'. The DN of an object is the DN of the object's parent in the IM tree hierarchy prefixed by the RDN of the object. For example, the RDN of AmfSU1 in Fig. 2-9 is 'safSu=AmfSU_1,safSg=AmfSG_1, safApp=AmfApp_1'. In the IM, the tree is constructed from the objects' DN. Alongside the containment relationship that the child objects have with their parents, entities also exhibit other relations. For example, AMF assigns CSIs to components at runtime. In IM, this association relationship between a component and a CSI is mapped by selecting the DN of the object representing the parent (component) and the DN of the related object (CSI) as the DN of the association object itself. For example, in Fig. 2-9, the CSI with RDN 'safCsi=AmfCSI_1' is assigned to the component with RDN 'safComp=AmfComp_1'. The IM runtime object class 'SaAmfCSIAssignment' represents the association relation [15]. The runtime object of the class 'SaAmfCSIAssignment' between the aforementioned component and CSI has the RDN 'safCSIComp=AmfComp_1, safSu=AmfSU_1, safSg=AmfSG_1, safApp=AmfApp_1' and the DN 'safCSIComp=AmfComp_1, safSu=AmfSU_1, safSg=AmfSG_1, safApp=AmfApp_1,safCsi=AmfCSI_1, safSi=AmfSI_1, safApp=AmfApp_1' in the IM.

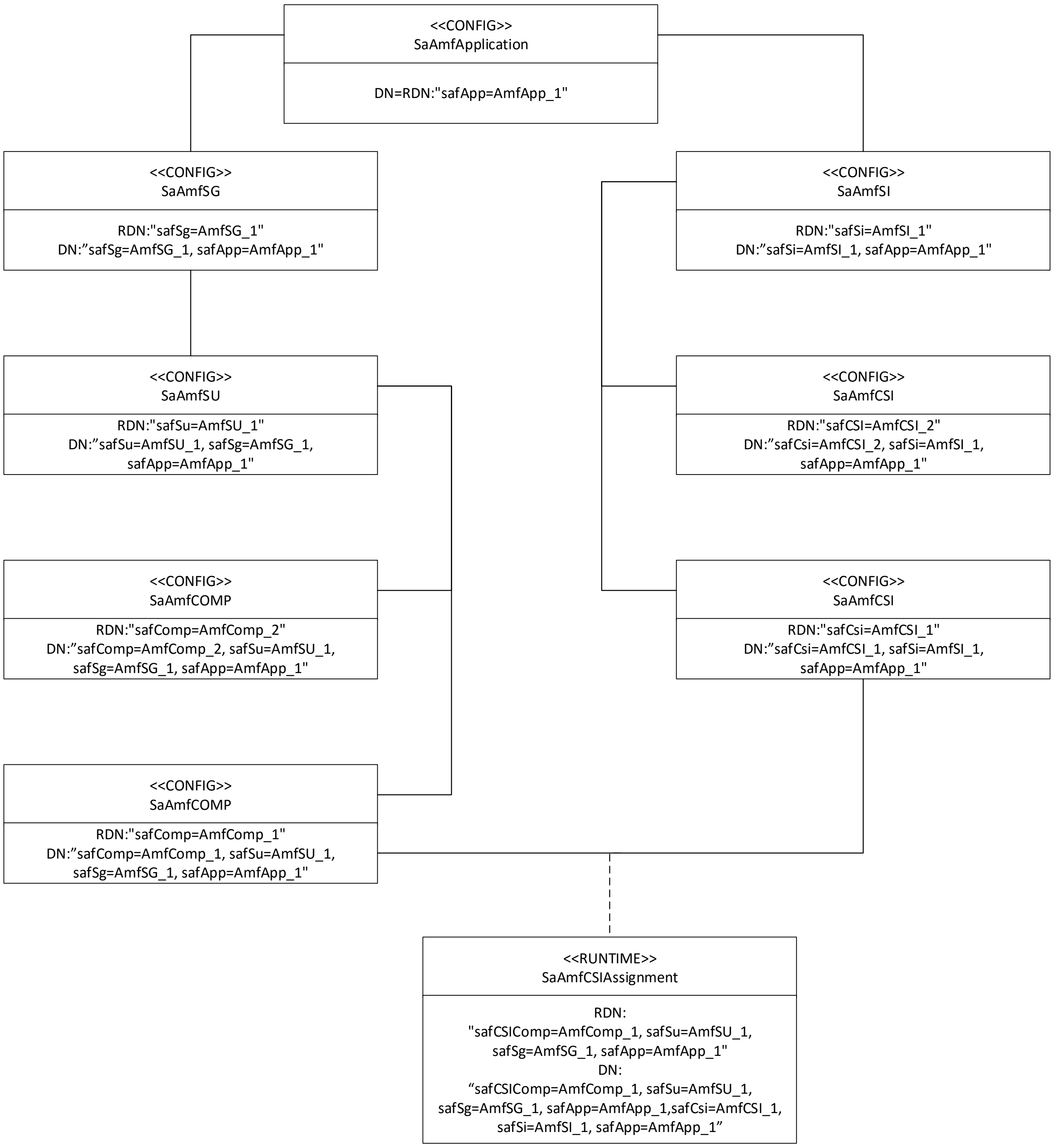

<<CONFIG>>
SaAmfApplication
DN=RDN:"safApp=AmfApp_1"
<<CONFIG>>
SaAmfSG
RDN:"safSg=AmfSG_1"
DN:"safSg=AmfSG_1, safApp=AmfApp_1"
<<CONFIG>>
SaAmfSI
RDN:"safSi=AmfSI_1"
DN:"safSi=AmfSI_1, safApp=AmfApp_1"
<<CONFIG>>
SaAmfSU
RDN:"safSu=AmfSU_1"
DN:"safSu=AmfSU_1, safSg=AmfSG_1, safApp=AmfApp_1"
<<CONFIG>>
SaAmfCSI
RDN:"safCSI=AmfCSI_2"
DN:"safCsi=AmfCSI_2, safSi=AmfSI_1, safApp=AmfApp_1"
<<CONFIG>>
SaAmfCOMP
RDN:"safComp=AmfComp_2"
DN:"safComp=AmfComp_2, safSu=AmfSU_1, safSg=AmfSG_1, safApp=AmfApp_1"
<<CONFIG>>
SaAmfCSI
RDN:"safCsi=AmfCSI_1"
DN:"safCsi=AmfCSI_1, safSi=AmfSI_1, safApp=AmfApp_1"
<<CONFIG>>
SaAmfCOMP
RDN:"safComp=AmfComp_1"
DN:"safComp=AmfComp_1, safSu=AmfSU_1, safSg=AmfSG_1, safApp=AmfApp_1"
<<RUNTIME>>
SaAmfCSIAssignment
RDN:
"safCSIComp=AmfComp_1, safSu=AmfSU_1, safSg=AmfSG_1, safApp=AmfApp_1"
DN:
"safCSIComp=AmfComp_1, safSu=AmfSU_1, safSg=AmfSG_1, safApp=AmfApp_1,safCsi=AmfCSI_1, safSi=AmfSI_1, safApp=AmfApp_1"

Figure 2-9: Example of information model

2.5. Elasticity Engine

An Elasticity Engine [10] has been proposed recently for AMF managed applications. The Elasticity Engine requires as input the load changes in terms of SI DNs.

When notified of a workload change, the Elasticity Engine reacts by manipulating SI or SG attributes in the writable configuration based on a number of strategies [10]. AMF in turn applies the changes made in the configuration by the Elasticity Engine by implementing the new changes.

As shown in Fig. 2-15, the Elasticity Engine is composed of the 'Elasticity Controller', 'Redundancy Model Adjustor' and the 'Buffer Manager'.

The activities of the Elasticity Engine is described below:

The Elasticity Engine may scale resources in a cluster due to two reasons:

a) The Elasticity Engine may receive triggers from the Monitoring Engine described in this thesis due to workload change associated to an SI as shown in Fig. 2-10.
b) The Elasticity Engine may scale workload due to addition or removal of services. I.e. the number of SIs may change in the AMF configuration. The Elasticity Engine receives information about such changes from the IMM.

Once alerted about workload change, the Elasticity Engine Controller reads the AMF configuration in the IM to identify the SG protecting the SI that has changed workload. Depending on the identified SG's redundancy model, the Elasticity Engine Controller calls the Redundancy Model Adjustor [10].

The Redundancy Model Adjustor responds to the call by reading the AMF configuration attributes of the SG in the IM using IMM and calculating the configuration changes required to adjust the SG's configuration to scale the cluster [10].

In order to speed up future adjustments, some nodes may be reserved for the SG. To accommodate that, the Redundancy Model Adjustor calls the Buffer Manager to reserve nodes or free up allocated nodes via additional Configuration Change Bundles [15] [10].

If the aforementioned actions taken are not effective, the Elasticity Engine Controller will do one or more of the following:

a) Alert the administrator or cloud manager: The Elasticity Engine Controller will inform the administrator or the cloud manager to add or remove node to/from the cluster if the cluster size is insufficient/nodes are not being utilized [10].
b) Alert the administrator or software management: The Elasticity Engine Controller will inform the administrator or the software manager if new nodes are required to cope with the workload increase in the cluster [10].

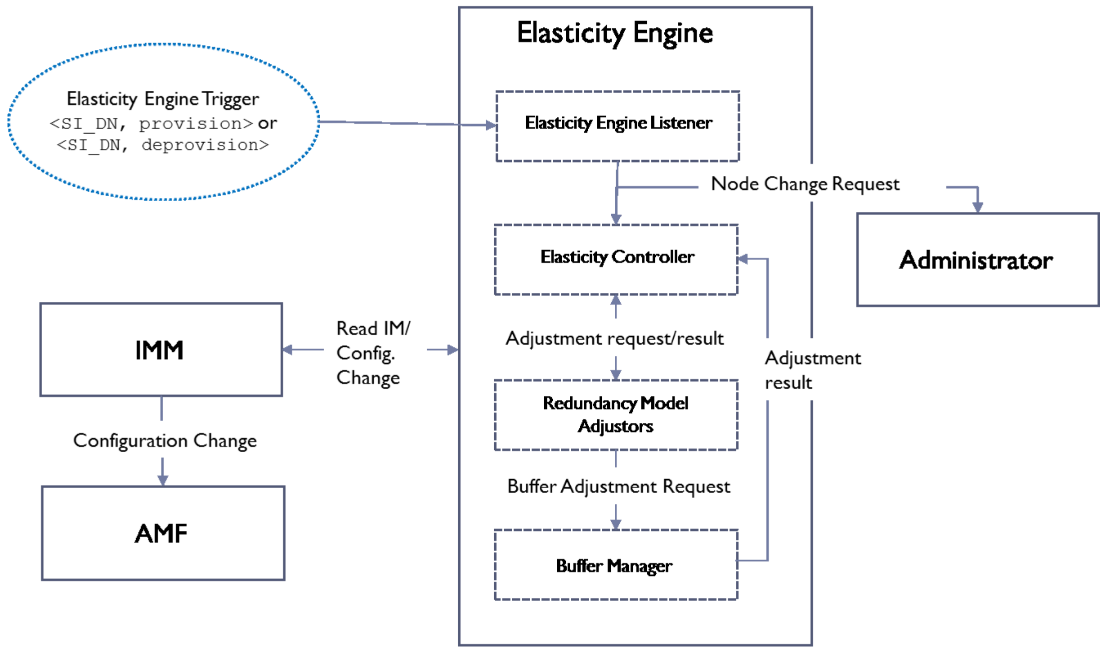


Figure 2-10: Elasticity Engine Architecture

2.6. Monitoring and tracing tools

There are a number of tools that can provide system usage metrics readily like top [17], vmstat [18], uptime [19], PSUtil [20] [21], etc. Most of such tools come with the support to provide CPU usage, Linux server status, process monitoring and such. OpenStack's [22] Ceilometer can be configured with its Heat engine to enable AutoScaling [23]. Ceilometer can be extended to use it as a monitoring solution. However, since it does not operate on real-time data, the solution offered by it has not been considered in this thesis [23].

LTTng, short for "Linux Trace Toolkit: next generation" is an open source system software package for correlated tracing of the Linux kernel, user applications and libraries. Its User Space Tracing (UST) feature enables tracing the interactions amongst C/C++/Java based multiple applications [24].

2.6.1. LTTng (Linux Tracing Tool, next generation)

There are a number of ways LTTng can be used to trace the running kernel, application and services. Three of LTTng features have primarily been used in this work.

2.6.1.1. LTTng Kernel Tracing

LTTng can trace the running Linux kernel processes and create a data dump for a tracing session that can be read later using any of the LTTng Trace Viewers [24].

The data from kernel tracing includes the active tasks running on the CPU as well as their scheduling information, memory allocation, etc. against timestamps.

2.6.1.2. LTTng User Space Tracing (LTTng UST)

LTTng User Space Tacing facilitates tracing specific applications that has pre-defined trace points in them. A trace point acts like break points in common IDEs that provides debugging information. It is a short C code snippet that sends data about the state of the application to the LTTng session daemon [24].

Among other features, a trace point can provide the timestamps of the starting point and the ending point of a specific section of an instrumented application source code's execution.

2.6.1.3. LTTng Live

LTTng Live [24] feature is used to obtain LTTng trace data during a programs execution. In LTTng Live, for each session, a maximum amount of trace is instructed to be cached. The cached trace is then processed in runtime by a Trace Viewer. Each session daemon caches a specific number of events.

2.6.1.4. Babeltrace

Babeltrace [25] is the open source LTTng Trace Viewer that is used to convert Common Trace Format (CTF) data into text format. Babeltrace's Python binding can be used to convert LTTng live data stream in runtime into text format.

2.6.2. Python PSUtil

PSUtil (Python System and Process Utilities) [20] is a cross-platform library/module for retrieving information on running processes and system utilization (CPU, memory, disks, network, etc.) in Python. It is useful mainly for system monitoring, profiling and limiting process resources and management of running processes. It currently supports Linux, Windows, OSX, FreeBSD and Sun Solaris, both 32-bit and 64-bit architectures [20].

2.7. Cloud Computing

Cloud computing can be thought of as a computing over network approach where an application runs on a group of remote servers owned by a service provider to serve the end users [26] [27]. The provider rents the computational power to their customers in an on-need basis, which introduces the pay-as-you go model [26]. In this model, the customers pay for only the amount of resource they use. This model is one of the primary contributors to cloud computing's popularity. It is also possible for an individual or a company to create a cloud infrastructure on their own data center. Such infrastructures are called private clouds. Private clouds are created, operated and managed by a single organization. In the previous example where a cloud service provider *rents* the computational power, the cloud infrastructure is called public cloud. Public cloud is the more common infrastructure solution [26].

Cloud services are offered following three different service models: IaaS, PassS and SaaS; which stand for Infrastructure as a Service, Platform as a Service and Software as a Service, respectively. In IaaS the customers have full control over their infrastructure and are provided with computing resources such as Virtual Machines (VMs), power supply, network connection, load balancers, firewalls, IP addresses, storage, etc. PaaS is the intermediate service model where the customers can deploy their own application and take care of it while the cloud service provider manages all the underlying infrastructural aspects. In the SaaS model, complete software packages are offered as ready to use, on-demand at a monthly or yearly fee. However, the end users cannot customize it more than the provider allows [26] [27].

2.8. Related work

In this section, predominant trends and examples of currently available monitoring solutions are discussed, which is followed by a subsection discussing the limitations of the existing solutions to provide service level workload in a cluster managed by an SA Forum middleware.

2.8.1. Available Monitoring Solutions

There are two predominant trends of monitoring in the cloud. One is monitoring at the platform level, which provides the resource usage based on system usage metrics such as CPU usage, memory usage, bandwidth, etc. Boundary [28], Amazon CloudWatch [6], Rackspace [29] and Microsoft Azure [30] among others offer such solutions. The other trend is monitoring at the application or the service level, which provides resource usage based on the aggregated performance of different entities used by the application, i.e. a typical web application's overall performance depends on bandwidth usage, its host VM's CPU usage, memory usage, disk usage, etc. AppDynamics [31], Rackspace [29], Aternity [32] among others belong to this category. Some of these solutions monitor applications based on the responsiveness and availability of the application using an outside agent [31] [29], following the trend of application-level monitoring solutions. This allows the monitor to evaluate the performance of the application from the perspective of an end user, but not the workload imposed on the system and its distribution.

Monitoring solutions offered by either of the aforementioned trends cannot be used directly in the context of SA Forum middleware to monitor workload changes at the service level as they cannot relate to the SA Forum concept of service.

There have been some notable attempts to solve the problem of monitoring applications in the cloud. In [5], the authors looked into the application deployment on to the mOSAIC framework and introduced multi-layered monitoring. mOSAIC is an open source framework [33] which offers an abstraction somewhat similar to that used by the SA Forum middleware. The concept of 'component' in mOSAIC is similar to the concept of the component with assigned CSI in the SA Forum terminology, although the SA Forum specifications have a clearer distinction between service and service provider. Components are stateless in the context of mOSAIC, unlike the components in the SA Forum middleware. A cloud application in mOSAIC is essentially a set of interconnected components forming a cloudlet, which is deployed redundantly in a cloudlet container, similar to the concept the SG containing a number of SUs with NWay-Active redundancy model in SA Forum context. mOSAIC components are developed based on cloudlet APIs. The cloud provider can scale and manage availability of an application by managing the size of the cloudlet container of that application. In the monitoring solution for mOSAIC, the cloud resource usage from the different cloud providers is detected by the 'observer' in a similar manner to which the PSUtil tool has been used to measure system workload in this work. Also, the resource usage of a mOSAIC application is collected using the 'connector' in a similar manner to which LTTng UST probes are used to detect AMF callbacks in this work. In the monitoring solution discussed for mOSAIC, the cloud application developer needs to develop/update the connector, observer and the warning components for each application based on the mOSAIC API, the application's architecture and its requirement. In contrast, in this work, depending on the component implementation of an application can be instrumented automatically using the auto-instrumenter tool for monitoring. If the automatic instrumentation is not possible, it needs to be done manually for the components. Apart from that, the rest of the application deployment procedure with respect to monitoring is the same for all applications. Since AMF is the entity that manages the resources and assigns/reassigns services to resources for all deployed applications based on a set of well-defined rules, it is possible to devise a general solution to map workload to services for most applications.

In [34], the authors introduced a multi-layered monitoring service, i.e. a monitoring service that is capable of monitoring at IaaS, PaaS and SaaS levels simultaneously. It is called CLAMS—Cross-Layer Multi-Cloud Application Monitoring-as-a-Service Framework. The objective of this work is to collect and present a complete view of Quality of Service (QoS) of the applications in the cloud. It achieves it by taking a monitoring agent based approach. The agents are deployed in various levels of the cloud provider to collect monitoring data. The framework is compatible with a number of popular cloud service providers. While this provides a more comprehensive and detailed monitoring data, it lacks support for service-assignment driven application monitoring. The problem presented in this thesis requires a monitoring solution that can measure workload on a platform and dynamically associate the measured workload to a service based on its assignments. Therefore, the same resource may be associated with multiple services throughout its life cycle and the service workload measurements associated with it need to be taken into account accordingly. In spite of the vast QoS metrics offered by this framework, a solution to the problem at hand is not available in this work.

In [35], the authors introduced a monitoring solution (DoLen) to detect distributed denial-of-service (DDoS) attacks using monitoring probes in a server-client architecture where the monitoring server bears the responsibility of aggregating usage data and detecting possible DDoS attacks. While this monitoring architecture is similar to the one used in this thesis, the monitoring objective is very different from ours. They correlate events to detect DDoS attacks while we aggregate resource usage to map to the service level workload. Moreover, this work also does not address the problem of dynamic workload association to services as previously discussed.

2.8.2. Resource usage representation: Hardware vs. AMF SI

In a cluster setup, the workload measurements are collected from each of the cluster's node or VMs by a Monitoring Engine. As shown in Fig. 2-11, workload measurements are collected from each node of the cluster by the Monitoring Engine. In the context of this description, VMs and nodes refer to the same entity. The Monitoring Engine then aggregates the workload data and outputs a summary of the cluster's overall workload, which may be expressed in term of the cluster's VMs' resource usage metrics, e.g. CPU usage, memory usage, network bandwidth usage, etc. The limitation of above approach is that the workload of the VMs are associated to the services they are providing permanently. This approach does not consider the possibility that the services can be removed/re-assigned from the VMs over time. By assuming that the VMs maintain the same service assignments at all-time result in incorrect monitoring output.

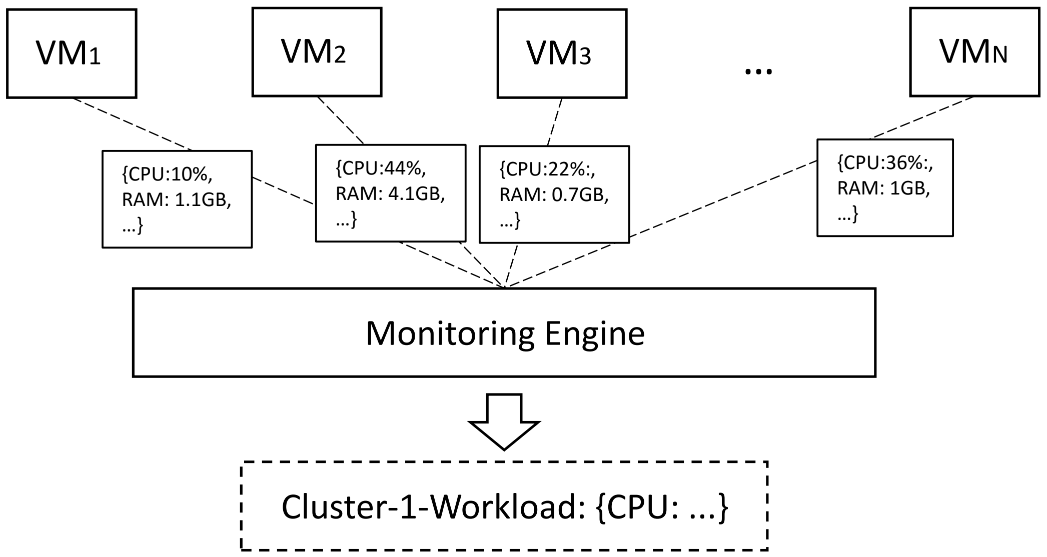

VM1
VM2
VM3
...
VMN
{CPU:10%, RAM: 1.1GB, ...}
{CPU:44%, RAM: 4.1GB, ...}
{CPU:22%:, RAM: 0.7GB, ...}
{CPU:36%:, RAM: 1GB, ...}
Monitoring Engine
Cluster-1-Workload: {CPU: ...}

Figure 2-11: Monitoring in terms of hardware entities

In the case where each VM has multiple components, each of which provides one or more service(s) and the services can be assigned/removed from the components over time, a fine-grained monitoring and data collection is essential to estimate the workload of services.

For example, as illustrated in Fig. 2-12 where VMs from VM_1 to VM_3 host components that provide Service-1; VMs from VM_2 to VM_N host components that provide Service-2. The services can be assigned/re-assigned to the components dynamically. The existence of a service provider entity capable of providing service (I.e. a component) does not necessarily imply that the resource usage of that service provider must be associated with the service that it is providing intermittently. There needs to be a valid assignment of a service to the service provider entity to correctly associate the service provider's load with the service. In other words, a component can exist and run on a VM at all-time but whether the workload of that component should be associated to a service depends on the *assignment* of a service to the component. Without the assignment of a service to a service provider, the workload of the service provider is irrelevant with respect to services. For example, in Fig. 2-12, comp-1 of VM-2 does not have any valid service assignment, hence in the solution, its workload is not taken into account while measuring the workload of Service-1. The resource usage of a service needs to be continuously updated as services are assigned/re-assigned to components at runtime.

Different component types are tied to the types of service they can provide. I.e. one type of component can provide a set of defined of services and is not capable of providing a service beyond its capability. A VM may host many components providing many different services. It is possible for multiple services to be provided from the same VM as a VM can host many types of components and those components can have many types of services assigned to them. In a system where VMs are dynamically assigned to applications or services and it is possible for different applications and services to collocate, monitoring VM level measurements would provide incorrect output.

Note that the service collocation problem is not completely solved by the approach introduced in this thesis. While it is possible to detect service assignment-reassignment at the process level over time following the approach introduced in this thesis, it is not possible to differentiate between two different services provided by the same process at the same time. Similarly, it is not possible to measure load of two different services provided by the same component simultaneously using the approach introduced in this thesis. For the approach to be effective, it is important that a process and components run by that process do not have one-to-many relationship, and a component and its services do not have one-to-many relationship.

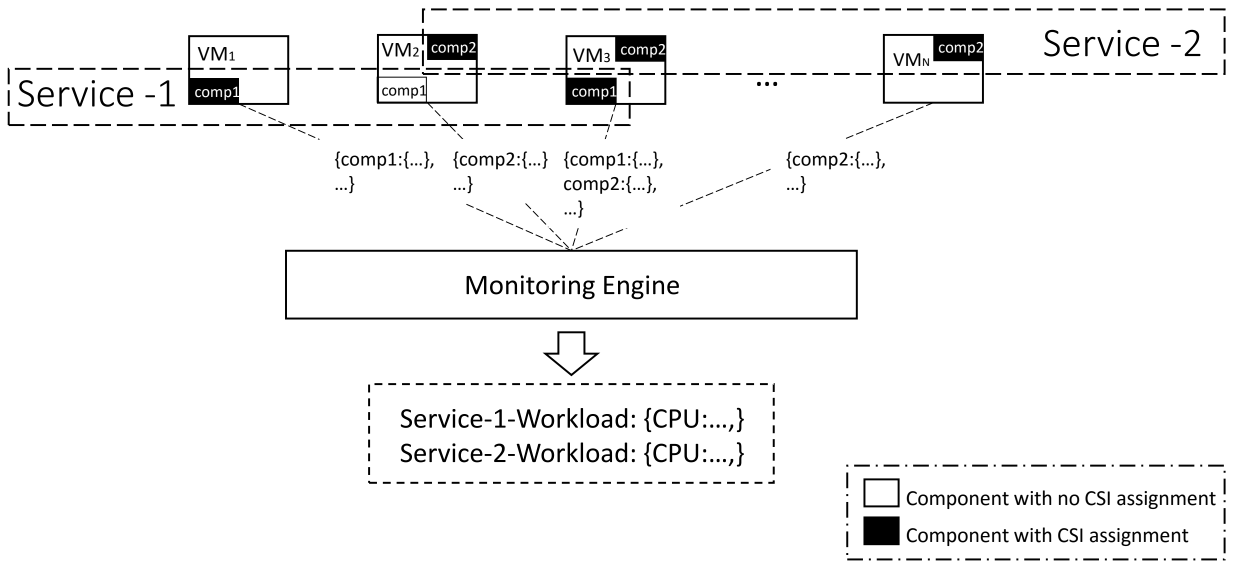

Service -2
VM1
comp1
VM2
comp2
comp1
VM3
comp2
comp1
VMN
comp2
Service -1
...
{comp1:{...}, ...}
{comp2:{...} ...}
{comp1:{...}, comp2:{...}, ...}
{comp2:{...}, ...}
Monitoring Engine
Service-1-Workload: {CPU:...,}
Service-2-Workload: {CPU:...,}
Component with no CSI assignment
Component with CSI assignment

Figure 2-12: Service level system resource usage representation

In the setup illustrated in Fig. 2-12, VM-2 and VM-3 host components capable of providing both Service-1 and Service-2. Therefore, in VM level workload data, Service-1 and Service-2 are collocated in terms of VMs. A monitoring solution that only measures VM load will associate some load of Service-2 with Service-1 and vice versa.

Comparing the clusters and their corresponding monitoring solutions illustrated in Fig. 2-11 and Fig. 2-12, we show that the existing monitoring solutions are not capable of adapting to the dynamic nature of the services in a cluster managed by an SA Forum middleware. We conclude that a new monitoring solution needs to be introduced where the solution will take into account the states and the dynamic nature of the services managed by the SA Forum middleware.

3. Monitoring Approach Overview

In this chapter, a monitoring approach to interpret resource usage in terms of services is discussed. First, a monitoring approach to measure workload in terms of services and its related architecture are discussed. This is followed by the overall view of the integration with the Elasticity Engine [10] in system running an SA Forum middleware. In the final sub-section, we discuss the overall activity of the integrated system to show the interactions among the proposed Monitoring architecture and the other entities involved in the integrated system and conclude the chapter.

3.1. Monitoring architecture

In this section, a monitoring architecture is presented to measure workload (or system usage) in terms of SIs for AMF managed applications deployed in the cloud primarily to enable elasticity management. The Monitoring Engine follows a client-server architecture, hence architecture is divided in two main sections: Monitoring Client and Monitoring Server. The discussion on architecture is concluded by a section discussing the overall activity breakdown of the Monitoring Engine.

Monitoring architecture is illustrated in Fig 3-1. Each node in a cluster running a SAF middleware that hosts a number of AMF managed components and also hosts a Monitoring Client. For each cluster, a node hosts a Monitoring Server. The Monitoring Clients communicate with the Monitoring Server over TCP. It is possible to configure a standby Monitoring Server keeping the potential failure of a single Monitoring Server in mind. For simplicity of discussion, we will consider only one Monitoring Server while discussing the architecture. An architecture with multiple Monitoring Servers (active and standby) is shown in Chapter 6.

3.1.1. Monitoring Client

For the Monitoring Clients to function, the AMF components hosted on the nodes need to be instrumented. The instrumentation enables the Monitoring Client to detect the AMF callbacks to the components. The methods to detect AMF callbacks to the components and resource usage of components are as follows.

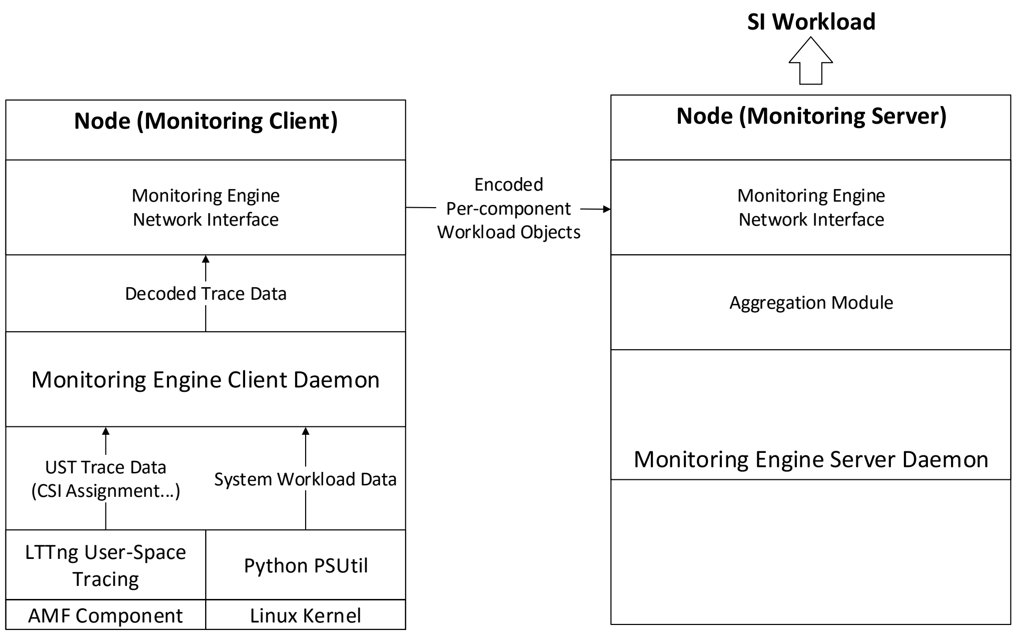

SI Workload
Node (Monitoring Client)
Monitoring Engine Network Interface
Decoded Trace Data
Monitoring Engine Client Daemon
UST Trace Data (CSI Assignment...)
System Workload Data
LTTng User-Space Tracing
Python PSUtil
AMF Component
Linux Kernel
Encoded Per-component Workload Objects
Node (Monitoring Server)
Monitoring Engine Network Interface
Aggregation Module
Monitoring Engine Server Daemon

Figure 3-1: Monitoring architecture

3.1.1.1. AMF Callback Detection

The prerequisite to map system usage to SI workload at runtime is the instrumentation of the AMF components. It is possible detect interactions between the AMF the components using LTTng UST [24].

Once instrumented with LTTng UST, the instrumented components generate an ‘event’ every time the instrumented portion of code is executed. This enables the Monitoring Client to receive a component's life-cycle events as LTTng UST events. These UST events also carry data related to the state of the component such as assigned/removed CSI, HA state change, component's process ID, etc. The methods to detect callbacks dispatched by the AMF to the components are described below:

SA Aware components: To manage the life-cycle of an SA Aware component, AMF interacts with the component using the AMF APIs [9]. For each new CSI assignment, CSI assignment change or CSI assignment removal from a component, AMF dispatches a callback to the component using this interface. The instrumentation of the AMF-component interface ensures that the AMF callbacks are detected at runtime by the monitoring system as shown in Fig. 3-2.

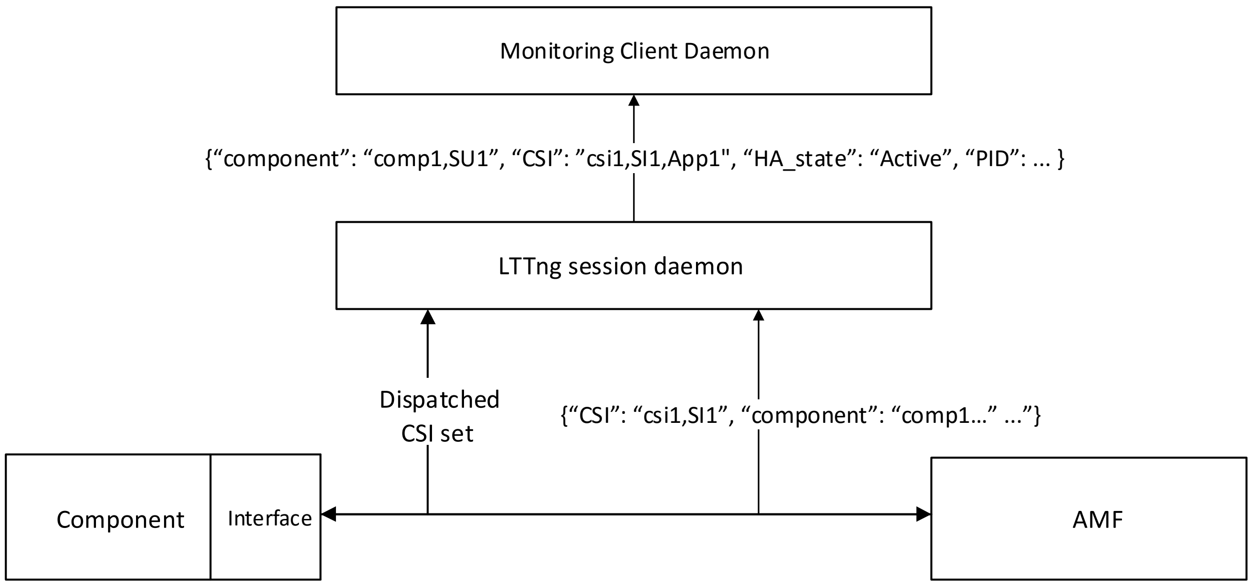

Monitoring Client Daemon
{"component": "comp1,SU1", "CSI": "csi1,SI1,App1", "HA_state": "Active", "PID": ... }
LTTng session daemon
Dispatched CSI set
{"CSI": "csi1,SI1", "component": "comp1..." ..."}
Component
Interface
AMF

Figure 3-2: AMF callback dispatch detection using LTTng UST for SA-Aware Components

Each callback from AMF to the component generates an LTTng UST event and is saved in the LTTng UST session trace. Such an event includes the component DN, the DN of the CSI assigned to it, the HA state assigned to the component on behalf of the CSI, and the ID of the process implementing the assignment. The Monitoring Client on each node of the cluster collects such UST events and creates a list of component to CSI map. This component to CSI map is updated periodically. By these means, the state of all the components present in each of the node is collected and updated by the Monitoring Client.

The instrumentation of AMF components is a critical for the Monitoring Client to function. Manual instrumentation of AMF components is a time-consuming process. Therefore the entire process is automated including searching for function declaration patterns, creating trace-points by extracting function parameters and inserting them into the AMF component source code. The instrumentation procedure and its automation are discussed in further details in Chapter 4.

Non Proxied Non-SA-Aware components: Since AMF's interaction with Non Proxied Non-SA-Aware components are limited to CLI-commands [9], the instantiation and termination scripts for these components are used with a wrapper that is pre-instrumented with LTTng UST probes to detect interactions between the AMF and components of this category. As shown in Fig. 3-3, all administrative commands issued by the AMF are first received by the instrumented wrapper, which forwards the commands to the administrative command script. Meanwhile, the instrumented wrapper generates traces based on the commands issued by the AMF and the target component's respective environment variables.

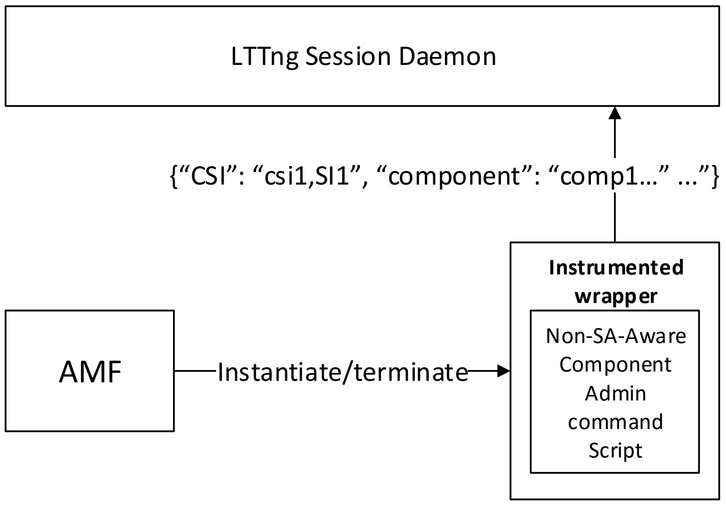

LTTng Session Daemon
{"CSI": "csi1,SI1", "component": "comp1…" …"}
Instrumented wrapper
Non-SA-Aware Component Admin command Script
AMF
Instantiate/terminate

Figure 3-3: AMF callback dispatch detection using LTTng UST for Non-SA-Aware Components

3.1.1.2. Per-Component Workload Measurement

Mapping component to process: In the context of SA Forum Specifications component is the smallest service provider entity recognized by AMF. On the other hand, in the context of hardware entities, the smallest entity in terms of which system resource usage is measured in this work is a process. Therefore to determine the resource usage of a component, the workload of the process or processes associated to the component needs to be mapped to the component's identity. The dual identity of a component that is a process in a system is referred to as *component-process* in this work.

Mapping component-process to CSI: Each component must have a valid CSI assignment to participate in providing a service. A program can be considered as a component when AMF can control its lifecycle and assign/remove services to/from it. I.e. AMF can instantiate, terminate, assign and remove CSIs from the executing program. The process created at starting the program and any process that is spawned due to a CSI assignment dispatched call from AMF to the program is considered as a component-process in the context of this work. Note that this is not the case for all components. It is possible for some components to get different CSI assignments for different threads of the same process or even get CSI assignment that are to be assigned to threads of a different process altogether. Such cases have not been covered in this work.

The workload of a component-process with a CSI assignment is essentially a part of the workload of an SI.

Measuring component-workload: As introduced in Chapter 2, the resource usage of a process can be measured based on its process ID using, for example, the Python PSUtil [36]. All component-process' process ID can be detected by analyzing the trace data obtained by component instrumentation as discussed in sub-section 3.2.1.1.

With the considerations above, the workload of a process or a set of processes related to component is considered to be the workload of a component. The relationship between components, process IDs and CSIs are detected from the trace events received from the instrumented components. The mappings described so far are summarized in table 3-1.

Table 3-1: Mappings among component, CSI, process ID and process workload

Source	**Collected data**	**Example**	**Mapping**
AMF component interface, Instrumented component wrapper (Collected in LTTng UST session)	Event type, component DN, CSI DN, process ID, HA state	{'Wed Jun 24 07:36:03 2015':{'type':'csi_assignment', 'CSI':'safCsi=CSI_1,safSi=SI_1,safApp=app_1', 'component':'safComp=comp_1,safSu=SU_1,safSg=SG_1,safApp=app_1' , 'HAState':'Active' , 'CSIFlags':'Add One', 'PID':18671}}	component DN to process ID, component DN to CSI DN
Python PSUtil	Workload from selected process IDs	18671: {CPU_usg: 4.4, mem_usg: 0.03}	Process ID to workload metrics

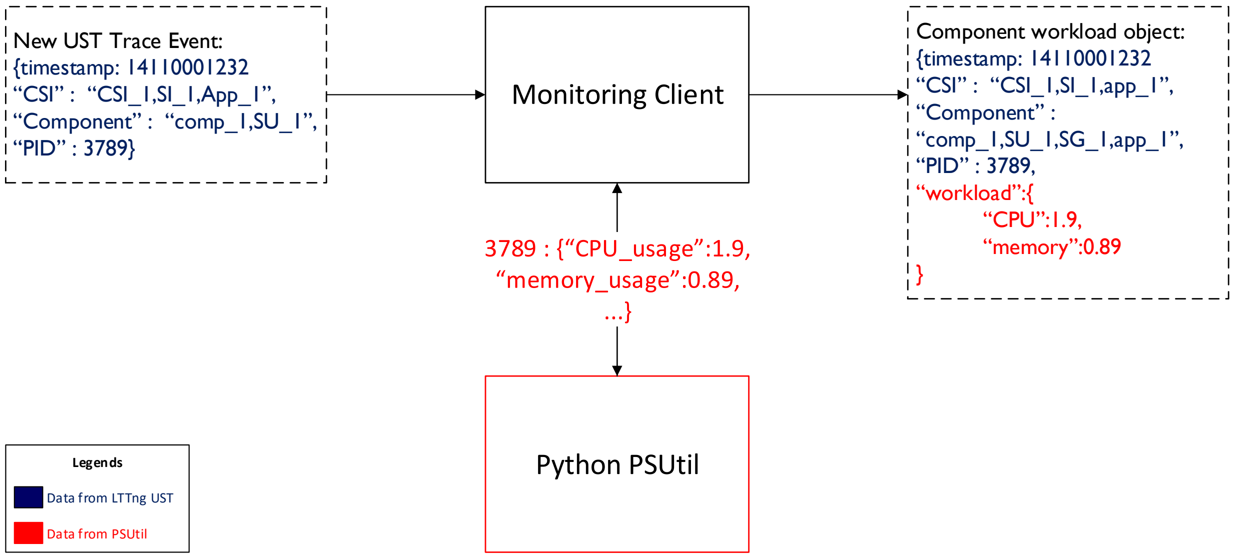


Figure 3-4: Collecting workload-per-component

As illustrated in Fig. 3-4, on each new UST event, the Monitoring Client determines whether or not to collect the usage data for that component based on the analysis of that event. Each UST event contains data regarding the type of an AMF callback to a component from which the event was originated. For an event showing the CSI assignment to a component as shown in Fig. 3-4, the Monitoring Client starts collecting workload for that component using the process ID of that component obtained from the event trace. The workload data collected using Python PSUtil tool and the data collected from the UST event trace are merged to create a data structure that is transmitted to the Monitoring Server as shown in Fig. 3-1. The merged data structure containing per-component workload is referred to as component-workload-object. The workload data of a component is collected and transmitted to the Monitoring Server as long as the component-process is not dead or there is no UST event showing either CSI remove or component termination callback has been dispatched from AMF to the component. In case of CSI removal or component termination dispatch call, the Monitoring Client stops collecting and sending workload data for that component to the Monitoring Server.

3.1.2. Monitoring Server

The Monitoring Server receives the workload data from all Monitoring Clients periodically. After decoding the per-component workload data from the Monitoring Clients, the Monitoring Server Aggregation Module generates, for the first time, a tree as illustrated in Fig. 3-5. The tree is populated from the top according to the following hierarchy: SIs, their CSIs, the components serving the CSI assignments, and the CSI related component workload. The tree is updated periodically with each new component-workload-object received. The SI workload is calculated by aggregating the CSI related component workloads following its associated sub-tree. The Monitoring Server Aggregation Module performs this aggregation periodically. The SI workload is analyzed by the workload analyzer to detect any condition violation. The Monitoring Server and its aggregation module algorithms are discussed in further details in Chapter 6.

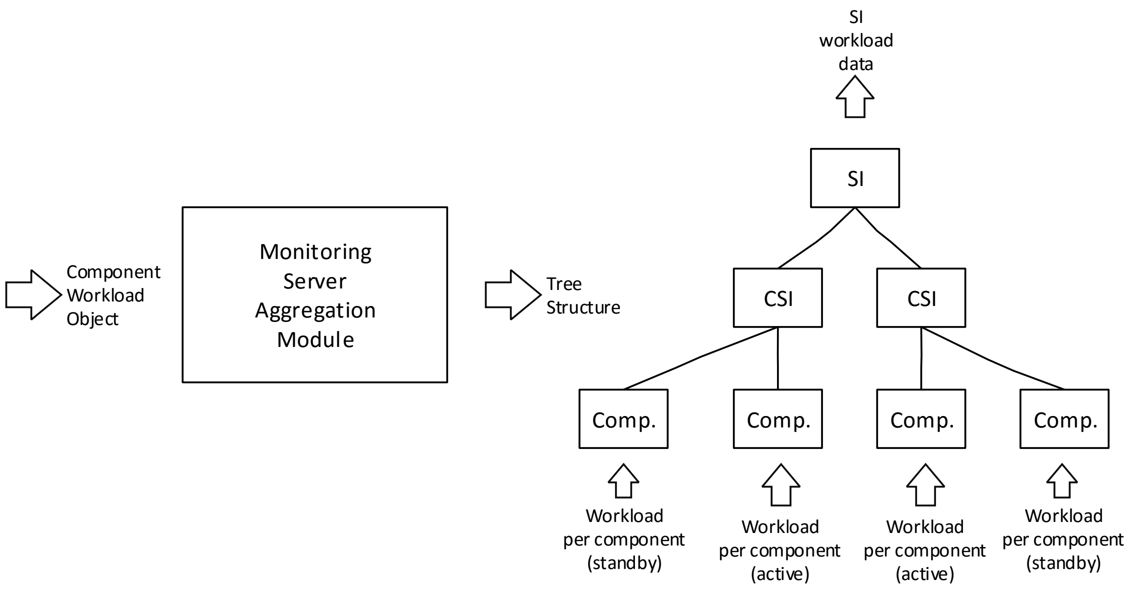


Figure 3-5: Workload aggregation in the Monitoring Server

3.2. Integration with the Elasticity Engine

The purpose of monitoring of the SI workload is to alert the Elasticity Engine about significant workload changes. Fig. 3-6 shows the architecture integrating the monitoring approach with the Elasticity Engine and AMF. In this architecture, the Monitoring Server sends a constant stream of SI workload measurement data to the Workload Analyzer that maintains a number of policies for triggering overprovisioning/under-provisioning alerts. For example, if the workload of any SI exceeds a threshold set in the Workload Analyzer, it sends an alert to the Elasticity Engine, notifying it of the DN of the SI and its workload status such as workload increase or workload decrease. The Elasticity Engine [10] then reads the current configuration of the SG protecting the SI through the IMM service, calculates any necessary configuration changes at the SG and possible at the cluster level and commits those changes through the IMM service. IMM in turn notifies AMF of the configuration changes. AMF applies the configuration changes by dispatching callbacks to redistribute the CSI assignments to the components in the nodes of a cluster in such a way that matches the best with a new configuration [9]. As a result, in the nodes of the cluster, the Monitoring Clients detect the new CSI assignments to the instrumented components by detecting the dispatched callbacks from AMF. The new distribution of CSI assignments is reflected in the component workload objects, which are sent from Monitoring Clients to the Monitoring Server. Based on these new component-CSI assignment relations the Monitoring Server adjusts the SI-tree as discussed in sub-section 3.2.2 and in more details in Chapter 6.

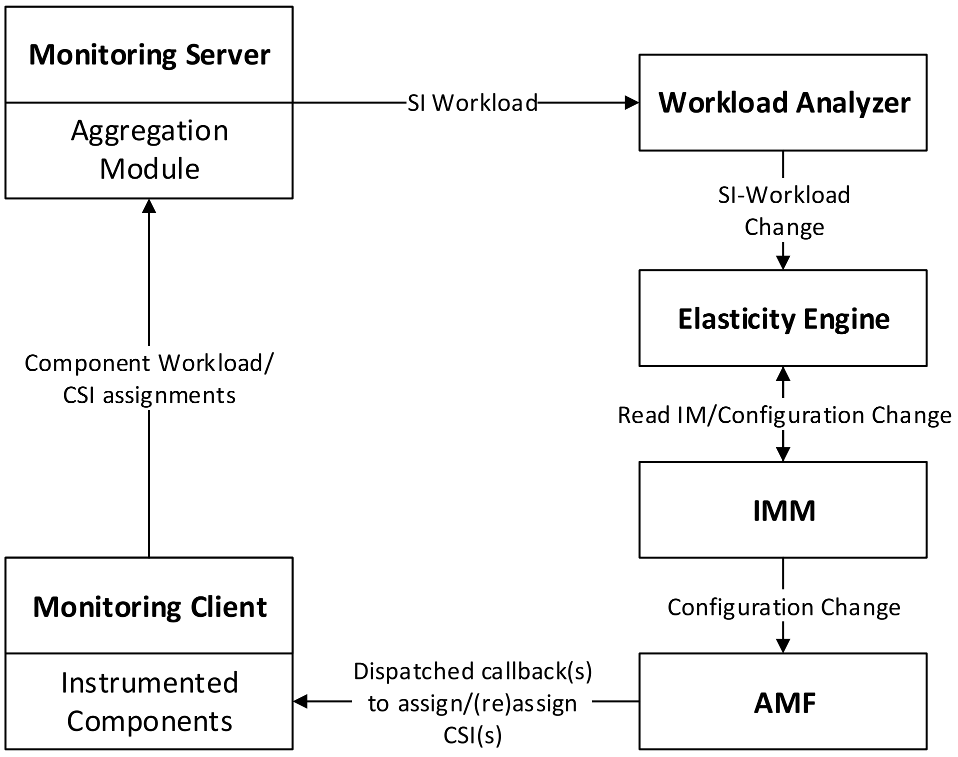


Figure 3-6: Monitoring approach/architecture integrated with the Elasticity Engine and AMF

3.3. Activity overview

The interactions among the elements of Monitoring Client, Monitoring Server, Workload Analyzer and the Elasticity Engine can be summarized by Fig. 3-7.

- In each node of the cluster, an LTTng UST session is started when the Monitoring Client is initiated.
- When any component receives a CSI-set callback from the AMF, a UST-event-trace is created, which is detected by the UST Session Daemon. The Monitoring Client Daemon detects all new events from the UST trace.
- The Monitoring Client Daemon maps the component's DN, CSI DN, process ID and HA state collected from the trace and collects the workload of the component-process using Python-PSUtil tool on fixed intervals. The workload data collection continues as long as the component has a valid CSI assignment and its process ID is alive.
- The component workload data is appended to the data collected from the UST trace to form component-workload-objects which are sent to the Monitoring Server.
- The Aggregation Module of the Monitoring Server receives component-workload-objects from all Monitoring Clients in the cluster and aggregates the data into SI workload, which is then sent to the Workload Analyzer.
- The Workload Analyzer checks if the SI workload breaches any condition to trigger elasticity alert(s). The alerts triggered from the Workload Analyzer consists of the SI DN which breached the condition and the condition type. For example, if the Workload Analyzer determines that the cluster is at under-provisioned state, the trigger would consist of the SI DN for which the SG is in under-provisioned state and a flag to notify that the trigger is for under-provisioned status. Similarly, if the

Workload Analyzer detects that the SG protecting an SI is at overprovisioned state, the trigger will consist of the SI DN for which the SG is at overprovisioned state and a flag to notify the overprovisioned state.

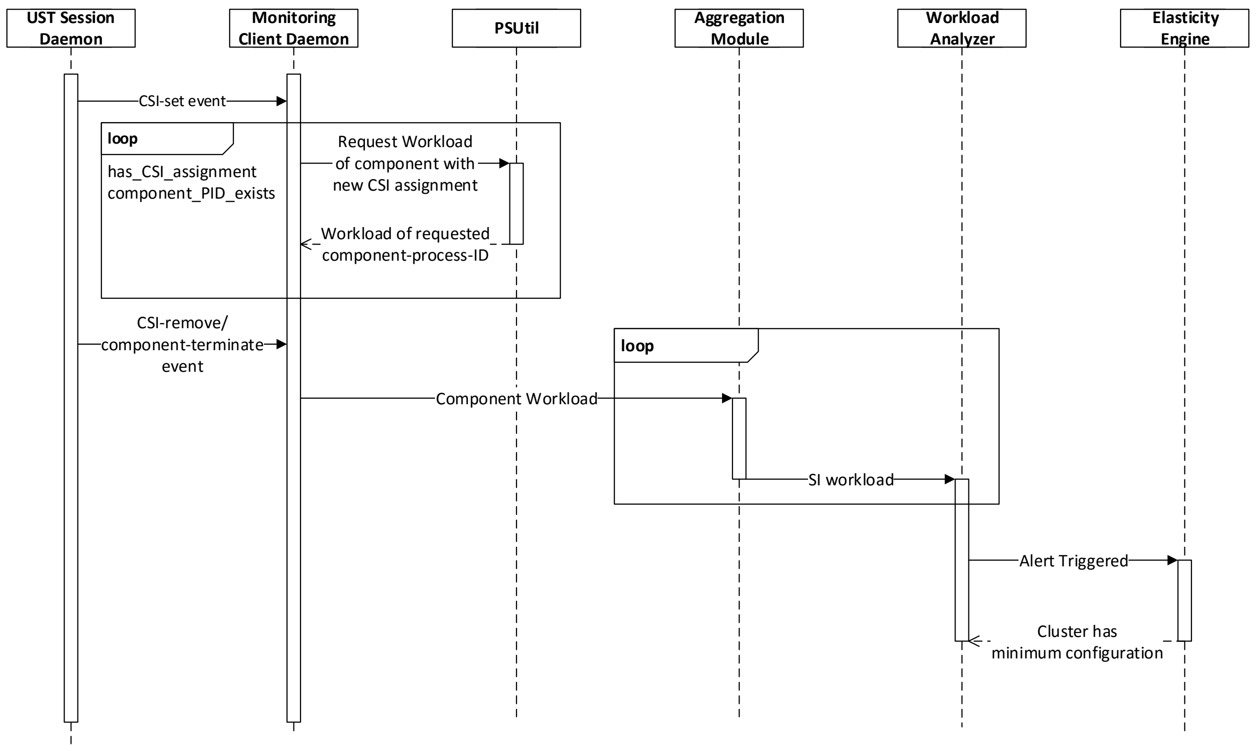


Figure 3-7: Sequence diagram of interactions between Monitoring Engine and Elasticity Engine

3.4. Conclusion

In this chapter, we introduced an approach and an architecture for the monitoring of workload at the service level applicable to the different services that may be provided by application components collocated in the same VM and where the service to application component assignments change dynamically in the system over time. In the subsequent subsections in this chapter, we discussed how the state of each component in the system is detected and used to measure service level workload in the architecture introduced. We concluded that by keeping track of the state of all components in the system while mapping the system load to components and then aggregating the components' load to their corresponding service assignments allows us to measure the load of collocated services in an environment where the service assignments are dynamic. In the subsequent chapters we discuss the methodologies to map system load to components and aggregating component workload to service workload.

4. Instrumentation of AMF Components

In the context of an SA Forum middleware, a component is the smallest unit of resource that is capable of performing a task [9]. A component transits through a number states that are driven by the state's corresponding life-cycle events during its service time. These life-cycle events are controlled by the AMF. Depending on the nature of a component's life-cycle event, the state of a component can also change. In order to monitor an AMF application, it is important for the Monitoring Client to be aware of the components' state in a system. Based on the component state information sent from the Monitoring Client, the Monitoring Server determines if the workload of the component should be associated with a service or not.

In order to make the Monitoring Client and subsequently the Monitoring Server aware of a component's state, the first step is to instrument the application components with a tracing tool like LTTng [24], following a method.

The possibilities of tracing applications using LTTng UST at runtime is vast; therefore the instrumentation instructions provided for LTTng UST does not include any specific instruction on where to put the tracing probes or how the trace results should be used. Formulating a purpose-specific LTTng User Space instrumentation method for any large application is a unique, one time solution (I.e. the instrumentation method for one application is not likely to be portable to another application). That said, the instrumentation method for similar applications built for same platforms are similar. AMF compliant applications are similar in nature as they run on components comprising of similar interfaces. Therefore, it is possible to formulate a method to instrument the AMF component interface source code that would be effective for all components of the same type. We have devised a method to instrument AMF components using LTTng. The method devised allows the user to provide a template that specifies the information that is to be obtained from the instrumented component.

Components are categorized into two types based on their service-availability-awareness: SA-aware components and Non-SA-aware components. Unlike SA-Aware components, AMF does not interact with Non-SA-Aware components directly via any interface, therefore, components of this category cannot be instrumented. As mentioned in Chapter 3, the callbacks from AMF to Non-SA-Aware components are detected using a pre-instrumented wrapper.

In this chapter, we discuss SA Aware components' lifecycle events and the method to instrument the component interfaces to detect such component lifecycle events. We also discuss a method that automates the instrumentation procedure and conclude with discussion on advantages and limitations of the instrumentation method.

4.1 SA Aware Components

SA-Aware components are chosen or written in a way that enables error detection, isolation and repair [9]. Components of this category interact with AMF via an interface. This interface implements specific workload assignment and recovery policies according to the API specified in the SA Forum Specifications. Such components must be designed in a way that the AMF can dispatch callbacks that dynamically assign CSIs to the target components and choose the roles in which the components will operate for each specific CSI assignment [9].

SA-Aware components are highly integrated with AMF and are under direct control of the framework. Each SA-aware component includes at least one process that is linked to the AMF library. One of these processes registers the component with AMF by invoking the `saAmfComponentRegister()` API function. This process, called the *registered process* for the component provides to the AMF references to the availability control functions it implements. These control functions are implemented as callbacks [9] [10].

Throughout the life of an SA-Aware component, AMF dispatches callbacks to the component to execute the following:

- assigning CSI to the component,
- removing CSI assignment from the component,

4.2 Component life-cycle API

The components interact with the AMF via an interface that implements a number of callback functions according to SA Forum API. The proper callback functions need to be identified and understood in order to instrument an SA-aware component to detect callbacks to the component from AMF.

The interactions and life-cycle events between AMF and an SA-Aware component has been summarized in Fig. 4-1 [9] [1].

- The SA-Aware component is instantiated by the instantiation script.
- Once instantiated, the `saAmfInitialize_4()` function is called from the component's interface. Two important parameters are passed to this function (italicized portions signify data type): *`SaAmfHandle`* `*amfHandle` and *`SaAmfCallbacksT_4`* `*amfCallbacks`.
    1. `amfHandle`: AMF replies to this function call by returning a handle to the component as a future reference. AMF uses this handle for all future communication with the component [9].

2. `*amfCallbacks`: If not set to NULL, this parameter specifies the callbacks the AMF may invoke. This essentially is a pointer pointing to a structure containing the callback function types and their names [9]. In the example below, if during a component's initialization, `amf_callbacks` is passed to the `saAmfInitialize_4()` AMF will be aware that it can invoke `amf_csi_set_callback`, `amf_csi_remove_callback` and `amf_comp_terminate_callback` functions on the component .

```
SaAmfCallbacksT amf_callbacks = {
 .saAmfCSISetCallback = amf_csi_set_callback;
 .saAmfCSIRemoveCallback = amf_csi_remove_callback;
 .saAmfComponentTerminateCallback = amf_comp_terminate_callback;
}
```

This function must be invoked before invocation of any other AMF API function.

- `saAmfSelectionObjectGet()` function returns the operating system handle associated with the handle returned by the function `saAmfInitialize_4()`. The invoking process can use the operating system handle to detect pending callbacks [9].
- `saAmfComponentNameGet()` function is called from the component which returns the DN of the component to which the invoking process belongs. This function is invoked by the process before its component has been registered with the AMF [9].
- `saAmfComponentRegister()` function registers the component with the AMF. Registering a component informs the AMF that the component is successfully instantiated and is ready to take CSI assignments.

4.3. Component CSI management

There are a number of functions that are used to manage the HA state of components on behalf of the CSIs they support. As mentioned earlier, each of these function names and their respective types are provided to the AMF during a component's initialization using the `*amfCallbacks` parameter while calling the `saAmfInitialize_4()` function in the beginning of the component's life-cycle.

Three callback functions are critical for instrumentation purposes, they are described below:

***SaAmfCSISetCallbackT():** This callback request has three important parameters (italicized portions signify data type).

1. *`SaNameT`* `*compName`: This is a pointer pointing to the name of the component to which a new CSI is to be assigned or for which the HA state of one or all supported CSIs is to be changed [9].

2. *`SaAmfHAStateT`* `haState`: This parameter signifies the new HA state to be assumed by the component identified by the name to which `compName` points for the CSI identified by `csiDescriptor`, or for all CSIs already supported by the component [9].

3. *`SaAmfCSIDescriptorT`* `csiDescriptor`: The descriptor with information about the CSI(s) including the CSI name targeted by this callback invocation [9].

The AMF invokes this callback to request that the component identified by the name to which `compName` points assume the HA state specified by `haState` for one or all CSIs [9].

***SaAmfCSIRemoveCallbackT():** This callback request has two important parameters (italicized portions signify data type).

1. *`SaNameT`* `*compName`: This is a pointer pointing to the name of the component from which all CSIs or the CSI name signified by the `csiName` parameter is to be removed [9].

2. `SaNameT *csiName`: This is a pointer pointing to the name of the CSI that must be removed from the component identified by the name to which `compName` points [9].

With this callback, the AMF requests the invoked process to remove from the component identified by the name referred to by `compName`, one or all CSIs from the set of CSIs being supported [9].

SaAmfComponentTerminateCallbackT(): This callback request has one important parameter (italicized portions signify data type).

1. *`SaNameT`* `*compName`: This is a pointer pointing to the name of the component which is to be terminated [9].

With this callback, AMF requests the component identified by the name referred to by `compName` to terminate.

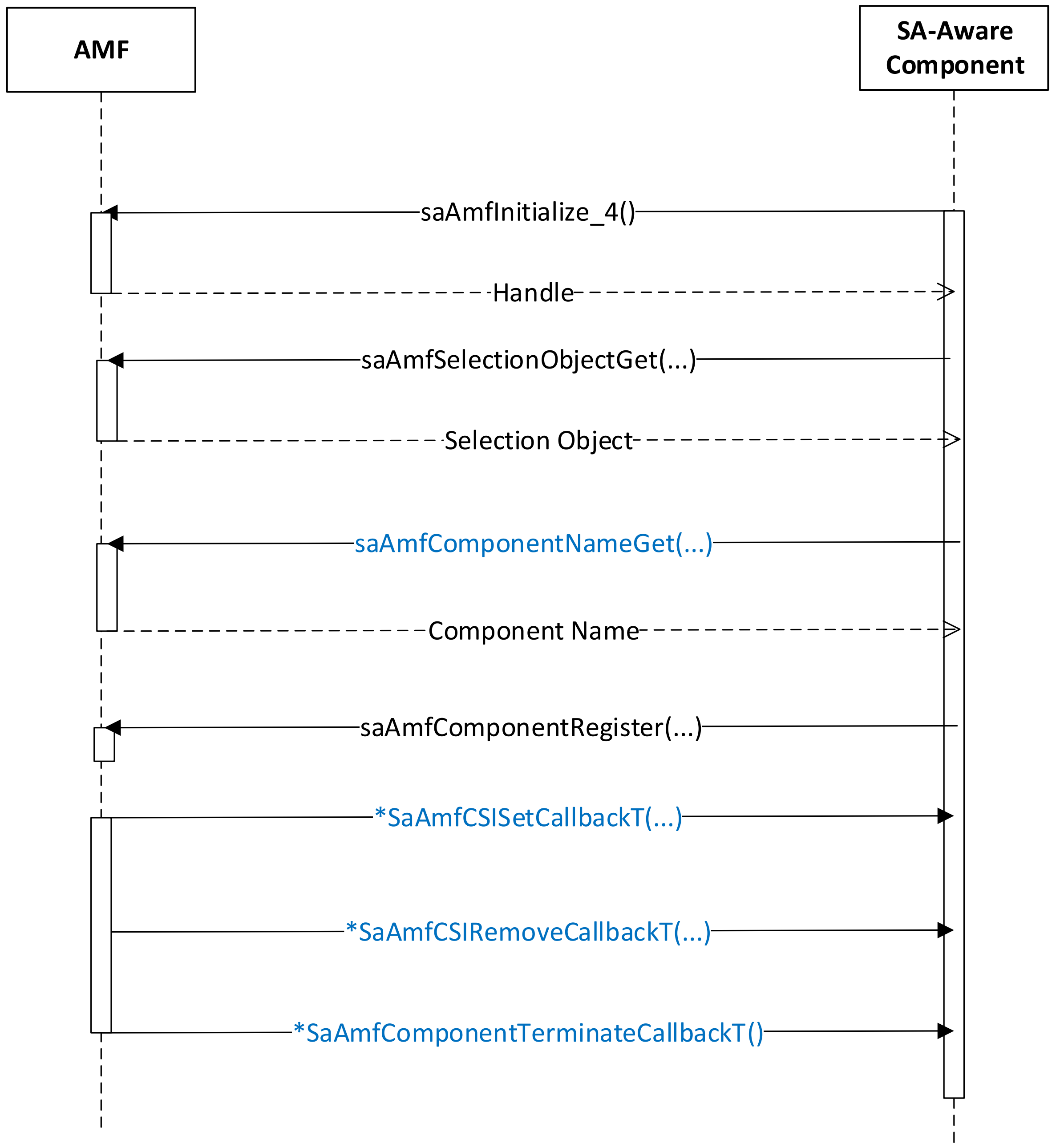

AMF
SA-Aware Component
saAmfInitialize_4()
Handle
saAmfSelectionObjectGet(...)
Selection Object
saAmfComponentNameGet(...)
Component Name
saAmfComponentRegister(...)
*SaAmfCSISetCallbackT(...)
*SaAmfCSIRemoveCallbackT(...)
*SaAmfComponentTerminateCallbackT()

Figure 4-1: The main interactions between an SA-Aware component and AMF

The function `saAmfComponentRegister(...)` is a synchronous. The component expects a response from AMF that indicates successful component registration, which has not been shown in the diagram (Fig. 4-1). Similarly, AMF expects a response for each callback function call which have not been shown. If a component fails to respond to a dispatch call, it is declared to be faulty by the AMF.

The required functions to be instrumented in a component's life-cycle and CSI management with their respective mappings that are to be collected from the traces are summarized in Table 4-1.

Table 4-1: Functions to instrument and the mappings obtained from the instrumentation of each function

Instrumented Function/Function Type	Mapping from Instrumentation
saAmfComponentNameGet()	Instantiated Component DN: Process ID
***SaAmfCSISetCallbackT()**	Component DN: Assigned CSI DN, Component DN: HA State
***SaAmfCSIRemoveCallbackT()**	Component DN: Removed CSI DN
***SaAmfComponentTerminateCallbackT()**	Terminated Component DN: Process ID

4.4 Instrumentation Method

In this section, a method that instruments SA-Aware component interfaces using LTTng UST probes is discussed.

The steps to instrument an SA-Aware component can be summarized in the following steps:

1) In the component source code, determine the mapping of targeted function names to their respective function types.
2) Locate the mapped AMF callback function implementations in the source code.
3) For each function found
    a. Map the AMF callback function parameter names with the names used in the function implementation
    b. Construct and insert the trace points based on the functions' utility in the code.
4) Update the header files of the instrumented source code.
5) Update the library linkers with the trace libraries.
6) Recompile the updated source code.

Architecture and Functions

The instrumentation method takes as input the location of the source code to be instrumented and the instrumentation template. The instrumentation template specifies details of the instrumentation to-be-performed. A template can be reused to instrument all components of the same component-type. The output of the method is the instrumented source code and updated auto-configuration file(s).

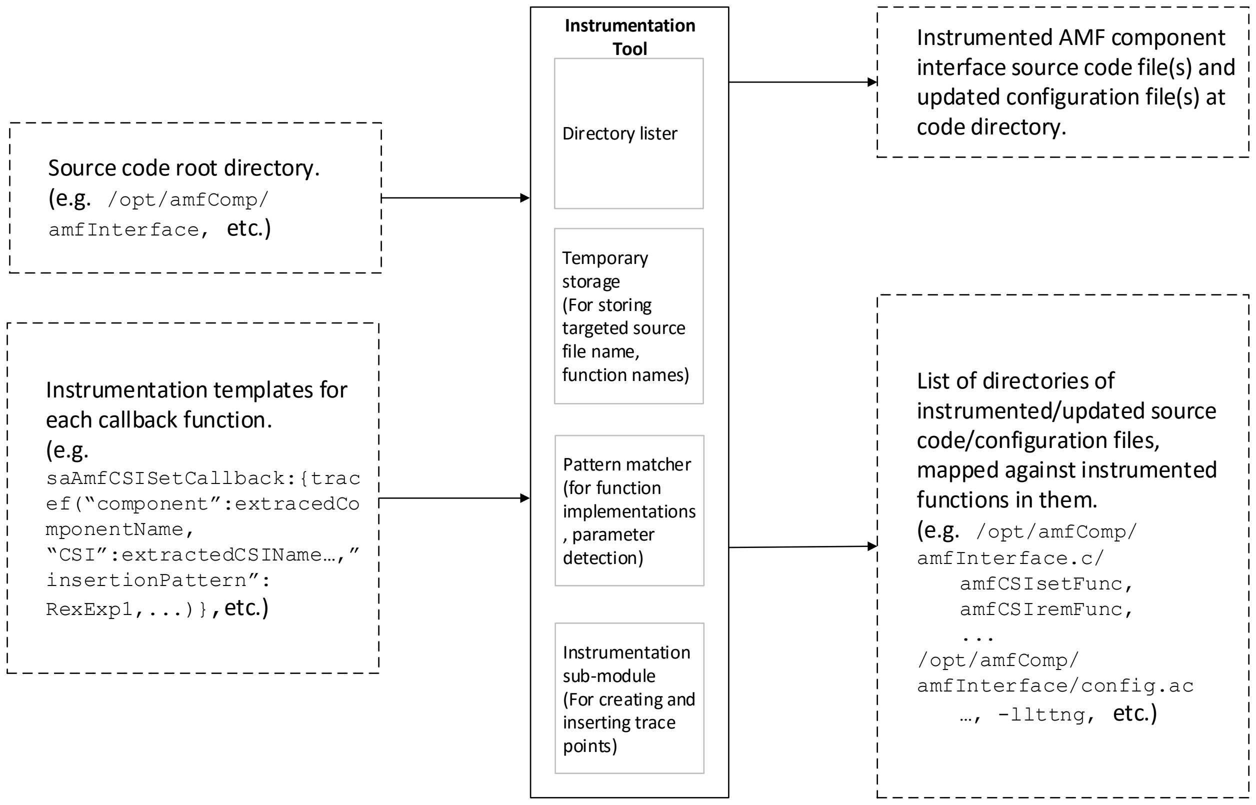

Instrumentation Tool
Directory lister
Temporary storage (For storing targeted source file name, function names)
Pattern matcher (for function implementations , parameter detection)
Instrumentation sub-module (For creating and inserting trace points)
Source code root directory. (e.g. /opt/amfComp/ amfInterface, etc.)
Instrumentation templates for each callback function. (e.g. saAmfCSISetCallback:{tracef("component":extracedComponentName, "CSI":extractedCSIName…," insertionPattern": RexExp1,...)},etc.)
Instrumented AMF component interface source code file(s) and updated configuration file(s) at code directory.
List of directories of instrumented/updated source code/configuration files, mapped against instrumented functions in them.
(e.g. /opt/amfComp/ amfInterface.c/ amfCSIsetFunc, amfCSIremFunc, ... /opt/amfComp/ amfInterface/config.ac …, -llttng, etc.)

Figure 4-2: Overall view of Instrumentation method

As shown in Fig. 4-2 the instrumentation method is divided into multiple modules according to their functionalities which include directory lister, temporary storage, and pattern matcher and trace point creator. Their functions are as follows:

Directory Lister: This sub-module takes a directory location as an input and returns the list of all files within that directory and sub-directories as an output. In the flowchart (Fig. 4-3), this sub-module is used in the steps where the pattern-matcher sub-module searches for the callback functions and also for the linker signatures throughout all the files.

Temporary Storage: This sub-module temporarily stores the input, templates and the resulting mappings for each callback function that are to be instrumented. It is used in each of the steps while instrumenting the callback functions. For example, in the step in the flow-chart (Fig.

4-3) within which the callback function implementations are searched for, the found source code file locations are mapped against each of the callback function names and temporarily saved using this sub-module. Similarly, the instrumentation template from the input with the pattern to identify the function implementation and the pattern used to identify and extract the different parameters for each callback function are mapped against each of them and saved using this module for later use. The module is used similarly to update the header and linker configuration files.

Pattern Matcher: This sub-module matches a pattern provided in the template with some string in the files to be instrumented and extracts the matched sections. It is also used to add, remove or replace parts of source code using the pattern to identify the sections to perform such actions. In Fig. 4-3, this sub-module is used in all the steps that involve pattern matching or adding/replacing code; namely, create trace points for each function, instrument all matched function, update headers and search for linkers and update files with linkers.

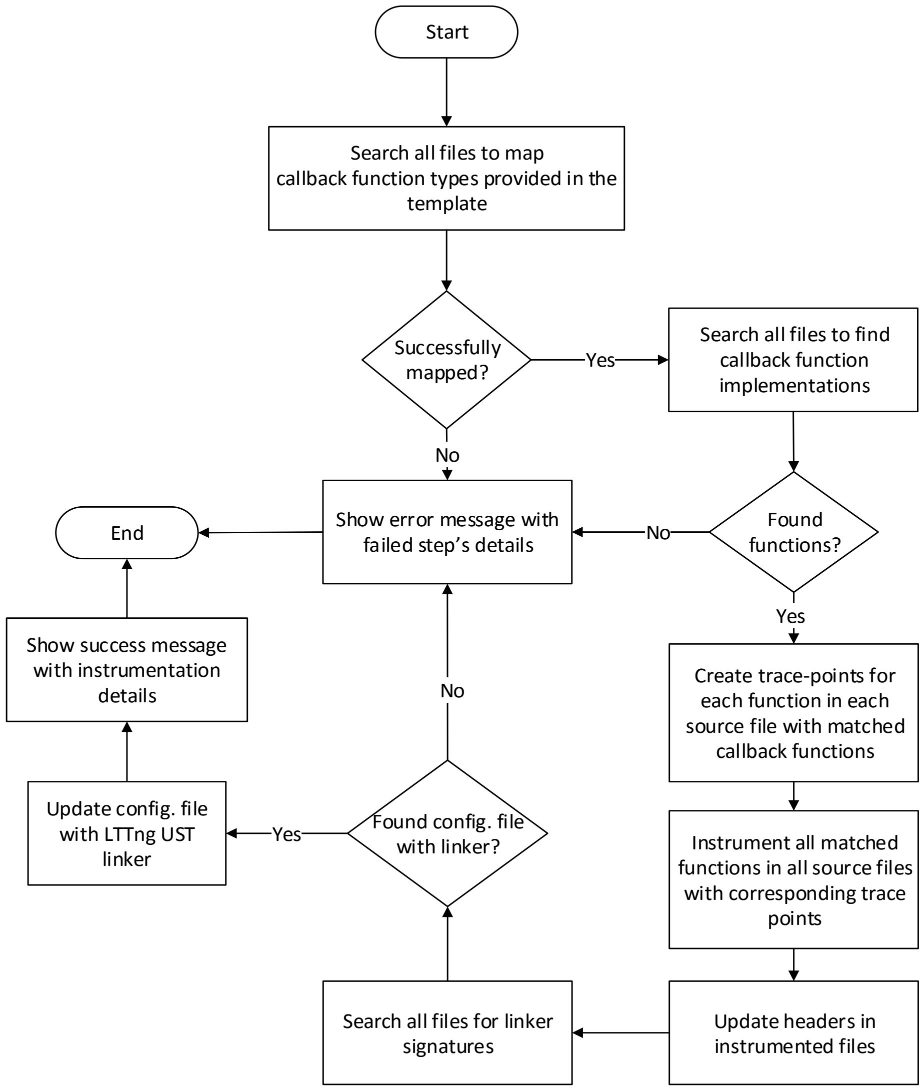
Instrumentation sub-module: This sub-module uses the directory-lister, pattern-matcher and temporary-storage sub-modules to create trace points according to the provided template for each of the callback functions and instruments the source code. This sub-module also updates the headers and linker configuration files.

The instrumentation module takes two inputs:

1) The location of the source code of the AMF component and
2) Instrumentation templates shown (Table 4-2), which include the
    a. Callback function name(s) to instrument
    b. Template(s) of trace-points to be created for the callback function

c. Pattern(s) associated with each of the trace point template to detect the location of the source code to insert their corresponding trace-points.

As output, the instrumentation module generates the instrumented source code and the details about the instrumentation.



Start
Search all files to map callback function types provided in the template
Successfully mapped?
Yes
Search all files to find callback function implementations
No
Found functions?
No
Show error message with failed step's details
End
Yes
Create trace-points for each function in each source file with matched callback functions
Show success message with instrumentation details
No
Instrument all matched functions in all source files with corresponding trace points
Update config. file with LTTng UST linker
Yes
Found config. file with linker?
Search all files for linker signatures
Update headers in instrumented files

Figure 4-3: AMF component interface instrumentation method

The AMF-component-interface instrumentation (shown in the flow chart in Fig. 4-3) starts with the instrumentation of the AMF callback functions.

First, the callback function names provided in the instrumentation template are mapped to the callback function implementation names. According to the AMF specification, the AMF callback function names are declared in a C-struct declaration of the type `SaAmfCallbacksT_4` [9]. To find out the callback function names for a given implementation of the AMF component interface, the instrumentation method first lists all source code files using the directory-lister sub-module and then uses the pattern-matcher sub-module to search through all files to find the function name declarations against the callback function names. If found, the corresponding function names mapped against the callback function names are stored temporarily for future reference using the temporary storage sub-module. For example, a mapping of "`saAmfCSISetCallback`": "`app_CSI_set_callback`" suggests that the name of the function that implements the standard `saAmfCSISetCallback` callback is `app_CSI_set_callback`. If the mapping fails, the instrumentation module shows an error message displaying which callback function name(s) could not be mapped.

If the callback function names are successfully mapped in the previous step, the instrumentation sub-module continues with the step to find the implementations for each of the callback functions. The pattern-matcher sub-module gets the list of all files from the directory-lister sub-module and searches through them to locate each of the callback function implementations. If the implementations are found, the instrumentation module moves onto the step of creating trace-points for each of the callback function implementation. If the implementations are not found, an error message is displayed that shows which callback function implementation was not found.

Once the callback function implementations are found, the instrumentation method needs to prepare the unique trace points based on the associated template for each of the functions found. The template describes the trace point to be inserted in terms of the function parameter defined by the standard function signatures. Hence to construct the trace points first the standard parameter need to be mapped to the parameter used by the function implementation.

Each callback function has a specific order of its parameters, each of which has a type. E.g. the CSI assigned to a component can be identified by accessing the *fourth* pointer parameter passed to a `saAmfCSISetCallback` function type; the component name can be obtained by accessing the *second* pointer parameter passed into the `saAmfComponentTerminateCallback` type function [9], etc.

The instrumentation method detects and maps the callback function parameters based on the order of their declaration in the callback function implementation. Each callback function has a specific order of parameters defined in the SA Forum API and all correct implementation of such callback functions follow this order. The mapping takes into account that some parameters are complex structures only part of which is used in a particular trace point. For instance, in the `saAmfCSISetCallback` function types, the CSI name is an external property, hence the CSI name is extracted from `saAmfCSIDescriptorT` type parameter.

The pattern-matcher sub-module is used to extract the parameters associated with each of the callback functions. Once the instrumentation sub-module extracts and maps the necessary parameters for each callback function they are stored.

Once the parameters have been mapped the instrumentation method proceeds with the instrumentation of the different occurrences of the implementations of the different callback function. The occurrences of the callback functions are listed in the temporary storage sub-module. The instrumentation sub-module generates trace points based on the mapped implementation parameters and the template provided and inserts the trace points using the insertion point detection pattern. Each trace point can have one or more insertion points, depending on where and how many times a user wants to insert the trace point into the source code. The instrumentation process is repeated for each file that contains any of the callback function implementation.

In the next step, the header file entries of each of the instrumented source code files are updated with the necessary C library header file entries (e.g. `stdlib.h, lttng/tracef.h`, etc.)

The instrumented AMF interface source code needs to be linked with the LTTng UST library objects for proper compilation/recompilation. Hence, after successfully instrumenting the callback function, the instrumentation method updates the linker files with the LTTng UST linkers in the same manner. For example, the "`-lSaAmf -lSaCkpt`" pattern will be updated to "`-lSaAmf -lSaCkpt -ldl -llttng`.

If all callback functions have been instrumented and configuration linkers were updated, the instrumentation method shows a success-message alongside the details of the instrumentation procedure. The shown details include the location of the source code files that have been instrumented and the configuration files that have been updated. At this point the AMF component source code has been instrumented and ready to be compiled.

Table 4-2: Instrumentation template sample input and its sample mapping

Input	Example input	Example mapping
Callback function name	saAmfCSISetCallback	app_saAmfCSISetCallback
Callback function parameters	component_name CSI_name HA_state	comp_name csi_desc.csiName ha_state
Trace point pattern	tracef("{'type':'dispatch_set', 'CSI':'%s', 'component':'%s' , 'HAState':'%d"', component_name, CSI_name, HA_state);	tracef("{'type':'dispatch_set', 'CSI':'%s', 'component':'%s' , 'HAState':'%d"', comp_name->value, csi_desc.csiName.value, ha_state);
Insertion point pattern	'\([^)]*\)\s*\n*\s*\{'	app_saAmfCSISetCallback(…){ …

4.5 Summary

In this chapter, we first discussed the component categories, a component's states and the states' related life-cycle events. Based on the component's life cycle, we introduced a method to instrument SA Aware components using LTTng UST probes. As a result of this instrumentation, it is possible to detect at runtime the dynamic assignment and removal of CSIs to the components, which in turn enables the Monitoring Engine to associate the workload of the components to the CSIs assigned to them.

Although the automatic instrumentation method greatly reduces the amount of time required to inspect and instrument the source code of an SA Aware component interfaces, its effectiveness is limited as it can only instrument the source code using one specific sort of trace point probe (E.g. `tracef` probes). The component interfaces need to be instrumented manually if more comprehensive instrumentation is required. Moreover, if the instrumentation fails to locate the mapping for the callback functions' implemented names to their types at any stage as shown in Fig. 4-3, the instrumentation will be incomplete.

5. Service Instance Usage Mapping and Aggregation

In this chapter, the method of aggregating workload-per component to workload-per SI is discussed. As mentioned in the earlier chapters, an Elasticity Engine [10] has been proposed to be integrated with OpenSAF, a middleware compliant with the SA Forum specifications [9]. Since AMF interprets and manages the workload in terms of SIs, the proposed elasticity engine requires as input the load changes in terms of SIs as well.

The proposed Monitoring Engine in this thesis solves the problem of measuring the system usage that is the workload in terms of services, i.e. SIs. The SIs of AMF managed applications are managed dynamically and assigned according to the runtime state of the available SUs and the applicable redundancy model. Mapping system usage measured in a system to the SIs is difficult because of the dynamic nature of the SI distribution. The Monitoring Engine proposed in this thesis adjusts itself according to the SI distribution of the system to measure workload in terms of SIs correctly.

In the previous chapters we discussed the methods of retrieving system resource usage in terms of components and relating the retrieved resource usage of the components to their respective CSI assignments. In this chapter, we address the issue of aggregating the system resource usage of components and expressing it in terms of AMF services (i.e. SIs). The Monitoring Server carries out the task of aggregating workload of the components sent from the Monitoring Clients into SI-workload.

5.1. Aggregation approach overview

The actions taken by the aggregation method that has been described so far can be summarized by the flowchart shown in Fig. 5-1.

The complete steps to get SI workload from component-workload are as follows:

1. De-serialize a newly arrived component-workload-object to get the following: CSI DN, component DN, HA State, node-name, usage hash values from component-workload object.
2. Obtain SI DN from CSI DN
3. If an SI-tree for newly de-serialized component workload object exists, update the SI tree with the usage values. If an SI tree with newly extracted SI DN root node does not exist, create a new SI-tree and populate it with the new node-names and usage values.
4. Check for obsolete components in the SI-tree; if found, delete the obsolete component node.
5. Duplicate SI-tree to aggregate the usage values to workload values. The duplicated tree with aggregated workload value is called workload-tree.
6. Keep adding and normalizing the usage values of the nodes at the last level of the workload tree until the last level is SI. At the end of this step, the workload-trees should have only the SI level with aggregated workload values attached to the root node of the workload-tree.
7. Repeat steps 1 through 5 to generate a workload-tree for each SI-tree generated.
8. Return the workload-trees' root node-names and usage values as key-value pairs to show SI DNs and their respective workload values.

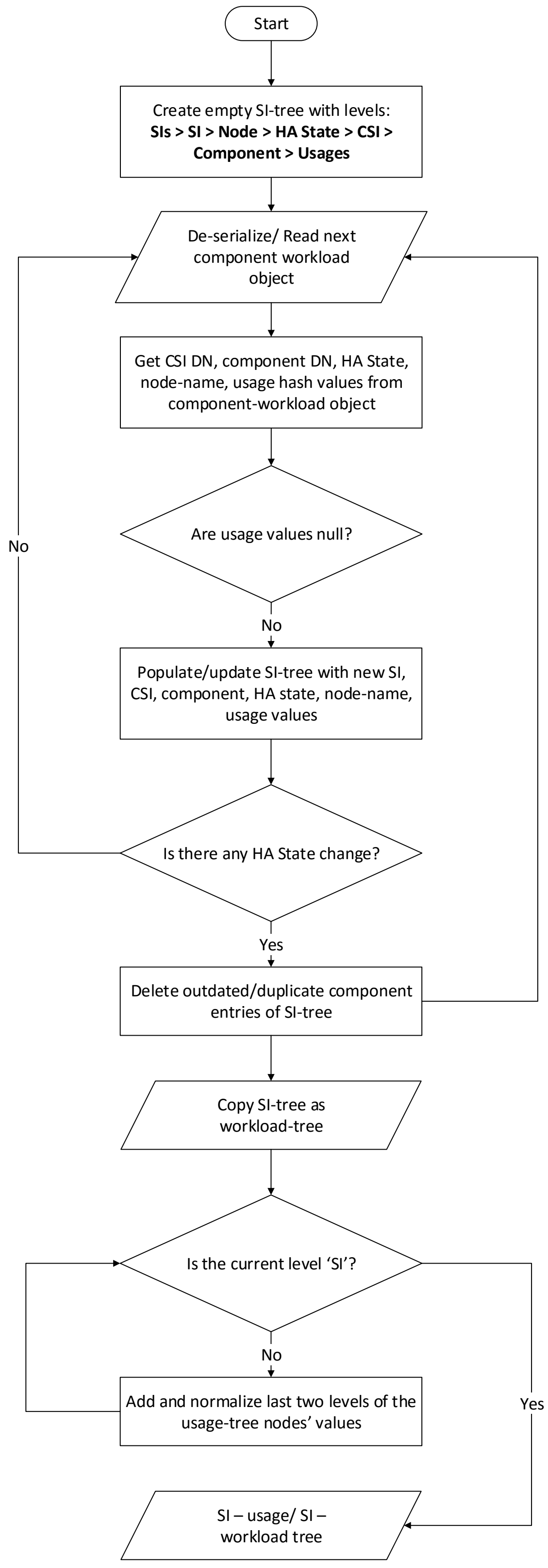

Start
Create empty SI-tree with levels: SIs > SI > Node > HA State > CSI > Component > Usages
De-serialize/ Read next component workload object
Get CSI DN, component DN, HA State, node-name, usage hash values from component-workload object
Are usage values null?
No
Populate/update SI-tree with new SI, CSI, component, HA state, node-name, usage values
Is there any HA State change?
No
Yes
Delete outdated/duplicate component entries of SI-tree
Copy SI-tree as workload-tree
Is the current level 'SI'?
No
Add and normalize last two levels of the usage-tree nodes' values
Yes
SI – usage/ SI – workload tree

Figure 5-1: Overall view of the aggregation approach

5.2. Algorithms and data structures for aggregation

In this section, the main algorithms and the actions of the aggregation method are discussed in further details.

To aggregate component-workload into SI-workload, we create two types of tree data structures.

a) SI-tree and b) Workload tree. SI-trees keep track of all SIs, CSIs and components in the cluster and are updated according to any assignment change. Workload trees are generated by aggregating along SI trees-paths to calculate workload for each SI in the cluster. Both kind of data structures are discussed in details in the following sections.

5.2.1. SI-tree

The SI-tree is structured from the top according to the hierarchy summarized in table 5-1 and is populated dynamically with the data received from the component-workload-objects sent from the Monitoring Clients.

As component-workload-objects arrive from the Monitoring Clients on fixed intervals, the SI-tree is populated by putting the data extracted from component-workload-objects in appropriate paths of the SI-trees.

The data structure used to create SI-trees has four main parts/features: a) node b) level c) value and d) path.

Each tree has one or more **node**(s) in it (not to be confused with a VM node). The topmost node is called the root node. Each node belongs to a **level**, each level has a unique name. Each node has a name which must be unique within that node’s level. There must be a valid **path** from each node to the root node. A path is the chain of node-names from a given node to the root node. For example, in Fig. 5-2 (a), the path from the node with name '*C_1,SU_1,SG_1,app_1*' to the root node would be as follows: ['*C_1,SU_1,SG_1,app_1*', '*CSI_1,SI_1, app_1*', '*Active*', '*Host_1*', '*SI_1,app_1*']. A node may or may not have **value(s)** attached to it. The value of a node is not the same entity as the name of the node. Setting value to a path means assigning value to the node at the end of the path. For example, in Fig. 5-2 (a), only the node at path ['*C_1,SU_1,SG_1,app_1*', '*CSI_1,SI_1, app_1*', '*Active*', '*Host_1*', '*SI_1,app_1*'] has values attached to it. In this work, two kinds of values have been used to quantify resource usage:

a) Relative values: This is the percentage of any resource a component-process uses with respect to a VM node's total resource. This kind of workload values are normalized while aggregating across VMs. Example: memory usage of a process in percentage.
b) Absolute values: This is the real value of the resource usage by a component-process. The sum of this kind of values are taken while aggregating across VMs. Example: memory usage of a process in Mega Bytes (MB).

The value types have been discussed in further details in Chapter 6 of this thesis.

Table 5-1: Names and short descriptions of the levels in an SI-tree

Level	Description
SI	DN of the SI of the current SI-tree.
Node	Component-workload-object source host-name. If a workload object is sent from a Monitoring Client hosted on a node with a hostname ‘node-1’, this level will be populated with ‘node-1’.
HA state	HA state assigned to the component by the AMF on behalf of the component’s CSI [9]. E.g. active, standby, quiescing and quiesced.
CSI	DN(s) of the CSI(s) assigned to the component(s)
Component	DN(s) of the components
Process ID	Process ID(s) of the component-processes

In Fig. 5-2, both the trees have six levels. Each level’s name is shown on the left side. Each connection from one node to the node(s) below its level signifies a parent-child relationship. A dotted line signifies node-value relationship. In the trees shown in Fig. 5-2, only the nodes at the component-PID have values attached to them.

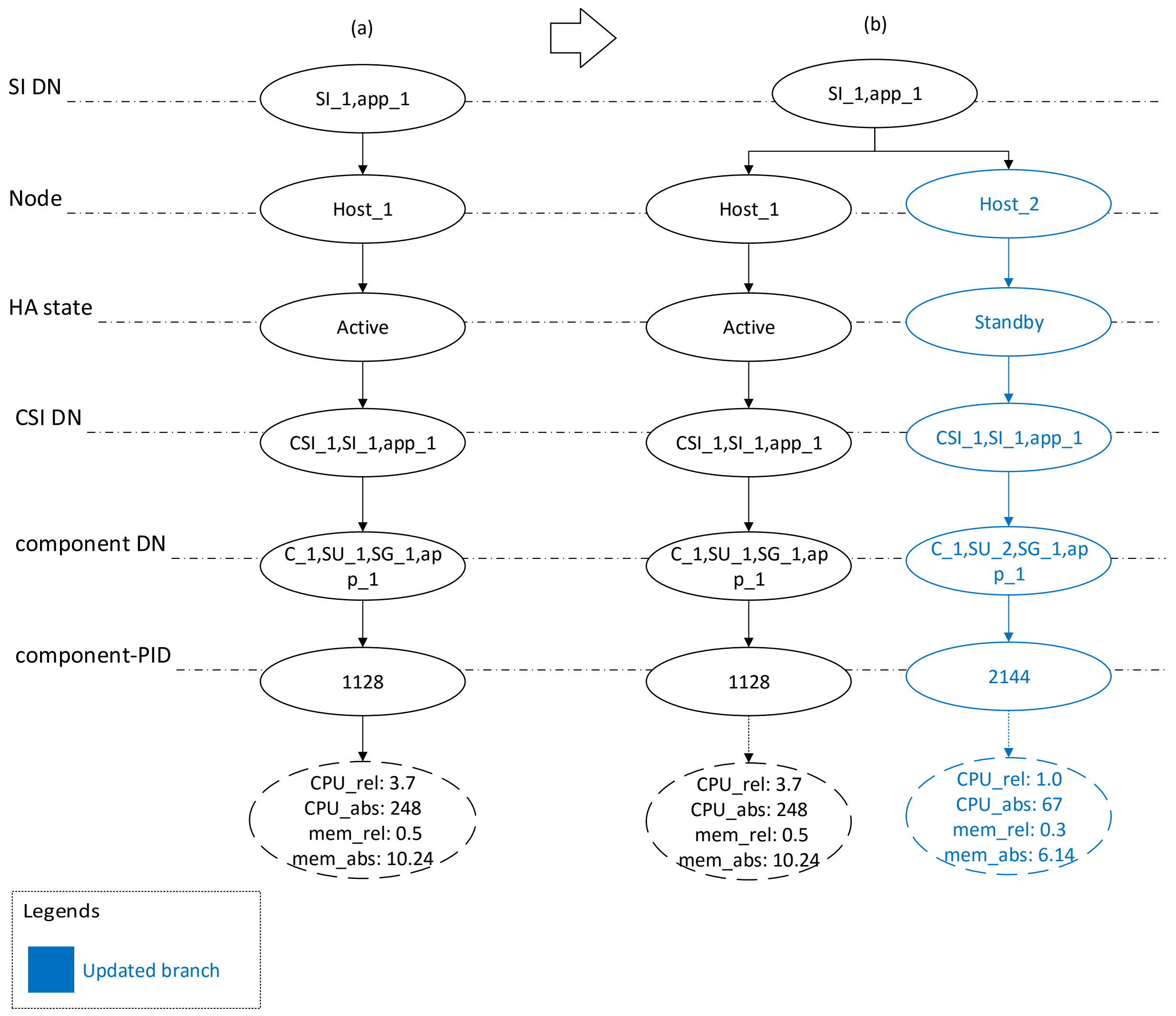


Figure 5-2: Populating an SI-tree

In this sub-section, we also discuss the algorithm used to create and populate an SI tree based on the component workload objects received from the Monitoring Clients.

The aggregation method consists of algorithm 5-1 which populates/updates SI-trees:

```
CreateSITree: SI-tree creation/population
Input:
        $C_{WL}$: component workload object
        SIs: tree comprising of SI-trees
Output:
        SIs: updated SI-trees
if $C_{WL} \neq \emptyset$ then
    #retrieving data from the component workload object to populate/create
    for each component $C \in C_{WL}$ do
        $C_{DN} \leftarrow$ getValueFromObject($C_{WL}$, component)
        $CSI_{DN} \leftarrow$ getValueFromObject($C_{WL}$, CSI)
        HAstate $\leftarrow$ getValueFromObject($C_{WL}$, HA)
        H $\leftarrow$ getValueFromObject($C_{WL}$, hostname)
        $C_{PID} \leftarrow$ getValueFromObject($C_{WL}$, PID)
        $C_{PID, usg} \leftarrow$ getValueFromObject($C_{WL}$, usages)
        $SI_{DN} \leftarrow$ splitDNwithDelimeter($CSI_{DN}$, 'safSI=')
        #Updating the appropriate SI node of the 'SIs' tree
        if $C_{PID, usg} \neq \emptyset$ then
            setValueAtPath(SIs , $C_{PID, usg}$ , [$SI_{DN}$, H, HAstate, $CSI_{DN}$, $C_{DN}$, $C_{PID}$])
        else
        #Deleting failed component entries since workload value is null
            deletePath([SIs, $SI_{DN}$, H, HAS, $CSI_{DN}$, $C_{DN}$])
        end if
        #Checking and deleting duplicate components under same CSI entries in different HA levels
        for each HAstate HAS $\in$ getFromPath(SIs, [SI,node]) do
            if HAS $\neq$ HAState and $CSI_{DN} \in$ getNodesFromLevel(CSI, [SIs, $SI_{DN}$]) and $C_{DN} \in$
        getNodesFromLevel(Component, [SIs, $SI_{DN}$]) then
                deletePath([SIs, $SI_{DN}$, H, HAS, $CSI_{DN}$, $C_{DN}$])
            end if
        end for
    end for
end if
```

Algorithm 5-1: Creating and populating SI-trees

According to the aggregation method, the component DN, CSI DN, component HA state, the source host name, the component PID(s) and their usages are fetched from a new component-workload-object to update existing SI trees or create a new SI tree. Note, that the SI DN is retrieved based on the CSI DN of the component by splitting the CSI DN by a delimiter (I.e. to get the SI

DN from a CSI DN, the CSI DN is split based on the delimiter ',safSi='). Once all necessary values are fetched, the 'SIs' tree is updated according to the values fetched from the workload object. The 'path' to update any value depends on the data fetched from the component workload object. The aggregation method creates an SI-tree for each SI configured in the cluster at any given time and populates a larger tree named 'SIs' with the SI-trees created.

Fig. 5-2 illustrates the process of populating an SI-tree. On the left side (a), the entries of an existing SI-tree is shown which is being updated based on the values extracted from a new component-workload-object. The tree on the right side (b) shows the values in the same tree after the update is completed. In the new branch of the tree, the node name at the 'node' level differed from any previously existing value at that level, which prompted the creation of a new branch in the tree at that level. If the data fetched from a component-workload-object refers to an existing path, the values of that path are over-written. For example, in Fig. 5-2, to create the initial tree, the aggregation method first constructs a single path based on component DN, CSI DN, component HA state, VM hostname and SI DN that are extracted from a component-workload-object originated from *Host_1*. This results in the initial tree as shown on side (a). Similarly, with the arrival of component-workload-objects from *Host-2*, a path is constructed that differs from the existing path for the same SI at the 'Node' level (*Host_1* vs. *Host_2*). Therefore a branch is added to the existing SI tree on that level to append the standby component workload at the end of the path. This results in the updated tree as shown in Fig. 5-2 (b).

The aggregation method first checks for duplicated CSI entries in a newly updated SI-tree after each time it updates the SI-tree using algorithm 5-1 and then it corrects the SI-tree by deleting the detected duplicate entries. In the example illustrated in Fig. 5-2, if the component '*C_1,SU_1,SG_1,app_1*' fails and the system is configured to fail-over the SI - '*SI_1,app_1*' for a component failure, the process ID mapped against the failed component will be dead in the VM hosting it and a standby component will be assigned '*Active*' HA state. Reacting to the component's failure, Monitoring Client will be sending a 'null' object as the usage of the failed component to the Monitoring Server until the failed component gets repaired. The 'null' value sent as the resource usage of a component notifies the Monitoring Server that the component-process no longer exists. Once the aggregation method detects a null object as usage of a component, it removes that component and its associated entries from the SI-tree. For example, in Fig. 5-2 (b), if the component with the '*Active'* assignment (component-PID 1128) fails, the aggregation method will receive a 'null' value to attach at the end of the path ['*C_1,SU_1,SG_1,app_1*', '*CSI_1,SI_1, app_1*', '*Active*', '*Host_1*', '*SI_1,app_1*'] from the newly arrived component-workload-objects originated from '*Host_1'* instead of component workload values. This prompts the aggregation method to delete the branch defined by this path.When the standby component gets the '*Active*' assignment, the aggregation method receives new component-workload-objects with the recent changes, and creates a new path in the SI tree as shown in Fig. 5-3. The SI-tree on the left hand side of Fig. 5-3 has an obsolete branch that shows the component '*C_1,SU_1,SG_1,app_1*' was in '*Standby*' HA state previously. This obsolete branch is eventually deleted from the SI tree by checking for duplicate component entries under different HA levels of any SI tree as shown in algorithm 5-1. In the right side of Fig 5-2, the corrected SI tree is shown. Without this check, the aggregation method would be unaware of the duplicate component branches under two different HA states of the same SI tree, which would result in incorrect workload measurement for the affected SI.

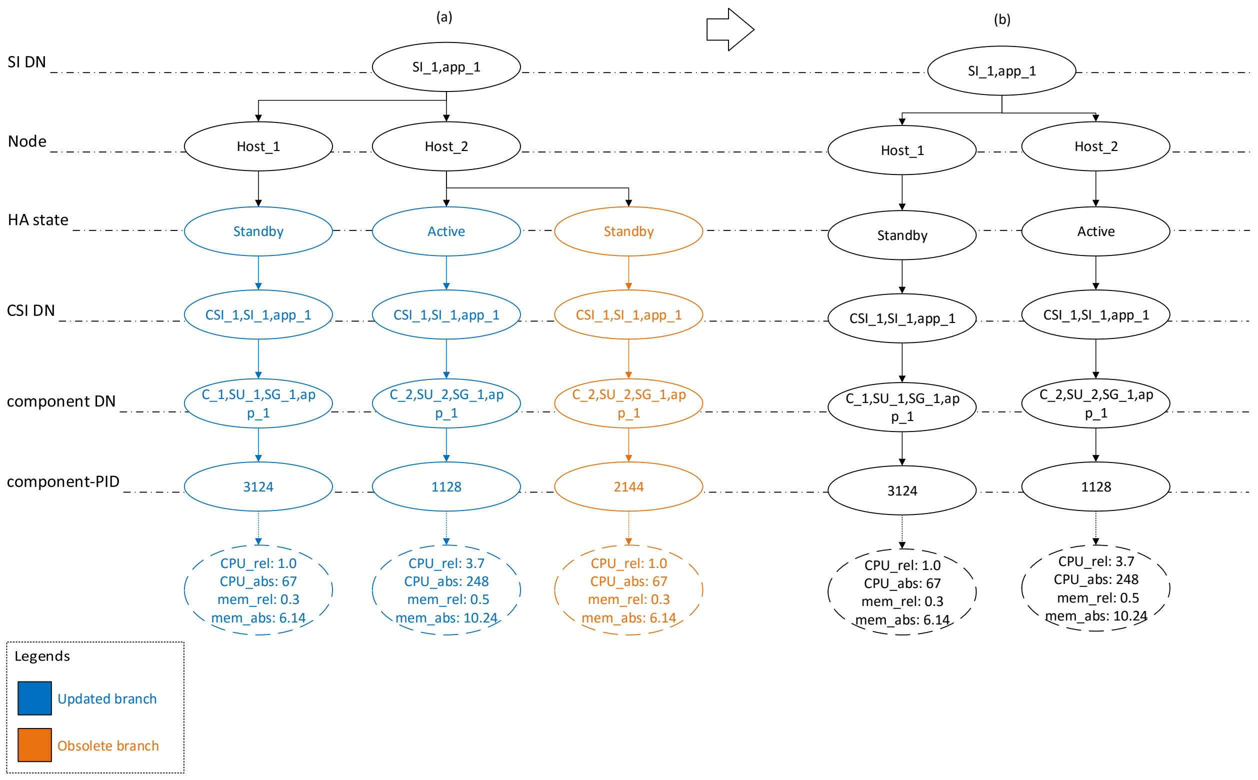


Figure 5-3: Obsolete branch deletion in SI-tree after a fail-over

5.2.2. Workload-tree

In order to aggregate component-workload to SI workload, the workload at the leaves of the existing SI-trees need to be aggregated along the tree-paths so that the nodes at the SI level have aggregated usage values attached to them. The aggregation method uses algorithm 5-2 and algorithm 5-3 to aggregate each SI-tree to a workload-tree. In the workload tree, the aggregated workload values are stored at the SI level as shown in Fig. 5-4.

In algorithm 5-2, the aggregation method first duplicates a given SI tree to aggregate resource usage/workload values of the components to the tree's root SI. The duplicated tree is called a workload-tree.

In the workload tree, except for the ‘Node’ level, the summed workload value of all nodes at each level is assigned to the parent nodes. In case of the ‘Node’ level, for ‘Relative’ workload values, the *mean* workload value of all nodes at that level is assigned to the nodes at their parent level (SI nodes). For ‘Absolute’ workload values, the *sum* of the workload values for all nodes at the ‘Node’ level is assigned to the nodes at their parent level.

This way, we get one workload tree for each HA state. In order to obtain the final workload tree, all the workload trees for each HA state of an SI is aggregated again using algorithm 5-3.

In algorithm 5-3, the workload for each SI under different HA states is summed to provide the total workload of that SI as shown in Fig. 5-4. Algorithm 5-2 provides the workload of each SI for each of their HA states (The transition shown in Fig. 5-4-a to Fig. 5-4-b). Algorithm 5-3 provides the total workload of each SI based on the output from algorithm 5-2 (The transition shown in Fig. 5-4-b to Fig. 5-4-c).

Once the SI-level is the only level in the final workload tree, the workload tree manipulation is complete. Each workload tree has only one node with an SI DN as its name and that SI’s aggregated workload as its value.

```
updateHAstateWorkloadTree: Workload-tree creation/aggregation for a given HA state
Input:
        $T_i$: tree to be aggregated/reduced
        HAS: HA State based on which tree is to be aggregated
Output:
        $W_{t\_HA}$ : workload tree

$W_{t\_HA} \leftarrow duplicateTree(T_i)$
$L \leftarrow getLastLevel(W_{t\_HA})$
while $L \neq SI$ do
    $P \leftarrow \emptyset$
    $totalValue \leftarrow 0$
    $VMCount \leftarrow 0$
    #set mean usage values at all children nodes of level L to the nodes at level L
    for each node $N \in getNodesFromLevel(L, [T_i])$ do
    #trimming the nodes not belonging to the HA state provided by the variable 'HAS'
        if $L =$'HA_State' and $getNameOfNode(N) \neq HAS$ then
            $deleteNode(N)$
            continue
        end if
        $P \leftarrow getPathToRootFromNode(getParentNode(N))$
        $totalValue \leftarrow totalValue + getValueOfNode(N)$
        #usages are averaged only at the node level, otherwise the sum of usage values are taken
        if $L =$'Node' then
            $VMCount \leftarrow VMCount + 1$
        end if
    end for
    if $P \neq \emptyset$ then
        if $VMCount > 0$ and $type(totalValue) =$ 'Relative' then
            $setValueAtPath(W_{t\_HA}, totalValue/VMCount, P)$
        else
            $setValueAtPath(W_{t\_HA}, totalValue, P)$
    end if
    $deleteLevel(W_{t\_HA}, L)$
    $L \leftarrow getLastLevel(W_{t\_HA})$
end do
```

Algorithm 5-2: Aggregating SI-tree to workload tree for one HA state

```
updateWorkloadTree: Workload-tree creation/aggregation
Input:
        $T_i$: SI-tree to be aggregated/reduced
Output:
        $W_t$ : workload tree
#Final workload tree
$W_t \leftarrow$ createEmptyTree()
#list of values from HA-state trees for each SI
$V_L \leftarrow$ []
#list of possible HA-states with workload values
$HAS_L \leftarrow$ ['Active', 'Standby', 'Quiescing']
for each state in $HAS_L$ do
    $W_{t\_HA} \leftarrow$ updateHAstateWorkloadTree($T_i$, state)
    for each node in $W_{t\_HA}$ do
        appendValueToList($V_L$, getValueFromNode(node))
    end for
end for
#Assigning sum of workload values from of HA state to root node of the workload tree.
setValueAtPath($W_t$, sumOfList($V_L$), getRootNode($W_t$))
```

Algorithm 5-3: Aggregating SI-tree to workload tree

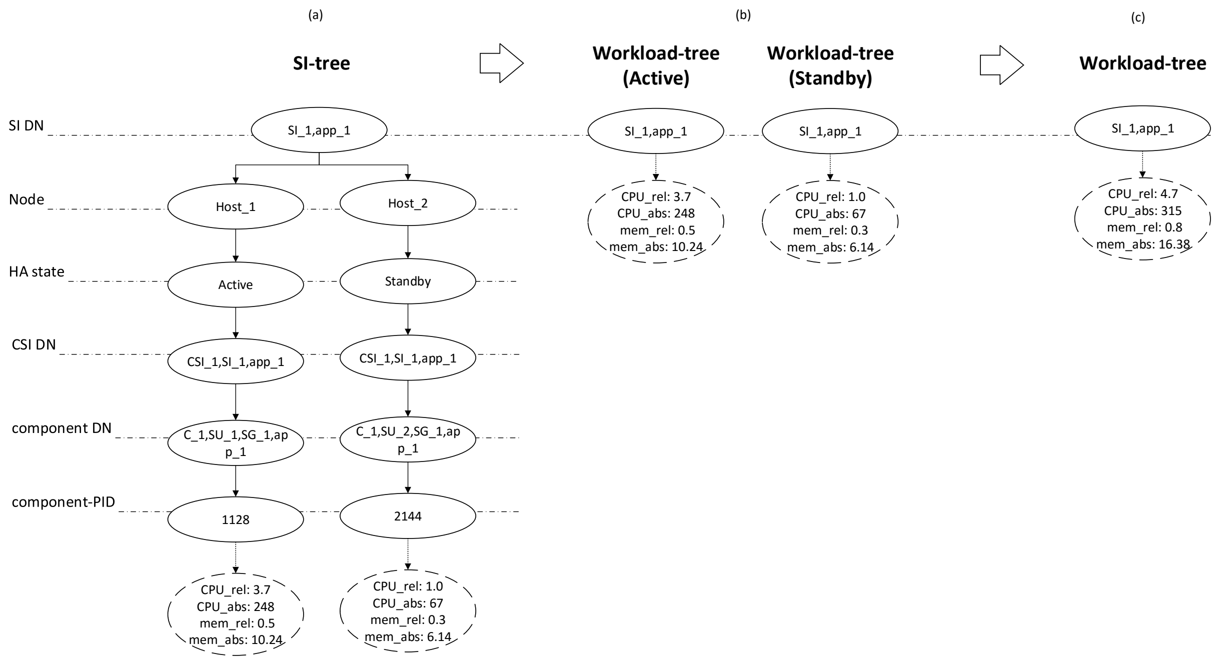


Figure 5-4: Aggregated workload-tree from an SI-tree

5.3. Summary

In this chapter, we first described the overall process of calculating SI workload from the component-workload-objects sent from the Monitoring Clients. Then we discussed in details how the Monitoring Server performs the aggregation of workload by using two sets of tree data structures. In the detailed discussion, we first described the algorithm to construct/update an SI-tree based on the data retrieved from each workload object. Then we discussed the method of constructing/updating a workload-tree from by aggregating each SI-tree along its existing path. The node-value mapping of the workload-trees is the desired output from the aggregation approach.

6. Monitoring Prototype and Overhead Evaluation

A Monitoring Engine has been implemented as a proof of concepts. In this chapter, first, the architecture of the Monitoring Engine prototype and the test beds for testing the Monitoring Engine are discussed. In the test bed discussion, the two different services configured in the system are then discussed followed by a discussion on the integration of the Monitoring Engine prototype with the Elasticity Engine prototype. The subsequent section includes different test cases discussing the Monitoring Engine's ability to trigger under-provisioning/overprovisioning elasticity alerts, its adaptation to service state changes such as SI fail-over and SI switch over. The Monitoring Engine's overhead is discussed before concluding this chapter.

6.1. Prototype architecture

The Monitoring Engine prototype has been developed using the python programming language [37]. During implementation, different functionalities of the Monitoring Engine have been implemented as different modules. In Fig. 6-1, the important modules of the Monitoring Engine and their interactions with other entities in the system are shown. The prototype consists of seven main modules.

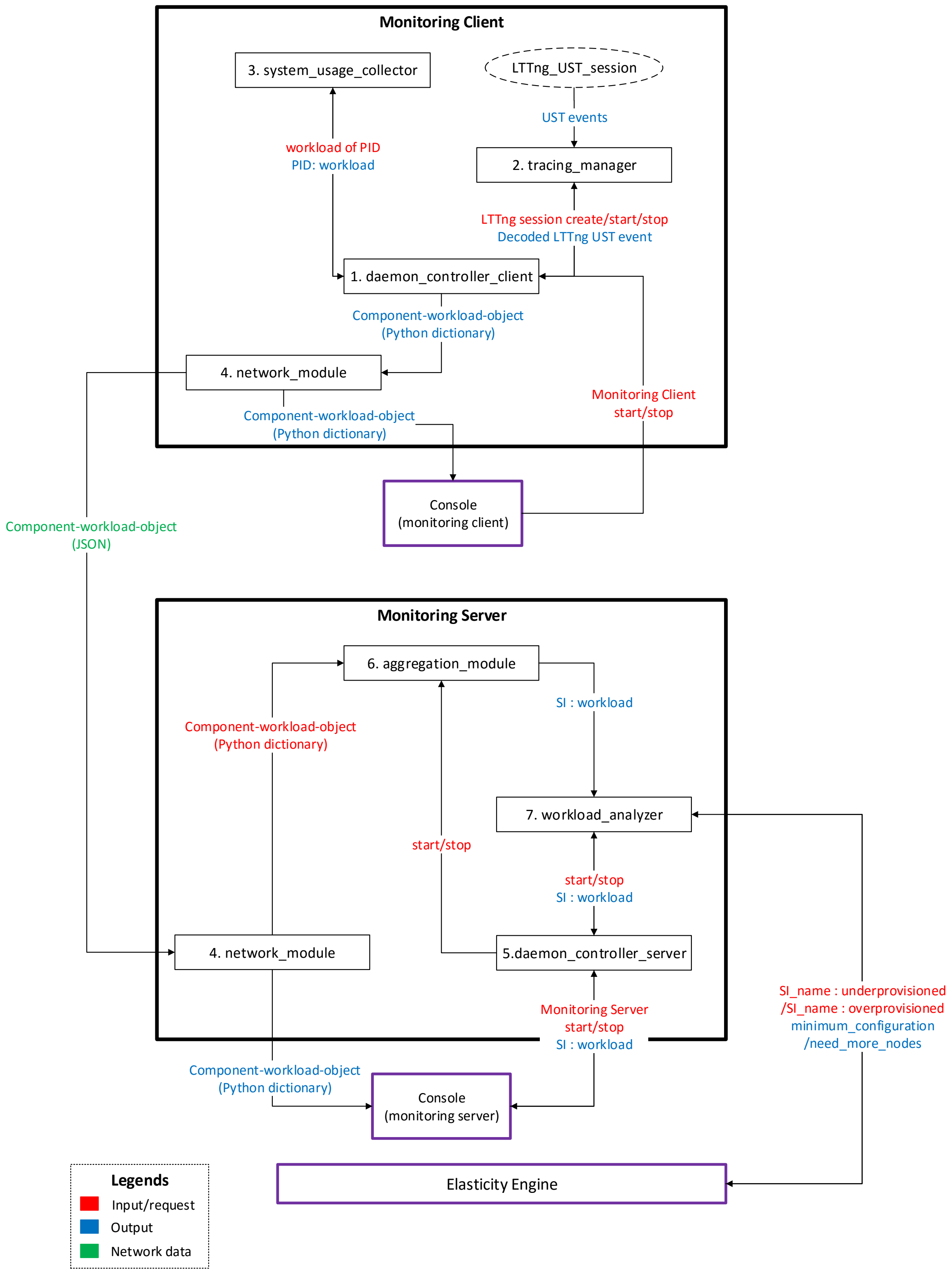


Figure 6-1: Monitoring prototype architecture

1. **daemon_controller_client**

This module controls all other modules of the Monitoring Client. A cloud administrator initializes the Monitoring Client Daemon controller specifying the IP address of the Monitoring Server passed as a parameter. Once started, this module runs as a daemon, starts to send component-workload-objects to the Monitoring Server and does not stop running unless it is specifically instructed to stop or encounters an error. In case of an error at the Monitoring Client Daemon, it stops sending component-workload-objects to the Monitoring Server. It is detected at the Monitoring Server by noting the absence of new component-workload-objects from the VM where the Monitoring Client Daemon has encountered the error or the connectivity from the node hosting Monitoring Client Daemon and Monitoring Server Daemon has been lost.

2. **tracing_manager**

This module is used by the Monitoring Client daemon module (daemon_controller_client) to initialize an LTTng UST [24] session if there isn't one already running. Once a tracing session is initialized, daemon_controller_client uses the tracing_manager module in runtime to check for new UST events. If there is a new LTTng UST event, this module decodes the event from LTTng Common Trace Format (CTF) [24] [38] to a python dictionary data structure [39] and returns it to daemon_controller_client. In essence, tracing_manager is responsible for detecting dispatched callbacks from AMF to all the components in a system.

3. **system_usage_collector**

The daemon_controller_client module uses this module to collect the resource usage of all the processes related to the components present in a system. This module takes a nested python dictionary consisting of all detected components and the process IDs associated with them as input and returns the dictionary updated with the resource usages mapped against each process. The daemon_controller_client module uses this module periodically to gather resource usages of the components. This module uses python PSUtil tool [36] to measure resource usage of the processes. E.g.: a sample component-workload-object sent from Monitoring Client to Monitoring Server:

```
{
    'nstime': 1413934753293198449,
    'msg': '',
    'time': 'Tue Oct 21: 19: 39: 102014 to Tue Oct 21: 19: 39: 132014',
    'cpu_core_usages': [
        28.82,
        13.15
    ],
    'component_info': {
        'safComp=AmfDemo_44,safSu=SC-1,safSg=AmfDemo,safApp=AmfDemo1': {
            'PID': 26750,
            'cpu_usage': 4.4,
‘mem_usage’:1.3,
'cpu_cycles_abs: 71.03,
‘mem_usage_abs’:26.6,
'CSI': 'safCsi=AmfDemo_44,safSi=AmfDemo,safApp=AmfDemo1',
            'HAState': 'Active',
            'CSIFlags': 'AddOne',
            'type': 'csi_assignment'
        },
    },
    'from': 'node1'
}
```

The components and their associated data are mapped as nested key-value maps within the ‘component_info’ key in the component-workload-object as shown. The usage metrics collected are discussed in further details in section 6.2.

4. network_module

This module is used by both Monitoring Client daemon and Monitoring Server daemon to communicate over the network. This takes python dictionaries and a destination IP-port pair as input when it acts under the Monitoring Client daemon. If it is successful to establish a connection to the provided IP-port pair, it serializes the provided python dictionaries into a JSON objects and sends them to their destination over TCP.

This module takes JSON objects and an IP-port pair as inputs when it acts as a part of a Monitoring Server daemon. At the receiving end, it listens to a given IP-port pair; if it receives any data, it de-serializes the data received into python dictionaries. In ‘debug’ mode, this module also displays the data transmitted/received in the console.

The component-workload-objects discussed in the previous sections are essentially nested JSON objects with components and their associated process IDs’ workload mapped to them.

5. daemon_controller_server

This module controls all other modules of the Monitoring Server. Like the Monitoring Client Daemon, a cloud administrator initializes the Monitoring Server Daemon controller (daemon_controller_server) specifying the IP address to listen for component-workload-objects, passed as a parameter. Once started, this module runs as a daemon, and initializes the aggregation_module and workload_analyzer modules.

6. aggregation_module

The Aggregation Module (aggregation_module) receives component-workload-objects from the network_module in the Monitoring Server and aggregates them into simple python dictionaries consisting of SI DNs and their respective workloads mapped as key-value pairs. E.g. a sample SI-workload object:

```
{
    'safSi=AmfDemo,safApp=AmfDemo1': {
    'cpu_usage': 4.4,
    'cpu_cycles_abs: 71.03,
    'mem_usage':1.3,
    'mem_usage_abs':26.6
},
    'safSi=AmfDemo_1,safApp=AmfDemo1': {
    'cpu_usage': 3.1,
    'cpu_cycles_abs':59.23,
    'mem_usage':0.8,
    'mem_usage_abs':16.4
    },
}
```

7. workload_analyzer

Note that this is a simple module developed for the purpose of showing the effectiveness of the prototype developed in this work as a proof of concept and is not the main contribution of this work.

The workload_analyzer module receives SI-workload-objects from aggregation_module and determines if any elasticity action is necessary. It detects the required elasticity action by comparing the moving-average of the workload of each of the SIs against a set of simple rules set by the administrator.

E.g. A sample rule in workload analyzer:

if

a) the average workload of SI_1 (SI name) in last 10 seconds (rolling average data point length) exceeds 70% (upper threshold) usage **and**

b) no alert has been triggered in the last *60 (cool-down period)* seconds **and**

c) elasticity_engine output from last 'underprovisioned alert' did not show 'add more nodes'

or

nodes have been added to the cluster since the elasticity_engine returned 'add more nodes'

then

dispatch trigger underprovisioned alert for SI_1(trigger for scaling operation)

The italicized portions in the rule above are set/updated by the administrator for each rule. A number of such rules are set in the workload analyzer to ensure elasticity in the cluster.

6.2. Workload metrics

In this prototype, the following four workload metrics are collected from the nodes through the Monitoring Client.

6.2.1. Normalized CPU usage

The CPU usage of each component is measured in percentage using the 'system_usage_collector' module of the Monitoring Client. The relative CPU usage of each component is measured in percentage in each VM. This measured usage is then sent to the

Monitoring Server, which aggregates and normalizes to express the relative CPU usage of the components in terms of SIs. The normalized CPU usage shows the relative CPU usage of each SI with respect to the capacity of the cluster in percentage. The normalized relative CPU usage for SIs is measured by expressing the ratio of the aggregated CPU cycles being used by the SI to total CPU cycles available for the SI in percentage. This way, the measurement does not discriminate between homogeneous and heterogeneous systems.

6.2.2. Total CPU cycle usage

The normalized CPU load of SIs expresses a relative load, which changes every time SI assignments are changed. The value of normalized CPU load can show a change in workload even if the total workload on an SI remains the same. Hence, another measure of CPU workload is required in order to understand the true workload of the SIs in terms of CPU usage.

In a heterogeneous system, different VMs have different CPU capabilities. The CPU usage of a component in one VM is not necessarily equivalent to the CPU usage of a similar component in another VM. Therefore, simple summation of the CPU usages of the components in a cluster would reflect an incorrect total CPU usage. Getting a precise measure of CPU usage of any application for a given processor is specific to the application and the VM [40]. The performance of a processor mainly depends on three characteristics of it: a) workload execution speed. I.e. CPU cycles per second b) pipeline effects. I.e. Threads per CPU core and c) memory hierarchy. I.e. CPU cache memory size and speed [40]. Keeping up with the performance of different components on these varying processors in runtime is a complicated and time consuming task. In order to get a quick and simple estimate of CPU performance for a given component, the CPU cycles used to execute the instructions of a given component is measured using Python PSUtil [36] tool in fixed intervals. The sum of CPU cycle usage of all components in the SUs of an SI is considered as the total CPU usage of that SI. It also allows the workload_analyzer module to measure the potential number of instances required to meet a certain amount of workload. In this prototype, the CPU cycles are measured in MHz units.

I.e.: CPU-cycle-usage = $\frac{\text{number of CPU cycles used per second}}{10\char`^6}$ MHz

6.2.3. Normalized memory usage

This is a similar measurement to normalized CPU usage. The relative memory usage of each component is measured in each VM, which is normalized and aggregated to express memory usage of each SI in percentage. The normalized memory usage of an SI shows the memory usage of it with respect to the cluster's capacity.

6.2.4. Total memory usage

The real value of memory usage for each component is measured in the VMs where they are hosted. For each SI, the sum of the memory usage of all components under that SI is considered to be the total memory usage of the SI. The memory usage is measured in mega-bytes (MB) in this prototype.

6.3. Test-beds and test cases

The prototype has been tested with two HA applications deployed on the OpenSAF middleware [11]. OpenSAF is an open source implementation of several SA Forum specifications. It has been installed and configured on each node of the clusters prepared for each of the test cases discussed in the subsequent sections.

VMware Workstation [41] has been used as a hypervisor in a system summarized in table 6-1. The clusters used in each of the test beds discussed in the subsequent sections has a homogeneous setup. I.e. The VMs in the clusters managed by VMWare Workstation have identical specification. The system specification of the VMs in the clusters is also summarized in table 6-1.

Table 6-1: Hardware specifications of the system running the hypervisor and the VMs

Node-type	Hardware Specifications
Hypervisor	**CPU**: Intel(R) Core(TM) i7-2600 CPU @ 3.40GHz
	Memory: 16GB
	OS: Ubuntu 14.04.2 LTS
VM	**CPU:** Intel(R) Core(TM) i7-2600 CPU @ 3.40GHz (Virtual, 2 cores)
	OS: Ubuntu 14.04.2 LTS
	Memory: 2GB

6.3.1. HTTP service

In this setup, the effectiveness of the Monitoring Engine is tested by dispatching Elasticity Engine [10] Triggers based on an HTTP SI-workload measured by the Monitoring Engine.

6.3.1.1. Test bed

The cluster to test Monitoring Engine integrated with the Elasticity Engine is illustrated in Fig. 6-2. The cluster has two controller VM nodes and two payload VM nodes. Each node of the cluster has a Monitoring Client daemon running in it. The controller nodes run the Monitoring Server daemon as a service with 2N redundancy model and AMF manages the availability of it. The controller nodes also have the Workload Analyzer and Elasticity Engine deployed in them as the triggers for the Elasticity Engine is dispatched from the Workload Analyzer to the Elasticity Engine at the same node.

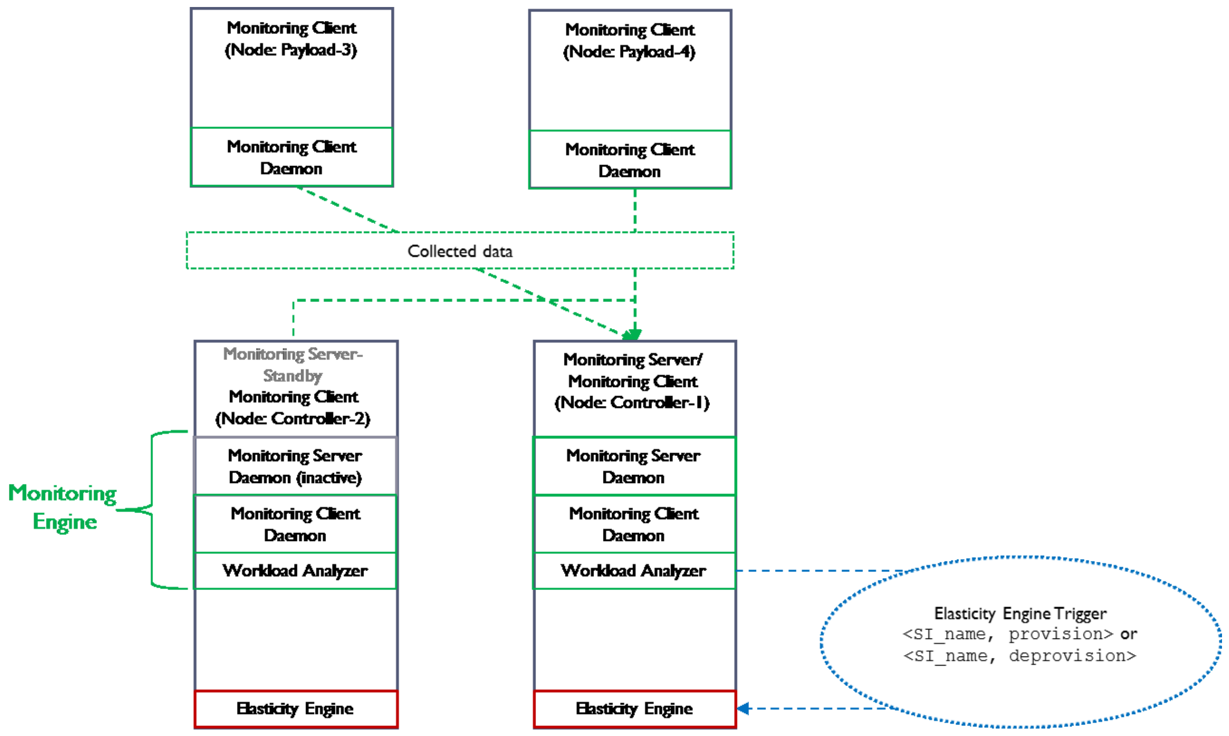


Figure 6-2: Monitoring Engine integrated with Elasticity Engine in a cluster

An HTTP application is configured in an OpenSAF managed cluster comprising of four nodes as shown in Fig. 6-2. The HTTP application is configured with N-way active redundancy model. Each of the SUs in the cluster has one HTTP component configured in it. The HTTP component is created using Python BaseHTTPserver module [42]. The cluster used in this test bed has four SUs in its SG, each SU is configured in a separate node. OpenSAF starts the HTTP application initially with minimum configuration where the two SUs, SU-1 and SU-2 in the SG have active SI assignment and the other two SUs, SU-4 and SU-5 are configured as buffer SUs as shown in Fig. 6-3.

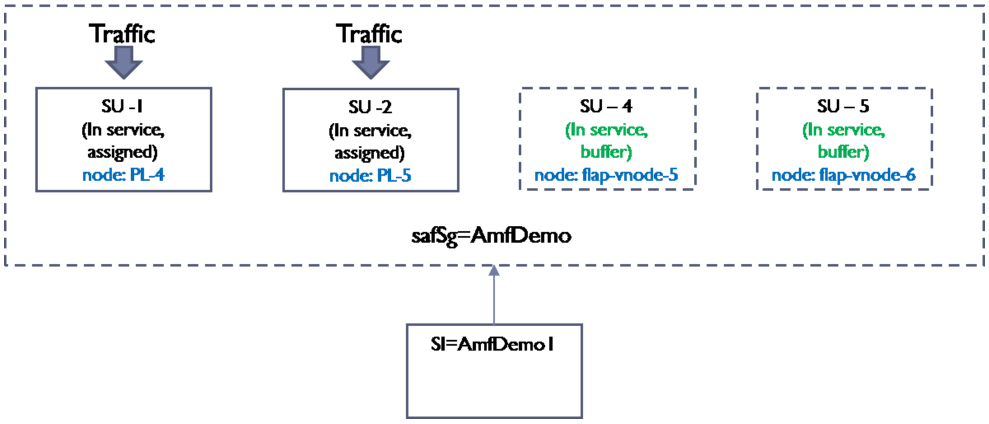
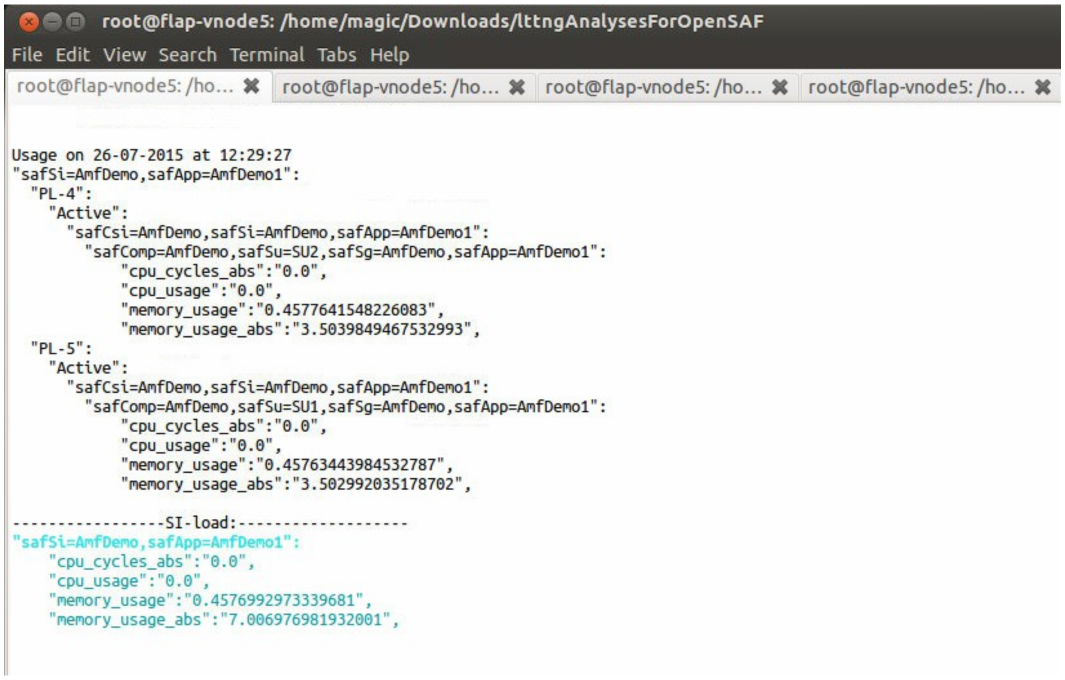

Traffic
Traffic
SU -1 (In service, assigned) node: PL-4
SU -2 (In service, assigned) node: PL-5
SU – 4 (In service, buffer) node: flap-vnode-5
SU – 5 (In service, buffer) node: flap-vnode-6
safSg=AmfDemo
SI=AmfDemo1

Figure 6-3: HTTP service with N-Way-Active redundancy model in minimum configuration


root@flap-vnode5: /home/magic/Downloads/lttngAnalysesForOpenSAF
File Edit View Search Terminal Tabs Help
root@flap-vnode5: /ho...
root@flap-vnode5: /ho...
root@flap-vnode5: /ho...
root@flap-vnode5: /ho...

```
Usage on 26-07-2015 at 12:29:27
"safSi=AmfDemo,safApp=AmfDemo1":
  "PL-4":
    "Active":
      "safCsi=AmfDemo,safSi=AmfDemo,safApp=AmfDemo1":
        "safComp=AmfDemo,safSu=SU2,safSg=AmfDemo,safApp=AmfDemo1":
            "cpu_cycles_abs":"0.0",
            "cpu_usage":"0.0",
            "memory_usage":"0.4577641548226083",
            "memory_usage_abs":"3.5039849467532993",
  "PL-5":
    "Active":
      "safCsi=AmfDemo,safSi=AmfDemo,safApp=AmfDemo1":
        "safComp=AmfDemo,safSu=SU1,safSg=AmfDemo,safApp=AmfDemo1":
            "cpu_cycles_abs":"0.0",
            "cpu_usage":"0.0",
            "memory_usage":"0.457634439845327 87",
            "memory_usage_abs":"3.502992035178702",

----------------SI-load:------------------
"safSi=AmfDemo,safApp=AmfDemo1":
    "cpu_cycles_abs":"0.0",
    "cpu_usage":"0.0",
    "memory_usage":"0.45769929733339681",
    "memory_usage_abs":"7.006976981932001",
```

Figure 6-4: Monitoring Output of HTTP server application at minimum configuration

The incoming HTTP requests to the active servers are interpreted as load on the servers and this load is associated with their respective SIs. The Monitoring Engine Server starts to display the details of SI-load once the service is started as shown in the screenshot of the console in Fig. 6-4. The top portion shows the system's current SI-trees maintaining the hierarchy: SI-name, node-name, HA-state, component-name, CSI-name, workload-metrics. Each level of the tree maintains a fixed indentation in the output. The bottom portion, highlighted in shades of cyan shows the system's current Workload-trees. The SI names are at the top and its aggregated load metrics are at the bottom.

6.3.1.2. Test Case: Triggering Elasticity

As seen in Fig. 6-3, SU-1 (node-name: 'PL-5' in Fig. 6-4) and SU-2 (node-name: 'PL-4' in Fig. 6-4) are the only in-service SUs in the SG 'AmfDemo'. Apache JMeter [43] was used to generate traffic towards HTTP servers with the active SI assignments. In this test case, simulated HTTP traffic was used to trigger two underprovisioned alerts which brought the cluster to its maximum capacity and then the simulated traffic was stopped to cause two overprovisioned alerts to be triggered which returned the cluster back to its minimum configuration.

- Rule to trigger an 'underprovisioned alert' for HTTP service - 'safSi=AmfDemo,safApp=AmfDemo1' :

if

a) the average workload of *'safSi=AmfDemo,safApp=AmfDemo1'* in last *10 seconds* exceeds *70%* usage **and**

b) no alert has been triggered in the last *60 (cool-down period)* seconds **and**

c) elasticity_engine output from last 'underprovisioned alert' did not show 'add more nodes' **or**

nodes have been added to the cluster since the elasticity_engine returned 'add more nodes'

then,

dispatch trigger *underprovisioned_alert* for *'safSi=AmfDemo,safApp=AmfDemo1'*.

- Rule to trigger an 'overprovisioned alert' for HTTP service - 'safSi=AmfDemo,safApp=AmfDemo1':

if

a) the average workload of *'safSi=AmfDemo,safApp=AmfDemo1'* in last *10 seconds* is below *20%* usage **and**

b) no alert has been triggered in the last *60* seconds **and**

c) elasticity_engine output in response to last 'overprovisioned alert' did not show 'minimum configuration'

then,

dispatch trigger *overprovisioned_alert* for '*safSi=AmfDemo,safApp=AmfDemo1'*

Initially, the cluster was at its minimum configuration with two in-service SUs protecting the HTTP SI: 'safSi=AmfDemo,safApp=AmfDemo1'. Once JMeter sent traffic to the active HTTP servers and the workload_analyzer module of the Monitoring Engine detected that the normalized CPU usage of SI 'safSi=AmfDemo,safApp=AmfDemo1' was over 70% for longer than 10 seconds, the scaling action for 'underprovisioned' state was triggered by the Elasticity Engine and SU-5 (node-name: 'flap-vnode-6') was brought into service as illustrated in Fig. 6-5 and shown in the console output in Fig. 6-6 and Fig. 6-7.

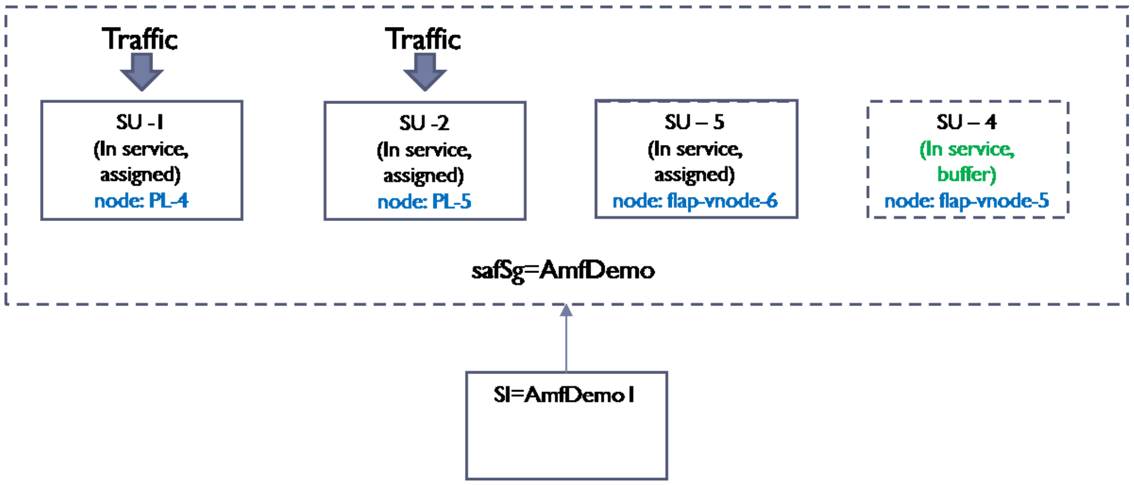
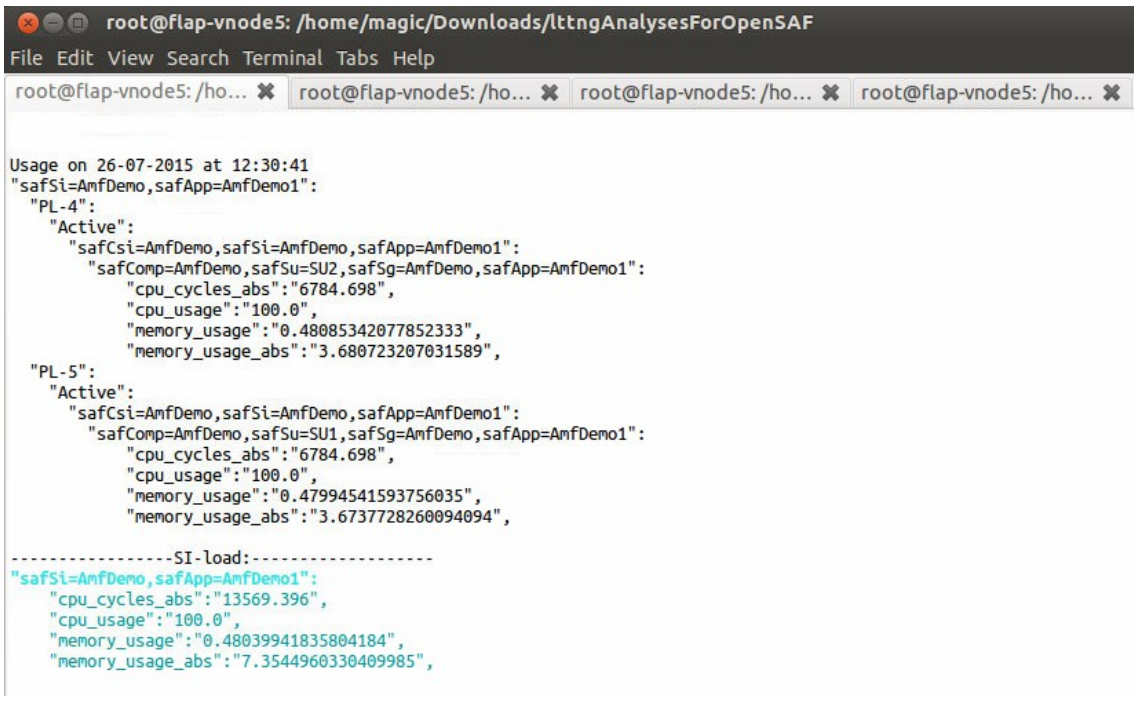


Figure 6-5: HTTP service after triggering underprovisioned alert


root@flap-vnode5: /home/magic/Downloads/lttngAnalysesForOpenSAF
File Edit View Search Terminal Tabs Help
root@flap-vnode5: /ho...
root@flap-vnode5: /ho...
root@flap-vnode5: /ho...
root@flap-vnode5: /ho...

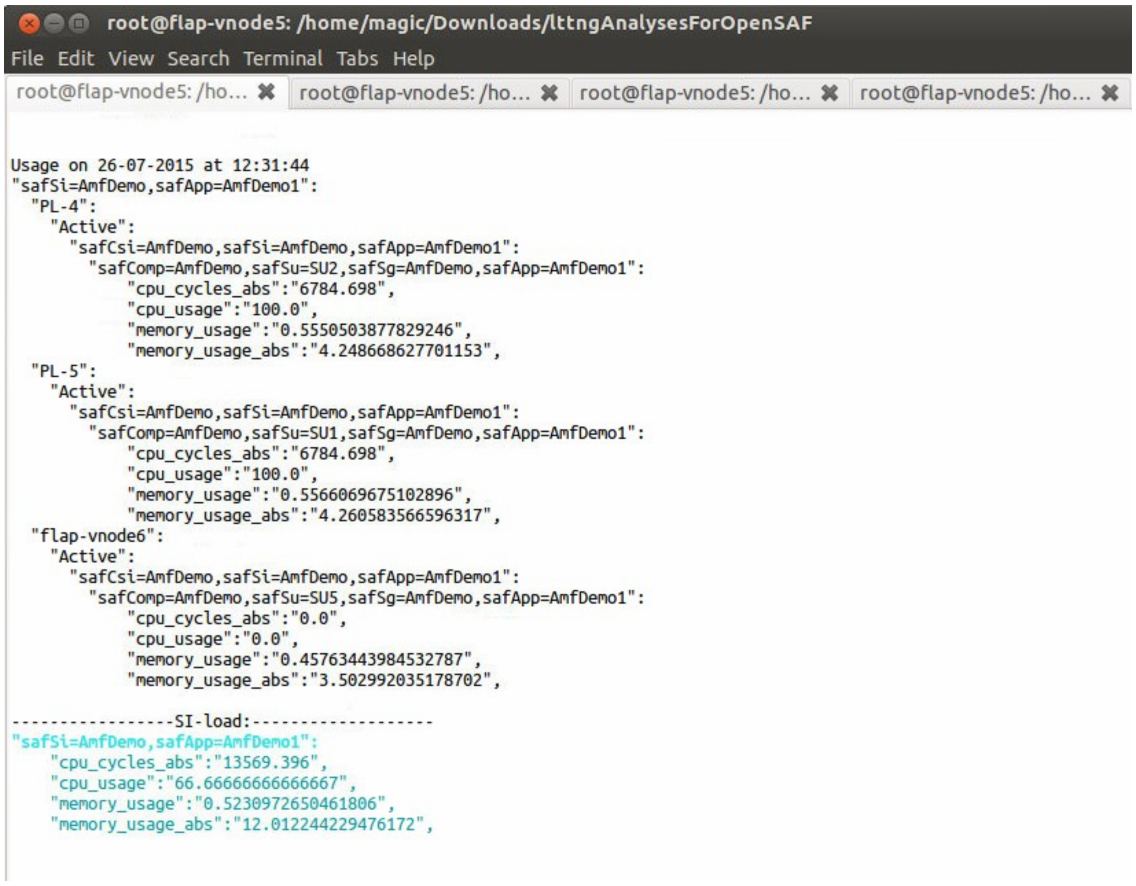
```
Usage on 26-07-2015 at 12:30:41
"safSi=AmfDemo,safApp=AmfDemo1":
  "PL-4":
    "Active":
      "safCsi=AmfDemo,safSi=AmfDemo,safApp=AmfDemo1":
        "safComp=AmfDemo,safSu=SU2,safSg=AmfDemo,safApp=AmfDemo1":
            "cpu_cycles_abs":"6784.698",
            "cpu_usage":"100.0",
            "memory_usage":"0.48085342077852333",
            "memory_usage_abs":"3.680723207031589",
  "PL-5":
    "Active":
      "safCsi=AmfDemo,safSi=AmfDemo,safApp=AmfDemo1":
        "safComp=AmfDemo,safSu=SU1,safSg=AmfDemo,safApp=AmfDemo1":
            "cpu_cycles_abs":"6784.698",
            "cpu_usage":"100.0",
            "memory_usage":"0.47994541593756035",
            "memory_usage_abs":"3.6737728260094094",

----------------SI-load:------------------
"safSi=AmfDemo,safApp=AmfDemo1":
    "cpu_cycles_abs":"13569.396",
    "cpu_usage":"100.0",
    "memory_usage":"0.480399418358O4184",
    "memory_usage_abs":"7.3544960330409985",
```

Figure 6-6: Monitoring Output of HTTP components in SU-1 and SU-2 receiving heavy traffic

As seen in Fig. 6-6, The HTTP component in both SU-1 and SU-2 were receiving significant traffic, which was reflected in their normalized CPU usages, jumping up to 100%. The absolute

CPU usage by the service on each VM jumped up to the maximum capacity of the VMs' CPU: 6.784 GHz.

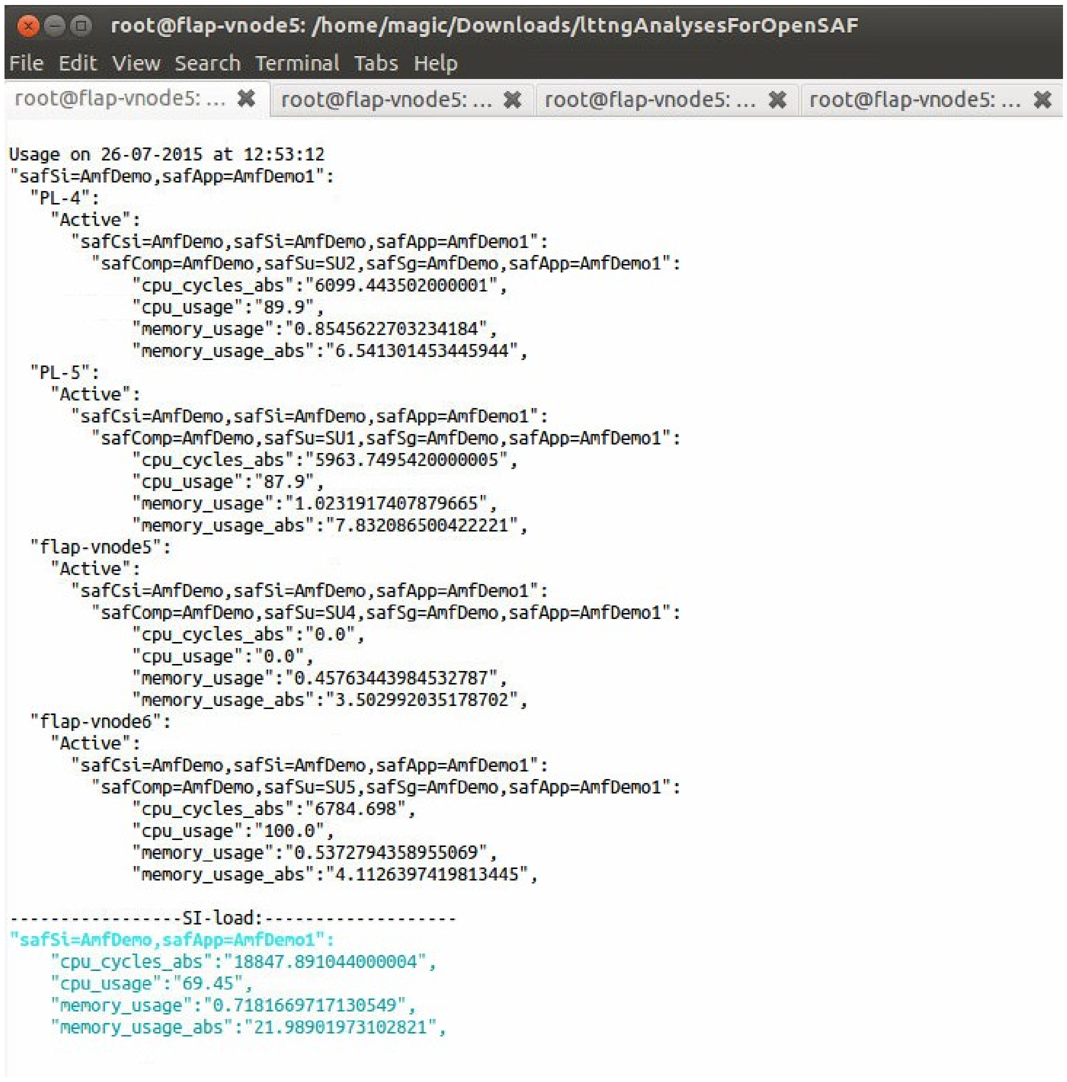
The HTTP component in the newly provisioned VM node where SU-5 is configured, was yet to receive traffic. In Fig. 6-7, the scaling operation causes the total normalized CPU usage of the SI 'safSi=AmfDemo,safApp=AmfDemo1' to drop to 66.67%, even though the total CPU usage of the SI remains the same at 13.56 GHz.


root@flap-vnode5: /home/magic/Downloads/lttngAnalysesForOpenSAF
File Edit View Search Terminal Tabs Help
root@flap-vnode5: /ho...
root@flap-vnode5: /ho...
root@flap-vnode5: /ho...
root@flap-vnode5: /ho...

```
Usage on 26-07-2015 at 12:31:44
"safSi=AmfDemo,safApp=AmfDemo1":
  "PL-4":
    "Active":
      "safCsi=AmfDemo,safSi=AmfDemo,safApp=AmfDemo1":
        "safComp=AmfDemo,safSu=SU2,safSg=AmfDemo,safApp=AmfDemo1":
            "cpu_cycles_abs":"6784.698",
            "cpu_usage":"100.0",
            "memory_usage":"0.5550503877829246",
            "memory_usage_abs":"4.248668627701153",
  "PL-5":
    "Active":
      "safCsi=AmfDemo,safSi=AmfDemo,safApp=AmfDemo1":
        "safComp=AmfDemo,safSu=SU1,safSg=AmfDemo,safApp=AmfDemo1":
            "cpu_cycles_abs":"6784.698",
            "cpu_usage":"100.0",
            "memory_usage":"0.5566069675102896",
            "memory_usage_abs":"4.260583566596317",
  "flap-vnode6":
    "Active":
      "safCsi=AmfDemo,safSi=AmfDemo,safApp=AmfDemo1":
        "safComp=AmfDemo,safSu=SU5,safSg=AmfDemo,safApp=AmfDemo1":
            "cpu_cycles_abs":"0.0",
            "cpu_usage":"0.0",
            "memory_usage":"0.45763443984532787",
            "memory_usage_abs":"3.502992035178702",

----------------SI-load:------------------
"safSi=AmfDemo,safApp=AmfDemo1":
    "cpu_cycles_abs":"13569.396",
    "cpu_usage":"66.66666666666667",
    "memory_usage":"0.5230972650461806",
    "memory_usage_abs":"12.012244229476172",
```

Figure 6-7: Monitoring Output of HTTP server after triggering underprovisioned alert

After being brought into service, the HTTP component in SU-5 also starts receiving traffic which caused the normalized CPU usage of the SI 'safSi=AmfDemo,safApp=AmfDemo1' to rise above the upper threshold of 70% CPU usage again. Since a scaling operation had just been performed by the Elasticity Engine, the workload_analyzer module did not trigger another alert during the cool-down period. If the cluster was still in underprovisioned state after the cool-down period had elapsed, another underprovisioned alert would be dispatched to the Elasticity Engine by the workload_analyzer module. The Elasticity Engine changed the configuration to bring SU-4 in service from the buffers, which caused the cluster reach its maximum capacity. Fig. 6-8 shows the console output of the Monitoring Server at this state.


root@flap-vnode5: /home/magic/Downloads/lttngAnalysesForOpenSAF
File Edit View Search Terminal Tabs Help
root@flap-vnode5: ...
root@flap-vnode5: ...
root@flap-vnode5: ...
root@flap-vnode5: ...

```
Usage on 26-07-2015 at 12:53:12
"safSi=AmfDemo,safApp=AmfDemo1":
  "PL-4":
    "Active":
      "safCsi=AmfDemo,safSi=AmfDemo,safApp=AmfDemo1":
        "safComp=AmfDemo,safSu=SU2,safSg=AmfDemo,safApp=AmfDemo1":
            "cpu_cycles_abs":"6099.443502000001",
            "cpu_usage":"89.9",
            "memory_usage":"0.8545622703234184",
            "memory_usage_abs":"6.541301453445944",
  "PL-5":
    "Active":
      "safCsi=AmfDemo,safSi=AmfDemo,safApp=AmfDemo1":
        "safComp=AmfDemo,safSu=SU1,safSg=AmfDemo,safApp=AmfDemo1":
            "cpu_cycles_abs":"5963.7495420000005",
            "cpu_usage":"87.9",
            "memory_usage":"1.0231917407879665",
            "memory_usage_abs":"7.832086500422221",
  "flap-vnode5":
    "Active":
      "safCsi=AmfDemo,safSi=AmfDemo,safApp=AmfDemo1":
        "safComp=AmfDemo,safSu=SU4,safSg=AmfDemo,safApp=AmfDemo1":
            "cpu_cycles_abs":"0.0",
            "cpu_usage":"0.0",
            "memory_usage":"0.45763443984532787",
            "memory_usage_abs":"3.502992035178702",
  "flap-vnode6":
    "Active":
      "safCsi=AmfDemo,safSi=AmfDemo,safApp=AmfDemo1":
        "safComp=AmfDemo,safSu=SU5,safSg=AmfDemo,safApp=AmfDemo1":
            "cpu_cycles_abs":"6784.698",
            "cpu_usage":"100.0",
            "memory_usage":"0.5372794358955069",
            "memory_usage_abs":"4.1126397419813445",

----------------SI-load:------------------
"safSi=AmfDemo,safApp=AmfDemo1":
    "cpu_cycles_abs":"18847.891044000004",
    "cpu_usage":"69.45",
    "memory_usage":"0.7181669717130549",
    "memory_usage_abs":"21.98901973102821",
```

Figure 6-8: Monitoring Output of HTTP server reaching maximum capacity of the cluster

Once the cluster reached its maximum capacity, the Elasticity Engine notified the workload_analyzer module of the Monitoring Engine that more node needed to be added to the cluster in order to respond to any further underprovisioned alert, hence no underprovisioned alert was dispatched to the Elasticity Engine once the cluster reached its maximum capacity until more nodes were added to the cluster.

At this stage, the simulated web traffic from JMeter was turned off. As the relative CPU load on the SI 'safSi=AmfDemo,safApp=AmfDemo1' remained below 20%, the workload_analyzer module kept dispatching overprovisioned alerts to the Elasticity Engine after every cool-down period until the cluster reached its minimum configuration again. When the workload_analyzer module detected that the cluster had reached its minimum configuration from the output of the Elasticity Engine's scaling action, it stopped dispatching overprovisioned alerts.

The memory and CPU load of the SI 'safSi=AmfDemo,safApp=AmfDemo1' while scaling the cluster has been plotted in Fig. 6-9 and 6-10 respectively. In both figures, the relative workload metric is plotted at the top and the absolute workload metric is plotted at the bottom. From the pattern seen in the total memory usage of the SI in Fig. 6-9, it is evident that the Elasticity Engine provisioned VMs to the service at 103rd second and 215th second, and it deprovisioned VMs at 312th second and 420th second. The times of scaling the cluster for the service is less evident in the CPU usage plots in Fig. 6-10.

In the plot for relative memory usage, we observe that the usage usually remained near or below the 1% of the cluster's total memory. It had the least percentage of memory usage when the service was using the most amount of memory in total, at about 23 MBs from 215th second to 312th second. The relative usage of memory dropped with provisioning of VMs as absolute memory usage rose. Conversely, we also observe that the relative memory usage rose with deprovisioning of VMs as absolute memory usage dropped.

The CPU usage pattern in Fig. 6-10 suggests that the cluster started to receive heavy traffic from time 87th seconds till 245th second. The provisioning events can be spotted by observing the sudden drop in relative CPU usage at 103rd second and 215th second. Comparing the plots in Fig. 6-10, we observe that the provisioning or de-provisioning VMs do not necessarily affect the total CPU usage.

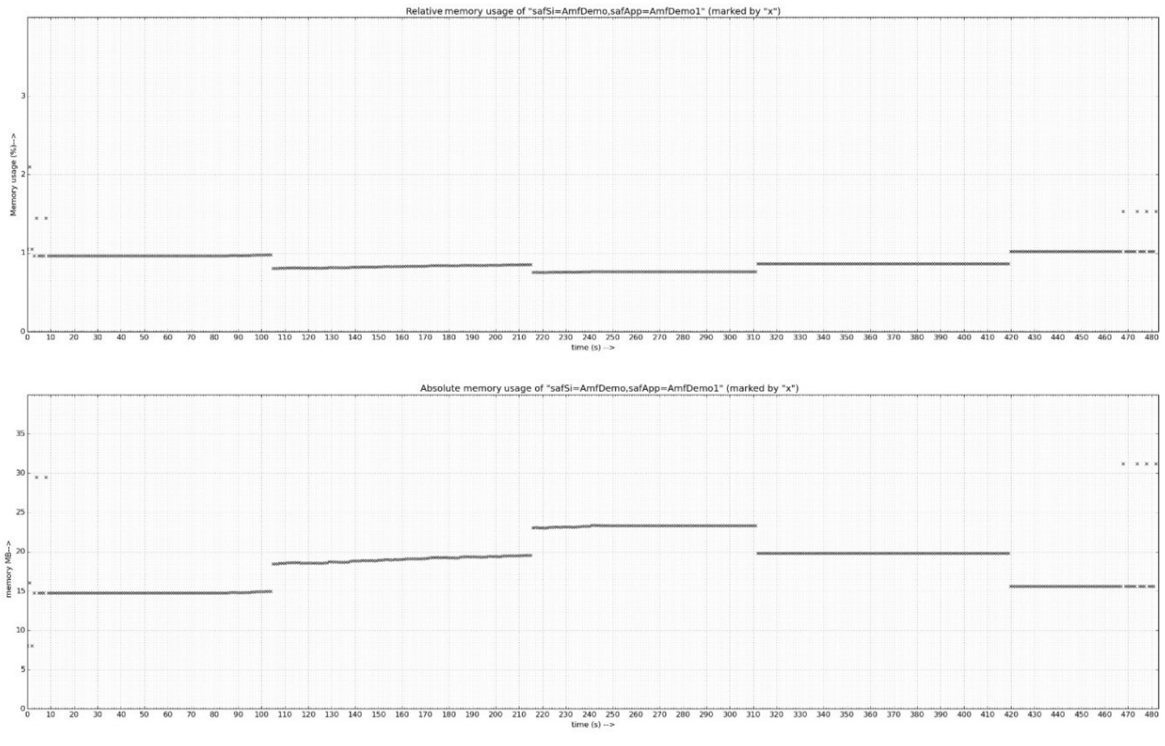


Figure 6-9: Memory usage metrics during provisioning/deprovisioning of VMs

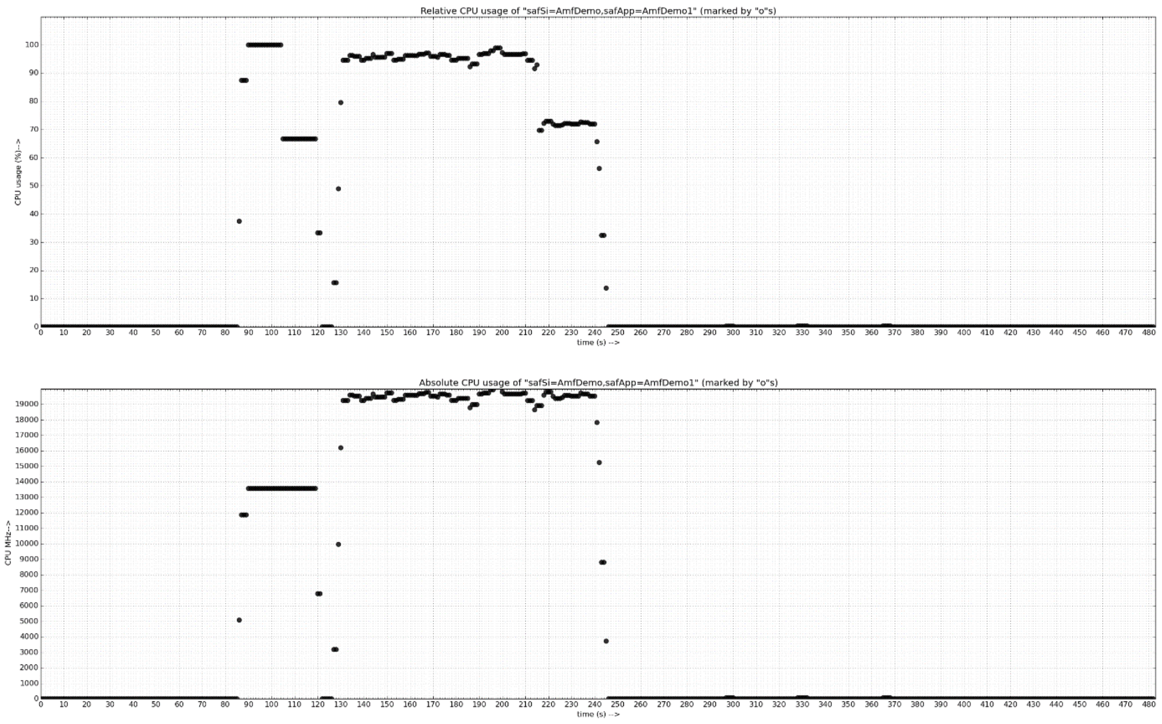


Figure 6-10: CPU usage metrics during provisioning/deprovisioning of VMs

By observing the plots, we can conclude that relative workload metrics indicated how well the workload is distributed over the cluster while absolute workload metrics indicate the actual workload of the service. The two kind of workload metrics can be analyzed further to reach more effective scaling decisions like the number of required VMs to meet workload at a certain time while provisioning or deprovisioning VMs, decisions on SIs' assignment distribution, etc.

6.3.2. Video streaming service

The VideoLAN [44] software has been configured as a component with 2N redundancy model in OpenSAF in order to test the Monitoring Engine's adaptability with the dynamic nature of the HA services.

6.3.2.1. Test bed

The cluster to test Monitoring Engine's adaptability to HA state change had two VMs as illustrated in Fig. 6-11, each VM node in the cluster had a Monitoring Client running on it. Since no scaling operation was performed in this test case, no Elasticity Engine was included in this test bed setup. Each VM of this cluster has similar system specification as shown in table 6-1.

In this test-bed, the SG 'SG-1' was configured with 2N redundancy model and comprises of two SUs: 'SU-1' and 'SU-2'. Each SU was configured on a separate node. Each SU in this setup comprised of two components.

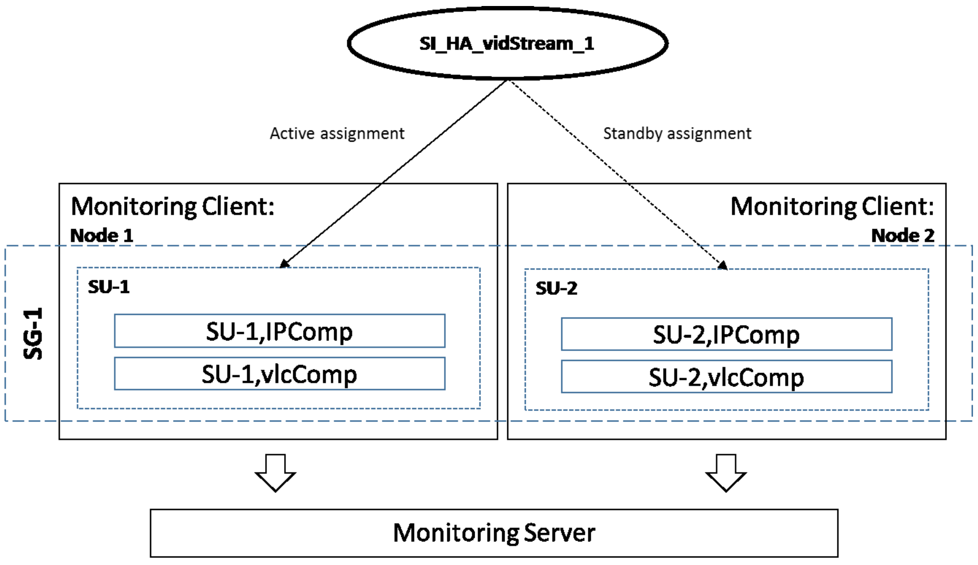


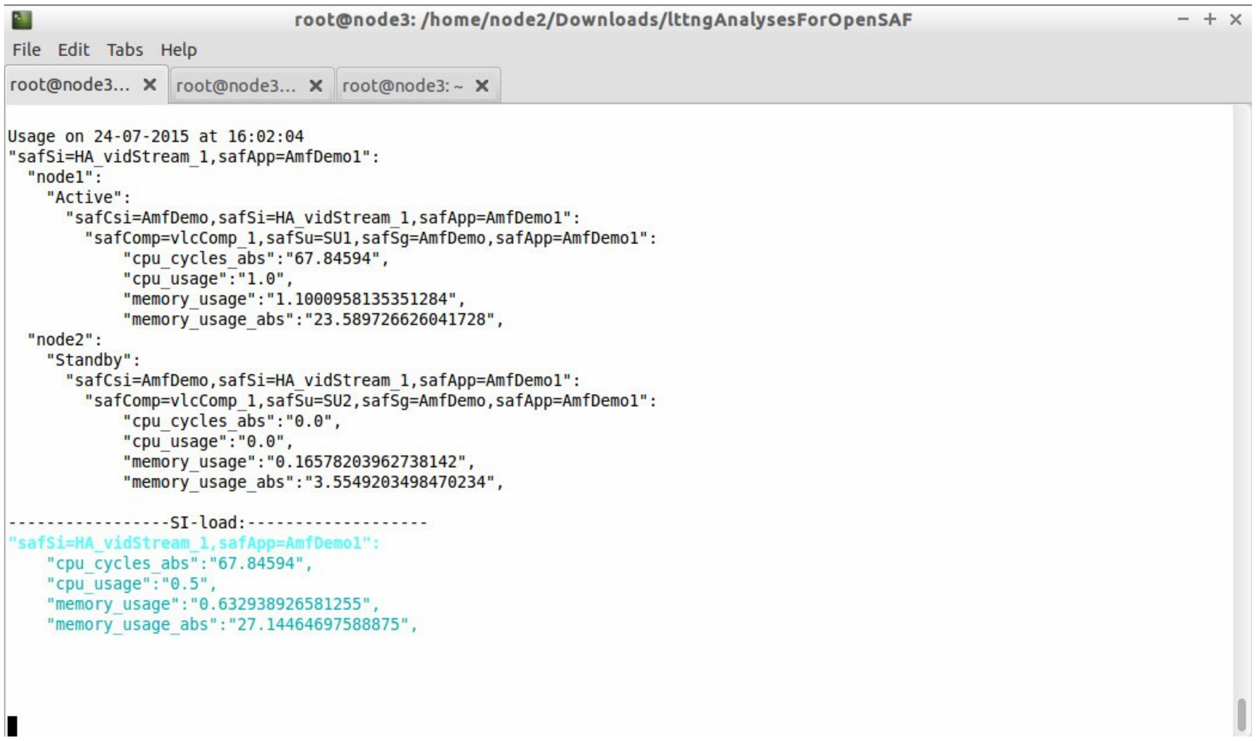
Figure 6-11: Video streaming HA service with a 2N redundancy model

- **VLC Component:** A VideoLAN application module has been developed according to SA Forum APIs to manage the application's lifecycle and streaming services, which enables us to use this application as a pre-instantiable, SA-Aware component. All VLC components used in this test bed were instrumented using LTTng UST [24] for monitoring purposes using the procedure discussed in Chapter 4. The video streaming service was configured in the SUs to stream a particular video according to the SI assignment using this component. Video streaming starts at the time when an SU hosting this component received an ‘active’ assignment for the SI it was protecting.

- **IP Component:** This is a simple component which sets the target IP address to stream the video service. Each VLC component is dependent on an IP component to provide video streaming service.

On service start, each vlcComp component ran the modified media player code, while the IPComp component specified the IP address where the vlcComp component with active assignment provided the video streaming service for a given video stream. IPComp component ran briefly when the service was assigned to the vlcComp component, but it did not create any continuous workload.

Conforming to the redundancy model, at startup for the video streaming service, one of the SUs got the active assignment and the other received the standby assignment. For example, the SU-1 in ‘Node-1’ got the active assignment for the service ‘SI_HA_vidStream_1’. This means that the vlcComp component in SU-1 received the callback from AMF with the active CSI assignment for the SI: ‘SI_HA_vidStream_1’, while the vlcComp component in SU-2 received the callback with standby CSI assignment for the same SI. These callbacks were detected by the monitoring clients in the respective nodes and communicated as resource usage to the monitoring server. The status and load of the video streaming service observed from the Monitoring Server output at this stage is shown in the console output in Fig. 6-12.

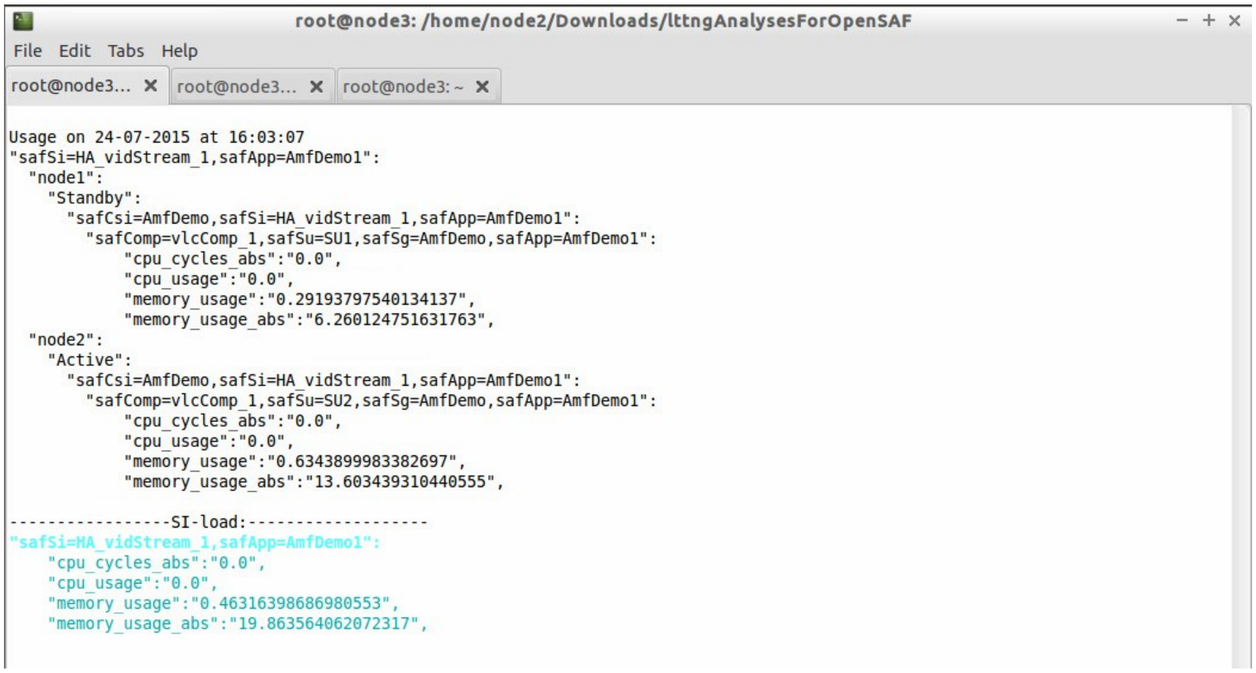


```
Usage on 24-07-2015 at 16:02:04
"safSi=HA_vidStream_1,safApp=AmfDemo1":
  "node1":
    "Active":
      "safCsi=AmfDemo,safSi=HA_vidStream_1,safApp=AmfDemo1":
        "safComp=vlcComp_1,safSu=SU1,safSg=AmfDemo,safApp=AmfDemo1":
            "cpu_cycles_abs":"67.84594",
            "cpu_usage":"1.0",
            "memory_usage":"1.1000958135351284",
            "memory_usage_abs":"23.589726626041728",
  "node2":
    "Standby":
      "safCsi=AmfDemo,safSi=HA_vidStream_1,safApp=AmfDemo1":
        "safComp=vlcComp_1,safSu=SU2,safSg=AmfDemo,safApp=AmfDemo1":
            "cpu_cycles_abs":"0.0",
            "cpu_usage":"0.0",
            "memory_usage":"0.16578203962738142",
            "memory_usage_abs":"3.5549203498470234",

----------------SI-load:------------------
"safSi=HA_vidStream_1,safApp=AmfDemo1":
    "cpu_cycles_abs":"67.84594",
    "cpu_usage":"0.5",
    "memory_usage":"0.632938926581255",
    "memory_usage_abs":"27.14464697588875",
```

Figure 6-12: Monitoring Server console output for video streaming service

6.3.2.2. Test-case: Adaptability to failover

In order to test the Monitoring Engine's adaptability with HA state change, the video streaming service the active component-process was killed in SU-1 was killed using a SIGKILL [45] command to invoke a failover. As a result, AMF failed over the SI 'SI_HA_vidStream_1' to the standby SU-2. Subsequently, it recovered the failed SU-1 and assigned its components as standbys. The status and load of the video streaming service observed from the Monitoring Server output at this stage is shown in the console output in Fig. 6-13, which shows that the HA state of the nodes had changed due to the failover and the Monitoring Engine correctly detected that change.



```
Usage on 24-07-2015 at 16:03:07
"safSi=HA_vidStream_1,safApp=AmfDemo1":
  "node1":
    "Standby":
      "safCsi=AmfDemo,safSi=HA_vidStream_1,safApp=AmfDemo1":
        "safComp=vlcComp_1,safSu=SU1,safSg=AmfDemo,safApp=AmfDemo1":
            "cpu_cycles_abs":"0.0",
            "cpu_usage":"0.0",
            "memory_usage":"0.29193797540134137",
            "memory_usage_abs":"6.260124751631763",
  "node2":
    "Active":
      "safCsi=AmfDemo,safSi=HA_vidStream_1,safApp=AmfDemo1":
        "safComp=vlcComp_1,safSu=SU2,safSg=AmfDemo,safApp=AmfDemo1":
            "cpu_cycles_abs":"0.0",
            "cpu_usage":"0.0",
            "memory_usage":"0.6343899983382697",
            "memory_usage_abs":"13.603439310440555",

-----------------SI-load:-------------------
"safSi=HA_vidStream_1,safApp=AmfDemo1":
    "cpu_cycles_abs":"0.0",
    "cpu_usage":"0.0",
    "memory_usage":"0.46316398686980553",
    "memory_usage_abs":"19.863564062072317",
```

Figure 6-13: Console output at the Monitoring Server after failover

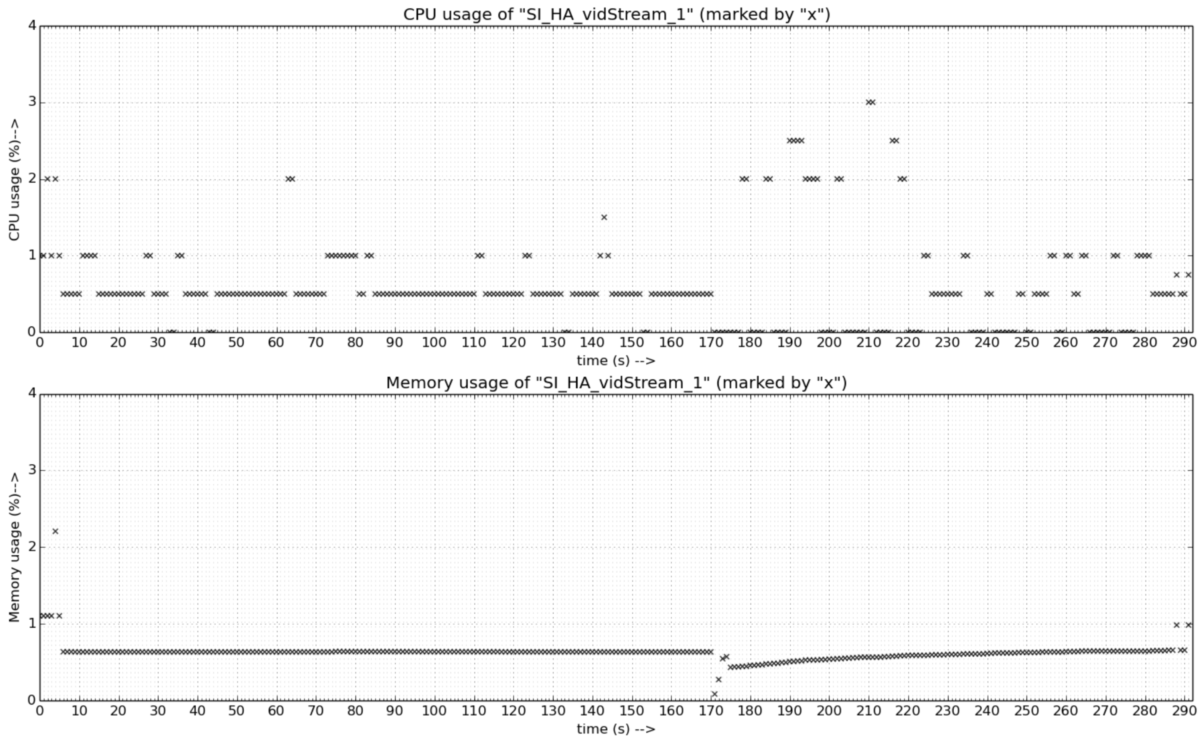


Figure 6-14: Relative CPU and memory workload of the video streaming service during failover

The workload measurements during this failover are shown in Fig. 6-14. The plot at the top shows the relative CPU usage of the SI during this test case and the bottom portion shows the relative memory usage of it. The short dip at the 170th second in the measurements indicates the moment of the failover.

6.4. Monitoring overhead

The monitoring overhead on the Monitoring Server and Monitoring Client nodes have been measured to evaluate the monitoring architecture for both of the test beds discussed in this chapter. The processes responsible for creating overhead and a short description of it is summarized in table 6-1.

Table 6-2: Processes responsible for monitoring overhead and their descriptions

Process	**Origin VM node**	**Description**
Monitoring Server Daemon	Monitoring Server	Process that controls /manages all modules including the workload analyzer in Monitoring Server.
Monitoring Client Daemon	Monitoring Client	Process that controls /manages all modules in Monitoring Client.
LTTng Session Daemon	Monitoring Client	LTTng process started as a part of LTTng service. This process manages all LTTng sessions [24].
LTTng Consumer Daemon	Monitoring Client	LTTng process that translates trace events in a buffer and then converts and saves them as trace data in runtime [24].
LTTng Relay Daemon	Monitoring Client	LTTng process that converts trace data received over network into local trace data. For the LTTng Live feature, all trace data are relayed as network data at first. I.e. LTTng Live trace data are sent to ‘localhost’ if the trace session is to run locally [24].

In order to ensure that no other LTTng session impacts the overhead values, only one LTTng session was started on each of the Monitoring Clients for each test bed.

The measurements were taken in one second interval using python PSUtil tool. For CPU usage, the tool provides the usage in percentage for any given process. However, the percentage is provided considering the usage of all CPU cores by the process. Therefore, in a system with 2 cores, it is possible for a process to consume up-to 200% CPU. To get an average CPU usage in the range of 0 to 100, the sum of the CPU the usage of considered processes has been divided by the number of CPU cores present in the system. Since the system is homogeneous, i.e. all the monitoring client nodes in the cluster have identical system specification as shown in Table 6-1, the mean overhead-per-node has been measured by dividing the sum of monitoring overhead on all nodes by the number of nodes in the cluster. With the considerations above, the following equations have been used to measure overhead.

$$\mathrm{CPU_{MC}} = \left(\sum_{n=0}^{N} \frac{\sum_{t=0}^{T} CPUusageOfProcs(\{LTTngSD, LTTngCD, LTTngRD, MCD\}, t)}{T \times numberOfCPUcores} \right) / \mathrm{N} \qquad (6\text{-}1)$$

In Eq. 6-1, CPU_{MC} refers to the mean CPU monitoring overhead of the monitoring clients per second, T refers to the total number of seconds the overhead has been measured, *CPUusageOfProcs* function provides the sum of CPU usage of a set of process IDs at time t. *LTTngSD, LTTngCD, LTTngRD* and *MCD* refer to the process IDs of the LTTng session daemon, the LTTng consumer daemon, LTTng relay daemon and the monitoring client daemon, respectively. *numberOfCPUcores* refers to the number of CPU cores of a system used for the measurements. N refers to the number of nodes in the cluster.

Similarly, for the monitoring server the overhead is calculated as

$$\mathrm{CPU}_{\mathrm{MS}} = \frac{\sum_{t=0}^{T} CPUusageOfProcs(\{MSD\},t)}{T\times numberOfCPUcores} \tag{6-2}$$

In Eq.2, *CPU_{MS}* refers to the mean CPU monitoring overhead of the monitoring server per second and *MSD* refers to the process ID of the monitoring server daemon.

PSUtil can also provide the memory usage of any set of processes in the range of 0 to 100%. In a similar manner to Eq.1 and Eq.2, memory overhead can be calculated by the following equations:

$$\mathrm{M}_{\mathrm{MC}}= \left(\sum_{n=0}^{N} \frac{\sum_{t=0}^{T} MemUsageOfProcs(\{LTTngSD,LTTngCD,LTTngRD,MCD\},t)}{T} \right)/\mathrm{N} \tag{6-3}$$

$$\mathrm{M}_{\mathrm{MS}} = \frac{\sum_{t=0}^{T} MemUsageOfProcs(\{MSD\},t)}{T} \tag{6-4}$$

In Eq. 6-3 and Eq. 6-4, M_{MC} and M_{MS} refers to the mean memory overhead – per second – for the monitoring client and for the monitoring server, respectively. *MemUsageOfProcs* provides the memory usage of a set of processes at time *t*.

To measure the overhead, we ran both the video streaming service and HTTP service separately and simultaneously for close to 12 hours for each cases. The monitoring overhead has been calculated using Eq. 6-1 through Eq. 6-4. The resulting overhead measurement is summarized in Table 6-3. The CPU and memory usage of the monitoring client processes and monitoring server process in their respective nodes are shown to remain within 5%. We also show that monitoring multiple applications at the same time increases the total monitoring overhead slightly.

Any notable resource usage in the cloud equates to a monetary value. A large monitoring overhead would require a bigger investment in monitoring. We tried to ensure that the overhead and the subsequent required investment in monitoring remains minimal to encourage the adoption in practice of the approach/architecture described in this work.

Table 6-3: Results over monitoring overhead measurement

Node-type	**Hardware Specifications**	**Mean Overhead/sec (Test bed -1, HTTP Service)**	**Mean Overhead/sec (Test bed -2, Video streaming Service)**	**Mean Overhead/sec (collocated services: HTTP and Video streaming)**
Monitoring Server	**CPU**: Intel(R) Core(TM) i7-2600 CPU @ 3.40GHz	CPU_{MS} = 2.02 %	CPU_{MS} = 1.94 %	CPU_{MS} = 2.16 %
	Memory: 16GB	M_{MS} = 1.14 %	M_{MS} = 1.08 %	M_{MS} = 1.18 %
Monitoring Clients	**CPU:** Intel(R) Core(TM) i7-2600 CPU @ 3.40GHz (Virtual, 2 cores)	CPU_{MC} = 1.8 %	CPU_{MC} = 1.67 %	CPU_{MC} = 1.85 %
	Memory: 2GB	M_{MC} : 2.14 %	M_{MC} : 2.5 %	M_{MC} : 2.72 %
Total runtime (*T*)	44227 seconds (12 hours, 17 minutes, 7 seconds)	44082 seconds (12 hours, 14 minutes, 42 seconds)		42391 seconds (12 hours, 6 minutes, 31 seconds)
Number of nodes (*N*)	2	4		4

The yellow shade in table 6-3 indicates the overhead measurements from Test bed -1, the green shade indicates the measurements from Test bed -2, the blue shade indicates the measurements taken from a test bed where the setup of both Test bed -1 and Test bed -2 were present and the services were run simultaneously.

6.5. Summary

In this chapter, we discussed the architecture of the Monitoring Engine prototype developed as proof of concepts discussed in the earlier chapters. In order to evaluate the effectiveness and performance of the prototype, we prepared two separate test beds with different applications and analyzed the result of a unique test case on each of the test beds. We showed that the monitoring engine prototype is able to trigger alerts to the Elasticity Engine while keeping up with the dynamic nature of service assignments. We also measured the overhead of the monitoring engine prototype for both of the test cases where we observed that the monitoring engine's overall overhead remains within 5% of the total resources for all metrics. Since the prototype is developed using python, it can be made to be very portable across VMs by sandboxing [46] the repository, which eliminates most of the dependency issues during deployment.

As the limitation of most applications following the client-server architecture goes, failure of the Monitoring Server ensues the failure of monitoring altogether and the Monitoring Server and Monitoring Clients on each of the nodes must be restarted to start monitoring again.

7. Conclusion and Future Work

7.1. Conclusion

In this thesis, we introduced an approach and an architecture for the monitoring of workload at the service level applicable to the services provided by application components, which may be collocated in the same VM and where the service to application component assignments change dynamically over time. Thus, the approach is applicable to SA Forum compliant systems where the high availability of services is ensured by AMF dynamically by assigning the application services to application components based of their current operational status.

We proposed and implemented a method to automatically instrument SA-aware components. The automatic instrumentation method speeds up the otherwise tedious instrumentation procedure. As a result of this instrumentation, the dynamic assignment and reassignment of services to the processes of application components is detected and tracked at runtime.

We devised algorithms to map and aggregate the resource usage of processes used by application components to the SIs using the trace-data obtained from the instrumented components. Therefore, workload changes at the service level can be detected and the Elasticity Engine [8] [10] can be triggered for resource provisioning and de-provisioning at the application level.

In order to prove the proposed concepts, the approach and architecture have been implemented and integrated with OpenSAF [11], an open source implementations of the SA Forum middleware for HA management. Accordingly, the implemented Monitoring Engine prototype adapts to the situations where different components can be in different HA states, active or standby, on behalf of a service and this HA state assignment changes dynamically due to for example component failures or configuration changes. We also integrated the Monitoring Engine prototype with an existing Elasticity Engine prototype [8] in one of the test beds to show that the Monitoring Engine can trigger alerts to the Elasticity Engine when the workload on any service exceeds a threshold. We measured the Monitoring Engine's overhead on the VMs running it as a part of its preliminary evaluation, which shows that the overhead for relative CPU and memory usage remain within 5% on average for the nodes in the cluster. The overhead is fairly low compared to most available solutions especially considering the fact that the overhead has been measured with respect to nodes that had configurations on par with 'micro-instances' [47] or 'mini-instances' [48] (I.e. the most basic VMs provided in IAAS services) [49]. The low overhead makes the solution more desirable to the current and prospective stakeholders.

7.2. Limitations and Future Work

The Monitoring Engine has a number of limitations that may need to be worked on in the future.

The Monitoring Engine maps the workload of a single process to a CSI assignment of one component. If there are multiple CSI assignments leading to a single process, there is no way yet to distinguish between the workload of the multiple assignments. Improving this will make the measurements of the Monitoring Engine more precise.

The current integration of the Monitoring Engine and the Elasticity Engine is only a proof of concept, there has been no significant research done on the integration yet. The rules based on which the Elasticity Engine is triggered need to take into account the nature of workload change, the reaction time of the Elasticity Engine, the probable strategy to react to the workload change etc.

Bibliography

[1] M. Toeroe, F. Tam, D. Penkler, R. Hyerle, J. Jensen, M. Angelic, U. Kleber, A. Mishra, A. Kanso, M. Angelic and F. Khendek, Service Availability: Principles and Practice, M. Toeroe and F. Tam, Eds., Montreal, Quebec: John Wiley & Sons. Ltd., 2012.

[2] Amazon Inorporated, "What is Cloud Computing? - Benefits of the Cloud," Amazon Inorporated, 2015. [Online]. Available: http://aws.amazon.com/what-is-cloud-computing/. [Accessed July 2015].

[3] M. Armbrust, A. Fox, R. Griffith, A. D. Joseph, R. Katz, A. Konwinski, G. Lee, D. Patterson, A. Rabkin, I. Stoica and M. Zaharia, "A View of Cloud Computing," *Communications of the ACM (CACM),* vol. 53, no. 4, pp. 50-58, April 2010.

[4] N. R. Herbst, S. Kounev and R. Reussner, "Elasticity in Cloud Computing: What It Is, and What It Is Not," in *International Conference on Autonomic Computing (ICAC)*, San Jose, CA, June 26-28, 2013.

[5] M. Rak, S. Venticinque and T. Ma´hr, "Cloud Application Monitoring: the mOSAIC Approach," in *2011 Third IEEE International Conference on Cloud Computing Technology and Science (IEEE CloudCom)*, Athens, Greece, August 28, 2011.

[6] Amazon CloudWatch, "Amazon CloudWatch Developer Guide: Publish Custom Metrics," Amazon, 1 August 2010. [Online]. Available: http://docs.aws.amazon.com/AmazonCloudWatch/latest/DeveloperGuide/publishingMetrics.html. [Accessed August 2015].

[7] Service Availability Forum, "Service Availability Forum - Home," Service Availability Forum, 2015. [Online]. Available: http://www.saforum.org/. [Accessed July 2015].

[8] N. Pawar, ""Managing High-Availability and Elasticity in a Cluster Environment" Masters Thesis," Concordia University, Montreal, 2014.

[9] Service Availability Forum, "Service Availability Forum Availability Management Framework SAI-AIS-AMF-B.04.01," September 2011. [Online]. Available: http://www.saforum.org/hoa/assn16627/images/SAI-AIS-AMF-B.04.01.AL.pdf. [Accessed July 2015].

[10] M. Toeroe, N. Pawar and F. Khendek, "Managing Application Level Elasticity and Availability," in *10th Conference on Network and Service Management and Workshop (CNSM)*, Rio de Janeiro, 17-21 November, 2014.

[11] OpenSAF Foundation, "OpenSAF Foundation - About Us," OpenSAF, 2015. [Online]. Available: http://www.opensaf.org/page/14944~155299/About-Us. [Accessed July 2015].

[12] Service Availability Forum, "Service Availability Forum: Service Availability Forum - Open Specifications for Service Availability," Service Availability Forum, 2015. [Online]. Available: http://www.saforum.org/page/16627~214723/Service-Availability-Forum-Open-Specifications-for-Service-Availability. [Accessed 4 August 2015].

[13] Service Availability Forum, "Service Availability Forum Specification : Hardware Platform Interface," Service Availability Forum, 2015. [Online]. Available:

http://www.saforum.org/Page/16627~217308/Service-Availability-Forum-Specification-Hardware-Platform-Interface-28HPI29. [Accessed 4 August 2015].

[14] Service Availability Forum, "Service Availability Forum - Tutorial Downloads," 2015. [Online]. Available: http://www.saforum.org/HOA/assn16627/images/SAIOverview.ppt. [Accessed 4 August 2015].

[15] Service Availability Forum, "Information Model Management Service SAI-AIS-IMM-A.03.01," September 2011. [Online]. Available: http://www.saforum.org/HOA/assn16627/images/SAI-AIS-IMM-A.03.01.AL.pdf. [Accessed July 2015].

[16] T. L. D. Project, "LDAP Linux HOWTO," August 2015. [Online]. Available: http://tldp.org/HOWTO/LDAP-HOWTO/whatisldap.html. [Accessed August 2015].

[17] A. C. C. S. James Warner, "top(1) - Linux man page," [Online]. Available: http://linux.die.net/man/1/top. [Accessed September 2015].

[18] F. F. Henry Ware, "vmstat," [Online]. Available: http://linuxcommand.org/man_pages/vmstat8.html. [Accessed September 2015].

[19] M. K. J. Larry Greenfield, "uptime(1) - Linux man page," [Online]. Available: http://linux.die.net/man/1/uptime. [Accessed September 2015].

[20] Python Software Foundation, "psutil 3.1.1 - Python Package Index," Python Software Foundation, 2014. [Online]. Available: https://pypi.python.org/pypi/psutil. [Accessed August 2015].

[21] Nixcraft, "Nixcraft," 27 June 2009. [Online]. Available: http://www.cyberciti.biz/tips/top-linux-monitoring-tools.html. [Accessed August 2015].

[22] OpenStack, "Home: OpenStack Open Source Cloud Computing Software," [Online]. Available: https://www.openstack.org/. [Accessed 22 November 2015].

[23] Red Hat, "Ceilometer Quick Start - RDO," 2015. [Online]. Available: https://www.rdoproject.org/install/ceilometerquickstart/. [Accessed 22 November 2015`].

[24] The LTTng Project, "The LTTng Documentation," 2015. [Online]. Available: http://lttng.org/docs/. [Accessed July 2015].

[25] Efficios Inc., "Babeltrace," Efficios Inc., 2015. [Online]. Available: http://www.efficios.com/babeltrace. [Accessed August 2015].

[26] A. Colangelo, "What is Cloud Computing? - Introduction to Cloud Computing," [Online]. Available: https://cloudacademy.com/cloud-computing/introduction-to-cloud-computing-course/. [Accessed November 2015].

[27] National Institute of Standard and Technology (NIST), "NIST Cloud Computing Program," 15 November 2010. [Online]. Available: http://www.nist.gov/itl/cloud/. [Accessed November 2015].

[28] Boundary, "Product- Boundary," 2014. [Online]. Available: http://www.boundary.com/product/. [Accessed July 2015].

[29] Rackspace, "Cloud Monitoring - Server, App & Website Monitoring by Rackspace," Rackspace, 2015. [Online]. Available: http://www.rackspace.com/cloud/monitoring/features/. [Accessed July 2015].

[30] Microsoft Corporation, "Enabling Diagnostics in Azure Cloud Services and Virtual Machines," Microsoft, [Online]. Available: http://azure.microsoft.com/en-gb/documentation/articles/cloud-services-dotnet-diagnostics/. [Accessed July 2015].

[31] AppDynamics, "Application Performance Management," AppDynamics, 2015. [Online]. Available: http://www.appdynamics.com/product/application-performance-management/. [Accessed July 2015].

[32] Aternity, "Aternity Workforce APM," Aternity, [Online]. Available: http://www.aternity.com/products/workforce-apm/. [Accessed July 2015].

[33] mOSAIC, "Towards a Cross Platform Cloud API Components for Cloud Federation," mOSAIC, [Online]. Available: http://mosaic-cloud.eu/dissemination/poster/1305227346_posterCLOSER11-1.pdf. [Accessed July 2015].

[34] K. Alhamazani, R. Ranjan, K. Mitra, P. P. Jayaraman, Z. (. Huang, L. Wang and F. Rabhi, "CLAMS: Cross-Layer Multi-Cloud Application Monitoring-as-a-Service Framework," in *IEEE International Conference on Services Computing (IEEE SCC)*, Anchorage, Alaska, USA, June 27- July 2, 2014.

[35] D. L. Quoc, L. Yazdanov and C. Fetzer, "DoLen: User-side multi-cloud application monitoring," in *International Conference on Future Internet of Things and Cloud (FiCloud)*, Barcelona, Spain, 27-29 August, 2014.

[36] Python Software Foundation, "psutil 2.2.1," Python Software Foundation, January 2015. [Online]. Available: https://pypi.python.org/pypi/psutil. [Accessed July 2015].

[37] Python Software Foundation, "About Python," Python Software Foundation, 2001-2015. [Online]. Available: https://www.python.org/about/. [Accessed August 2015].

[38] EfficiOS Inc., "Common Trace Format (CTF)," EfficiOS Inc., 2015. [Online]. Available: http://www.efficios.com/ctf. [Accessed July 2015].

[39] Python Software Foundation, "Data Structures," Python Software Foundation, 8 February 2015. [Online]. Available: https://docs.python.org/3/tutorial/datastructures.html. [Accessed July 2015].

[40] P. G. Emma, "Understanding some simple processor-performance limits," *IBM Journal of Research and Development - Special issue: performance analysis and its impact on design,* vol. 41, no. 3, pp. 215-232, May 1997.

[41] VMware Incorporated, "VMware Workstation," VMware Incorporated, 2015. [Online]. Available: https://www.vmware.com/products/workstation. [Accessed July 2015].

[42] Python Software Foundation, "BaseHTTPServer — Basic HTTP server," 23 May 2015. [Online]. Available: https://docs.python.org/2/library/basehttpserver.html. [Accessed July 2015].

[43] Apache Software Foundation, "Apache JMeter™," Apache Software Foundation, 2015. [Online]. Available: http://jmeter.apache.org/. [Accessed July 2015].

[44] VideoLAN non-profit organization, "VideoLAN," VideoLAN non-profit organization, [Online]. Available: http://www.videolan.org/. [Accessed July 2015].

[45] D. C. Johnson, "Kill Commands and Signals," Linux.org, 12 July 2015. [Online]. Available: http://www.linux.org/threads/kill-commands-and-signals.4423/. [Accessed July 2015].

[46] I. Bicking, "Virtualenv," The Open Planning Project, PyPA, 2014. [Online]. Available: https://virtualenv.pypa.io/en/latest/. [Accessed September 2015].

[47] Amazon Incorporated, "Amazon EC2 Instances," Amazon Incorporated, 2015. [Online]. Available: https://aws.amazon.com/ec2/instance-types/. [Accessed September 2015].

[48] OpenStack Foundation, "Flavors," OpenStack Foundation, 2015. [Online]. Available: http://docs.openstack.org/openstack-ops/content/flavors.html. [Accessed September 2015].

[49] Google Incorporated, "Google Cloud Platform: Linux Getting Started Guide," Google Incorporated, 2015. [Online]. Available: https://cloud.google.com/compute/docs/linux-quickstart. [Accessed September 2015].

[50] M. Desnoyers and M. R. Dagenais, "LTTng, Filling the Gap Between Kernel Instrumentation," [Online]. Available:

http://events.linuxfoundation.org/slides/lfcs09_desnoyers_paper.pdf. [Accessed 21 Nivember 2013].

[51] "LTTng Project," [Online]. Available: http://lttng.org/viewers. [Accessed 21 November 2013].

[52] T. Willhalm, R. Dementiev and P. Fay, "Intel® Performance Counter Monitor - A better way to measure CPU utilization," 16 August 2012. [Online]. Available: https://software.intel.com/en-us/articles/intel-performance-counter-monitor. [Accessed July 2015].

[53] M. Desnoyers, J. Desfossez and D. Goulet, "LTTNG," 18 July 2013. [Online]. Available: http://lttng.org/files/doc/man-pages/man1/lttng.1.html. [Accessed July 2015].