CORA Cork Open Research Archive
Cartlann Taighde Oscailte Chorcaí

Title	Self-organising, self-managing frameworks and strategies
Authors	Xiong, Huanhuan;Filelis-Papadopoulos, Christos K.;González-Castañé, Gabriel;Dong, Dapeng;Morrison, John P.
Publication date	2018
Original Citation	Xiong, H., Filelis-Papadopoulos, C., Castañe, G. G., Dong, D. and Morrison, J. P. (2018) 'Self-organising, self-managing frameworks and strategies', in Lynn, T., Morrison, J.P. & Kenny, D. (eds.) Heterogeneity, High Performance Computing, Self-Organization and the Cloud. Cham: Springer International Publishing, pp. 63-88. doi: 10.1007/978-3-319-76038-4_3
Type of publication	Book chapter
Link to publisher's version	https://link.springer.com/chapter/10.1007%2F978-3-319-76038-4_3 - 10.1007/978-3-319-76038-4_3
Rights	© 2018, the Author(s). This chapter is licensed under the terms of the Creative Commons Attribution-NonCommercial-NoDerivatives 4.0 International License (http://creativecommons.org/licenses/by-nc-nd/4.0/), which permits any noncommercial use, sharing, distribution and reproduction in any medium or format, as long as you give appropriate credit to the original author(s) and the source, provide a link to the Creative Commons license and indicate if you modified the licensed material. You do not have permission under this license to share adapted material derived from this chapter or parts of it. The images or other third party material in this chapter are included in the chapter's Creative Commons license, unless indicated otherwise in a credit line to the material. If material is not included in the chapter's Creative Commons license and your intended use is not permitted by statutory regulation or exceeds the permitted use, you will need to obtain permission directly from the copyright holder. - http://creativecommons.org/licenses/by-nc-nd/4.0/
Download date	2026-11-03 13:05:01

Item downloaded from	https://hdl.handle.net/10468/6916





CHAPTER 3

Self-Organising, Self-Managing Frameworks and Strategies

Huanhuan Xiong, Christos Filelis-Papadopoulos, Gabriel G. Castañe, Dapeng Dong, and John P. Morrison

Abstract A novel, general framework that can be used for constructing a self-organising and self-managing system is introduced. This framework is independent of the application domain. It embodies directed evolution, can be parameterised with different strategies, and supports both local and global goals. This framework is then used to apply the principles of self-organisation and self-management to resource management within the CloudLightning architecture.

Keywords Directed evolution • Self-organisation • Self-management • Strategies • Goal state

H. Xiong (✉) • G. G. Castañe • D. Dong • J. P. Morrison
Department of Computer Science, University College Cork, Cork, Ireland
e-mail: h.xiong@cs.ucc.ie; gabriel.gonzalezcastane@ucc.ie; d.dong@cs.ucc.ie; j.morrison@cs.ucc.ie

C. Filelis-Papadopoulos
Democritus University of Thrace, Komotini, Greece
e-mail: cpapad@ee.duth.gr

© The Author(s) 2018

T. Lynn et al. (eds.), *Heterogeneity, High Performance Computing, Self-Organization and the Cloud*, Palgrave Studies in Digital Business & Enabling Technologies,
https://doi.org/10.1007/978-3-319-76038-4_3

3.1 Introduction

A general framework for self-organisation and self-management (SOSM) is needed to support hierarchical architectures composed of autonomous components such as those described in the CloudLightning (CL) architecture discussed in Chap. 2. This chapter introduces a novel framework for SOSM developed to support CloudLightning. The next section presents key concepts in SOSM and how they are used to augment the CloudLightning architecture. The various SOSM mechanisms that enable components within CloudLightning to communicate, modify behaviour, make decisions, and cooperate with each other are then presented. Components may use different strategies for SOSM. As such, exemplar strategies are presented and illustrated in the context of CloudLightning through example scenarios.

3.2 Key Concepts

As discussed in Chap. 2 and mentioned above, the CloudLightning architecture is composed of autonomous components. Each component is equipped with various Strategies. These can be self-managing and/or self-organising strategies, and define how components at various levels in the hierarchy should evolve towards some ideal state known as the component's local goal.

In general, decisions being made by components at a particular level in the hierarchy can directly influence evolution in the adjacent levels. These influences may come from the top down, or from the bottom up. When coming from an upper level in the hierarchy, the process is called Directed Evolution. Directed Evolution signals the desire of the upper level to have the components, in the level underneath, change in operation or in configuration, to align with the goal of the upper level. Since components at a particular level also have local goals, the overall evolution that is brought about at that level should respect progress towards those local goals, while simultaneously accommodating the Impetus associated with the Directed Evolution process. An Impetus is communicated in the form of a tuple of values (i.e., a vector), known as a Weight. In a similar manner, a lower level in the hierarchy may directly influence the level above. This can be seen as Feedback from the lower level. This Feedback, in the form of tuples of values (i.e., vectors), known as Metrics, is derived from the operations of the components at the lower level and gives the upper level a Perception of

how the lower layer is changing and evolving. Perceptions can be used to determine subsequent Directed Evolution decisions.

As part of the self-organisation process, the interaction of two or more components, in any level of the hierarchy, may result in component creation, component destruction, component splitting, and/or component merging.

A measure of how close a component is to stasis, and hence how suitable its operating characteristics are for contributing to the global goal, is referred to as its Suitability Index (SI). In principle, any component subject to Impetus and possessing a Perception has an associated SI. Thus, in the CloudLightning framework, the goal state of those components, and the global goal of the systems, can be cast in terms of maximising the respective SIs.

In summary, the CloudLightning framework defines a number of mechanisms as follows:

- A mechanism to communicate Impetus, through the transmission of weights, from a level in the hierarchy to the level below. This mechanism allows a component, higher in the hierarchy, to steer the evolution of components immediately below them in the hierarchy.
- A mechanism to allow components to communicate Feedback, through the transmission of metrics, to components in the next level up in the hierarchy.
- A mechanism to modify the behaviour of components in response to Impetus and Feedback.
- Mechanisms to allow components to make decisions in accordance with various strategies to maximise their individual SIs.
- Mechanisms to allow components at the same level in the hierarchy to cooperate with each other in accordance with various strategies to maximise collective and/or individual SIs.

All of these concepts, and their interactions, are visualised in Fig. 3.1.

The CloudLightning framework provides these mechanisms to enable the SOSM strategies being deployed and performed by individual components to move nearer to their goal state. Within this framework, each component can make local decisions in accordance with various SOSM strategies based on its current state (from the feedback loops) and imposed Impetus (from the directed evolution processes), maximising its SI. Overall, self-management is implemented at a system level, allowing the whole system to evolving towards its business/system objectives.

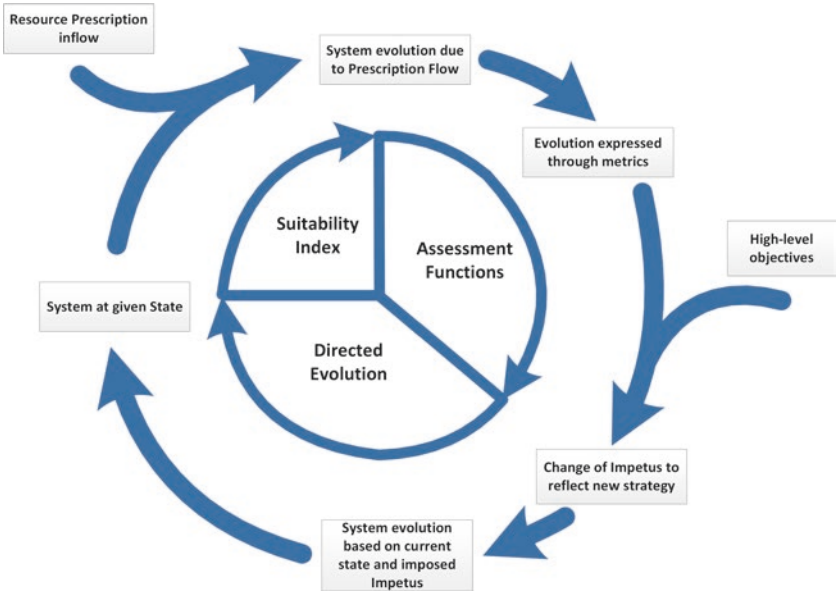


Fig. 3.1 Directed Evolution

3.3 Augmenting the CloudLightning Architecture

The CloudLightning architecture is initially augmented to include explicit entry points to the vRack Manager Groups. It can be seen from previous Chapter that these groups partition the resource space into different types of CL-Resources. This partitioning speeds up resource selection, since at most one CL-Resource type can be returned by the CloudLightning system for each service. The entry points into the differently typed vRack Manager Groups add an additional component to the CloudLightning architecture. Because of its routing characteristics described above, this component is called a pRouter. Figure 3.2 depicts this component in the augmented architecture.

From Fig. 3.2, it can be seen that there is an entry point into each vRack Manager Group, of the same CL-Resource type, hanging from each pRouter. These partition the space into smaller sets of CL-Resources of the same type. These entry points add yet another component to the

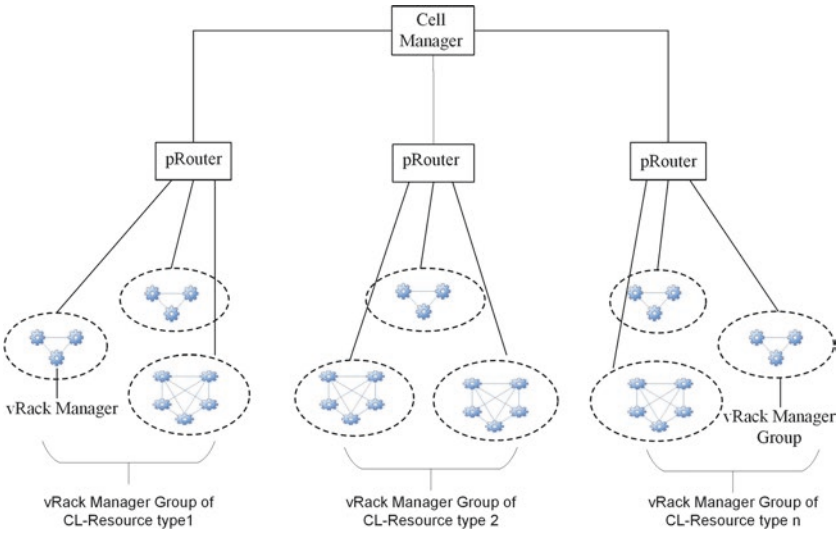


Fig. 3.2 Augmented CloudLightning architecture to include pRouters

CloudLightning architecture. Because this component connects all vRack Managers in the same group, it acts as a switch and is called a pSwitch. Figure 3.3 depicts this component in the augmented architecture.

It can be seen that the final augmented architecture forms a tree structure in which the root node corresponds to the Cell. The children of the Cell are pRouters, and there is at least one pRouter for each distinct CL-Resource type. The children of a pRouter are pSwitches. pSwitches partition the Virtual Rack Managers (vRMs), managing the same CL-Resource type, into groups. The number of pSwitches per pRouter is not fixed over time, neither is the size of the vRM groups managed by each pSwitch. In the following sections and chapters of this deliverable, it will be seen that pSwitches and vRMs can self-organise within groups, which are called Cooperatives, to emphasise their self-organising nature. To prohibit the creation of Cooperatives with different CL-Resource types, pSwitch Cooperatives cannot span pRouters. Similarly, to minimise administrative overhead and to simplify coalition formation, vRM Cooperatives (formerly called vRack Manager Groups) cannot span pSwitches.

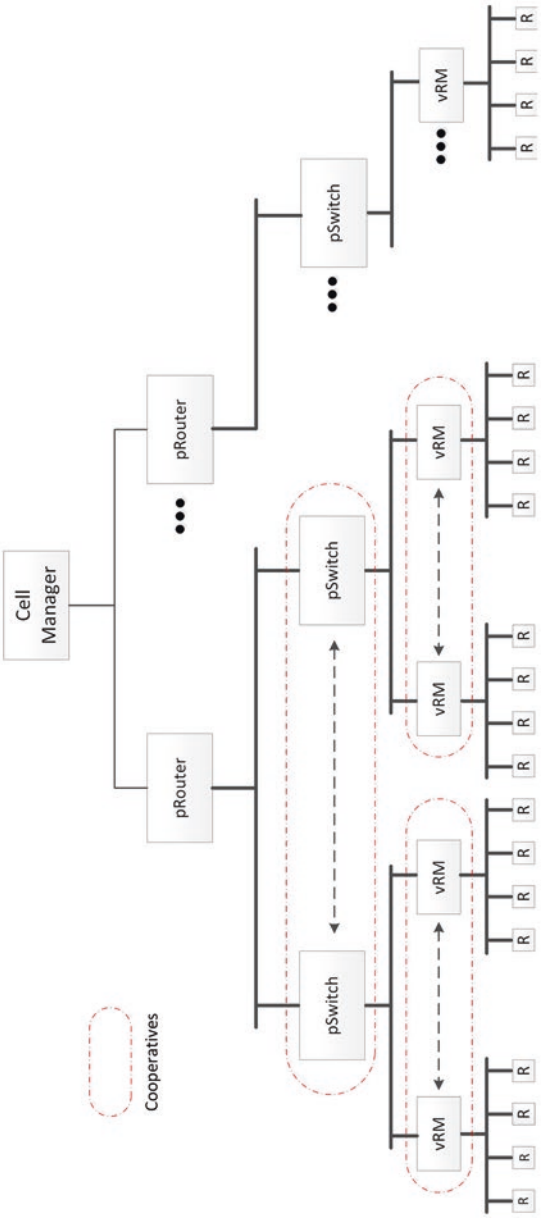


Fig. 3.3 Final augmented CloudLightning architecture illustrating its hierarchical nature with pRouter and pSwitch components

As the CloudLightning system evolves, it is anticipated that the number of pSwitches connected to a pRouter will change and will converge to some optimal number with respect to the global goal. This goal is derived from the Directed Evolution coming from the pRouter and from the pSwitch's efforts to achieve its local goal state. As part of the self-organisation process, pSwitches can be created, destroyed, merged, and split. In addition, pSwitches, within the same Cooperative, may exchange vRMs to optimise management. Together, the pRouters and the pSwitches form a reconfigurable and self-optimising switching fabric.

Similarly, it is anticipated that the number of vRMs connected to a pSwitch will change and will converge to some optimal number derived from the Directed Evolution coming from the pSwitch and from the vRM's efforts to achieve its local goal state. As part of the self-organisation process, vRMs can be created, destroyed, merged, and split. In addition, vRMs, within the same Cooperative, may exchange CL-Resources in an effort to maximise CL-Resource utilisation, minimise energy consumption, and facilitate coalition formation and management optimisation.

An important driving force behind the evolution of the CloudLightning system is the sequence of services/tasks that the system is required to execute. From the previous chapter, it can be seen that the process of maintaining a separation between resource and service life-cycles involves using the CloudLightning system to autonomously locate appropriate resources to execute each specific service/task. As part of this process, a description of these resources is passed to the CloudLightning system in an attempt to match appropriate resources with the service/task request. The term resource prescription (subsequently referred to simply as prescription) is introduced to refer to this description, and hence the pRouter is a prescription Router and the pSwitch is a prescription Switch.

vRMs form the lowest software level in the hierarchical organisation of the CloudLightning system. The next level up in this hierarchy is formed by grouping vRMs of the same type into Cooperatives. The elements of the Cooperatives, that is, its vRMs, self-organise by exchanging CL-Resources appropriately, to enable optimal management. Similarly, the elements of the pSwitch level self-organise by exchanging vRMs appropriately to enable optimal management. Finally, the elements of the pRouter level, that is, groups of pSwitches, self-organise by exchanging pSwitches appropriately to enable optimal management. All of these self-organising actions take place simultaneously resulting in the emergence of pathways through the hierarchy designed to optimise the ongoing propagation of resource prescriptions through the system.

3.4 Self-Organisation and Self-Management in CloudLightning Architecture

The general SOSM framework is mapped to the augmented hierarchical CloudLightning architecture outlined in the previous chapter. In the CloudLightning architecture, the autonomous components are the Cell, the pRouters, the pSwitches, and the vRMs. This framework provides Directed Evolution, self-management, and self-organisation mechanisms.

3.4.1 Directed Evolution

Directed Evolution is a mechanism to communicate a changing force throughout the system in a manner which effectively allows a component, higher in the hierarchy, to steer the evolution of the components immediately below them.

3.4.1.1 The Goal State

The goal of each component at all levels in the hierarchy is to maximise its SI.

The SI, η, is defined to be a combination of the Impetus and Perception expressed through a function $\eta\left(\vec{P},\vec{I}\right)$, such that $\vec{I}\in R^N, \vec{P}\in R^N \rightarrow \eta\left(\vec{I},\vec{P}\right)\in R,$ where N is the number of parameters used to express Impetus and Perception.

Note that, in the Cell the SI is calculated per resource type.

The goal state for the pRouter and the pSwitch is:

$$\arg\max \eta\left(\vec{I}\left(\vec{w}\right),\vec{P}\left(\vec{m}\right)\right), \vec{w},\vec{m}\in R^N \tag{3.1}$$

where $\vec{w}$ is an N-dimensional vector of weights corresponding to the Impetus and $\vec{m}$ is an N-dimensional vector of metrics obtained from the lower levels. Equivalently the goal state for the vRM is:

$$\arg\max \eta\left(\vec{I}\left(\vec{w}\right),\vec{P}\left(\vec{d}\right)\right), \vec{w},\vec{d}\in R^N \tag{3.2}$$

where $\vec{w}$ is an N-dimensional vector of weights corresponding to the Impetus and $\vec{d}$ is an M-dimensional vector of metrics obtained from the Telemetry service.

3.4.1.2 Cell State

The Cell state is a set of vector tuples and function tuples of the form:

$$\left\{\left\{\left(\vec{w}, \vec{m}_1\right), \left(\vec{\mu}_1, \vec{\varphi}_1\right)\right\}, \left\{\left(\vec{w}, \vec{m}_2\right), \left(\vec{\mu}_2, \vec{\varphi}_2\right)\right\}, \ldots, \left\{\left(\vec{w}, \vec{m}_n\right), \left(\vec{\mu}_n, \vec{\varphi}_n\right)\right\}\right\} \tag{3.3}$$

where n is the number of different pRouter types and $\vec{w}$ is the weight calculated by the Cell to effect steering. The tuple $\left(\vec{w}, \vec{m}_1\right)$ represents metrics and weights of the i-th pRouter, respectively, where $\vec{w} \in R^N$, $\vec{m}_i \in R^N$. The function tuple $\left(\vec{\mu}_i, \vec{\varphi}_i\right)$ is used to calculate the Impetus and Perception vectors, respectively, for each CL-Resource type maintained by each pRouter.

Since the Cell is at the highest level in the hierarchy, weights may be determined by the flow of tasks into the system and/or by local decisions made in an effort to move towards an objective goal state.

3.4.1.3 pRouter State and pSwitch State

The pRouter and pSwitch states can be described as a vector tuple $\left(\vec{w}, \vec{m}\right)$, representing weights and metrics where $\vec{w} \in R^N$, $\vec{m} \in R^N$, and a function tuple $\left(\vec{\mu}, \vec{\varphi}\right)$ is used to calculate Impetus and Perception, respectively.

3.4.1.4 vRM State

vRM state can be described as a vector tuple $\left(\vec{w}, \vec{d}\right)$, representing weights and metrics where $\vec{w} \in R^N$, $\vec{d} \in R^N$, and a function tuple $\left(\vec{\mu}, \vec{\varphi}\right)$ is used to calculate Impetus and Perception, respectively.

3.4.1.5 Steering by the Cell

There are at least two mechanisms for specifying a global goal state, G. The first is an objective goal specified to meet a specific business case. This can be set in a Cell, and in conjunction with the current local state of that Cell, adjustments can be made to the weights and applied to the underlying pRouters to steer them in that direction. By responding to this Impetus appropriately, the system will tend towards the goal state:

$$\vec{I}_{Cell} = \mu\left(\vec{I}'_{Cell}, \vec{G}_{Cell}, T_i\right) \tag{3.4}$$

where $\vec{I}'_{Cell}$ is the current Impetus of the Cell, $\vec{I}_{Cell}$ is the new Impetus of the Cell, $\vec{G}_{Cell}$ is the goal state of the Cell, and T_i are resource prescriptions.

Alternatively, the global goal state of the system can be expressed as a maximisation of the local goal state of the Cell. That is:

$$\arg\max \eta_i \left(\vec{I}, \vec{P}\right),\ i = 1,\ \ldots\ ,\ n,\ \vec{I},\ \vec{P} \in R^N \tag{3.5}$$

where η_i is the suitability of i-th pRouter attached to the Cell.

3.4.1.6 Steering by the pRouter

Steering by a pRouter is a mechanism for calculating and transmitting an Impetus to its attached pSwitches:

Impetus is a function such that:

$$\vec{I}_{pRouter} = \mu\left(\vec{I}'_{pRouter}, \vec{I}_{Cell}\right),\ \vec{I}'_{pRouter} \in R^N,\ \vec{I}_{Cell} \in R^N \tag{3.6}$$

where $\vec{I}'_{pRouter}$ is the previous Impetus of the pRouter. Here $\vec{I}_{Cell}$ represents the weight coming from the Cell.

3.4.1.7 Steering by the pSwitch

Steering by a pSwitch is a mechanism for calculating and transmitting an Impetus to its attached vRMs:

$$\vec{I}_{pSwitch} = \mu\left(\vec{I}'_{pSwitch}, \vec{I}_{pRouter}\right),\ \vec{I}'_{pSwitch} \in R^N,\ \vec{I}_{pRouter} \in R^N \tag{3.7}$$

where $\vec{I}'_{pSwitch}$ is the previous Impetus of the pSwitch. Here $\vec{I}_{pRouter}$ represents the weight coming from the pRouter.

3.4.2 Self-Management Mechanisms

The self-managing components in the system include (a) pRouters and pSwitches, managing prescription routing, metrics, and weights; and (b) vRMs, managing task execution, metrics, weights, and CL-Resources.

3.4.2.1 Mechanism to Send Metrics from a vRM to pSwitch

A separate assessment function corresponding to one of N metrics is executed in each vRM, and the result is passed as an N-dimensional vector to the respective pSwitch associated with that vRM.

3.4.2.2 Mechanism to Send Metrics from a pSwitch to pRouter

A number of *N*-dimensional vectors will arrive at a pSwitch (one from each vRM in the cooperative defined by that pSwitch), and each of these is combined to derive a new *N*-dimensional vector. This represents the pSwitch's Perception of the suitability of the underlying vRM cooperative to accept new tasks. This Perception can be customised by choosing the specific manner in which the input *N*-dimensional vectors are combined. The resulting *N*-dimensional vector is passed to the pSwitch's pRouter.

3.4.2.3 Mechanism to Send Metrics from pRouter to Cell

A number of *N*-dimensional vectors will arrive at a pRouter (one from each pSwitch in the cooperative defined by that pRouter), and each of these is once again combined to derive an *N*-dimensional vector representing the local state of that pRouter. This state can be viewed as being the pRouters Perception of the suitability of the underlying pSwitch cooperative to accept new tasks. This perception can also be customised by choosing the specific manner in which the input *N*-dimensional vectors are combined. This *N*-dimensional vector is passed to the Cell.

3.4.2.4 Mechanism to Send Weights from Cell to pRouters

Weights sent from a level in the hierarchy to a lower level represent the desire of the transmitting level to evolve in a particular direction. Since the Cell is at the highest level in the hierarchy, the sending of weights to the pRouters is the first step in the process of Directed Evolution. There are many strategies that the Cell can employ to determine how these weights change from time to time in the CloudLightning system. In all cases, these weights are sent to each pRouter as an *N*-dimensional vector representing the desired/calculated change to the progression of the Directed Evolution.

3.4.2.5 Mechanism to Send Weights from pRouters to pSwitches

After receiving an updated *N*-dimensional vector from the Cell, a pRouter will transform it using a customizable function, which will dictate the rate at which the next level down in the hierarchy is expected to change. This transformed *N*-dimensional vector is passed to the underlying pSwitches.

3.4.2.6 Mechanism to Send Weights from pSwitch to vRMs

After receiving an updated *N*-dimensional vector from the pRouter, a pSwitch will transform it using a customizable function, which will dictate

the rate at which the next level down in the hierarchy is expected to change. This transformed *N*-dimensional vector is passed to the underlying vRMs.

The same weights are propagated to every component in the same level (in the same pRouter). This ensures that the underlying level does not return metrics that cannot be meaningfully compared at that level. For example, if the weights associated with the calculations of power efficiency in two different servers of the same type are grossly different, one will appear to be more power efficient than the other even if both are equally power efficient.

Figure 3.4 depicts an example propagation of weights and metrics through the CL hierarchy in eight distinct time-steps. These vectors are propagated asynchronously from level to level. The metrics originate at the bottom level of the hierarchy, where they are derived from the application of CL-specific assessment functions applied to data gathered from the resource monitor. As they travel up through the hierarchy, they are aggregated to give successive perceptions of the underlying system at each successive component. The propagation of weights begins at the Cell and is modified as they are passed down through the hierarchy to reflect successive inflections of the Impetus coming from the Directed Evolution.

3.4.2.7 A Mechanism in the Cell to Modify Local Behaviour in an Effort to Respond to Impetus Provided by the Directed Evolution and Metrics Coming from Attached pRouters

Perception is a function such that:

$$\vec{P}_{Cell} = \varphi\left(\vec{m}_1, \vec{m}_2, \ldots, \vec{m}_r\right),\ \vec{m}_1 \in R^N,\ \vec{m}_2 \in R^N, \ldots, \vec{m}_r \in R^N \qquad (3.8)$$

Here, each $\vec{m}_i$ is a metric (an *N*-dimensional vector) coming from each of the **r** pRouters attached to the Cell.

Impetus $\vec{I}_{Cell} = \mu\left(T_i\right)$, where T_i is the task prescription under consideration.

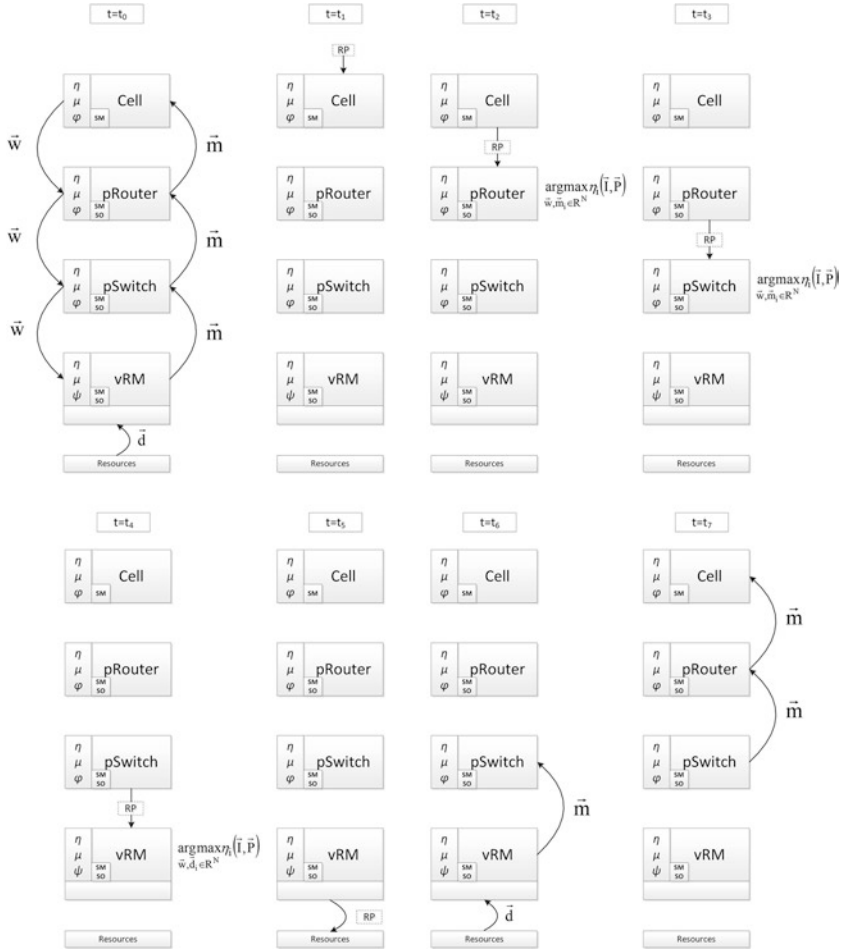


Fig. 3.4 An example propagation of weights and metrics through the CL hierarchy, with respect to a resource prescription

3.4.2.8 *A Mechanism in a pRouter to Modify Local Behaviour in an Effort to Respond to Impetus Transmitted by the Cell and Metrics Coming from Attached pSwitches*

Perception is a function such that:

$$\vec{P}_{pRouter} = \varphi\left(\vec{m}_1, \vec{m}_2, \ldots, \vec{m}_s\right),\ \vec{m}_1 \in R^N,\ \vec{m}_2 \in R^N, \ldots, \vec{m}_s \in R^N \quad (3.9)$$

Here, each $\vec{m}_i$ is a metric (an *N*-dimensional vector) coming from each of the **s** pSwitches attached to the pRouter.

Impetus is a function such that:

$$\vec{I}_{pRouter} = \mu\left(\vec{I}'_{pRouter}, \vec{I}_{Cell}\right),\ \vec{I}'_{pRouter} \in R^N,\ \vec{I}_{Cell} \in R^N \quad (3.10)$$

where $\vec{I}'_{pRouter}$ is the previous Impetus of the pRouter. Here $\vec{I}_{Cell}$ represents the weight coming from the Cell.

3.4.2.9 *A Mechanism in a pSwitch to Modify Local Behaviour in an Effort to Respond to Impetus Transmitted by its pRouter and Metrics Coming from Attached vRMs*

Perception is a function such that:

$$\vec{P}_{pSwitch} = \varphi\left(\vec{m}_1, \vec{m}_2, \ldots, \vec{m}_v\right),\ \vec{m}_1 \in R^N,\ \vec{m}_2 \in R^N, \ldots, \vec{m}_v \in R^N \quad (3.11)$$

Here, each $\vec{m}_i$ is a metric (an *N*-dimensional vector) coming from each of the **v** vRMs attached to the pSwitch.

Impetus is a function such that:

$$\vec{I}_{pSwitch} = \mu\left(\vec{I}'_{pSwitch}, \vec{I}_{pRouter}\right),\ \vec{I}'_{pSwitch} \in R^N,\ \vec{I}_{pRouter} \in R^N \quad (3.12)$$

where $\vec{I}'_{pSwitch}$ is the previous Impetus of the pSwitch. Here $\vec{I}_{pRouter}$ represents the weight coming from the pRouter.

3.4.2.10 A Mechanism in a vRM to Modify Local Behaviour in an Effort to Respond to Impetus Transmitted by its pSwitch and Metrics Coming from its vRack

Perception is a function such that:

$$\vec{P}_{vRM} = \vec{m} = \psi\left(\vec{d}\right),\ \vec{d} \in R^M \tag{3.13}$$

where $\vec{d}$ represents an *M*-dimensional Telemetry data obtained from the Telemetry service running on the physical resources belonging to the associated vRack.

Impetus is a function such that:

$$\vec{I}_{vRM} = \mu\left(\vec{I}'_{vRM}, \vec{I}_{pSwitch}\right),\ \vec{I}'_{vRM} \in R^N,\ \vec{I}_{pSwitch} \in R^N \tag{3.14}$$

where $\vec{I}'_{vRM}$ is the previous Impetus of the vRM. Here $\vec{I}_{pSwitch}$ represents the weight coming from the pSwitch.

3.4.2.11 Sample Events that Trigger the Transmission of Metrics at each Level in the Hierarchy

Options:

- Periodically, at a rate suitable for that level in the hierarchy
- From the vRM to the pSwitch:
  - After the receipt of a task prescription
  - When resources are freed
  - As a result of a self-organisation activity
  - Periodically to reflect utilisation, power consumption, and other low-level metrics of interest

3.4.2.12 Sample Events that Trigger the Transmission of Weights at Each Level in the Hierarchy

Options:

- As a result of steering
- Periodically, at a rate appropriate for each level in the hierarchy

3.4.3 Self-Organisation Mechanisms

vRMs self-organise within the same pSwitch to optimally manage CL-Resources and to satisfy resource prescriptions, thus, maximising their SI and evolving towards the local goal state. Similarly, pSwitches can self-organise within the same pRouter to maximise their SI to identify those parts of the system that are evolving towards their local goals. In principle, pRouters of the same CL-Resource type can also self-organise; however, that level of re-organisation is not considered further here since the added advantages are thought to be minimal. One example of Self-organisation scenarios can be described as follows.

Within the vRMs

1. A task comes into the pSwitch.
2. The pSwitch sends the task to an attached vRM with the highest SI.
3. The vRM checks to see if it has sufficient resources to execute the task.
   (a) If yes, no problem.
   (b) If no, the vRM initialises a self-organisation event within its cooperative.
4. The vRMs send updated metrics with their pSwitch.

Within the pSwitches

1. A task comes into the pRouter.
2. The pRouter sends the task to an attached pSwitch with the highest SI.
3. The pSwitch checks to see if there are sufficient resources to execute the task.
   (a) If yes, it passes the task to the vRM with the highest SI.
   (b) If no, the pSwitch initialises a self-organisation event within its co- operative.
4. The pSwitch sends updated metrics to its pRouter.

Within the pRouter

1. A task comes into the Cell.
2. The Cell sends the task to an attached pRouter with the highest SI of the desired type.
3. The pRouter checks to see if there are sufficient resources to execute the task.
   (a) If yes, passes the task to the pSwitch with the highest SI.
   (b) If no, the pRouter initialises a self-organisation event within its co- operative.
4. The pRouter sends updated metrics to the Cell.

Sample events that trigger re-organisation at each level in the hierarchy

- When weights are updated.
- As a result of an autonomous, periodic, housekeeping action designed to maximise the SI of the initiating component.
- After the arrival of a resource prescription that cannot be satisfied without re-organisation.

When all else fails: sample resource prescription rejection strategies

- Outright reject.
- Return prescription to the previous level and possibly trigger a re-organisation there.
- Recycle the task prescription into the system at the Cell level and record its recycle iterations until an upper limit is reached. If this limit is reached, reject.

3.5 CloudLightning SOSM Strategies

3.5.1 Self-Management Strategies

In the CloudLightning SOSM framework, each component is autonomous, which allows the component using different self-management strategies accordingly to achieve its local goal state.

Some self-management strategies may include:

- Static weights and dynamic weights (only for Cell Manager)
- Average aggregation (suitable for pRouters, pSwitches, and vRMs)
- Modifying weights for smoothing changes towards local goal state (suitable for pRouters, pSwitches, and vRMs)
- Bin-packing for energy efficiency (only for vRMs)
- Functions for management efficiency (only for vRMs)
- Isotropy preservation for task process parallelism (only for vRMs)

3.5.1.1 An Example Self-Management Scenario

Here, an example of examining the effect of different choices of management cost functions is presented. Four different functions are selected for inspection, characterising different types of evolution, which are described by the equations that follow.

(a) Small vRacks

$$1 - \int_{0}^{\frac{2N_{total}}{N_{total}-2}} e^{-t^2 dt} \tag{3.15}$$

Equation 3.15 favours small capacity vRacks enabling them to evolve; while when a vRack has large capacity, the output of the management cost function approaches zero resulting in a reduced SI. Thus, large vRacks are not capable of undertaking more requests, and they have to transfer their servers to other smaller vRacks in order to slowly achieve the ideal size.

(b) Large vRacks

$$1 - \int_{0}^{\frac{2\left(2\hat{N}_{total}-N_{total}\right)}{\hat{N}_{total}-2}} e^{-t^2 dt} \tag{3.16}$$

Equation 3.16 favours large capacity vRacks; when a vRack has small capacity, the output of the management cost function approaches zero

resulting in a reduced SI. Thus, small vRacks are not capable of undertaking more requests, and they have to transfer their servers to other larger vRacks merging with them.

(c) Medium vRacks

$$e^{\frac{-\left(-4+4\frac{N_{total}}{\hat{N}_{total}}\right)^2}{2}} \tag{3.17}$$

Equation 3.17 favours medium capacity vRacks; when a vRack has very small or very large capacity, the output of the management cost function approaches zero resulting in a reduced SI. Thus, very small and very large vRacks are not capable of undertaking more requests, and they have to transfer their servers or merge with other vRacks.

(d) Extreme vRacks

$$1-e^{\frac{-\left(-4+4\frac{N_{total}}{\hat{N}_{total}}\right)^2}{2}} \tag{3.18}$$

Equation 3.18 favours very small capacity or very large capacity vRacks; when a vRack has medium capacity, the output of the management cost function approaches zero resulting in a reduced SI. Thus, medium capacity vRacks are not capable of undertaking more requests, and they have to transfer their servers or merge with other smaller or larger vRacks.

Overall, the optimal number of servers per vRack is given by $\hat{N}_{total}=\frac{1}{N_v}\sum_{i=1}^{N_v}\left(N_{total}\right)_i$. This number is dynamic and is changing with the creation/destruction of vRacks or with the merging/splitting of vRacks. The management cost functions can be depicted schematically by (a), (b), (c), and (d) in Fig. 3.5.

However, the choice of management cost function significantly affects the evolution as well as other parameters and metrics of the systems such as utilisation and number of rejected resource prescriptions.

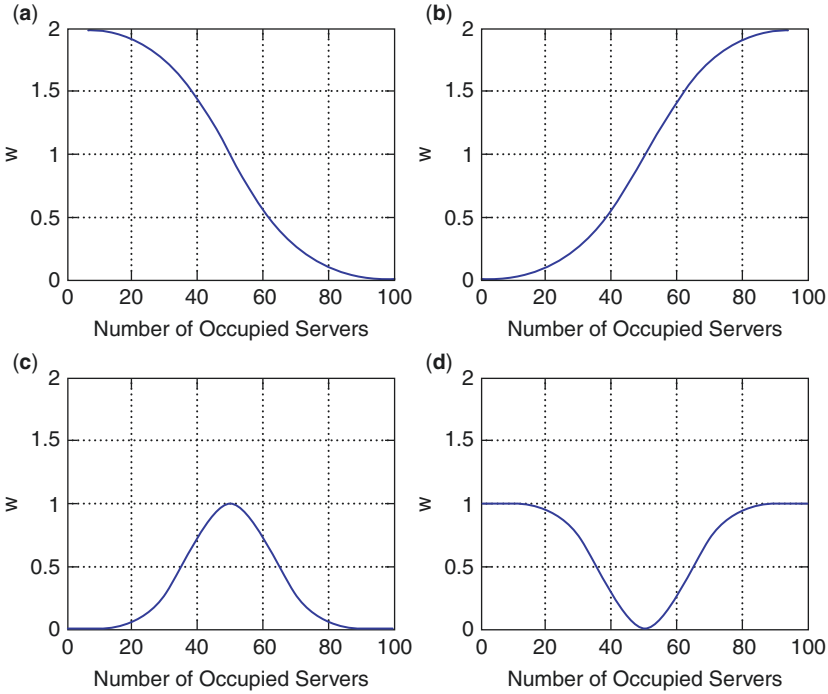


Fig. 3.5 Different types of management cost functions

3.5.2 Self-Organisation Strategies

The self-organising components in the system include vRMs and pSwitches. vRMs self-organise within the same pSwitch to optimally manage CL-Resources and to satisfy resource prescriptions, thus maximising their SI and evolving towards the local goal state. Similarly, pSwitches can self-organise within the same pRouter to maximise their SI to identify those parts of the system that are evolving towards their local goals. In principle, pRouters of the same CL-Resource type can also self-organise; however, that level of re-organisation is not considered further in this book since the added advantages are thought to be minimal.

Some self-organisation strategies may include:

- *Dominate*: the component with the greater SI has precedence and can demand another component of the same type, but with a lower SI, to transfer some resources.
- *Win-Win*: components may cooperate to exchange resources to maximise the SI of each.
- *Least Disruptive*: minimise disruption with respect to management and administration.
- *Balanced*: maximise load-balancing among each cooperating component.
- *Best Fit*: minimise server fragmentation and/or minimise network latency (this strategy may come from some vRM-specific objectives).
- Any meaningful combination of the above.

3.5.2.1 An Example Self-Organisation Scenario

An example of a Least Disruptive algorithm that can be used by vRMs for self-organisation is presented. This algorithm can be used by vRMs to exchange resources to minimise their management cost. This algorithm has two steps: the first function endeavours each vRM to minimise the number of administrative actions, and the second function is taking virtualisation and fragmentation into account, which can be used to avoid the creation of very large vRMs for management efficiency purpose. This two-stage self-organising scheme can be described by the algorithmic procedure given by the following algorithm.

Algorithm 1

Let ρ be the minimum number of vRacks allowed per pSwitch

Let j be the index of the vRack with maximum Suitability Index

Let rp be a resource prescription arriving to vRM_j

Let p_j be the set of free resources belonging to vRM_j

function MINADMINCOSTS(rp)

$a = \emptyset$

$t = \emptyset$

if $p_j < rp$ **then**

$required = rp\text{-}|p_j|$

for $i \leftarrow 1$ to N_v with $i \neq j$ **do**

$send\ request\ to\ acquire\ free\ resources\ from\ vRM_i$

$receive\ p_i\ from\ vRM_i$

$a = a \cup \{i\}$

$t = t \cup p_i$

if $required \leq 0$ **then**

$remove\ exceeding\ resources\ from\ t$

$required = 0$

break

$send\ request\ to\ vRMs\ in\ a\ to\ acquire\ resources\ in\ t$

$receive\ resource\ handlers\ from\ vRMs\ in\ a$

if $|p_j| \geq \frac{rp}{2}$ **then**

return $resource\ handlers\ to\ Gateway\ Service$

else

$create\ new\ vRM_k\ with\ resources\ p_i \cup t$

return $resource\ handlers\ to\ Gateway\ Service$

function TWOSTAGESO(rp)

if MINADMINCOSTS(rp) does not return $resource\ handlers$ **then**

if $|p_j| \leq \widehat{N}_u$ and $N_u \geq \rho$ **then**

for $i \leftarrow 1$ to N_u with $i \neq j$ **do**

Probe i-th $vRM\ for\ resources, so\ that\ p_j \cup p_i\ can\ service\ rp$

if $p_j \cup p_i\ can\ service\ rp$ **then**

Merge vRM_i with vRM_j

return $resource\ handler\ from\ resulting\ vRM\ to\ Gateway\ Service$

$rejection\ to\ Gateway\ Service$

else

return $resource\ handlers\ obtained\ from$ MINADMINCOSTS(rp) $to\ Gateway\ Service$

Figure 3.6 presents the increased system utilisation and requests reject rate of this two-stage self-organisation algorithm merging with the minimum free resources. However, because the system accommodates larger tasks through merging, the smaller tasks arriving at the system are continuously rejected due to lack of resources.

In the case of merging with the vRack with maximum free resources, the utilisation of the system, depicted in Fig. 3.7a, is slightly increased but oscillates around 80%. As a consequence, the percentage of rejected requests increases, since the system is accommodating an increased number of larger requests, as schematically represented in Fig. 3.7b.

Overall, this two-stage self-organisation strategy has been employed for enhancing utilisation and reducing fragmentation with virtualisation in mind.

3.6 Conclusion

The SOSM framework described in this chapter provides a general and scalable mechanism for hosting and executing SOSM strategies that, in principle, could be associated with any hierarchical architecture.

The key elements of the self-management and self-organisation framework include the process of Directed Evolution; an Impetus that drives the evolutionary process at all levels in the hierarchy; a Perception, associated with each component, indicating the effectiveness of the system underlying that component; and an SI, associated with each component, that determines how close that component is to achieving its goal state. Specifying an objective global goal state may be based on business decisions and/or technology constraints, however, to optimise the CloudLightning system in its entirety; it is suggested that the goal states for components of the system should be chosen to maximise their respective SIs.

This approach introduces a great deal of flexibility into the evolution of a system by allowing it to achieve stasis while attempting to balance local constraints with the external Impetus derived from the directed evolutionary process. Over time, the system as a whole evolves to optimise typical service usage, to achieve the dynamic equilibrium. The local constraints are most evident at the vRM level where they are embodied in assessment functions capturing the essential characteristics of the underlying resources.

The framework endows the system being specified with the flexibility to extend the resource fabric in a seamless fashion. This elegantly addresses the CloudLightning objective of readily supporting heterogeneous hardware now and into the future.

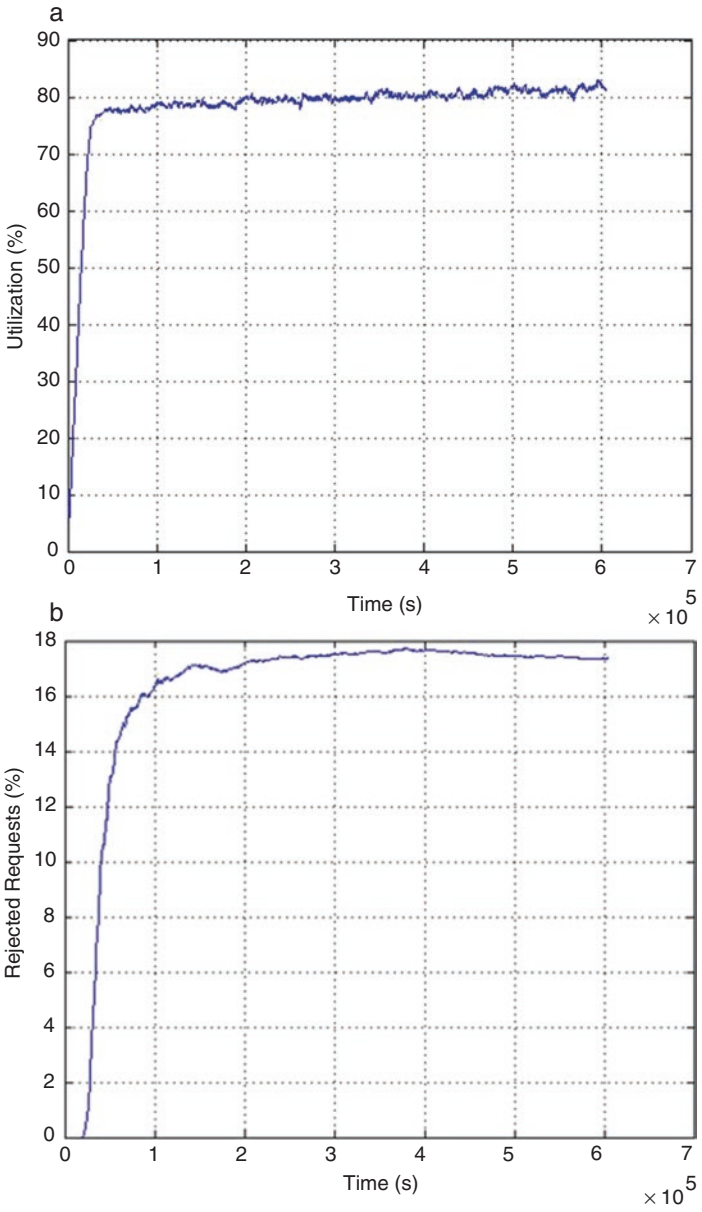


Fig. 3.6 The system utilisation (**a**) and requests reject rate (**b**) of two-stage self-organisation algorithm merging with the minimum free resources ($\rho = 3$)

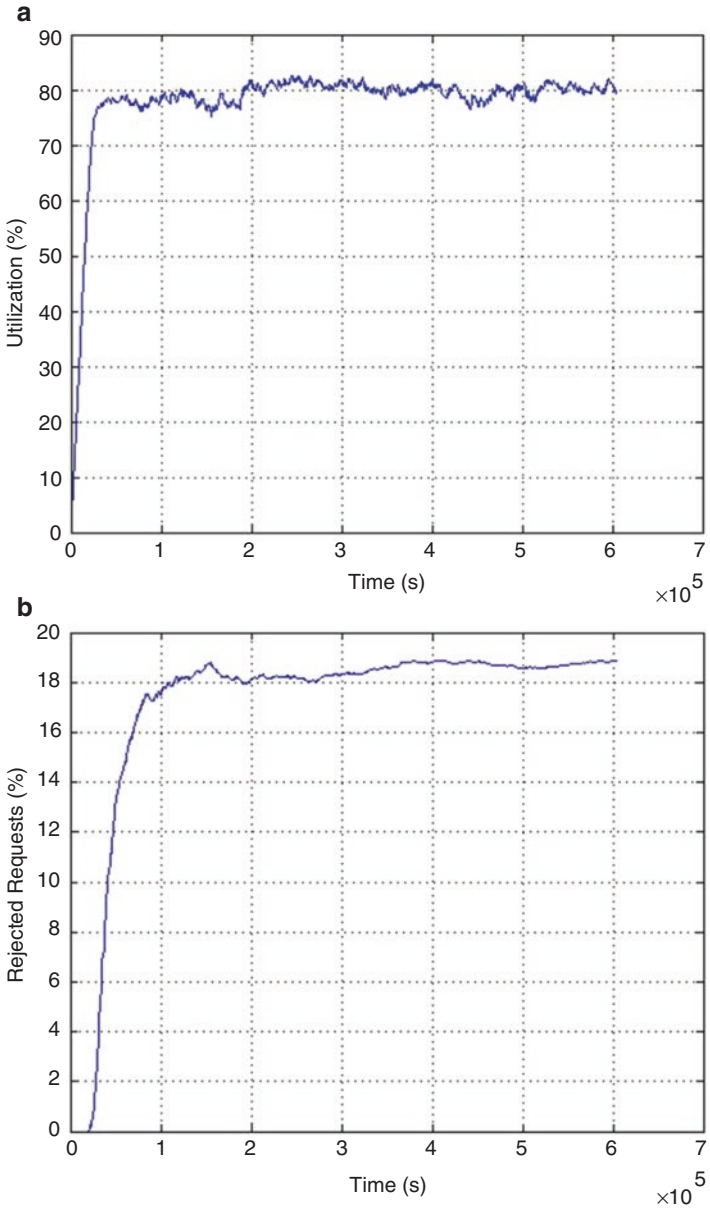


Fig. 3.7 The system utilisation (**a**) and requests reject rate (**b**) of two-stage self-organisation algorithm merging with the maximum free resources ($\rho = 3$)

3.7 Chapter 3 Related CloudLightning Readings

1. Drăgan, I., Fortiş, T. F., Iuhasz, G., Neagul, M., & Petcu, D. (2017). Applying self-* principles in heterogeneous cloud environments. *Cloud Computing*, 255–274. Springer International Publishing.
2. Filelis-Papadopoulos, C., Xiong, H., Spataru, A., Castane, G., Dong, D., Gravvanis, G., et al. (2017, July). A generic framework supporting self-organisation and self-management in hierarchical systems. In *The 16th International Symposium on Parallel and Distributed Computing (ISPDC 2017)*. Innsbruck, Austria.
3. Petcu, D. (2015). On autonomic HPC Clouds. In *Proceedings of the Second International Workshop on Sustainable Ultrascale Computing Systems (NESUS 2015)* (pp. 29–40).
4. Stack, P., Xiong, H., Mersel, D., Makhloufi, M., Terpend, G., & Dong, D. (2017). Self-healing in a decentralised Cloud management system. In *Proceedings of the 1st International Workshop on Next generation of Cloud Architectures*, Vol. 3. ACM.

Open Access This chapter is licensed under the terms of the Creative Commons Attribution-NonCommercial-NoDerivatives 4.0 International License (http://creativecommons.org/licenses/by-nc-nd/4.0/), which permits any noncommercial use, sharing, distribution and reproduction in any medium or format, as long as you give appropriate credit to the original author(s) and the source, provide a link to the Creative Commons license and indicate if you modified the licensed material. You do not have permission under this license to share adapted material derived from this book or parts of it.

The images or other third party material in this chapter are included in the chapter's Creative Commons license, unless indicated otherwise in a credit line to the material. If material is not included in the chapter's Creative Commons license and your intended use is not permitted by statutory regulation or exceeds the permitted use, you will need to obtain permission directly from the copyright holder.

