Automatic Configuration Generation for Service High Availability with Load Balancing

A. Kanso[1], F. Khendek[1], M. Toeroe[2], A. Hamou-Lhadj[1]

[1]Electrical and Computer Engineering Department, Concordia University,
Montreal, Canada
{al_kan, khendek, abdelw@ece.concordia.ca}
[2]Ericsson, Montreal, Canada
Maria.Toeroe@ericsson.com

Abstract
The need for highly available services is ever increasing in various domains ranging from mission critical systems to transaction based ones such as banking. The Service Availability Forum (SAForum) has defined a set of services and related API specifications to address the growing need of commercial-off-the-shelf high availability solutions. Among these services, the Availability Management Framework (AMF) is the service responsible for managing the high availability of the application services by coordinating redundant application components deployed on the AMF cluster. To achieve this task, an AMF implementation requires a specific logical view of the organization of the application's services and components, known as an AMF configuration. Any AMF configuration must be compliant to the concepts and constraints defined in the AMF specifications. Developing manually such a configuration is a complex error prone task that requires extensive domain knowledge. In this paper, we present an approach for the automatic generation of AMF compliant configurations from a set of requirements given by the configuration designer and the description of the software as provided by the vendor. The objective is to alleviate the need of configuration designers to acquire profound domain knowledge and deal with the complexity of handling large number of AMF entities and their relations. One important aspect of the AMF configuration is ranking the service units, when it is required by the redundancy model, for the assignment of the workload by AMF at runtime. Our approach includes a technique for generating these rankings in such a way that guarantees load balancing even after the occurrence of a failure.

Keywords: High Availability, Load Balancing, Redundancy Models, Clustered Systems, Availability Management Framework, Configuration Generation.

1 Introduction

High availability of a software system is achieved when the system's services are accessible (or available) to its users 99.999% of the time [1]. The Availability of a system depends mainly on its reliability and on the extent to which it can be rapidly repaired. Ensuring reliability is not always possible, and therefore the work towards sustaining highly available systems has shifted towards improving the recovery of the services provided by the system. The recovery of the system’s services is significantly improved by adding redundant components that can take over the services provided by faulty components [1].

The Service Availability Forum (SAForum) [2] is a consortium of telecommunications and computing companies working together to define and standardize high availability solutions for systems and services. The SAForum has developed an Application Interface Specification (AIS), which includes the Availability Management Framework (AMF) [3]. The role of AMF is to manage the availability of the services provided by an application. This is achieved through the management of its redundant components and by shifting dynamically the workload assigned to a faulty component to a redundant and healthy one when a fault is detected or reported.

The AMF service requires a configuration for any application it manages. An AMF configuration can be seen as an organization of some logical entities. It also comprises information that can guide AMF in assigning the workload to the application components. AMF managed applications are typically deployed as distributed system over a cluster of nodes, load balancing is therefore an important aspect to be considered when designing the configuration.

Designing an AMF configuration requires a deep understanding of AMF entities, their relations, and their grouping. This grouping is guided by the characteristics of the software that will be deployed in an AMF managed cluster. These characteristics are described by the software vendor in terms of prototypes delivered in an Entity Types File (ETF) [4].

The design of an AMF configuration consists of selecting a set of AMF entity types from a set of ETF prototypes, specifying the entities and their attributes in order to provide and protect the services as required by the configuration designer. Creating manually such a configuration can be a tedious and error prone task due to the large number of required types, entities and attributes. This is combined with the complexity of selecting the appropriate ETF prototypes and deriving from them the necessary AMF types to be used in the configuration. During the type selection and derivation, several constraints need to be satisfied. Some of these constraints require calculations and extensive consistency checks. Moreover, the configuration ideally needs to be designed in such a way that load balancing is preserved even in case of failure.

In this paper, we present a technique for the automatic generation of AMF configurations from ETF(s) to provide and protect the services as required by the configuration designer, for a given cluster of nodes. We describe an algorithm to select the appropriate ETF prototypes and the method of deriving the appropriate AMF types. The configuration generation completes with populating the configuration with entities of the AMF types and their attributes. For some redundancy models, one of these attributes requires complex calculations and has a big impact on load balancing. For these cases, we also present a method for determining this attribute such that the distribution of the workload is balanced before and after a failure.

The remaining part of this paper is organized into seven sections. In Section 2, we introduce AMF and the concepts needed in this paper. In Section 3, we introduce our approach of generating automatically AMF configurations. In Section 4 we present our method for load balancing followed, in Section 5, with a discussion summarizing some issues not covered in this paper. The related work is reviewed in Section 6. We conclude in Section 7 and present some future directions.

2 Availability Management Framework

To provide fault tolerance by clustered systems, applications are composed of several processes running on different nodes in a redundant manner. The role of the AMF service is to protect the services provided by these applications by managing the redundancy of their processes. For this purpose, these application processes need to be organized and described according to the following logical view.

The smallest entity AMF manages is the *component*. It represents a set of hardware and/or software resources that provide some functionality. AMF manages each component through APIs. A *service unit* (SU) is a logical entity that consists of one or more components that combine their functionalities into some service. The workload associated with providing or protecting some functionality and which can be assigned to the component is represented by a *component service instance* (CSI). A set of CSIs requited for a service that can be assigned to the components of the same SU is represented by a *service instance* (SI). Thus, the SI is the workload that is assigned to an SU by AMF at runtime either to actively provide the service represented by the SI or protect it as a standby.

AMF maintains the availability of the SIs by managing their assignments among a set of redundant SUs. For this purpose, SUs are grouped into service groups. A *service group* (SG) consists of a set of SUs that collaborate to protect a set of SIs. They collaborate according to a certain redundancy model. There are five different redundancy models: 2N, N+M, N Way, N Way Active, and No-Redundancy. These redundancy models differ in the number of active and standby state assignments each SI may have and their distribution of these assignments among the SUs. The five redundancy models can be summarized as follows:

- The *2N* redundancy model specifies that in an SG at most one SU will have the active HA (High Availability) state for all SIs protected by this SG and it is referred to as the active SU, and at most one SU will have the standby HA state for all SIs and it is called the standby SU.

- An SG with the *N+M* redundancy model is similar to 2N, but has N active SUs and M standby. An SU can be either active or standby for all SIs assigned to it. That is to say, no SU can be simultaneously active for some SIs and standby for some other SIs. An SI can be assigned to at most one SU in the HA active state and to at most one SU in the HA standby state.

- In the *No-Redundancy* redundancy model we have no standby SUs, but we can have spare SUs, i.e. available for assignment. An SI can be assigned to only one SU at a time. An SU is assigned the active HA state for at most one SI.

- An SG with the *NWay* redundancy model contains N SUs that protect multiple SIs. An SU can simultaneously be assigned the active HA state for some SIs and the standby HA state for some other SIs. At most, one SU may have the active HA state for an SI, but one, or multiple SUs may have the standby HA state for the same SI.

- An SG with the *NWayActive* redundancy model contains N SUs. An SU has to be active for all SIs assigned to it. An SU is never assigned the standby HA state for any SI. From the service side, for each SI, one, or multiple SUs can be assigned the active HA state according to the preferred number of assignments, *numberOfActiveAssignments*, configured for the SI. The *numberOfActiveAssignments* should always be less or equal to the number of SUs in the SG.

An AMF *application* is composed of one or more SGs and the SIs they provide.

There are two additional AMF logical entities used for deployment purpose: The cluster and the node. The cluster consists of a collection of nodes under the control of AMF.

Figure 1 illustrates an example configuration of an application and the services it provides. The application consists of one SG protecting three SIs according to the *NWayActive* redundancy model. Here, each SI has two active assignments. Each active assignment is assigned to a different SU. The CSIs of the SIs represent the workload assigned to the components and each SU has two components, which can take these assignments. At runtime AMF distributes the SI assignments among the SUs based on the SUs' ranking for each SI, more details are presented in Section 4. Since each SI has two active SUs, the SI is immune to any single component, SU or node failure, as the other SU can carry on the provision of the service abstracted by the SI.

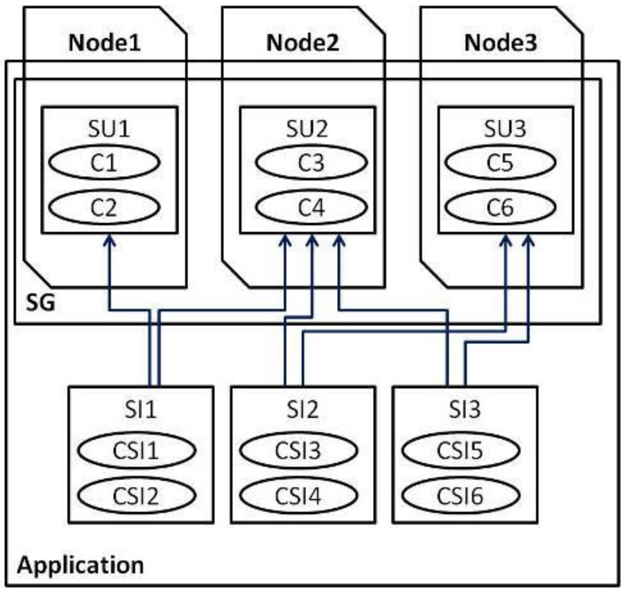


Figure 1. An example of AMF configuration

AMF entities, except the cluster and nodes, are typed. An entity type represents a particular implementation (e.g. a software version) and therefore it provides AMF with the information about the shared characteristics of its entities. For example, the component type indicates the component service types any component of this type can provide. AMF uses this information to determine to which component within an SU to assign a particular CSI. Therefore, when building an AMF configuration, we need to provide the AMF type for each of the entities. Table 1 lists the AMF configuration entities and their corresponding types.

AMF types are derived from the prototypes defined in the Entity Types File (ETF) [4], which is a standardized XML file provided by the software vendor and which describes the application software developed to be managed by an AMF implementation. ETF prototypes, their relations, and main characteristics can be summarized as follows.

Component Type[1]: A component type describes a particular version of a software implementation designed to be managed by AMF. The component type specifies the component service types (i.e. the functionalities) that the components of this type can provide. It defines for each component service type the component capability model and any dependency on the functionality of other component types. The component capability model characterizes the total workload a component of the type can take for a given functionality. It is described as a triplet `(x, y, b)`, where `x` represents the maximum number of active CSI assignments, `y` the maximum number of standby CSI assignments a component can handle for a particular component service type and `b` indicates whether it can handle active and standby assignments simultaneously.

Component Service Type (CS type): It describes the set of attributes that characterizes the workload that can be assigned to a component of a particular type in conjunction of providing some functionality.

Service Unit Type (SU type): The service unit type specifies the service types an SU of the type can provide, and the set of component types from which an SU of the type can be built. It may limit the maximum number of components of a particular type that can be included. Thus, the SU type defines any limitation on the collaboration and coexistence of component types within its SUs.

Service Type: A service type defines the set of component service types from which its SIs can be built. The service type may limit the number of CSIs of a particular CS type that can exist in a service instance of the service type. It is also used to describe the type of services supported by an SU type.

Service Group Type (SG type): The service group type defines for its SGs the redundancy model. It also specifies the different SU types permitted for its SGs. Thus the SG type plays a key role in determining the availability of services.

[1] Although the ETF file defines a set of prototypes, they are still named types, but they act as meta-types from which the AMF types are derived or built.

Application Type: The application type defines the set of SG types that may be used to build applications of this type.

It should be noted that the only mandatory types in an ETF are the component type and the component service type. Further types (i.e., SU types, SG types, service types and application types) may be used by the vendor to specify software implementation constraints and limitations to be considered at the integration of this software with an AMF implementation, but they are not mandatory. If no optional type is provided at a particular level (e.g. there is no SU type), then types of the level below (i.e. the component types) can be combined as needed without any restriction. If any optional type is provided at a particular level, it is assumed that all allowed types at that level are specified by the vendor [4]. Table 1 summarizes the concepts introduced by AMF.

Table 1. The AMF configuration elements.

Entity	Acronym	Brief description	Corresponding type
Component	−	A set of hardware and/or software resources	Component type
Service Unit	SU	A set of collaborating components	SU type
Service Group	SG	A set of SUs protecting a set of SIs	SG type
Application	−	A set of SGs and the SIs they protect	Application type
Service Instance	SI	The workload assigned to the SU	Service type
Component service Instance	CSI	The workload assigned to the component	CS type
Node	−	A cluster node	−
Cluster	−	The cluster hosting AMF and the applications	−

3 AMF Configuration Generation

The configuration generation method takes input from two sources: The configuration designer and the software vendor.

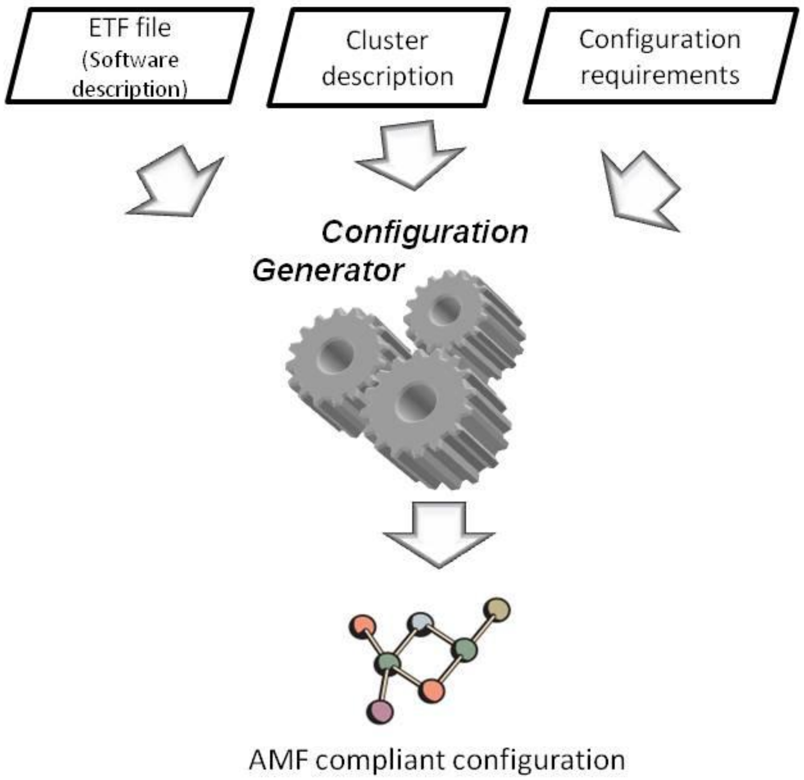


Figure 2. Configuration generation overview.

The configuration designer defines the services to be provided in terms of SIs and CSIs together with their desired protection i.e., the redundancy model. We refer to this as configuration requirements. The configuration designer provides also the information about the deployment cluster and its nodes, e.g., the number of nodes. The AMF cluster entity and the node entities can be added to the configuration without any further processing. The software to be used to provide the required services is characterized by each vendor in an ETF as described in the previous section.

To facilitate the specification of the required services and the cluster, we define templates: An SI-template represents the SIs that share common features including the service type, the desired redundancy model: (e.g., 2N, N+M, etc.), the number of active and standby assignments, and the associated CSI-templates. A CSI-template defines the CS type with its number of CSIs. The same concept applies for a node-template.

Once the services, cluster description and the software are specified, the configuration generation method proceeds with selecting the appropriate ETF types for each required service. In the next sections, we present how component, SU, SG, and application types are selected.

Figure 3 illustrates how each portion of the input is used during the configuration generation. The ETF content will be processed in order to find the proper types that will be used to derive or build the AMF types. The cluster description is going to be directly used to create the nodes.

Finally, the information in the SI and CSI template will be processed to be used for two main purposes (1) to create the SIs and CSIs and (2) to select the appropriate ETF types that can support these services[2]. The configuration generator will create the rest of the AMF entities based on the calculations performed on the information that is extracted from the input.

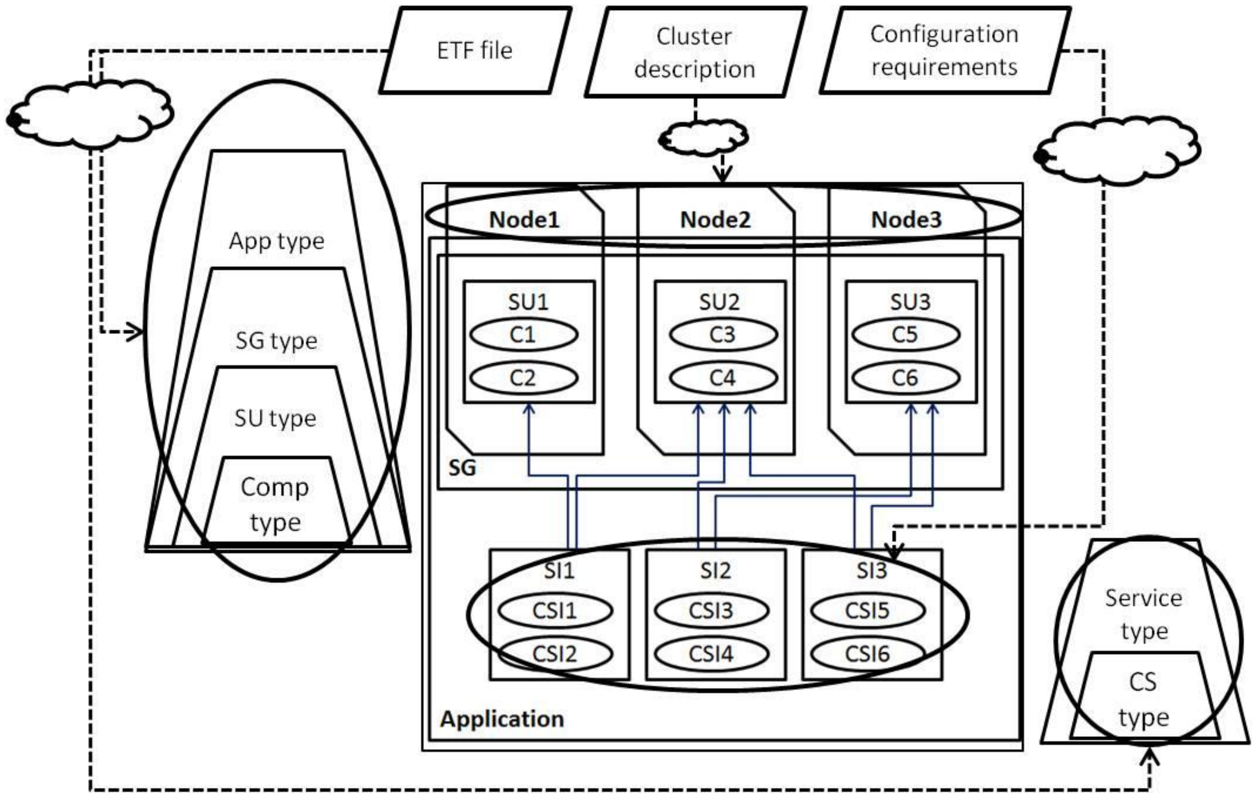


Figure 3. Input usage for generating AMF configurations.

3.1 Workload Calculation

The SI template specifies the required redundancy model, and the number of SUs that will protect the SIs of this template. For instance if the specified redundancy model is 2N, then the SG protecting the SIs will only have one active SU and one standby, and thus any SU of this SG must be capable of supporting all the SIs in their active or standby HA state. The capacity of the SU in supporting the SIs is determined by two factors: the capacity of its components in supporting the CSIs of the SIs, and the number of components that can be grouped in this SU. The component capacity in supporting the CSIs is specified in the component type's capability model defined for the CS type of the CSIs. The maximum number of components of a particular type that can be grouped within an SU is specified by the SU type of the SU. In other words, the types hold all the required information about the capacities of the SUs and their components.

[2] The terms services and workload are used interchangeably but the context of configuration generation they bear the same meaning, which is the SIs and CSIs that will be assigned by an implementation of AMF to components and SUs.

Therefore, an essential criterion in selecting any type is the expected load of SIs or CSIs an entity of this type is expected to handle within the SG.

3.1.1 Service Unit Load Calculations

The load of SIs each SU needs to handle depends on the total number of SIs the SG has to protect, the protection it needs to provide for each SI, i.e., the number of active and standby assignments, and the number of SUs among which these assignments are distributed.

More precisely, for each SI-template, we determine the minimum expected load of an SU for the active and standby assignments as defined by the redundancy model using Equations 1 and 2, respectively, where:

- `numberOfSUs` refers to the total number of SUs of the SG that will protect the SIs of the template,
- `numberOfSIs` refers to the number of SIs specified in the SI template,
- `numberOfActiveSUs` refers to the number of SUs that have active assignments only,
- `numberOfStdbSUs` specifies the number of SUs that have standby assignments only,
- `numberOfActiveAssignments` refers to the number of active assignments for each SI,
- `numberOfStdbAssignments` refers to the number of standby assignments for each SI, and
- `redMod` is used for the redundancy model specified in the SI template.

Equation 1. The load of active SI assignments for an SU.

$$
ActiveLoadOfSIs = \begin{cases}
\mathrm{ceil}\left(\frac{numberOfSIs}{numberOfSUs}\right) & if\ \ redMod = "N\ Way" \\
\mathrm{ceil}\left(\frac{numberOfSIs \times numberOfActiveAssignment}{numberOfActiveSUs}\right) & if\ redMod = "N\ Way\ Active" \\
1 & if\ redMod = "NoRedundancy" \\
numberOfSIs & if\ redMod = "2N" \\
\mathrm{ceil}\left(\frac{numberOfSIs}{numberOfActiveSUs}\right) & if\ redMod = "N + M"
\end{cases}
$$

Equation 2. The load of standby SI assignments for an SU.

$$
StandbyLoadOfSIs = \begin{cases}
\mathrm{ceil}\left(\frac{numberOfSIs \times numberOfStdbAssignment}{numberOfSUs}\right) & if\ redMod = "N\ Way" \\
0 & if\ redMod = "N\ Way\ Active" \\
0 & if\ redMod = "No\ Redundancy" \\
numSIs & if\ redMod = "2\ N" \\
\mathrm{ceil}\left(\frac{numberOfSIs}{numberOfStdbSUs}\right) & if\ redMod = "N + M"
\end{cases}
$$

The above equations use the characteristics of the various redundancy models to determine the load in terms of the number of SI assignments an SU is expected to handle. For example, for the N-way redundancy model, the load of active SI assignments for an SU is calculated as the total number of SIs of an SG of the SI-template divided by the number of SUs since an SI is assigned active to at most one SU and all SUs may handle active assignments. To guarantee that there is always an SU, which can take over the active load of a failed SU, we may use `numberOfSUs – 1` as the number of SUs. For the same redundancy model, the number of SIs an SU can handle in a standby state (in Equation 2) is the number of standby assignments of the SIs divided by the number of SUs. The number of standby assignments of all SIs is calculated as `numOfSIs x numberOfStdbAssignments` since, in an N-way redundancy model, an SI can have multiple standby assignments.

We apply the same principle to other redundancy models and compute the load of SIs assigned to an SU in both active and standby states.

3.1.2 Components Load Calculation

AMF assigns the CSIs to the components based on the capability of the component type of the components to support the CS type of the CSIs. So, basically within the same SU, if two components of two different types can handle the CS type of a CSI, we cannot determine at configuration time to which component AMF will assign the CSI. In order to avoid having the CSIs of a particular CS type being assigned to a component of a component type that was not originally selected to support these CSIs, we opt to create our SUs in such a way that within the types of the component of the same SU, no two component types will overlap in the CS types they can provide. In other words the CSIs of the SI that share the same CS type will be handled by the SU components that share the same component type. This decision affects the component type selection. Since now, the load of CSIs of the same CS type cannot be shared among the components of different types, which puts all the load of CSIs on the same type on one component type. Therefore, an essential criterion for the component type selection is its capacity of handling the CSIs load. The question now is how to calculate this load?

The load of CSIs of a particular CS type that the components of the same SU will handle depends on two factors: (1) the number of SIs the SU will handle and (2) the number of CSIs of this CS type each of these SIs contain. Based on the assumption that within the CSI-templates of an SI template, each CSI-template has a distinct CS type, the load of CSIs of a particular CS type the components of the same SU must handle can be calculated according to Equation 3 where:

- `numberOfCSIs` refers to the number CSIs specified by the CSI-template for which we want to find the proper component type,
- `ActiveLoadOfSIs` refers to the result of Equation 1, and
- `StandbyLoadOfSIs` refers to the result of Equation 2.

Equation 3. The load of CSIs a component type must handle.

$$\boxed{\begin{aligned} ActiveLoadOfCSIs &= ActiveLoadOfSIs \times numberOfCSIs \\ StandbyLoadOfCSIs &= StandbyLoadOfSIs \times numberOfCSIs \end{aligned}}$$

3.2 Component Type Selection Criteria

To select an ETF component type, we match the CS type of the CSIs as specified in the CSI-template to one of the CS types of the component type. We also need to verify that the component capability model of the type is compliant with the one required by the redundancy model specified in the SI-template the CSI-template belongs to. Depending on the required redundancy model and the component capability model, certain component types may be more suitable than others, while other component types may not be usable at all. For example, a component type that can have only active or only standby assignments at a time can be used in an SG of the 2N redundancy model, but it is not valid for an SG with N-way redundancy model where the components may have active and standby assignments simultaneously.

In addition, if the component type is checked as part of an SU type, the constraint imposed by the SU type on the component type needs to be verified: The maximum number of components in an SU of this SU type must not be exceeded. This limits the capacity of the SU for a component service type. Therefore, we need to ensure that the maximum number of components of the selected component type can provide the capacity necessary for the load of CSIs, as calculated in Equation 3, imposed by the load of SIs that are expected to be assigned to the parent SU.

3.3 Selecting SU, SG, and Application Types

To determine the suitable SU, SG, and application types, we use an algorithm for searching from the highest defined level of types (i.e., the application type).

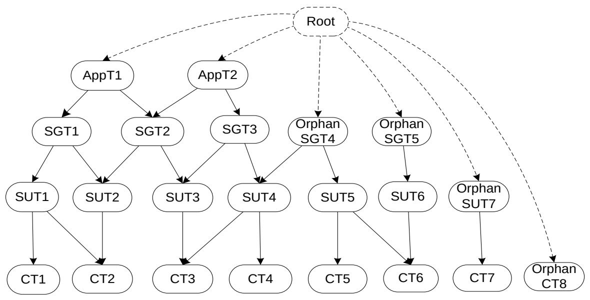


Figure 4. An Example of ETF.

ETF types can be structured in the form of a “disconnected non-ordered” directed acyclic graph (DAG) as shown in Figure 4. The nodes of the DAG represent ETF types, whereas the edges represent the relations among these types.

Except for the application type, all other types can have more than one parent type; and since not all types are mandatory in ETF, some types might come with no parents. For example, SG types SGT4 and SGT5 of Figure 4 do not have a parent application type. We refer to them as *orphan* types. We added a new node called `Root` to connect the sub-graphs of the DAG to facilitate type searching.

The algorithm for selecting ETF types is given in Figure 5. The function `SelectETFTypes` takes the DAG as input (i.e., its root node) and an SI-template, and returns as output a set, called `Types`, which contains the ETF types - if there are any - to provide the services defined by the SI template and that satisfy the constraints that applies to these types, that we refer to as *requirements*. Note that these requirements can change depending on the visited type. For instance, an orphan type may not have the same limitations as a non-orphan one. Moreover, the satisfaction of these requirements may depend on the parent type, e.g. the maximum number of components defined by the SU type and not by the component type. Therefore, we may need to visit the same node several times.

The initialization step sets `Types` to an empty set. The `SelectETFTypes` function is a recursive function that performs a depth-first search traversal of the DAG starting from the `Root` node to the component types. More precisely, it starts by visiting the first child of the `Root` node, which in our case, consists of an application type if provided. It checks if this application type satisfies the configuration requirements. If this is the case, then it checks if it is a leaf node and since it is not it continues by recursively visiting the application type's SG types. Otherwise, the algorithm proceeds to the next child of the `Root` node which could be another application type, an orphan SG type, an orphan SU type or an orphan component type. If it finds an SG type with a redundancy model identical to the one specified in the SI-template, its SU types are visited.

An SU type is selected if it provides the service type specified in the SI-template and has the capacity, through the components of its component types, to handle the load of SIs (and their CSIs) expected to be assigned to an SU of this type (see Section 3.2). To determine this, the component types of the SU type also need to be visited. When the first component type is reached and satisfies the requirements as described in Section 3.1, it also satisfies the leaf condition. This component type is added to the selected Types set, and to verify that service type of the SI template can be satisfied a new set is initialized with the CS types of this child and it is checked if the required service can be provided. If it cannot be provided yet, each sibling component type is checked and if satisfies the requirements, then the type itself and its provided CS types are added to the appropriate sets. After each sibling we verify if the service type of the SI template can be provided. If so, we return from the component type level. If we could not satisfy the service type after visiting all the siblings, then we clear the collected types and backtrack to the SU type level (or `Root` for orphan components) and select a new SU type.

```
Types = {}
SelectETFTypes(Node N, SI-template SItemp)
1. For each child of N do
2.  If child satisfies requirements then
3.   If child is leaf then
4.    Types = {child}
5.    ST = {all provided CSTs of child}
6.    If SItemp.ST ⊆ ST then        Return
7.    For each sibling of child do
8.     If sibling satisfies requirements then
9.      Types = Types ∪ {sibling}
10.      ST=ST ∪ {all provided CSTs of sibling}
11.      If SItemp.ST ⊆ ST then Return
12.    Enddo
13.    Types = {}
14.    Return
15.   Else
16.    SelectETFTypes(child, SItemp)
17.    If Types ≠ {} then
18.     Types = Types ∪ {child}
19.     Return
20. Enddo
End SelectETFTypes
```

Figure 5. The algorithm for ETF types selection.

On the return path depending on whether the Types set is empty or not we either continue the search with the next child, or add the current child to the Types set and return to the level above. When the algorithm terminates the Types set may be: (1) a complete type set, i.e. {AppType, SGType, SUType and CompTypes}, (2) a partial type set where some types are missing, e.g. {SUType and CompTypes}, or (3) empty.

If the Types set is empty, this means that the algorithm could not find any types to satisfy the configuration requirements. Therefore, no configuration can be generated to provide the requested services using the software described by the given ETF(s). In the case where the Types set is a complete set, we have all the necessary types.

```
CreateTypes(TypeSet Types, SI-template SItemp)
1. lastType = orphan types in Types
2. While AppType ∉ Types do
3.  Build newType from lastType for SItemp
4.  Types = Types ∪ {newType}
5.  lastType = newType
6.  Enddo
End CreateTypes
```

Figure 6. The algorithm for type creation.

For the second case in between, the creation of some missing types is required. We use the simple algorithm presented in Figure 6 to create the missing types. It selects the orphan type(s) in `Types` and as long as there is no application type in `Types` it builds a new type from the last selected type(s) according to the SI- template. In [6], we presented a method that targeted this type creation. Here we restrict that method by allowing type creation only from orphan types.

3.4 Completing the Configuration

Once ETF types are selected (or created), we need to derive the corresponding AMF types, and generate the corresponding AMF entities.

AMF types are created from the selected ETF types by deriving their attributes from the attributes provided in the ETF prototypes. In most cases this means using the default value provided for the type. In other cases, the AMF type is tailored to the intended use. For example, if an ETF component type can provide CST1 and CST2 CS types, but the type was selected to provide only one of them (e.g. CST1), in the AMF type only this component type is listed as provided CS type.

For created types, the AMF type attributes are derived based on the dependencies that can be established among the used ETF prototypes. For example, the default instantiation level of ETF component types of a created SU type is calculated based on the dependency of their provided CS types if any.

Once all the AMF types have been supplied, their entities are created. For this again for each SI-template the number of components is determined within each SU of the selected SU type. This is based on the load of SIs expected to be assigned to the SU (Equations 1 and 2), the number of CSIs within these SIs, as well as the capability of the components, specified in the component type of these components. For example, if the SU can handle one SI, which contains four CSIs in a 2N redundancy model and that the component capability is 2 active and 3 standby then the number of components for this SU should be 4/2 = 2. In other words, we only need two components to be active for the four CSIs. We also need two components of this type in the standby SU to handle the four CSIs (using the ceiling of 3/2).

From the first created SU we duplicate all the necessary SUs to create the SG, which might also be duplicated to satisfy all the SIs of the template. We repeat the process for each SI-template. SGs are combined into applications based on the selected application types.

So far we have created the skeleton of the AMF configuration. That is, we have created all the types and the entities; however we still need to populate the remaining attributes of the AMF entities. The values of these attributes come from three sources: (1) ETF, for instance the types' attributes values are mostly derived from the values specified in ETF (2) The configuration designer, for example the SI and CSI attribute values (3) The configuration generator, in the sense that some values must be computed.

Among the computed values, some are calculated in a straightforward manner while others require deeper analysis. The SU ranking for SIs is one of the latter attributes. It is a significantly important attribute that must be carefully set in order to ensure SI load balancing among the SUs of the same SG before and after failure. Section 4 discusses in details the importance of this attribute and our approach to calculate it.

4 Ranking SUs for SIs

As aforementioned, the workload assigned to the SUs is abstracted by SIs, and assigned to SUs by AMF at runtime. SUs collaborate to protect SIs within an SG, that is, when an SU fails, only an SU(s) of the same SG can take-over the load originally assigned to the failed SU. Two issues arise in the context of the workload assignment: (1) the SIs must be evenly distributed among the SUs of the same SG (2) when an SU fails, the workload assigned to this SU should be evenly redistributed among the other SUs of the SG in order to avoid a load misbalance.

In the *2N* redundancy model, we only have one active SU per SG and one standby, therefore when the active one fails the entire load is shifted to the standby one, and therefore we have no load balancing issue. The same applies to the '*no redundancy'* redundancy model, since each SU is only allowed to be assigned one SI. However in *NWay* and *NWayActive* redundancy models, we have several SUs that can be assigned several SIs. The SI assignment is performed by AMF at runtime based on a ranked list of SUs for each SI, determined at configuration time[3]. The example configuration shown in Figure 1 shows an SG of three SUs, protecting three SIs according to the *NWayActive* redundancy model. Each SI has been configured to have two active assignments. The runtime active assignment shown in Figure 1 corresponds to the ranked list of SUs configured for each SI as shown in Table 2.

Table 2. The ranked list of SUs for SIs.

	SU1	**SU2**	**SU3**
SI1	Rank=1	Rank=2	Rank=3
SI2	Rank=3	Rank=1	Rank=2
SI3	Rank=3	Rank=2	Rank=1

The ranking works according to the following rules:

(1) Each SI has a ranked list of all the SUs in the SG [3].

(2) The number of SUs that will be assigned the active/standby state is defined by the preferred number of active/standby assignments configured for the SI. The SUs with the highest ranks (lowest integer values) will be assigned the active/standby state on behalf of the SI.

(3) In case of an SU failure, the SU with the next highest rank will get an assignment. For example, according to Table 2, if SU1 fails then SI1 will lose one of its active assignments. AMF will restore this active assignment by assigning the SI to another SU. SU3 will get the active assignment for SI1 since SU2 already has an assignment and SU3 has the next highest rank. Note that Table 2 illustrates one possible ranking; other

[3] AMF does not specify how the SIs are assigned in case of N+M redundancy model

rankings are possible and other equivalent rankings could lead to the same assignment shown in Figure 1.

Since the ranking will determine the workload assignments, then we cannot specify the ranks in an ad-hoc manner.

4.1 Conventional Load Balancing Algorithms at Configuration Time

In this section we examine the problem of load balancing after a failure, using the conventional round robin algorithm. The rationale behind choosing round robin as a reference to compare our algorithm with is discussed further in Section 5.4.

As a case study we will discuss a scenario with 14 SIs protected with an SG of 6 SUs with an *NWayActive* redundancy model. The *numberOfActiveAssignments* for each SI is set to 3, i.e. each SI is configured to have 3 active SUs.

Table 3. A ranked list of SUs using a round robin algorithm.

	SU1	**SU2**	**SU3**	**SU4**	**SU5**	**SU6**
SI1	**1**	**1**	**1**	2	3	4
SI2	2	3	4	**1**	**1**	**1**
SI3	**1**	**1**	**1**	2	3	4
SI4	2	3	4	**1**	**1**	**1**
SI5	**1**	**1**	**1**	2	3	4
SI6	2	3	4	**1**	**1**	**1**
SI7	**1**	**1**	**1**	2	3	4
SI8	2	3	4	**1**	**1**	**1**
SI9	**1**	**1**	**1**	2	3	4
SI10	2	3	4	**1**	**1**	**1**
SI11	**1**	**1**	**1**	2	3	4
SI12	2	3	4	**1**	**1**	**1**
SI13	**1**	**1**	**1**	2	3	4
SI14	2	3	4	**1**	**1**	**1**

Table 3 shows a ranked list of SUs generated using a round robin algorithm. According to this ranking, at runtime, AMF will assign the HA active state for each SI in the following manner:

- SU1, SU2 and SU3 will be assigned the HA active state for SI1, SI3, SI5, SI7, SI9, SI11, SI13, and
- SU4, SU5 and SU6 will be assigned the HA active state for SI2, SI4, SI6, SI8, SI10, SI12, SI14.

The cells marked in blue correspond to the active assignments. Following this ranking, the load will be evenly distributed among the SUs. Each SU will have 7 active assignments. In absence of failure, a round robin algorithm solves the problem of ranking with load balancing.

Now let us assume that SU1 (or SU2 or SU3) fails, the active assignment of each SI assigned to SU1 will be shifted to the SU with the next higher rank and which is not already active for the SI. Therefore, according to the ranking in Table 3, all the workload of SU1 would be shifted to SU4 causing it to bear twice the load of any other SU. In this case SU4 is seen as a “backup” for SU1 with respect to all its assignments.

The shortcoming of the round robin algorithm is that it eventually develops a repetitive pattern as shown in Table 3. After SI2, the pattern keeps repeating itself every two SIs. SU1 and SU4 are the only SUs that are backing up the other SUs in case of failure as illustrated in Figure 7, which is derived from Table 3. SU1 is backing up SU4, SU5 and SU6, while SU4 is backing up SU1, SU2 and SU3.

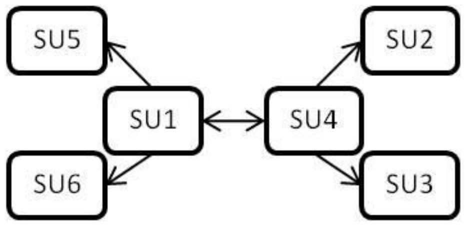


Figure 7. SU backup graph using the round robin algorithm.

4.2 Load Balancing Before and After one Failure

The problem in hand can be viewed as follows. We have a predefined set of SIs that will be assigned to a set of SUs. The portion of SIs that each SU is supposed to serve can be easily computed, and a corresponding ranking can be produced. However, one main concern as shown in the previous section is what happens after an SU failure. How will the load assigned to the failed SU be distributed among the remaining SUs? We want the SUs to be assigned equal initial load, and in case of failure, we want the load of the failed SU to be evenly assigned among the remaining SUs to maintain a balanced load and avoid cascading failures caused by overload. Our main objective is to capture and ensure this at configuration time through the SU ranking for SIs.

As shown earlier the problem of the round robin algorithms is in the re-assignment of the SIs to the remaining SUs. Only one SU, e.g. SU4 will be re-assigned all the SIs initially assigned to SU1 when the later fails. Therefore, the solution to this problem of load balancing after single failure must be tackled from the perspective of SUs backing up each other for SIs and make sure that SIs assigned to a particular SU are equally backed up by the remaining SUs.

Each SU can have at most one active assignment with respect to a particular SI. So, if an SU has *X* active assignments, then it is assigned *X* different SIs. We want to ensure that these *X* SIs are backed up evenly by the other SUs. In order to achieve this, we start by assigning each SU an equal number of SIs to backup, then we determine the SUs that will be active for those SIs, and finally we generate our ranked list of SUs for each SI.

The solution we are presenting in this section is for the *NWayActive* redundancy model, but the concept readily applies for the *NWay* redundancy model if it is properly adapted as we will discuss it at the end of this section. This solution is based on the following assumptions:

- The SUs have the capacity to support the SIs that AMF assigns to them even after an SU fails and its load is shifted to the other SUs.
- The SIs of the same SG have the same protection level, and therefore the *numberOfActiveAssignments* is the same for all of them.
- The SIs impose the same load on the SUs.
- There are at least two SUs in the SG, and they are all in service, that is, we have no spare SUs.
- The *numberOfActiveAssignments* is less than the number of SUs.

Systems managed by AMF intended to have no single point of failure and are expected to tolerate the failure of one SU without causing a service to be dropped. Therefore, we are targeting here a single failure. Load balancing after multiple failures is outside the scope of this paper.

Our approach consists of four steps:

1. Determine the total number of assignments to backup, or simply backup assignment[4], each SU will have,
2. Distribute the total backup assignments of each SU equally among the other SUs,
3. Balance the total number of active assignments for all SUs, and
4. Derive the ranked list of SUs for each SI from the assignment table.

For the *NWayActive* redundancy model one or many SUs will be assigned the active HA state on behalf of an SI. However, only one SU will serve as a backup for this SI if any of its active SUs fails. In this case we say that the backup SU is backing up the active SUs in terms of this SI. If this particular SI requires *x* active SUs where *x* is the *numberOfActiveAssignments*, we say that the backup SU is backing up *x* active assignments. It is important here to distinguish that although the backup assignment is in terms of SIs, we bring the *numberOfActiveAssignments* each SI has into the equation because if one SU is backing up an SI this means it is backing up all of its active assignments, while on the other hand being active for an SI means having a maximum of one of its active assignments.

[4] The backup assignment is not to be confused with the standby HA state assignment.

Equation 4. Backup assignment for each SU.

$$Backup = \begin{Bmatrix} \left\lfloor \dfrac{numberOfSIs}{numberOfSUs} \right\rfloor \times numberOfActiveAssignments \\ or \\ \left\lceil \dfrac{numberOfSIs}{numberOfSUs} \right\rceil \times numberOfActiveAssignments \end{Bmatrix}$$

A prerequisite for ensuring back up balancing is that each SU must back up an equal number of SIs. The backup value of an SU is given by Equation 4. Since the number of SUs is not always a divisor of the *numberOfSIs*, some SUs will get the floor of this division while others will get the ceiling with the constraint that the sum of the backup assignments for all SUs is equal to *numberOfSIs* * *numberOfActiveAssignments*. Of course, an SU does not back up its own active assignments, but the active assignments of the other SUs as we will see it in Table 4.

In order to ensure backup balancing, it is not enough that the SUs backup the same number of active assignments, because if all those active assignments are provided by one SU, and this SU fails, its entire load will go to the backup SU that is already active for its own SIs. We will end up in the same pitfall of the round robin algorithm. Therefore, the number of active assignments backed up by an SU must be the sum of equal contributions ± 1 from all the other SUs. In other words, if the SU is calculated to have *x* back up assignments, and we have *n* SUs, then we make sure that the SU will back up each of the other SUs in $x \div (n\text{-}1)$ active assignments. This division may result in a decimal value; some SUs may get an extra back up from the SU in question.

In order to render the solution more concrete, let us visualize our problem in terms of an assignment table as shown in Table 4, which shows simultaneously the active and backup assignments.

Table 4. Distributing the assignments of each SU.

	SU1	…	SU_j	…	SU_m	Backing Up
SU1	N/A					
…		N/A				
SU_i			N/A			
…				N/A		
SU_m					N/A	
Act						–

Table 4 represents the blueprint of our assignment scheme. A value *x* in cell SUij represents a number of assignments. However this value can either mean the number of active assignments, or the number of backup assignments. The SU rows represent the number of backup assignments each SU will have for the other SUs. The cells denoted with N/A (not applicable) means an SU cannot back up itself. A value *x* in cell SUij means that the SUi will back up SUj in *x* of SUj's active assignments. The "*Baking Up*" column is used to hold the value of the total backup assignments that we calculate with Equation 4 for each SU. We assign the SUs cell values of any row *i* in such a way that (1) their sum is equal to the "*Backing up*" cell value in row *i* (2) the cell values can only be the floor or the ceiling of the baking up value after it is divided by (the number of SUs -1). In other words if SUi is backing up *x* active assignments, we want those assignments to be equally distributed among other SUs.

The "*Act*" row is used to hold the values of the total active assignments each SU is expected to have. It is the sum of the column cells values. Note that the SUs of the columns and the rows of the table are identical. We simply used different indexing (*i* for row and *j* for column) to avoid ambiguity.

So far we have balanced the backup assignments. However, the total active assignments that each SU is handling from previous calculations may be uneven, i.e. that values of the *'Act'* row may be imbalanced. In order to solve this issue we need to make sure that the SUs have equal active assignments.

If we have a number of SIs each having a *numberOfActiveAssignments* to be assigned to the same number of SUs, the load will be evenly balanced among the SUs if each SU is assigned one of the values defined in Equation 5. Again since the number of SUs is not always a divisor of the value of the *numberOfSIs * numberOfActiveAssignments*, some SUs will get the floor of this division while others will get the ceiling[5] with the constraints that the sum of the active assignments for all SUs is equal to *numberOfSIs * numberOfActiveAssignments*. Some SUs may have an extra load of one active assignment compared to other SUs. The sum of the cells in the "*Act"* row and the sum of the cells in the "*Backing up*" column are both equal to *numberOfSIs * numberOfActiveAssignments*. This is due to the fact that this number represents the total active assignments an SG will protect, regardless of how we assign the active and backup assignments this number is always going to be the same.

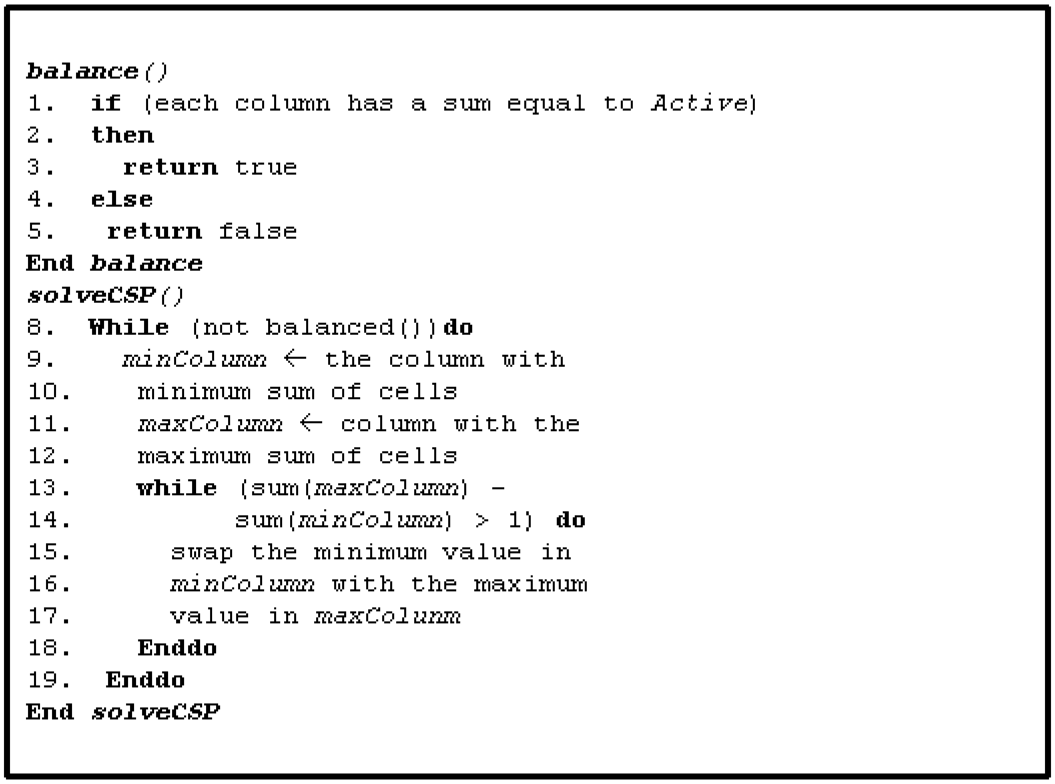
Equation 5. Active assignment for each SU.

$$Active = \left\{ \begin{array}{l} \left\lfloor \dfrac{numberOfSIs \times numberOfActiveAssignments}{numberOfSUs} \right\rfloor \\ or \\ \left\lceil \dfrac{numberOfSIs \times numberOfActiveAssignments}{numberOfSUs} \right\rceil \end{array} \right\}$$

[5] In Equation 1 we calculate the minimum load of SIs any SU must be able to handle, and this is why we use only the *ceiling*. While in Equation 5 we are looking for the exact load, and this is why we use the notion of *ceiling* and *floor*.

Our third step consists of making sure the total active assignments for each SU is one of the values in Equation 5. This must be completed without affecting any value in the "*Baking Up*" column. In other words we need to shuffle the values in the row cells in such a way that (1) their sum remains intact and equal to the value of the "*Baking Up*" cell in the row, and (2) achieve a balance so that the cells of the "*Act*" row have values as given by Equation 5. This reasoning converts our problem into a Constraint Satisfaction Problem (CSP), where we have a set of values within a table, and they are constrained by the fact that the sum of the row cell values and the sum of the column cell values must obey to certain magnitudes.

The algorithm in Figure 8 solves the constraints satisfaction problem of balancing the active load.



```
balance()
if (each column has a sum equal to Active)
then
  return true
else
  return false
End balance
solveCSP()
While (not balanced())do
  minColumn ← the column with
   minimum sum of cells
   maxColumn ← column with the
   maximum sum of cells
   while (sum(maxColumn) -
         sum(minColumn) > 1) do
     swap the minimum value in
     minColumn with the maximum
     value in maxColumm
   Enddo
 Enddo
End solveCSP
```

Figure 8. The algorithm for the CSP solution.

This algorithm consists of two main functions, the *balance()* function that test whether the SUs have equal active assignments, and the *solveCSP()* function that keeps swapping row values until the balance is obtained.

Notice that we could have started working with the columns of Table 6, and therefore balance the active load first and then work with the backup assignment, or work simultaneously with both as it is the case for such constraints solving problems.

Our final step is to automatically generate the ranked list of SUs for each SI. The rank values of this list are based on the assignment table. The list is generated in the following manner. For each row in the table that corresponds to SUi, we will calculate the number of different SIs that this SU is backing up by dividing the value in the "*Backing Up*" cell of the row over the *numberOfActiveAssignments*. The assignment table will tell us how many active assignments each of the other SUs will have on behalf of the SIs backed up by SUi, but we still need to determine to which particular SU each SI will be assigned. Here, again, we face another CSP problem, with the following constraints:

- The number of SUs assigned active to each SI must be equal to the *numberOfActiveAssignments*, and

- The number of SIs each SU (excluding SUi) will be active for is specified by the assignment table and must be respected.

By solving this CSP problem we determine the active SUs and assign them the lower ranks, the SU that was assigned the N/A value (i.e. SUi) will serve as the backup, and therefore will have the first rank higher than the SUs with the active assignments, the other SUs will have ranks greater than the one assigned to the backup SU. The above process will result in a sub-list of ranked SUs generated for the SIs backed up by SUi. The same process is repeated for all the rows, and the sub-lists of ranked SUs are joined together repeatedly until we get the ranked list of SUs for all SIs. The process is further illustrated in the next example.

4.3 Application

For illustration purpose, we reuse the example presented in Section 4.1. Step 1 would be to calculate backup assignment for each SU. According to Equation 4, the backup in terms of SIs is equal to the floor or ceiling of $(14 \div 6) = 2.33$. Some SUs will back up two SIs while others will back up three. We multiply those numbers with the *numberOfActiveAssignments* to get the baking up value each SU will handle in terms of active assignment. Knowing the backup assignments, we can proceed into the second step and populate our table with values as shown in Table 5. The values in the cells are assigned by taking the value in the Backing up cell of each row and dividing it evenly among the other cells of the same row.

Note that at this step of the table is still not balanced, we simply balanced the backup values among the SUs.

Table 5. The assignment table before balancing.

	SU1	SU2	SU3	SU4	SU5	SU6	Backing Up
SU1	N/A	1	2	2	2	2	9
SU2	1	N/A	2	2	2	2	9
SU3	1	1	N/A	1	1	2	6
SU4	1	1	1	N/A	1	2	6
SU5	1	1	1	1	N/A	2	6
SU6	1	1	1	1	2	N/A	6
Act	5	5	7	7	8	10	$\sum = 42$

The third step consists of balancing the active load among SUs, as aforementioned the problem here is a constraint satisfaction problem that involves shuffling the values in the table cells in such a way that the sum of the columns would be floor or ceiling of the value calculated according to Equation 5 [$Active = (14 \times 3) \div 6 = 7$].

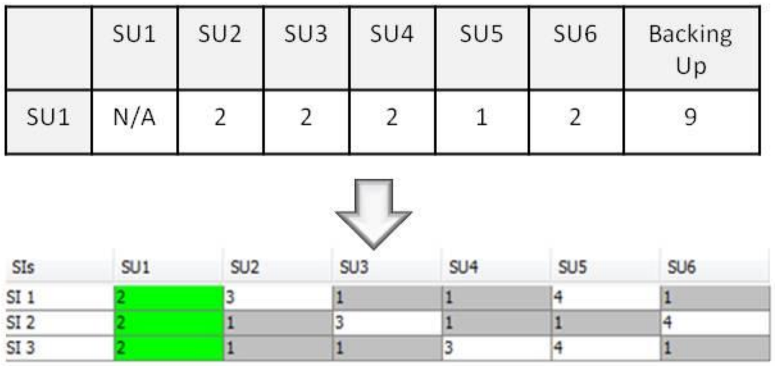
Solving the CSP by swapping the values in the row cell results in the balanced Table 6 shown below.

Table 6. The assignment table after balancing.

	SU1	SU2	SU3	SU4	SU5	SU6	Backing Up
SU1	N/A	2	2	2	1	2	9
SU2	2	N/A	2	2	2	1	9
SU3	2	1	N/A	1	1	1	6
SU4	1	2	1	N/A	1	1	6
SU5	1	1	1	1	N/A	2	6
SU6	1	1	1	1	2	N/A	6
Act	7	7	7	7	7	7	$\sum = 42$

The final step is to derive from Table 6 a ranked list of SUs for each SI. Each row of our assignment holds the information needed to generate the ranked list of SUs for a subset of SIs. We take a row at a time, extract the information encapsulated in this row, and based on this information, we generate our ranked list of SUs. Figure 9 shows the ranked list of SUs generated for the subset of SIs backed up by SU1.

The process is carried out as follows. Based on the first row of Table 6 we deduce that SU1 will back up three SIs ($9 \div 3$*;* where 3 is the *numberOfActiveAssignments*). So for SI1, SI2 and SI3 the backup is SU1, therefore SU1 is not allowed to be active for those SIs and as a result will be assigned a rank higher than the one we will assign to the active SUs of behalf of those three SIs. In order to determine which SU is active on behalf of which SI, we need to solve the CSP problem defined by assigning 9 active assignments to 5 SUs on behalf of 3 SIs in such a way that each SU will have the exact value of active assignments specified in the assignments table, e.g. the table specifies that SU5 has only one active assignment and therefore will be active for at most one of the SIs backed up by SU1. Figure 9 presents one possible solution to this problem. The SU with the active assignment is given the lowest rank. The rest of the SUs will be given a rank higher than the one assigned to the backup SU (i.e. SU1). This process is repeated for all the rows, until we have a complete list of ranked SUs for all SIs.

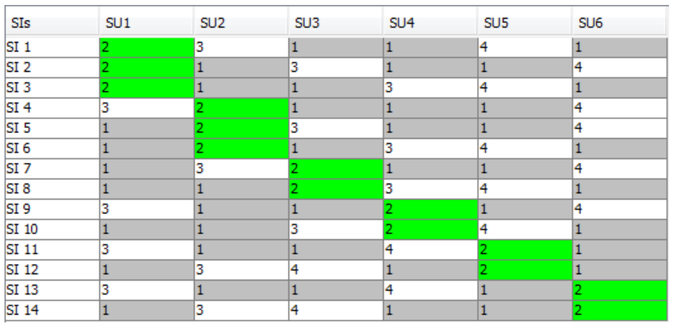


	SU1	SU2	SU3	SU4	SU5	SU6	Backing Up
SU1	N/A	2	2	2	1	2	9

SIs	SU1	SU2	SU3	SU4	SU5	SU6
SI 1	2	3	1	1	4	1
SI 2	2	1	3	1	1	4
SI 3	2	1	1	3	4	1

Figure 9. A mapping from the assignment table row to a ranked sub-list.

Figure 10 is a snapshot of our generated ranked list of SUs, the table cells are rendered such that the cells holding the backup assignment value are marked in green, while the cell holding the active assignment value are marked in gray.



SIs	SU1	SU2	SU3	SU4	SU5	SU6
SI 1	2	3	1	1	4	1
SI 2	2	1	3	1	1	4
SI 3	2	1	1	3	4	1
SI 4	3	2	1	1	1	4
SI 5	1	2	3	1	1	4
SI 6	1	2	1	3	4	1
SI 7	1	3	2	1	1	4
SI 8	1	1	2	3	4	1
SI 9	3	1	1	2	1	4
SI 10	1	1	3	2	4	1
SI 11	3	1	1	4	2	1
SI 12	1	3	4	1	2	1
SI 13	3	1	1	4	1	2
SI 14	1	3	4	1	1	2

Figure 10. A snapshot of our ranked list of SUs.

4.4 The NWay Redundancy Model

For the *NWay* redundancy model, an SI can have at most one SU active, but many standbys. In case of failure of the active SU, the highest ranking standby SU will be assigned the HA active state on behalf of this particular SI. We refer to that SU as the primary standby for the SI. However, even after the standby SU took the active role instead of failed SU, another SU needs to take over the standby assignment that was originally assigned to the primary standby. Moreover, since the failed SU is normally active for some SIs and standby for others, then it is not only the active load that must be evenly redistributed; the standby load must as well be evenly redistributed.

Our solution holds also for the *NWay* redundancy model if we consider the primary standby SU to be the backup SU that we introduced for the *NWayActive* redundancy model. Note that *numberOfActiveAssignments* is always equal to one for the *NWay* redundancy model. Nonetheless unlike the backup assignment, the standby assignment for each SI imposes a load on the SU and the node where the SU is deployed. Therefore, our approach will serve in determining the primary standby SUs for the SIs by distributing this load evenly. Additional steps not presented in this paper are needed to determine how the standby assignments are evenly distributed among SUs. Each of these steps constitute an additional CSP problem similar to the one we solved to generate our ranked list of SU for SIs.

5 Discussion

The issues uncovered in the previous sections are categorized in this section into four points: Selection of orphan types, selection of higher-level types, generation of multiple configurations, and ensuring SI load balancing dynamically at runtime.

5.1 Selection of Orphan Types

The selection of an orphan type requires the population of the attributes of the created parent type. These attributes may require a good understanding of the software implementation and therefore may not be derived automatically. An example of such a type attribute is the component restart probation period found in an SG type. This attribute applies to components in service units belonging to a service group of this type. If the SG type is created (instead of being selected from ETF) then this attribute needs to be determined without any guidance from the vendor. We need to introduce artificially a value, which will affect the software behavior at runtime and therefore impacts the quality of the generated configuration. It is still unclear how to determine safely the attributes of the different entity types when they are created.

A related observation is that it is necessary to maintain the information whether a type was created by our method or provided by the software vendor in the ETF. Created types do not reflect implementation limitation and therefore should not restrict the use of orphan types. However they may be reused whenever they are appropriate to ease the type creation task.

5.2 Selection of Higher Level Types

Our preference is to only analyze orphan types when the non-orphan ones have been checked and found unsatisfactory with respect to the configuration requirements. However, there might be situations where we have an ETF (or multiple ETFs) where more than one type can be a candidate. This is typically the case for different versions of software or that offers similar services but provided by different vendors. Some vendors (versions) may constrain the way their components should be configured by having the corresponding component types refer to SU types, while other vendors may provide components that do not need to be constrained from the ETF perspective. In other words, the component types corresponding to these components are orphans. Depending on the required services and the deployment system, there might be situations where orphan component types are a better choice than non-orphan types due to their flexibility. This comes at the price of the difficulties of type creation as aforementioned. Our technique is easily adaptable to check types in any desirable order.

5.3 Generating Multiple Configurations

Several configurations might be generated from the same set of ETF types and configuration requirements. Different sets of types may satisfy the requirements, and also from the same set of types different sets of entities can be created. Finally, the same set of entities may be distributed in the cluster in different manners leading to different configurations.

The algorithm presented in Section 3.3 stops when it finds the first set of types that satisfies the configuration requirements. However, there may be many other possible sets that are candidate solutions. To extract all possible paths that contain valid ETF types with respect to the configuration requirements, the algorithm can easily be adapted by changing the exit condition and saving each path that contains a set of qualified types.

For a single type set, for instance, determining the number of components in an SU is a criterion that can lead to multiple configurations. In the example presented in Section 3.4, we concluded that we needed 2 components in each SU to handle the CSIs associated to these SUs in both active and standby states targeting maximum utilization of the components based on the capability model. However, we could also have created four components of this component type in each SU. This would also have been a valid configuration.

Although there are many configurations that can be generated from the same input sets, not all configurations are equally suitable. There is a need to investigate the criteria by which configurations can be compared. Examples of these criteria include the availability level that a configuration offers, performance, the cost associated with the components, etc.

5.4 Ranking SUs for SIs

One of the main advantages of ranking SUs for SIs at configuration time in order to ensure load balancing at runtime is that it relieves the middleware implementation from implementing a load monitoring and distribution service, which could increase the complexity of the middleware implementation and the computation overhead needed for runtime monitoring. A round robin ranking technique can be used since it does not require any runtime information, and the rankings generated using round robin balance the load in the absence of failure. However this technique does not solve load balancing after failure.

The systems for which we are generating the configuration have no single point of failure, which means that they tolerate at least one failure, and that is completely covered by our ranking method. Nevertheless in cases where protection against several points of failures is needed, our solution will guaranty that the load is redistributed among several SUs, but it does not guaranty the equal redistribution.

6 Related Work

The work presented in this paper is part of a larger project, called the MAGIC project [5], which aims at investigating techniques and building tools for automatic generation of AMF configurations for clustered systems as well as the generation of upgrade campaigns to migrate these systems from one configuration to another.

The works presented in [7] and [8] are directly related to our domain, since they partially tackle the problem of AMF configuration generation. We are not aware of any other work that focuses on generating AMF configurations. This is mainly due to the fact that the SAForum specifications are relatively new and that the implementation of SAForum compliant middleware is still an ongoing effort. Existing middleware implementations such as OpenAIS [9], OpenSAF [10] and OpenClovis [11] offer limited if any support for the generation of AMF configurations. Other works described in this section relate to our work in the broader sense of system configuration generation.

The authors in [7] apply the Model Driven Approach (MDA) to the design of AIS configurations. In their approach an initial AIS compliant configuration is devised using predefined design patterns, gathered from previous experiences. This initial configuration is referred to as the Platform Independent Model (PIM), which is then transformed and specialized automatically into a Platform Specific Model (PSM) which corresponds to a specific implementation of AIS. Meta-models are used for the transformation and for the validation of configurations. The authors however did not specify any methodology to follow in the process of configuration generation for the PIM. Moreover they did not include ETF in their solution as their work predates [4]. Our view of the automatic configuration generation differs from their view. Their approach is a model transformation that takes an existing configuration from a PIM level to a PSM level. Instead we automatically generate this initial configuration or PIM.

In [8] the authors created a modeling framework for the automatic generation of middleware specific deployment descriptors. The implementation of the framework is based on the IBM Rational Software Architect (RSA) modeling tool. They have also implemented configuration generator facility for each different AIS implementation in the form of RSA pluglets. And they used these pluglets to generate AMF configurations for two middleware implementations: (1) for OpenAIS they generate a simple text file from the configuration UML model (2) for OpenSAF they generate an XML file using the DOM solution from the configuration UML model as well. Again the authors of this work have a different view of what the automatic configuration generation is, and they view it again as a model transformation from an existing configuration at PIM level to a PSM level. And therefore their approach is missing the methodology and concepts of generating AMF configurations.

In [12] the authors present an engine that automatically designs a service infrastructure which aims at meeting the service's availability requirements. The engine explores a design space consisting of multiple combinations of hardware/software and repair options configurations presenting various tradeoffs among cost, availability, and performance. Their approach is rather a combinatorial approach that finds all possible combinations of hardware, operating systems and applications, and filters out the ones that do not satisfy the availability. In this work the configuration is the design of the system stack from hardware to application, rather than configuring the components and the services that constitute the application.

More work on configuration generation has been done in the more general context of software configuration management, particularly using constraint satisfaction techniques and policies as reported in [13][14]. The authors in [14], for instance, propose an approach for generating a configuration specification and the corresponding deployment workflow from a set of user requirements, operator and technical constraints, which are all modeled as policies. An example of such constraints is that a given operating system can only run on certain processor architectures. Generating a configuration is formulated as a resource composition problem taking into account the constraints. Our approach is similar from this point of view; however, our focus is on the availability and AMF constraints instead of general utility computing environments. Challenging constraints inherited from the AMF domain such as the various redundancy models to be provided, are not taken into account in [14].

The work presented in [15] focuses on developing a system that will automatically select which data protection techniques to use, and how to apply them, to meet user-specified dependability. By being selective in choosing protection techniques, i.e. assigning different levels of protection to different kinds of information, the authors argue they can save money or free up resources for providing more protection to important data. Their input consists of (1) a description of the user requirements for data performability and data reliability, (2) a description of the failures to be considered, including their scope and likelihood of occurrence, and (3) a description of the data protection techniques. They have designed a rule-based engine that can select the appropriate protection technique based on these criteria. The focus of the approach is on reliability and performability for data, rather than for applications. They select and configure the protection technique for data rather than generating system configurations to provide and protect services. They work at a lower level of abstraction with data storage/replication techniques.

In [16] the authors investigate the applicability of Model-Driven Engineering to address the middleware Quality of Service (QoS) configuration challenges by automating the process of mapping the domain-specific QoS requirements onto the right middleware specific configuration options. In particular, their model transformation-based approach begins with domain-specific, platform-independent models (PIM) of Distributed-Realtime-Embedded systems. The QoS requirements are then used to iteratively transform the PIM to more refined and detailed middleware platform-specific models (PSMs). The variability in the middleware configuration spaces are captured in the form of parameterization of mapping rules of the platform-independent model transformations. Subsequently individual platform-specific transformations are instantiated by specializations, such as by providing exact values of these parameters. Their notion of parameterized transformations and specializations is similar in concept to C++ templates and Java generics. Like most of the automatic configuration generation techniques we have surveyed this work also starts from an existing PIM level configuration and automatically generates the PSM level configuration.

The workload assignment issues we address in this work is tightly related to the SAForum domain in the sense that the services that the system components provide are separately abstracted from the component themselves and the workloads are specified and configured during the system design phase, the particularity of our problem comes from the fact that the initial assignment of the services (SIs) to SUs and its evolution in case of failure is predetermined at configuration time using the ranking of SUs. Existing solutions handle load balancing at runtime [17][18][19][20], where it is much easier to handle because the load of the failing SU can be distributed evenly knowing the current load of each of the other healthy SUs. The most relevant conventional runtime load balancing algorithm that we can use as a reference is round robin [22]. Other algorithms in [17] and [23], such as *Central Manager Algorithm* or

Threshold Algorithm, or *Randomized* algorithms in [21] exist but they all work for runtime load balancing. An interesting work was presented in [20] where the authors present an approach for load balancing in the presence of a random node failure; however they made several assumptions like knowing the mean time to fail and the mean time to recover, as well as nodes knowing the initial workload of other nodes.

7 Conclusions

In this paper, we presented a method for the automatic generation of AMF configurations. Our method generates configuration that satisfies the user requirements, while taking into account the deployment cluster constraints and the software description from the vendors. Our method proceeds in a top-down fashion by visiting ETF types starting from the highest ones. We showed the detailed steps involved in our approach including estimating the load of SIs per SU, the selection of component types as well as SU, SG, and application types. We also discussed how AMF types and entities are created once the types are selected or created. We have also presented an approach for generating a ranked list of SUs for the SIs of an AMF configuration. This ranking is necessary in the case of NWayActive and NWay redundancy models. Our approach is based on balancing the number of backups as well as the load. The particularity of our approach is that it guarantees, at configuration time, load balancing before and after a single failure during runtime.

Our configuration generation technique has been implemented. Our future work will focus on defining techniques for comparing multiple configurations in terms of criteria such as the availability level they offer to the services. We also intend to investigate how our SU ranking approach can be integrated to support load balancing after multiple SU failures and dropping some of the assumptions such as the capacities of the SUs/nodes or the load imposed on SUs by SIs.

Acknowledgements

This work has been partially supported by the Natural Sciences and Engineering Research Council (NSERC) of Canada and Ericsson Software Research.

References

[1] Jim G., Daniel P., (1991) "High-Availability Computer Systems," Computer, vol. 24, no. 9, pp. 39-48.

[2] Service Availability Forum™, URL: http://www.saforum.org

[3] Service Availability Forum, Application Interface Specification. Availability Management Framework SAI-AIS-AMF-B.04.01.

[4] Service Availability Forum, Application Interface Specification. Software Management Framework SAI-AIS-SMF-A.01.01.

[5] MAGIC Project, URL: http://users.encs.concordia.ca/~magic/

[6] A. Kanso, M. Toeroe, F. Khendek, A. Hamou-Lhadj, "Automatic Generation of AMF Compliant Configurations", Proceedings of the International Service Availability Symposium (ISAS), LNCS Vol.5017, pp. 155-170, Tokyo, Japan, 2008.

[7] A. Kövi, D. Varró, "An Eclipse-Based Framework for AIS Service Configurations", In Proc. of the International Service Availability Symposium (ISAS), LNCS Vol.4526, pp. 110-126, Durham, NH, 2007.

[8] Z. Szatmári, A. Kövi, M. Reitenspiess "Applying MDA approach for the SA forum platform", In Proc of the 2nd workshop on Middleware-application interaction (MAI) ACM Vol. 306, pp 19-24 Oslo, Norway, 2008.

[9] OPENAIS, URL: http://www.openais.org

[10] OPENSAF, URL: http://www.opensaf.org/

[11] OPENCLOVIS, URL: www.openclovis.org/

[12] G. Janakiraman, J. Santos, Y. Turner: “Automated Multi-Tier System Design for Service Availability”, In Proceedings of the First Workshop on Design of Self-Managing Systems, June 2003

[13] T. Hinrich, N. Love, C. Petrie, L. Ramshaw, A. Sahai, S. Singhal, “Using Object-Oriented Constraint Satisfaction for automated Configuration Generation“, 15th IFIP/IEEE International Workshop on Distributed Systems Operations and Management (DSOM), LNCS Vol. 3278, Springer, pp.159-170, Davis, CA, November 15-17, 2004.

[14] A. Sahai, S. Singhal, R. Joshi, V. Machiraju, “Automated Generation of Resource Configurations through Policies”, Fifth IEEE International Workshop on Policies for Distributed Systems and Networks (POLICY'04), Yorktown Heights, New York, June 7-9, 2004.

[15] K. Keeton , J. Wilkes “Automating data dependability” In Proceedings of the 10th ACM-SIGOPS European Workshop Saint-Emilion, France, July 1, 2002.

[16] A. Kavimandan and A. Gokhale. A Parameterized Model Transformations Approach for Automating Middleware QoS Configurations in Distributed Real-time and Embedded Systems. In Proceedings of ASE Workshop on Automating Service Quality, (WRASQ 2007), Atlanta, GA, Nov. 2007.

[17] Z. Xu, R. Huang, "Performance Study of Load Balancing Algorithms in Distributed Web Server Systems", CS213 Parallel and Distributed Processing Project Report.

[18] P. L. McEntire, J. G. O'Reilly, and R. E. Larson, “Distributed Computing: Concepts and Implementations”, New York: IEEE Press, 1984.

[19] L. Rudolph, M. Slivkin-Allalouf, E. Upfal, “A Simple Load Balancing Scheme for Task Allocation in Parallel Machines”, Proceedings of the 3rd ACM Symposium on Parallel Algorithms and Architectures, pp.237-245, July 1991.

[20] S. Dhakal, M.M. Hayat, J.E. Pezoa, C.T. Abdallah, J.D. Birdwell, J. Chiasson, "Load balancing in the presence of random node failure and recovery", Proceedings of the 20th IEEE International Parallel & Distributed Processing Symposium, 2006.

[21] R. Motwani and P. Raghavan, “Randomized algorithms”, ACM Computing Surveys, 28(1):33-37, 1996

[22] M. Shreedhar, G. Varghese, "Efficient fair queuing using deficit round robin", IEEE/ACM Transactions on Networking, Vol. 4, No. 3 pp: 375-385, 1996.

[23] S. Sharma, S. Singh, and M. Sharma, "Performance Analysis of Load Balancing Algorithms" World Academy of Science, Engineering and Technology, Vol. 38, 2008.