Ranking Service Units for Providing and Protecting Highly Available Services with Load Balancing

A. Kanso, F. Khendek, A. Hamou-Lhadj
Electrical and Computer Engineering Department
Concordia University
Montreal, Canada
{al_kan, khendek, abdelw@ece.concordia.ca}

M. Toeroe
Ericsson Canada
Montreal, Canada
Maria.Toeroe@ericsson.com

***Abstract*—In highly available systems, continuous load balancing in the presence of failure is essential to avoid performance degradation. The Availability Management Framework (AMF) is a middleware service that manages the availability of the applications' services by coordinating their provision among the application's redundant components. The application's components that collaborate to provide a service are logically grouped into a service unit (SU). For management purpose, the SU workload is abstracted as a service instance (SI). Redundant SUs that collaborate to protect a set of SIs are grouped into a service group (SG). The assignment of the SIs to the SUs is a runtime operation performed by AMF. For some redundancy models, *NWay* and *NWayActive*, this assignment is performed for each SI according to a predefined ranked list of the SUs in the SG. In this paper we discuss the challenges of defining this ranking at configuration time and propose an approach that automatically generates the ranked list of SUs at that time and guaranties continuous load balancing.**

Keywords- High availability, redundancy models, load balancinge, Availability Management Framework, constraints solving.

I. INTRODUCTION

The Service Availability Forum (SAF) [1] is a consortium of telecommunications and computing companies that joined forces to standardize a high availability solution for systems and services. SAF has developed a set of middleware service specifications. Among them, the Availability Management Framework (AMF) [2] manages the availability of the services provided by an application. This is realized through the management of redundant application components and by shifting dynamically the workload (services) assigned to a faulty component to a redundant one when a fault is detected.

In order to manage the high availability of an application AMF requires a logical grouping of its components known as an AMF configuration [2][3][4] .

In an AMF configuration a component represents a set of hardware and/or software resources that implements the APIs that allow AMF to control its life cycle and its workload assignment. The components that combine their functionalities to provide a more integrated service are logically grouped into a service unit (SU). For control purposes, the workload assigned to a component is abstracted as a component service instance (CSI). CSIs are grouped into a service instance (SI), which represent an abstraction of workload assigned to the SU. In order to ensure the provision of the SI in case of an SU failure, SUs are grouped into a service group (SG). An SU can be assigned the active HA (High Availability) state for an SI, the standby HA state, or it can be a spare one. The SI is provided by the SU(s) assigned the HA active state. The SIs provisioning is protected by the SG according to a redundancy model. AMF will dynamically shift the assignment of the SI from a failed SU to another one in the same SG. An example of an AMF configuration is shown in Figure 1 where the components C1 and C2 are grouped into SU1, which collaborates with other similar SUs of the SG to protect the provision of the SIs. An AMF application is a grouping of SGs. More details on AMF configurations can be found in [2][3][4].

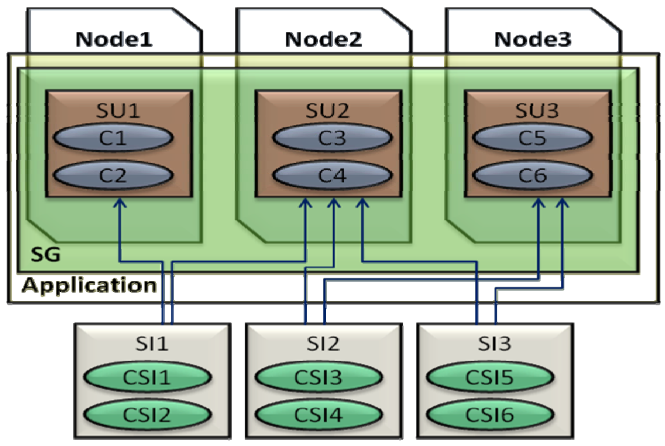


Figure 1. An example of AMF configuration.

As aforementioned, the assignment of SIs to SUs is performed at runtime. For the *NWay* and *NWayActive* redundancy models, this assignment, and the re-assignment after an SU failure, is performed for each SI according to its ranked list of SUs (the assignment criteria are defined in Section II). The ranking is established at configuration time. In this paper, we will demonstrate that if the ranked list is determined according to conventional algorithms, like round robin, the shifting of SIs from a failed SU to healthy ones may lead to an unbalanced workload among the SUs, which may lead to overload causing subsequent failures and performance degradation. To ensure a continuous load balancing in the presence of a failure, we propose an algorithm that views the issue as a Constraints Solving Problem (CSP).

The organization of the paper is as follows. In Section II we introduce the different redundancy models defined for AMF. In Section III we describe the problem of load balancing at configuration time using existing algorithms. These problems occur after one failure. In Section IV we present our solution for ensuring load balancing before and after a failure. In Section V we briefly review related work before concluding in Section VI.

II. REDUNDANCY MODELS DEFINED IN THE AMF SPECIFICATION

An SG redundancy model defines how the SUs of the SG will protect the SIs of the SG. There are five different redundancy models: *2N, N+M, NWay, NWayActive, and No-Redundancy* [2]. These redundancy models differ on the distribution of the active and standby SI state assignments among the SUs, and define how many active and standby state assignments an SI may have..

- The *2N* redundancy model specifies that in an SG at most one SU will have the active HA state for all SIs of the SG and is referred to as the active SU, and at most one SU will have the standby HA state and is called the standby SU. We have one active SU at all time and therefore no load balancing is needed.

- An SG with the *N+M* redundancy model is similar to 2N, but has N active SUs and M standby. An SU can be active for all SIs assigned to it 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. The AMF specification does not require a ranked list of SUs for the SIs in this case, and therefore this model is not considered in this paper.

- The *No-Redundancy* redundancy model is typically used with non-critical components, when the failure of a component does not cause any severe impact on the overall system [2]. We have no standby SUs, but we can have spare SUs. 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. Therefore, load balancing is not an issue for this redundancy model.

- 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* is always less or equal to the number of SUs in the SG.

Our work in this paper targets the *NWay* and *NWayActive* redundancy model since the assignment of SIs to SUs is governed by the SUs ranking for the SIs as illustrated hereafter.

Figure 1 shows an *NWayActive* SG, which consists of three SUs, and which is protecting three SIs. Each SI has been configured to have two SUs that are assigned the active state on its behalf. Thus, if one SU serving a given SI fails, while the second active one keeps servicing the SI, AMF will also assign the remaining third SU the active HA state on behalf of this SI, since the SI is 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 1.

Table 1. 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 have a ranked list of all the SUs in the SG [2].

(2) The number of SUs that will be assigned the active/stanby state is defined by the preferred number of active/stanby 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 1, 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 1 illustrates one possible ranking; other rankings are possible and other equivalent rankings could lead to the same assignment shown in Figure 1.

III. THE PROBLEM WITH 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.

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

Table 2. A ranked list of SUs using a round robin algorithm.

	SU1	SU2	SU3	SU4	SU5	SU6
SI1	1	2	3	4	5	6
SI2	4	5	6	1	2	3
SI3	1	2	3	4	5	6
SI4	4	5	6	1	2	3
SI5	1	2	3	4	5	6
SI6	4	5	6	1	2	3
SI7	1	2	3	4	5	6
SI8	4	5	6	1	2	3
SI9	1	2	3	4	5	6
SI10	4	5	6	1	2	3
SI11	1	2	3	4	5	6
SI12	4	5	6	1	2	3
SI13	1	2	3	4	5	6
SI14	4	5	6	1	2	3

Table 2 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.
- 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. The load will be evenly distributed among the SUs. Each SU will have 6 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, all its workload would be shifted to SU4 causing it to bear twice the load of any other SU. All the active assignments of SU1 are shifted to SU4 because it has the next highest rank for all the SIs (i.e. SI1, SI3, SI5, SI7, SI9, SI11, SI13) assigned to the above SU1 (and SU2 and SU3). In other words, SU4 can be looked as a "backup" for SU1 with respect to all its assignments.

The shortcoming of the round robin algorithm is that it develops a repetitive pattern as shown in Table 2. 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 2, which is derived from Table 2. SU1 is backing up SU4, SU5 and SU6, while SU4 is backing up SU1, SU2 and SU3.

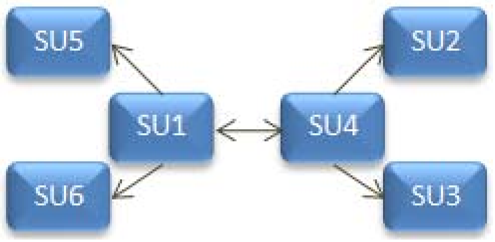


Figure 2. The SU back up graph using the round robin algorithm.

IV. A RANKING ALGORITHM TO ENSURE LOAD BALANCING BEFORE AND AFTER ONE FAILURE

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 cause by overload. Our main problem is to capture this at configuration time through the SU ranking for SIs.

As shown in the previous section 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 it applies for the *NWay* redundancy model as we will discuss it briefly 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 this work targets a single failure, and load balancing after multiple failures is outside the scope of this paper.

We first introduce our approach and then apply it to an example.

A. Approach for the NWayActive Redundancy Model

Our approach consists of four steps:

1. Determine the total number of assignments to backup (or simply backup assignment[1]) 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 the SUs, and
4. Derive the ranked list of SUs for each SI from the assignment table.

In 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 it is said 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, since one SI can only be backed up by one SU for all its 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.

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

Equation 1. Backup assignment for each SU.

A prerequisite for ensuring back up balancing is that each SU must back up an equal number of SIs. Therefore, the backup value of an SU is given by Equation 1. 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 3.

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 $\pm$ 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* ÷ *(n-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 3, which shows simultaneously the active and backup assignments.

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

Table 3. **Distributing the assignments of each SU.**

	SU_1	…	SU_j	…	SU_n	Backing Up
SU_1	N/A					
…		N/A				
SU_i			N/A			
…				N/A		
SU_n					N/A	
Act						–

Table 3 represents the blueprint of our assignment table. A value x in cell SU_{ij} 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 SU_{ij} means that the SU_i will back up SU_j in x of SU_j's active assignments. The "*Baking Up*" column is used to hold the value of the total backup assignments that we calculate with Equation 1 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 ceil of the baking up value after it is divided by the number of SUs -1. In other words if SU_i 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 2. 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 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 how we assign the active and backup assignments; this number is always going to be the same.

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

Equation 2. Active assignment for each SU.

Our third step consists of making sure the total active assignments for each SU is one of the values in Equation 2. 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 2. This reasoning converts our problem into a constraint satisfaction problem, 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.

Algorithm 1, shown hereafter, 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())
    minColumn ← the column with
    minimum sum of cells
    maxColumn ← column with the
    maximum sum of cells
    while (sum(maxColumn) –
          sum(minColumn) > 1)
      swap the minimum value in
      minColumn with the maximum
      value in maxColumm
    end while
  end while
end solveCSP
```

Algorithm 1. 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 5, 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 SU_i, 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 SU_i, 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 SU_i) 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. SU_i) 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 SU_i. 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.

B. *Application*

For illustration purpose, we reuse the example presented in Section 2. Step 1 would be to calculate backup assignment for each SU. According to Equation 1, 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 4. 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 value among the SUs.

Table 4. 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 ceil of the value calculated according to Equation 2 [*Active* $= (14 \times 3) \div 6 = 7$].

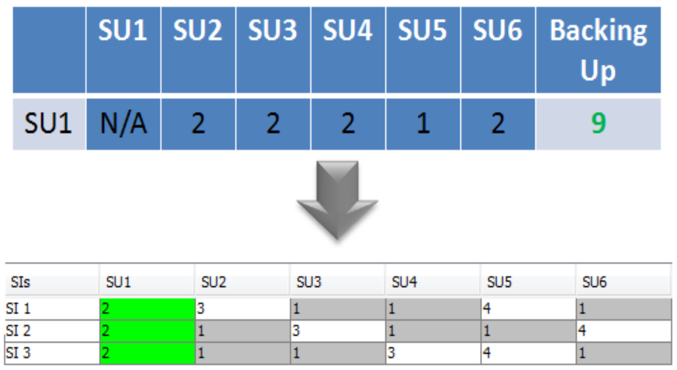
Solving the CSP by swapping the values in the row cell will result in the balanced Table 5 shown below.

Table 5. 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 5 a ranked list of SUs for the SIs. 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 3 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 5 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 3 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 successively for all the rows, until we have a complete list of ranked SUs for all SIs.

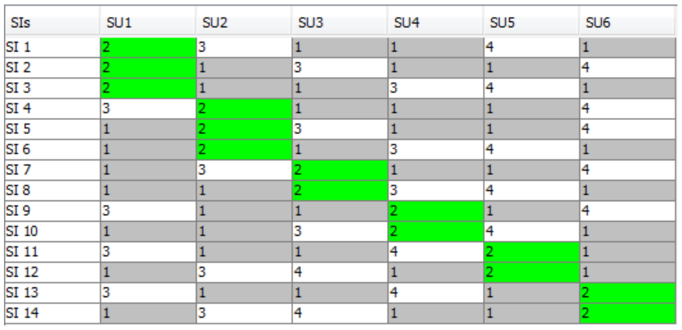


	SU1	SU2	SU3	SU4	SU5	SU6	Backing Up
SU1	N/A	2	2	2	1	2	9

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

Figure 4 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 4. A snapshot of our ranked list of SUs.

C. The NWay Redundnancy 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. 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. An additional step not presented in this paper is needed to determine how the secondary standby assignments are evenly distributed among SUs. This step is also considered as a CSP problem similar to the one we solved to generate our ranked list of SU for SIs.

V. RELATED WORK

The particularity of our problem comes from the fact that the initial assignment of 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 [6][7][8][9], 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 [10] Other algorithms in [6][11], such as *Central Manager Algorithm* or *Threshold Algorithm,* or *Randomized* algorithms in [5] exist but they all work for runtime load balancing. An interesting work was presented in [9] 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.

SU ranking is defined at configuration time, so any runtime algorithm that requires runtime information is not applicable to our problem. It targets applications in which the workload represents the use of distributed resources in the system, such as a distributed database, therefore the resource and the workload need to be collocated. This means that the configuration of the resource distribution determines the workload distribution therefore any load balancing need to be taken into account at configuration time.

VI. CONCLUSION

In this paper we presented an approach for generating a ranked list of SUs for the SIs of an AMF configuration. The approach is based on balancing the number of backups as well as the load. The particularity of our approach is that it guaranties load balancing before and after a single failure. Our approach is used at configuration time. It does not require runtime information. Moreover using our approach we can foresee at configuration time the system behavior in case failure at runtime. This work is complementary to the work presented earlier in [3][4] on automatic generation of AMF configurations. Our SU ranking approach can be integrated with the configuration generation technique.

Since it is not specified how an AMF implementation behaves in the absence of the ranked list of SUs, our solution will relieve an AMF implementation from incorporating runtime load balancing algorithms and thus reducing its complexity.

Our future step is to investigate how our work 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.

ACKNOWLEDGMENTS

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

REFERENCES

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

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

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

[4] A. Kanso, M. Toeroe, A. Hamou-Lhadj, F. Khendek, "Generating AMF Configurations from Software Vendor Constraints and User Requirements", *In Proc. of the fourth International Conference on Availability, Reliability and Security (ARES),* Fukuoka, Japan, March 2009.

[5] R. Motwani and P. Raghavan, "Randomized algorithms", ACM Computing Surveys (CSUR), 28(1):33-37, 1996

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

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

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

[9] 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," ipdps, pp.36, Proceedings 20th IEEE International Parallel & Distributed Processing Symposium, 2006

[10] M. Shreedhar, G. Varghese, " Efficient fair queueing using deficit round robin," IEEE/ACM Transactions on Networking (TON), Volume 4, Issue 3 pp: 375-385, 1996.

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