Autonomous Placement and Migration of Services in Heterogeneous Environments

CUNEYT CALISKAN

KTH Information and
Communication Technology

Master of Science Thesis
Stockholm, Sweden 2012

TRITA-ICT-EX-2012:266

KTH Computer Science
and Communication

Autonomous Placement and Migration of Services in Heterogeneous Environments

CUNEYT CALISKAN

Master's Thesis at Technische Universität München
Supervisor: Marc-Oliver Pahl
Examiner: Prof. Vlassov, V.

TRITA xxx yyyy-nn

Abstract

In this thesis, we present an autonomous placement protocol for services in smart spaces. Proposed design is a combination of computing Grids and Intelligent Agent Systems which is able to adapt to environmental changes. These changes are failing/joining/leaving nodes, changing node usage, failing/joining/leaving services, and changing service demands. Smart spaces are heterogeneous in terms of available resources for consumption and they are dynamic where available resources and services change over time. The system adapts to environmental changes by live service migration and load balancing, and provides high availability by maintaining backup replicas of services. Load in the system is balanced among available resources by taking into account heterogeneity of the environment. Complex nature of the problem space makes it difficult to manage the services and resources manually. Thus, all functionality provided by the system is fully autonomous. A novel approach is presented for migration decisions based on utility functions that represent characteristics of nodes. Fitness of the designed protocol is tested with simulations under different circumstances. Obtained test results show that it provides high degree of availability to services and adapts to environmental changes.

Referat

Autonoma placering och migration av tjänster i heterogena miljöer

I denna uppsats presenterar vi ett autonomt placeringsprotokoll för tjänster i smarta utrymmen. Den föreslagna utformningen är en kombination av datornät och intelligenta agentsystem som kan anpassa sig till förändringar i omgivningen. Dessa förändringar är felande/anslutande till/lämnande av noder, förändrat nodanvändande, felande/anslutande till/lämnande av tjänster och förändrat tjänsteanvändande. Smarta utrymmen är heterogena i termer av tillgängliga resurser för utnyttjande och de är dynamiska där tillgängliga resurser och tjänster förändras över tid. Systemet anpassar sig till förändringar i omgivningen genom tjänstemigration och belastningsbalansering, samt tillhandahåller hög tillgänglighet genom att bibehålla backup-kopior av tjänster. Belastningen i systemet balanseras mellan tillgängliga resurser genom att ta med omgivningens heterogenitet i beräkningen. Den komplexa naturen av problemutrymmet gör det svårt att hantera tjänsterna och resurserna manuellt. Därför är all funktionalitet som tillhandahålls av systemet fullständigt autonom. En ny metod presenteras för migrationsbeslut baserade på nyttofunktioner som representerar noders egenskaper. Det utformade protokollets lämplighet testas med simulationer under olika omständigheter. Erhållna testresultat visar att det tillhandahåller en hög grad av tillgänglighet till tjänster och anpassar sig till förändringar i omgivningen.

Contents

List of Figures

List of Algorithms

Chapter 1

Introduction

The interest on smart spaces is growing day by day because they bring ease to daily lives of human beings. As this interest grows, the expectations about the services provided by smart environments grow, too. A large number of services are provided already. Controlling the heating system, illumination, media devices, air conditioning, security, surveillance, kitchen appliances and many others are examples of such services. A situation where users want to turn on the heating system in cold weather but the system is not responding is not desirable. Another similar situation may occur when trying to turn on the security or surveillance system. Smart environments offer critical services to their users. Thus, unpleasant situations are not welcome. Users of a software need it to be available whenever they want to use it no matter what happens to the computers/devices on the system site.

The goal of this thesis is to perform a research to provide a reliable infrastructure for services designed for smart environments. The goal is aimed to be achieved by providing an autonomous service placement protocol. The reason for the autonomy is that the environment is complex to be maintained manually. Complexity of the system raises from the large number of computational resources in the environment, different characteristics and dynamism of these resources.

This thesis aims at using all computational resources at smart environments such as personal computers, notebooks, resources of smart devices such as refrigerators, TVs and other devices. This variety of resources results in a heterogeneous environment in terms of available resources. The aim is to take into account this heterogeneity while making decisions about service placement with balancing load on the resources proportional to their capacities. The load on a resource is the fraction of resource demand to resource's capacity. Resources are connected to each other with network links where connectivity of each resource is different than the others'. This connectivity leads to a distributed system where resources are located at different physical locations. For example, wind sensors can be installed outside of building, on roof for instance, controlling unit of these sensors where data are sent can be placed in the basement of the building and current state about wind speed can be displayed on a personal computer. Depending on the wind speed,

shutters on windows can be controlled by the controlling unit where actuators of the shutters are placed outside the windows.

Services in smart spaces are dynamic where new services can be added or existing ones can be removed. Some services can fail due to implementation or other unexpected factors. The same dynamism applies also for the computational resources in the environment. Some resources can be removed or new ones can be introduced by users. They can fail or lose connection with the network. Also the available resources can change over time. These changes on resources and services constitute environmental changes where the goal is to adapt to these changes autonomously.

Smart spaces are heterogeneous, distributed and dynamic environments. The goal is to provide a reliable infrastructure for services in these environments and providing an autonomous service placement protocol that will adapt to the dynamic nature of the problem.

Contributoin

In this thesis, a reliable infrastructure for services in smart environments is designed. Research is conducted in different domains for similar problems. To the best of our knowledge, there is no research in this area which aims at exactly the same goals that we do as the time of writing this document (28 September 2012). A prototype for the proposed design is implemented by adapting solutions from different domains. However, due to missing functionalities of DS2OS, the design was tested with simulations because our solution depends on DS2OS which is explained throughout the document.

Chapter 2

Analysis

Smart spaces are heterogeneous, distributed and dynamic. Heterogeneity is in terms of available resources for consumption and connectivity. The system is distributed with several nodes located in physically different locations. And the system is dynamic where nodes and services can join and leave the system, connectivity and available resources can change. The dynamism of the system requires failure detection mechanisms for node and service failures. A self-organizing mechanism for providing autonomy because of the complex nature of the system is required. Dynamic nature of the problem requires dynamic load balancing solutions. Mobility in the system is a requirement in terms of replication and migration as a result of the desired properties such as availability and efficient resource consumption. The heterogeneous and dynamic nature of the environment requires a runtime environment to handle these difficulties.

This chapter starts with the introduction of the currently existing smart environments in 2.1. Then, it continues with smart spaces and their characteristics in 2.1.1. After introducing smart spaces, managed entities in smart spaces are addressed in 2.1.2. Afterwards, desired properties are introduced in 2.2 including availability, autonomy, load balancing, mobility together with migration and replication, and a runtime environment that can handle our requirements. Finally, this chapter concludes with the questions that are going to be answered throughout this thesis in 2.3.

2.1 Existing Infrastructure

It is stated in Chapter 1 that all available resources in smart spaces are aimed to be used. Smart spaces may include many nodes with different capabilities such as storage, memory or processing power. The large number of nodes makes the environment *heterogeneous* in terms of available resources for consumption. All these nodes may be located in physically different locations and be responsible for performing specific operations. Some of them can be used for storing data while some others can be used for handling user requests. Although the nodes are responsible

for specific operations, they all provide service to smart space users. Thus, forming a *distributed* system to serve for a common goal. Beside being heterogeneous and distributed, the system is also *dynamic* in terms of node failures, connectivity speed, resources and deployed services. Heterogeneity, distribution and dynamism are the main challenges that we choose to address and overcome as they were the challenges for other problems in different domains [RLS+03, AH04, KS97, NS05, WS01]. Details about these challenges are presented in the following subsection.

2.1.1 Smart Spaces and DS2OS

Smart spaces contain sensors and actuators. Sensors in smart spaces provide information about the real world. This information is processed by electronic control elements and physical actions are reflected on real world via actuators. There is no common communication language for these sensors and actuators. Vendors have their own communication protocols specific for devices. These protocols are different even among different domains of a vendor. This causes a vast variety of communication protocols. Devices are distributed in different physical locations. For example, the temperature sensors can be located in the living area within different rooms, the control unit can be in the cellar whereas the actuator for adjusting the temperature is located in the basement. Temperature regulation becomes a distributed task. This requires a routing mechanism to address the devices and route information to actual receivers. Connectivity characteristics among these devices can differ. Devices in smart spaces are added and removed continuously which results in a dynamic environment. A single program would be very complex to orchestrate the whole smart space. Thus, small services for specific purposes will be deployed in smart spaces. For example, there can be an alarm service that can trigger another service to play music in the morning and can turn on the lights when the alarm is activated. Then, coffee machine and floor heating in the bathroom can be activated. A dynamic coupling of services is possible to create macros to perform a complicated task like mentioned. Isolated systems, such as the illumination system within a house, have dedicated communication channels between the control units to devices. In decentralized smart spaces, there are no such direct connections. This brings the problem of security and trust. Data may have been tampered and altered in gateways or other places inside the system [PUC].

All of the challenges mentioned above are overcome by the distributed smart space operating system, DS2OS, developed in Technical University of Munich [PUC]. DS2OS abstracts the heterogeneity of devices by maintaining a virtual model of the real world. The virtual model resides in the middleware and it is a tree structure that holds the state information of entities and services which provides location transparency. The abstraction makes it possible for services to control the devices in a vendor independent way. The virtual model is shared among all entities in the system and has access rules that can be set by the creator of the sub-tree. DS2OS offers a limited set of interaction with the model which are basically getting and setting the state of a device or service. DS2OS also provides a publish-subscribe

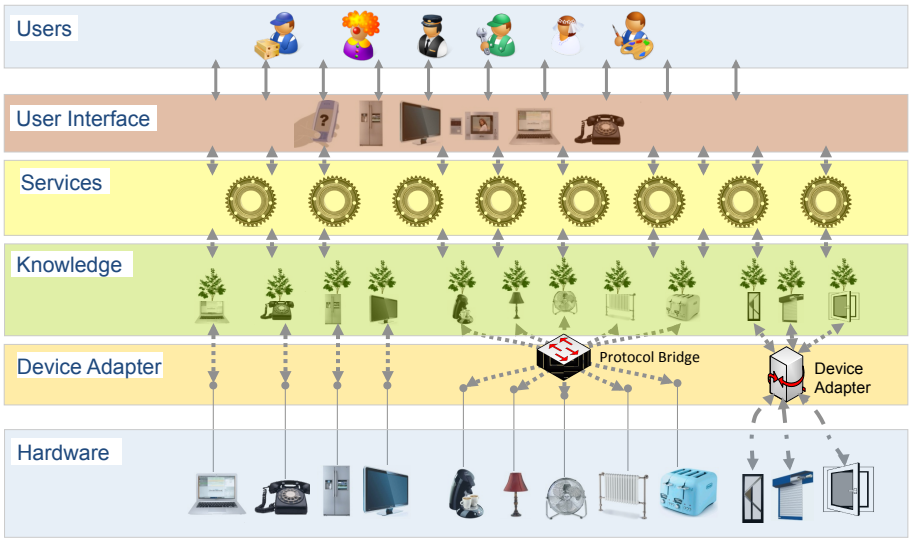


Figure 2.1. Layered system structure [PNS+09]

mechanism. Services can subscribe to an address in the virtual model tree and whenever a change occurs on the address, a notification is sent to the subscriber. This provides a dynamic coupling of services. For example, the scenario about the alarm given above. The bathroom floor heating, music and coffee making services can subscribe to the state address of the alarm service. When the state is changed, these services receive notification depicting the changed address. Then, these services can query the state and activate their own actions depending on the state.

Figure 2.1 depicts the layered structure of DS2OS. As it is seen in the hardware layer of the figure, a wide variety of devices are monitored and controlled. The device adapter layer connects the devices to the knowledge layer. Highly abstracted knowledge layer contains digital representation of these devices in a real world model. The service layer contains high level control services that run on nodes with sufficient resources. Finally, the user interface provides interaction between users and the system. The knowledge layer provides location transparency to services. All data required for the operation of a service are provided by the knowledge layer. Thus, a service can operate independent from its location and can be migrated among different runtime environments with sufficient resources. It is stated in Chapter 1 that the goal is to be achieved by providing a placement protocol for services. The operational layer of the placement protocol is the service layer of DS2OS.

Heterogeneous System

Smart spaces are equipped with various kinds of sensors and actuators. Examples of these sensors are: humidity, temperature, light, barometric pressure, GPS-receiver and accelerometer. Some examples of devices that can be orchestrated via actuators in smart spaces can be seen in the hardware layer of Figure 2.1. Also some examples of devices that a user can use for interaction with the system is shown in the user interface layer of the figure. As it is seen from the figure, users can interact through a notebook, personal computer, mobile phone, personal digital assistant (PDA), an embedded computer and many other devices. All these devices differ in capabilities such as storage capacity, memory (RAM), processing power (CPU), network connectivity and operating power (limited battery life for mobile devices). Thus,the environment is a heterogeneous system. Two sample nodes running in a system can be thought as an embedded computer and a server. The embedded computer with system parameters as 700 MHz of CPU, 256 MB of RAM and 1 Mbps bandwidth. The server with system parameters as 8 cores and each 3200 MHz of CPU, 32000 MB of RAM and 1 Gbps bandwidth.

It is not possible to treat all nodes equally in a heterogeneous environment. If we take the above mentioned two sample computers and deploy same amount of work to both of them. It can be said that the embedded computer operates much slower compared to the other one. This situation depends on the type of work deployed because embedded computers are desinged to process specific tasks. It is possible that the clients of the services deployed in the embedded computer are not satisfied by this situation. The logical decision about deploying the jobs would be to treat the computers according to their capabilities. This can be thought as a load balancing mechanism which uses system resources in a fair way. As it is stated in Chapter 1, we aim at dealing with the challenge of heterogeneity to place services on nodes proportional to their capabilities. The operational area of this thesis is at the service layer of DS2OS architecture where high level control services are located.

Distributed System

In the context of this thesis, a distributed system is a collection of software components operating over physically distributed and interconnected nodes. As it is stated in Chapter 1, smart spaces constitute distributed systems with several sensors, actuators and nodes placed on different locations and interconnected with communication links. Distributed systems are classified based on abstractions about processes/nodes and time [Gue06].

In the context of this thesis, a process p is said to be *correct* if it behaves as intended and sends messages according to specifications. An *incorrect* process is the one that does not send any messages at all because it has crashed or sends messages but not compliant with specifications. The abstraction about processes is that the model is either *crash-stop* or *crash-recovery*. In a crash-stop model a crashed process never sends and receives messages again which means the process is

not operating any more. In crash-recovery model, however, a process stops sending messages for a period of time due to its failure but it starts to send messages and operate again after it recovers or restarts. Nodes in smart spaces are classified in crash-recovery model because they can fail and recover or restart.

The abstraction about time defines if a system is synchronous, asynchronous or partially synchronous. A *synchronous* system is the system where there is a bounded time in which an operation is completed while there is no timing bound in an *asynchronous* system. However, there exists another kind of system which is in between, *partially synchronous* system. In partially synchronous systems there is a timing assumption but it is not fixed as in synchronous systems. Assume that the time for an operation to complete is t. If the operation does not complete within this amount of time, the node is suspected or detected as incorrect. However, this might not be the case. It is possible that the communication channel is slow and the response is received later. In this case, the timeout value t is increased with an amount of Δ, $t = t + \Delta$. The next time when an operation is performed, the timeout value to wait for the response will be larger. After a while, t reaches a value in which all the operations are completed and the system becomes synchronous. In this thesis, partially-synchronous system model is referred whenever time is referred. As it is said earlier, we have a distributed system where nodes can be located at different physical locations. But they are interconnected in order to serve for a common goal and these connections among nodes differ in terms of speed. Thus, there is no bounded time limit in which a message is delivered to its destination. It might take x milliseconds for a message to be delivered from node A to node B while it might take y milliseconds to deliver the same message from node A to node C where $x! = y$. The same message on the same route can also take a different amount of time when it is sent next time depending on the network traffic on the route it travels. Taking all these conditions into consideration, smart spaces are partially synchronous systems.

Dynamic System

It is already mentioned in the previous section about the dynamism of nodes that they can fail and recover. However, the dynamism of the system is not only because of node failures but also because of the dynamism of connectivity, resources and services. The dynamism of nodes also includes actions taken for maintenance purposes. For example, the system administrator may want to remove a node from the system manually and place a new one instead. When there are no longer available resources for satisfying the users, a new node may be necessary to install to the system. As it is mentioned about the heterogeneity of the connections in Section 2.1.1, the connectivity capabilities may also change over time. Because of the current Internet infrastructure, messages sent to the same destination can be routed on different routes depending on the traffic. This causes the messages sent to the same destination to take different times to travel. The services deployed in a system are also dynamic. New services can be installed, existing ones can be updated

with newer versions or uninstalled from the system. Services can also fail due to buggy implementation. There might be also services that are run only once for some special purpose such as installing a device driver. In this case, a service of this kind will join the system and leave right after completion. The nodes in the system do not necessarily need to be dedicated computers for the operation of system. A personal computer or a notebook can also host some services and the resources on these nodes can change whenever users start to use them for personal needs. All in all, nodes and services can leave and join the system and the connectivity can change over time just like the resources.

2.1.2 Managed Entities

Various types of services may be provided by smart spaces. Some examples of services are already given in Chapter 1. These services are deployed in the above mentioned heterogeneous, distributed and dynamic system. Services are designed and developed for specific purposes. They require resources to operate in order to fulfill their design goals. Resources demanded by services can be dynamic and may change over time. New services can be deployed or existing ones can be removed from the system. They can fail due to buggy implementation or unexpected reasons such as running out of resources. Hence, the resources demanded from a node also change over time depending on the services deployed on it. Nodes can also fail and recover or new ones can be introduced and existing ones can be removed.

Services and nodes are the managed entities in smart spaces and they can be managed in various ways. Placement of a service is one operation that comes to mind. The system is distributed and heterogeneous, so, problem is on which node to place a given service. If a service has high memory requirements, it would not make any sense to place it on an embedded computer with low memory capacity. Likewise, it would not be wise to place a service with high computation power requirements on an embedded computer with low computation power. These services are also controllable via user interfaces as it is mentioned in Section 2.1. Users can start, stop, uninstall, update or install services. Services are migrated and replicated for various goals such as load balancing, energy saving and providing availability. A service may be replicated in several different nodes to increase its availability or to decrease latency that users experience as a result of interaction with the system. Migration of a service is another operation that can be performed for load balancing purposes. Another reason for migration can be for having an energy efficient system. When there are few services in the system, all services can be migrated to only one or two nodes while others can be shut down to save energy.

2.2 Desired Properties

The goal of this thesis is stated in Chapter 1 as providing a reliable infrastructure for services in smart spaces by an autonomous service placement protocol that adapts to environmental changes. Reliability can be achieved by the availability of nodes

and services [AH04]. The complex nature of the problem space is difficult to manage entities manually but autonomic system behaviour can provide low complexity and eliminate manual (re)configuration [Her10]. Adaptation to environmental changes requires dynamic load balancing on the nodes through mobility of services [RLS+03]. Availability of services can be achieved with migration and replication of stateful services [LAB+].

2.2.1 Availability

In the context of this thesis, availability means that the system is responding whenever a user requests a service. It is almost inevitable that some nodes or services will stop functioning. Some reasons of being unavailable are failure of nodes, system maintenance, network partitioning and buggy implementations. Whenever a node stops functioning due to any reason, the services running on that node are no longer available to clients. Being aware of functioning defects requires a mechanism for *failure detection* [Gue06]. As described in Section 2.1.1, the system is not synchronous. This makes failure detection more difficult compared to failure detection in synchronous system which are introduced in Section 3.3. If time required to complete an operation such as delivering a message is known, then a node can be said to be crashed when the message is not delivered within that time. However, there is no such time limit in which every operation is completed. Thus, deciding on the failure of a node is not straightforward. There might be some cases such as some links between nodes are down or slow but the nodes are operating. The messages may take different times to be delivered because of the network traffic or the node capabilities. In such cases, making a decision about the failure of a node may be costly. False detection of nodes will put extra burden to the system because the services on the detected node are recovered on another node. Thus, there will be a waste of resources whereas efficient resource consumption is among the goals of this thesis.

Failure detectors have requirements regarding actually crashed nodes and actually alive nodes in a system. The former requirements regarding actually crashed nodes are called *completeness* requirements. The latter requirements are called *accuracy* requirements [Gue06].

Completeness has two different kinds and they are as follows:

- **Strong completeness:** every crashed node is *eventually* detected by **all** *correct* nodes. There exists a time after which all crashed nodes are detected by all correct nodes.

- **Weak completeness:** every crashed node is *eventually* detected by **some** *correct* node. There exists a time after which all crashed nodes are detected by some correct node, possibly by different correct nodes.

Accuracy has four different kinds and they are as follows:

- **Strong accuracy:** no correct node is ever suspected. It means that for all pairs of nodes p and q, p does not suspect q unless q has crashed. This type of accuracy requires synchrony.
- **Weak accuracy:** there exists a correct node which is never suspected by any node. It means that a correct node p is always "well connected".
- **Eventual strong accuracy:** after some finite time the failure detector provides strong accuracy.
- **Eventual weak accuracy:** after some finite time the failure detector provides weak accuracy.

Failure detectors are grouped according to their completeness and accuracy properties as well as the timing assumption whether they are applicable in synchronous or asynchronous systems. Table 2.1 summarizes these different kinds of failure detectors.

	Strong Completeness	**Weak Completeness**	
Synch.	Perfect Detector (P)	Detector (Q)	**S. Accuracy**
	Strong Detector (S)	Weak Detector (W)	**W. Accuracy**
Asynch.	Eventually Perfect Detector ($\Diamond P$)	Eventually D. ($\Diamond Q$)	**Eventually S. A.**
	Eventually S. D. ($\Diamond S$)	Eventually W. D. ($\Diamond W$)	**Eventually W. A.**

Table 2.1. Failure Detectors

Failure of nodes is one of the aspects in availability. Another aspect is the failure of a single service. Services running in the system can also fail due to buggy implementation or any other unexpected reason. Thus, a mechanism to monitor the state of a service can also be useful if it is operating properly or not. When a service failure is detected, a recovering protocol can be applied.

2.2.2 Autonomy

In the context of this thesis, "autonomy" means making decisions without any external intervention. The environment is heterogeneous, distributed and dynamic which makes it complex for manual maintenance. Thus, a self-organizing system which has no external intervention eliminates the complexity and manual intervention. The system needs to autonomously adapt itself to changes in the environment in order to provide availability. Whenever a node becomes unavailable due to any reason or runs out of available resource, the system needs to detect this and take actions based on the decisions it makes [Her10].

Some challenges arise during providing autonomy. One of the problems is choosing parameters that constitute the base for decisions. Another one is how to decide when to take action and what action to take. The system can decide to perform any operation on a service as mentioned in Section 2.1.2. If the decision is to migrate

a service, two new decision problems arise. First one is to decide which service to migrate. The second one is to decide where to migrate it. If the decision is to replicate a service, new problems arise again. First decision to make is to determine how many replicas are going to be maintained. Then, for each of these replicas, where to place it. The system itself needs to decide where to place a newly installed service according to the current state of the resources. While making decisions to manage services, some constraints need to be satisfied. For example, a service and its replica are never placed on the same node. Similarly, two replicas of a service are never placed on the same node. The capabilities of nodes should be taken into consideration while choosing the node to place a service or a replica.

2.2.3 Load Balancing

As it is explained in the existing infrastructure in Section 2.1, resources in smart spaces are heterogeneous and dynamic. Entities in smart spaces are services and nodes. A service requires certain amount of resources to fulfill its operations. And these resource demands may change over time. It is possible that a node hosting several services may run out of resources and become overloaded. An overloaded node means that the resource demands on the node are greater than its capacity. In case of existence of overloaded nodes, excess load can be shared with other nodes that are not overloaded, a load balancing needs to be performed. In other words, utilization of all nodes equally. By utilization, we mean the resource consumption of a node proportional to its capacity. For example, a node that can perform 100 units of work per time is assigned 75 units of work. Utilization of this node is %75. The goal of load balancing is to utilize all nodes at the same level [RLS+03].

2.2.4 Mobility

The term mobility in this thesis means transferring a service or a service replica from its current runtime environment to another one. Adaptation to environmental changes is possible through mobility of services among nodes with sufficient resources [RLS+03]. Availability of services or low service response time of systems is possible via migration and replication of services [Her10]. However, live migration of software requires capturing and re-establishing the state of a software [BHR+02, BLC02, Fün98]. Mobility has different forms depending on its degree as shown on Figure 2.2. When the program code together with its parameter set is transferred to a remote site and its execution is completed there, it is called **remote execution**. Upon completion, the computation results are returned back to its issuer. The issuer of the execution selects the remote site in this form of mobility. A similar form of mobility where the program code is transferred before starting the execution is called **code on demand**. The destination itself is the issuer of this mobility. These two forms are also said to provide only "code mobility" because the transfer is performed before the execution starts. When both program code and its data state are transferred, it is called **weak migration**.

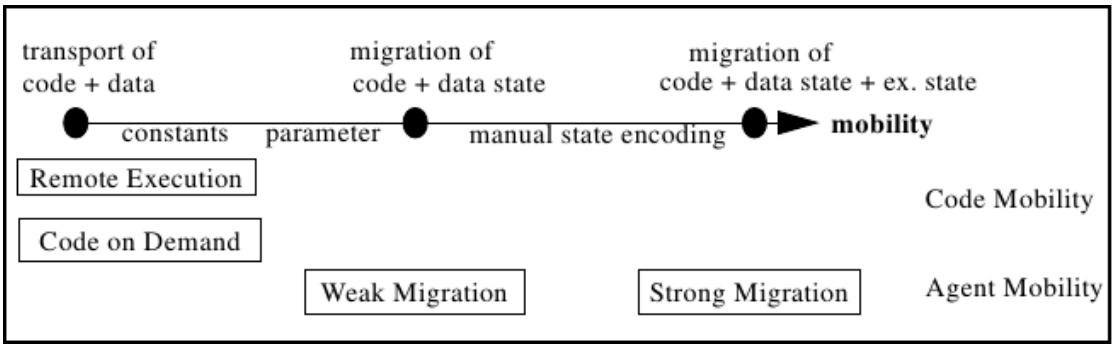


Figure 2.2. Taxonomy of Mobility [BHR+02]

Data state. In object-oriented programming languages data are organized in classes. A class is composed of data, stored in named fields and code structured into named methods. An instance of a class is called object. Objects encapsulate a collection of data, possibly references to other objects and a set of methods that can be invoked to manipulate that data. These methods can also have local fields that are valid only in the scope of methods. Thus, local variables are not considered in the concept of data state. Data contained in the fields of an object constitute the data state.

In weak migration, the migration takes place during execution. Thus, putting it in the category of "agent mobility". In weak migration, the execution after transfer continues from a predefined point such as a certain method to start. The last and strongest form of mobility which is also in the category of agent mobility is **strong migration**. In addition to weak migration, it also supports the transfer of the execution state of the program[BHR+02]. The execution continues exactly from the point that it was suspended.

Execution state. Each thread of a running program has its own program counter (PC). It is created when the thread is created and it contains the address of the current instruction being executed by that thread. PC is one of the entities in execution state. The local variables of methods are also included in execution state. Other additional information is also included depending on the runtime environment. For example, Java Virtual Machine (JVM) includes the Java stack and the stack frame in execution state. Stack is a data structure that works in last-in-first-out strategy. It has three operations: push, pop and top. Push stores an entry in the stack. Pop removes the entry on top of the stack. Top returns the entry on top of the stack. The Java stack stores frames per thread. When a method is invoked by a thread, a new frame is pushed onto that thread's Java stack. A stack frame includes local variables, operand stack and frame data. Local variables are stored on a zero-based array. The operand stack is used as a workspace. Values are popped from the stack, operations are performed on them and the result is pushed back to the stack. Frame data includes information such as exception dispatching

and method returns.

Mobility includes migration and replication in the context of this thesis. Migration and replication brings an additional dynamism to the system. We have a very dynamic environment where services and their replicas can be migrated among different runtime environments.

2.2.5 Migration

Migration of services in the context of this thesis means preemptively suspending the operation of a service in its current runtime environment. Then, transferring it to another runtime environment in a different physical location. Finally, resuming the operation of the service in its new location. The reasons for migration can be load balancing and energy saving as mentioned in Section 2.1.2. Migration needs capturing and reestablishing the state of a service. Detailed explanation about states is given in Section 2.2.4. The destination and the service to be migrated need to be selected by the system by considering the current state of the system resources. Migration also has a constraint that needs to be considered while selecting the destination. This constraint is that a service and its replica never exist in the same location.

2.2.6 Replication

Replication of services in this thesis means keeping a number of exact consistent copies of a service in different physical locations. Consistency means having the same state on all of the instances of a service. Software replication is a cheaper solution for reliability compared to hardware replication because producing new copies of a developed software is done at no cost. However, in hardware replication, multiple physical computers are deployed as backups to be activated in case of failure of the active machine. Replication has two different fundamental classes of techniques, *primary-backup* replication and *active* replication. In primary-backup replication strategy, one of the replicas, called *primary*, plays the role of a mediator for all the other replicas. All other replicas, called *backups*, just receive state update messages from the primary and do not interact with the issuer of requests. Upon receiving a request from a client, the primary processes it, updates its state and generates a response. After generating the response, it sends messages to backup replicas to update their states. Upon receiving acknowledgements from all of correct replicas, the primary sends back the response to the client. In active replication technique, there is no mediator and all of the replicas are playing active role in processing the requests. However, there is need for a front-end process to deliver requests to all replicas. Every request is sent to all replicas by the front-end process. Each replica processes the request, updates its state and sends back the response. It is up to the front-end which response to send back to the client. Sending the first response received is one of the options. Another option is to collect all responses and decide on the one that is common to all or majority of replicas.

2.2.7 Runtime Environment

The heterogeneity and dynamism of the environment brings some constraints on capabilities of runtime environments that can host services. We need a runtime environment that has low resource requirements because smart spaces can have nodes with low resources. It is also mentioned about the dynamism of the services that they can join and leave the system. Thus, smart spaces need a runtime environment that provides plug-and-play support and an update mechanism which is needed for deploying newer versions of services.

Services in smart spaces can collaborate with each other to assist users. For example, a service that provides entertainment services like playing music can collaborate with a location service that keeps track of current location of a user. When a user changes its location from living room to kitchen, location service can inform entertainment service about this change. When the entertainment service receives this information, it can simply deactivate speakers in living room and activate the ones in kitchen. Thus, a runtime environment that provides mechanisms for service discovery and collaboration is required.

2.3 Questions to be answered

After introducing the problems with the existing infrastructure, managed entities and desired properties, details about the goal of this thesis are introduced. The goal of this thesis is to answer the following central question which is further decomposed into more detailed sub-questions:

How to provide a reliable infrastructure for services within smart spaces?

The above central question can be split into more detailed and specific questions that will be answered throughout the thesis.

- *How to deal with heterogeneity?* A problem about this is to have a runtime environment with low resource requirements. Another problem is how to treat all nodes in a fair way so that efficient resource consumption is possible. Making decisions about service management is another trouble. For example, selecting the node to place a service or selecting a service to migrate.

- *How to deal with distribution?* Decision making in a distributed system is a difficult process. A simple decision to make sure if a node is failed or not becomes difficult. It is not straightforward to say that a node is failed if a message is not received in t seconds. It can be received in the next $t + 1$, second.

- *How to deal with dynamism?* Adaptation to environmental changes is the main problem which includes many sub problems. Nodes and services joining and leaving the system is another one of the sub problems. Another one is the changing resource demands and changing node capacities.

Chapter 3

Related Work

A general view about why autonomous and distributed systems need each other is addressed. After considering the current state of the art solutions in similar and different domains, solutions about the problems and desired properties mentioned in Chapter 2 are addressed. As the environment is heterogeneous, the solution about representing this difference between nodes with *utility functions* is addressed. The two strongest failure detectors among the ones summarized in Table 2.1 are introduced. Different solutions about decision making mechanisms are introduced. Load balancing mechanisms in structured P2P networks are addressed both for homogeneous and heterogeneous systems. Mobility concepts and their realizations are also addressed in different domains. Applications of migration and replication for availability and load balancing are addressed. OSGi framework is introduced as a runtime environment.

This chapter starts with the introduction of why Grids and agents need each other in 3.1 and then continues with utility function in 3.2. Then, solutions on failure detection are introduced in 3.3. And then, the current decision making solutions for autonomy are presented in 3.4. It continues with the concept of load balancing in 3.5. Afterwards, mobility concepts are addressed in 3.6. Next, realized migration and replication techniques are addressed in 3.7 and 3.8 respectively. Finally, it concludes with the runtime environment including the OSGi framework in 3.9.

3.1 Grids and Agents

A good example of distributed systems is *Grids*. Grids are collections of computational nodes that provide infrastructures for resource sharing for coordinated problem solving. The research on Grids have always been on the area of providing interoperable infrastructure and tools for reliable resource sharing among geographically distributed communities. In other words, Grids define protocols and middleware for resource discovery and harnessing. Grids are scalable and robust but they lack intelligence [FJK04].

According to the definition of Pattie Maes, Autonomous Agents are compu-

tational systems that inhabit complex dynamic environments, sense and act autonomously in this environment, and by doing so realize a set of goals or tasks for which they are designed. Agents are problem solving entities with well-defined objectives. They are situated in complex environments where they can sense and act with their sensors and actuators. Agents are flexible and autonomous in terms of making decisions to complete their objectives. They can cooperate or compete with other agents in the same environment or another environment. However, agents are not designed to be scalable and robust [FJK04].

It is seen from their design perspectives that both agents and Grids have a common thread. This thread is the creation of communities for achieving common goals. However, research on these areas has been focused on different aspects [FJK04].

We cannot adapt nor pure Grid solutions neither pure agent solutions because we need the features of both sides [Her10]. Design goals of this thesis include autonomic system behaviour to reduce complexity which is the strongest part of agent systems. Providing service availability and efficient resource consumption are also among the goals and they require scalability and resource sharing which are the strongest parts of Grids. Thus, a system which is a combination of Grids and agent systems is a relevant approach [FJK04].

3.2 Utility Function

Devices in heterogeneous environments differ in terms of available resources such as processing power, RAM, storage, network bandwidth, connectivity, limited battery life and others. Representing this difference between devices is possible with a ranking function or *utility function* [JB05]. In a peer-to-peer (P2P) system where each node has equal responsibilities, peers may exchange periodically their utility values with their neighbours to form an overlay network. Overlay network is a logical topology where the nodes are connected with logical links rather than physical links. For example, a line overlay can be constructed by selecting two neighbours where one of them has the closest utility value to its own but less and the other one with the closest utility value to its own but larger. Selecting neighbours is done through a preference function which selects neighbours according to their utility values. However, the parameters constituting the utility function differs according to the application domain. In a P2P video streaming system, the utility can be thought as the network bandwidth and available storage of a peer or the up-time of a server where the goal is to discover the most stable peers [SDCM06, SD07].

The concept of a utility function can be adapted into the design of autonomous service placement protocol. Utility function can be used to represent nodes' available resources over time.

3.3 Availability

Availability is addressed among desired properties in Section 2.2.1 which requires failure detection mechanisms. Different kinds of failure detectors have been designed as summarized in Table 2.1. Two strongest failure detectors are explained briefly.

Perfect Failure Detector is based on the abstractions of synchronous time and crash-stop processes. It requires strong completeness and strong accuracy. Special messages called heartbeat messages are periodically sent to inform other processes that the sender is functioning properly. The crashes in this model are detected by setting a timeout for the heartbeat messages. Upon timeout, a process p_i detects a process p_j as crashed if there is no heartbeat message received from process p_j. The process p_j is removed from the known processes set because it crashed and p_i will never receive messages from it again. Perfect failure detectors have strong assumptions and require synchrony. Thus, they are not deployed in distributed systems where there is no synchrony.

On the other hand, *Eventually Perfect/Imperfect Failure Detector* is based on the abstractions of asynchronous time and crash-recovery processes. It requires strong completeness and eventually strong accuracy. Timeouts for heartbeat messages is used in this model, too. When the timeout occurs, a process p_i suspects process p_j if there is no heartbeat message received from process p_j. The process p_j is marked as *suspicious* rather than being detected as crashed because it might not necessarily be dead. Upon receiving a message from process p_j, the decision about it is revised and removed from the suspicious set and the timeout interval is increased. The timeout delay used by a process p_i to suspect p_j will eventually be large enough because p_i keeps increasing it whenever it makes a false suspicion. This is because of the assumption that there is a time after which the system becomes synchronous, partially synchronous system [Gue06].

3.4 Autonomy

Autonomy is addressed among desired properties in Section 2.2.1 which requires decision making mechanisms. Human being always observes its environment, processes the information gathered, schedules an action and reflects this action to its environment. For example, we observe the weather before going out. If the weather is rainy, we plan to take an umbrella. If the weather is sunny and hot, we plan to wear light clothes. Finally, we put our plans into action and take an umbrella or wear light clothes before going out. IBM proposed an autonomic manager model that has a monitor, analyze, plan and execute (MAPE) loop [KC03]. The proposed model has been adapted by [PNS+09] and the adapted model is depicted in Figure 3.1. In the model, *managed entity* can be either a physical device or a software module that is orchestrated. *Monitor* module observes raw data from sensors of the managed entity and provides the data to *analyze* module. Analyze module performs the analysis of the raw data and provides it to the knowledge agent. The

plan module receives the knowledge events from the knowledge agent and plans the required actions based on the desired functionalities and provides the results to the *execute* module. The execute module performs the planned actions on the managed entity via actuators.

The application of this simple yet effective model can be observed on all decision making mechanisms.

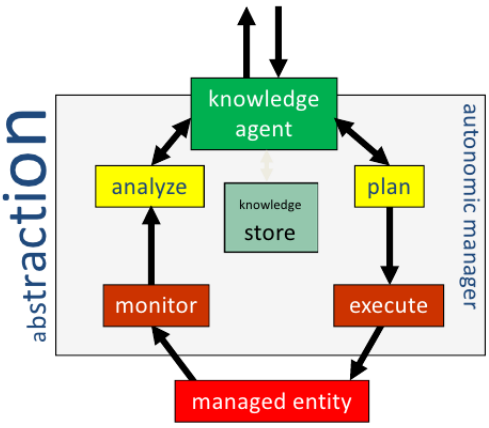


Figure 3.1. Monitor, Analyze, Plan, Execute [PNS+09]

Making decisions requires a base for the decisions and this base can change depending on the application domain. In a grid environment -a collection of computers connected to perform a common task- each node has to do some computations and decide on which node to migrate jobs in order to balance the overall load in the system. For this purpose, locally connected nodes make estimations about the CPU loads, service rates and job arrival rates of their neighbours [SVM07]. Nodes also take into account the job migration cost, resource heterogeneity and network heterogeneity while making decisions about migration. Based on the result of calculations of these parameters, they make decisions on which jobs to migrate to which neighbours. However, the aim in this method is to provide load balancing, not providing high degree of availability.

Mobile agents are defined as active objects that have behaviour, state and location. They are called autonomous because once they are invoked they will autonomously decide which locations to visit and what instructions to perform [BHR+02]. Ambient Intelligence is an information technology concept by which mobile users shall be seamlessly supported in their everyday activities. In such an environment, mobile agents can be deployed to assist users. When a user changes location in the environment, location agent can inform multimedia agent to activate music system in new location of the user. Another example application in such an environment is a search agent that can be used to search for a product in a shopping mall. In the shopping mall scenario, an agent may make decisions about replication

and migration for decreasing the response latency. It can make decisions according to locally collected information over some time interval such as incoming requests per second or number of hops -intermediary nodes/routers- a request has travelled [Her10]. If the number of requests it receives per time unit exceeds a threshold, it may decide to replicate itself in some other locations. If the number of hops a request has travelled exceeds a threshold, it may decide to migrate towards the direction it receives these requests. In this method, the aim is to increase quality of service by decreasing response time to user requests. It doesn't aim fault tolerance. This mechanism also requires a request routing system that routes the requests by determining the shortest path.

A market-like structure where mobile agents earn energy for giving service to users and expend energy for the resources they use is another example for decision making. When an agent runs out of energy, it dies of starvation and in case of abundance of energy, it reproduces or migrates to other runtime environments. The agents also migrate to other runtime environments with lower costs in order to save some energy [WS01, NS05]. This solution, too, doesn't aim for fault tolerance.

Another example of decision making mechanism is based on time [LAB+]. In this approach, a set of replicas called configuration are maintained for a specified time interval. Upon interval completion, the number of replicas may or may not change but the set of nodes hosting these replicas is changed. This solution aims for fault tolerance and load balancing. However, the heterogeneity of nodes is not taken into consideration.

3.5 Load Balancing

Load balancing is addressed among desired properties in Section 2.2.1. Many proposals have been made for load balancing in structured P2P systems and distributed systems. However, not all of these proposals address the heterogeneity and dynamism of systems. Chord [SMK+01] is a scalable P2P look-up service for Internet applications. It is a distributed hash table (DHT) where nodes in this system form an overlay network of a circle. Each node is responsible to store a certain interval of objects. Object to node assignments are done via a one way hash function in a static way. It is scalable in terms of joining and leaving nodes but it does not provide any kind of load balancing for the number of objects a node is responsible. This becomes an issue when some objects are more popular than others. In this case, nodes responsible for storing these popular objects handle most of the look-ups while the rest handles only a few queries. However, Waon [TOT11] solves this issue in structured P2P networks. Waon performs dynamic load balancing on the number of objects a node is responsible for. Unfortunately, it does not address the heterogeneity of nodes in a system.

In [RLS+03], three different methods for load balancing in structured P2P networks are introduced. Namely, *one-to-one*, *one-to-many* and *many-to-many* schemes. In one-to-one scheme, a lightly loaded node performs a DHT look-up

for a random object ID and picks up the node responsible for that object. If the node is overloaded, load transfer takes place.

In one-to-many scheme, excess loads of an overloaded node are transferred to many lightly loaded nodes. This is performed by maintaining directories about load information of light nodes in the system. These directories are stored in the system as normal objects in the DHT and some nodes are responsible for the operations on them. Lightly loaded nodes periodically advertise their loads and their capacities on these directories. And overloaded nodes periodically sample these directories. An overloaded node picks randomly one of these directories and sends the information about its capacity and the loads of its objects. The receiving node that maintains the directory chooses the best object to transfer on a light node.

In many-to-many scheme, the same directories in the one-to-many scheme are maintained with addition to advertising the heavy nodes also. These directories can be thought as a global pool where all overloaded nodes put their excess loads. The nodes responsible for maintaining these directories perform the matching of loads to be transferred to light nodes. This scheme is explained in more details in Section 4.5.1.

3.6 Mobility

Mobility is addressed among desired properties in Section 2.2.1 which includes capturing and re-establishing state of the software being mobilized. Weak migration is the choice for most of the mobile agent platforms such as JADE [BCPR03], Agentscape [age], Mole [BHR+02] and Aglets [Agl09]. All these platforms are written in Java (Agentscape mostly in Java) and are mainly using the Java Object Serialization API [Sun01]. It provides a stable object serialization mechanism. An object is a programming-language level software entity that encapsulates a collection of data, possibly references to other objects and a set of procedures that can be invoked to manipulate that data [Huc]. Object serialization means flattening an object in a way to be stored on permanent storage such as a file or transferred over network in order to be reused later by reconstructing the object [Gre].

Capturing the execution state of Java programs is not allowed by the Java security policy [LYBB12]. Thus, several techniques have been developed to capture the internal state of a Java application. These techniques can be categorized as Java Virtual Machine (JVM) manipulation, byte-code instrumentation, source code instrumentation and modification of the Java platform debugger architecture (JPDA). JVM manipulation means customizing the core JRE so that it provides the functionality of capturing the execution state. This method is efficient in terms of speed and overhead but it has the main drawback of not being portable which is the main goal of the Java platform [BHD03]. Byte-code instrumentation means manipulating the compiled source code by post-processing. A class file includes the byte-code instrumentations to be interpreted by the JVM [Dah99, Whi, BLC02, SSY00]. This method has the drawback of time and space overhead. Source code instru-

mentation means including special instructions in the source code to save snapshots/checkpoints of the state be pre-processing the source code before compilation [Fün98]. This method also has the same drawbacks as the byte-code instrumentation and also the disadvantage that the source code is not always available in case of using libraries. Performing some modifications on JPDA allows capturing the execution state as well. It is possible to access runtime information of an application in debug mode [AmSS+09]. Bytecode and source code instrumentation methods have been applied to preserve the portability of the Java programs by different techniques such as using the Java exception handling mechanism, debugging tools and functionality blueprint transformation [BOvSW02]. Migrating operating systems instances, virtual machines, across physical hosts is another use of strong migration. It provides a distinction between hardware and software, and facilitates fault management, load balancing and low-level system maintenance [CFH+05].

3.7 Migration

Migration is addressed among desired properties in Section 2.2.1. It is one of the techniques used for providing availability [Mar08]. However, it has its own challenges such as saving the state of the entity to be migrated and reestablishing at the destination. The reason for the decision to migrate an entity can be based on different facts. If it is possible to know apriori that a node will be unavailable, the services running on that node can be pre-emptively moved to other system nodes prior to node death. Apriori knowledge of failure can be battery exhaustion of a mobile device or system maintenance by administrators. In such cases, entities running on that node can be forced to migrate to another one [Huc].

The nodes don't necessarily have to die or crash to perform migration. Nodes in the systems have limited resources and when a node starts to run out of available resources, some of the services running on that node can be migrated to another one with more resources. This can serve for the purpose of load balancing among nodes [SVM07]. A new approach to migration has been introduced by [LAB+]. This approach enables migrations of services that replace non-failed nodes. It provides both load balancing and autonomy but does not take into consideration the heterogeneity of nodes.

3.8 Replication

Replication is addressed among desired properties in Section 2.2.1. It is another widely deployed approach for providing availability. It requires extra communication overhead for state consistency among replicas. It has two different techniques as described previously in Section 2.2.6. Active replication has been the choice for fault tolerance despite its high complexity for consistency [Mar08, FD02]. When active replication method is used, there is need for a total order/atomic broadcast mechanism that the messages are delivered in the same order to all of the replicas.

One way of achieving total order broadcast is by adapting a consensus mechanism which solves the problem of total order broadcast [Gue06]. Paxos, [Lam01], is the most widely known consensus algorithm proposed by Leslie Lamport and it is applied in [LAB+] for deciding on the execution order of the requests. In case of failure of primary replica in primary-backup replication, there is need for a primary election system [GS96]. Both techniques have their advantages and disadvantages over each other. Failures in active replication are transparent to the user while users may have to reissue requests and can experience some delay in primary replication. Primary replication does not require a total order broadcast or consensus mechanism for consistency. Finally, active replication uses more resources because all replicas are actively processing requests.

3.9 Runtime Environment (RTE)

Runtime environment is addressed among desired properties in Section 2.2.1. As soon as a software program is executed, it is in a running state. Within this state, the program can send instructions to the computer's processor, access the RAM, storage, network resources and other system resources [tec]. Runtime environments are components designed to support the execution of programs by providing them resources. The most commonly known runtime environment is the Java runtime environment (JRE). JRE provides an abstraction layer over the operating system that allows a java application or applet to be executed on any other computer with JRE installed [LYBB12]. OSGi Alliance (former Open Services Gateway initiative) is a framework that runs on top of JRE [OSG11]. It provides a general-purpose, secure and managed Java framework that supports the deployment of extensible and downloadable applications known as *bundles*. The main purpose of OSGi is to provide a modular approach to software development. Software is developed by creating standalone operational pieces called modules. Each module has a specific operational purpose and can be later used by other modules. In this approach, software development can be thought as putting together the pieces of a puzzle. In the context of OSGi, these modules are called bundles.

OSGi framework provides a very dynamic environment where bundles join and leave the system at runtime without restarting the framework. Bundles can be updated with newer versions or new versions can be installed while preserving the previous versions of bundles. Bundles are called dependant when one bundle requires another one to run. It can be thought as a relation similar to producer-consumer. Producer is the one that provides the service and the consumer is the one that depends on the producer. Bundles include all the required resources for their operation in a Java archive (JAR) file. All these requirements are defined in a file called *MANIFEST.MF* file located in *META-INF* directory of each bundle. The manifest file declares the other bundles together with their versions that the operation of the bundle depends. The framework ensures that all these dependencies are satisfied before starting a bundle. Manifest file also includes the list of exported packages of

a bundle that other bundles can consume.

Some of the most widely used bundle manifest headers are as follows:

- Bundle-Activator: specifies the name of the class used to start and stop the bundle.
- Bundle-ClassPath: defines a comma-separated list of JAR file path names or directories containing classes and resources.
- Bundle-Name: defines a human-readable name for the bundle.
- Bundle-SymbolicName: specifies a non-localizable name for this bundle. The bundle symbolic name together with a version identifies a unique bundle.
- Bundle-Version: specifies the version of this bundle.
- Export-Package: a comma-separated list of packages exported by this bundle for use of other bundles.
- Import-Package: a comma-separated list of packages imported by this bundle that are exported by other bundles.
- Require-Bundle: specifies that all exported packages from another bundle must be imported, namely the public interface of another bundle.

A bundle may be in one of the following states:

- INSTALLED - The bundle has been successfully installed.
- RESOLVED - All Java classes that the bundle needs are available. The bundle in this state is either ready to start or has stopped.
- STARTING - The bundle is being started.
- ACTIVE - The bundle has been successfully activated and is running.
- STOPPING - The bundle is being stopped.
- UNINSTALLED - The bundle has been uninstalled from the framework and it cannot move into another state.

Chapter 4

Design

Putting the problem and the current state of the art solutions in similar and different domains together, a centralized system is introduced. It is centralized in the sense that all information is gathered in a single service in the system. However, it does not exhibit the behaviour of a single point of failure because it is decentralized in the sense that it manages itself by migration and replication and treats itself as any ordinary service. The system is called SMYRNA and it manages all the services running in the system. SMYRNA itself is a service that is deployed in the runtime environment which is the OSGi framework. OSGi is adapted as the runtime environment because of its modular service oriented and dynamic nature and its low resource requirements. The nature of OSGi provides great ease in replication and migration because the framework enables deployment of services dynamically. The environment is distributed and asynchronous. Thus, eventually perfect failure detector algorithm is adapted with a slight modification to detect node failures. The modification is required because of the burden that will be caused to the system in case of false suspicion of a node failure. Finally, list of functionalities to be adapted by services are listed in order to be manageable by SMYRNA. These functionalities include operations such as starting, stopping, resuming and suspending services. Also some operations for collecting statistical information are listed.

Designed solution for the problem is introduced in an abstract way in 4.1. Then, we continue with utility functions in 4.2. Afterwards, service functionalities that are needed to manage services are introduced in 4.3. After that, failure detection mechanism for total failures together with its adaptation and worst case complexity scenario are introduced in 4.4. Then, management mechanism, SMYRNA, is introduced in details about decision making strategies in 4.5. Finally, this chapter concludes by addressing the functionalities of runtime environment in 4.6.

4.1 Introduction

We want to achieve the goals and answer the questions listed previously in Chapter 1 and Section 2.3. The path to conclusion goes through adapting the current state

of the art solutions for similar problems in different domains as listed in Related Work 3. Taking into account the goals to be achieved, challenges and current state of the art solutions, a system to meet the needs is designed.

In general, the system is responsible for placing and managing services in smart environments. The abstract view of the system is given in Figure 4.1. We have a virtual computing node which is composed of several physical computing nodes and we have services available for users. The idea is to give the view of a single computing node while in fact there are several nodes constituting a distributed system. Users can interact through a user interface which can be a control panel in home, an application store where users can buy new services, or any other interface such as a web page. A user can install a new service, start or stop it, uninstall an already installed service, give feedback about services or can interact with any service. When a new service is installed, the system selects an appropriate computing node according to the desired functionality parameters to place this service. Creates replicas of this service depending on the provided parameters and places them in different computing nodes. It also manages these services at runtime by migrating them to different computing nodes with more available resources. It allows the system to adapt to environmental changes without any external intervention. Placement actions are denoted by arrows in the figure. Solid arrows represent the placement of primary/active services while dotted arrows represent the placement of replicas. As it is seen from the figure, the number of service replicas may differ. This number depends on the parameters provided by the user.

Availability is one of the parameters and represents the desired level of availability of a service (e.g. 0.99999). *Latency* is another one which denotes how fast a service communicates (e.g. MAX 100 milliseconds). *Priority* can be another parameter indicating how critical a service is. Priority can be a numeric value on a predefined interval such as [1-10] where higher values represent higher priority as it is the situation with processes and threads in operating systems currently. This is an important parameter because critical services, such as building security, can be deployed in smart spaces. Another factor can be that many other services depend on this service and it is desired to have a high priority.

More detailed view of the system is given in Figure 4.2. On the left side of the figure there is a special case which is explained in details later in Section 4.5. At the bottom lays a physical computing node. A computing node is composed of an operating system (OS), Java Virtual Machine (JVM) on top of OS and the OSGi framework on the top most. As it is explained earlier in Section 3.9, applications deployed in OSGi are called bundles. On top of the computing node there is a special bundle called RTE. On the top most service bundles are located that can be used by users. The links represent the connectivity among entities.

Beside these entities, there is a special entity denoted with a pentagon named K. This entity is called the *knowledge store* which is already provided by the autonomous control and management platform, DS2OS, developed in Technical University of Munich. Knowledge store provides a high level abstraction of devices in smart environments. It is responsible for storing the data of these devices and

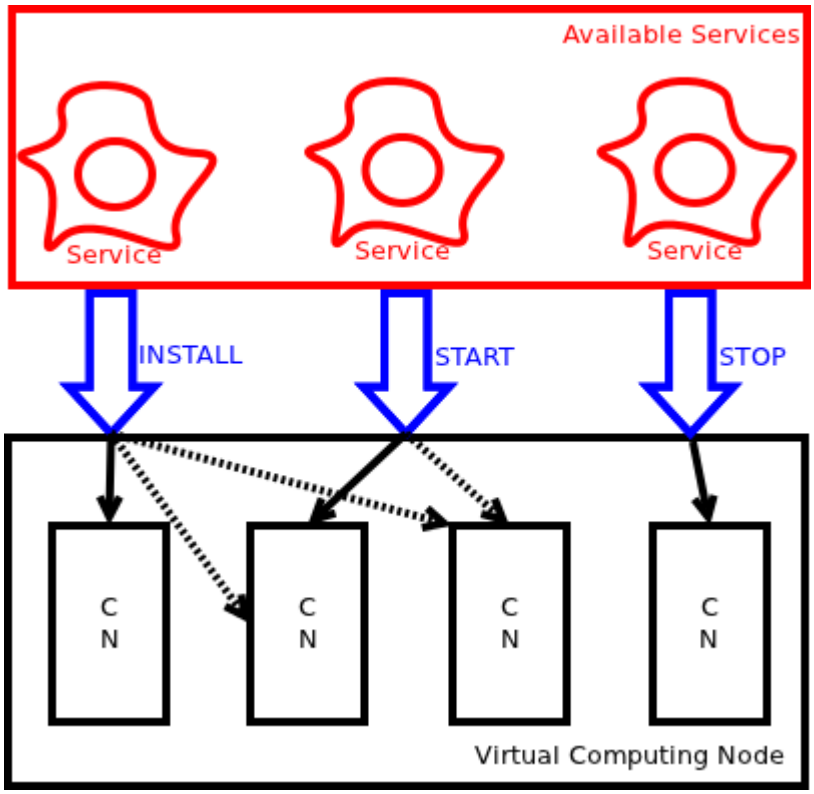


Figure 4.1. Abstract View of the System

services.

4.2 SMYRNA Utility Functions

Taking into consideration the problem about heterogeneity mentioned in Section 2.1.1 and the current state of the art solutions for dealing with this problem as introduced in Section 3.2, specific utility functions for the purpose of this thesis are designed. As it was explained previously, utility functions are based on some parameters. We decided to select *CPU*, *memory* and *network bandwidth* values of a node as the most important parameters to constitute our utility functions. There are 2 different utility functions, namely utility of a node and utility of a service on a node.

4.2.1 Utility of a Node

Smyrna has a novel migration strategy which is explained in details in Section 4.5.3. This strategy aims at migrating one service at a time from the worst RTE to the best

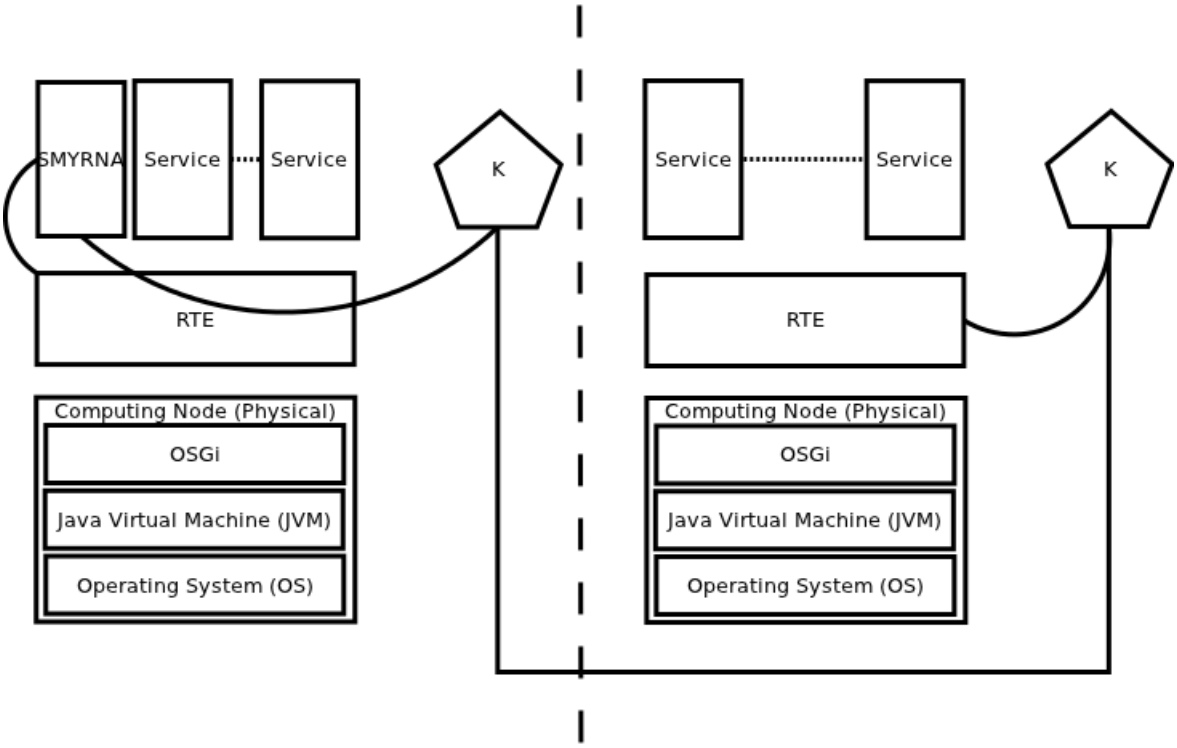


Figure 4.2. Detailed View of the System

fitting RTE for the selected service. Utility of an RTE is calculated to determine the worst RTE that a service is going to be migrated from. The strategy of selecting the service to be migrated is given in Section 4.5.3. The utility function of a node is formulated as

$$U_M = \frac{m + \frac{\sum_{j=1}^{n} H_M}{n}}{2} \tag{4.1}$$

where m is the slope of the fitted line of utility histories and $H_M = (U_1, U_2, ..., U_\eta)$ is the utility history of node M. Each utility of a node M is calculated as

$$U_i = \frac{\sum_{k=1}^{n} P_k}{n} \tag{4.2}$$

where $P = (C, M, N)$ is the set of parameters that constitute our utility functions. C is free CPU, M is free memory and N is free network available. C is calculated as

$$C = CPU_{idle} * CPU_{total} \tag{4.3}$$

where CPU_{idle} is the CPU idle percentage and CPU_{total} is the total CPU capacity.

$$M = M_{free} \tag{4.4}$$

where M_{free} is the current free system memory. N is calculated as

$$N = N_{free} * N_{max} \tag{4.5}$$

where N_{free} is the currently free percentage of network and N_{max} is the maximum amount of traffic that can be handled.

Designed utility function has a parameter, η, that denotes the history size. In this design, it is decided to keep the history of the last $\eta = 10$ utilities. Worst runtime environment is selected by investigating nodes' utility histories. While comparing runtime environments, an ultimate utility of each runtime environment is calculated as given in 4.1. This value is calculated by fitting the utility history values in a line by the least squares fitting method. Then, the slope of the line is taken into consideration to conclude if it shows increasing or decreasing characteristic. The slope of the fitted line constitutes half of the ultimate utility. Calculating only the slope of the fitted line is not enough to make a decision that the runtime environment is a good one or not. There can be a case that a runtime environment has a utility history with alternating values that are all close to each other. In this case, the slope of the fitted line is close to 0 even though it has many free resources. Thus, average of the utility history is also taken into account while calculating the ultimate utility. Average utility is normalized according to maximum average utility. The slope of the fitted line and the average utility values constitute the ultimate utility value.
Utility of a node is used in order to determine the worst RTE in the system so that a service can be migrated from it to free some resources.

4.2.2 Utility of a Service on a Node

Smyrna's novel migration strategy aims at relaxing worst RTEs by migrating services to other best fitting RTEs by migrating one service at a time. The worst RTE is selected by calculating the utility of a node as given in the previous subsection 4.2.1. Best fitting RTE for a given service is determined by calculating the utility of a service on each RTE. RTE which results in maximum utility is selected as the best fitting one for the given service. The utility of a service s on a node M is calculated as

$$U_{S,M} = \sum_{i=1}^{n} W_{i,S} * P_{i,M} \tag{4.6}$$

where $W = (W_c, W_m, W_n)$ is the set of weights that service S provided about CPU, memory and network respectively. $P = (C, M, N)$ is the set of free resources of node M as explained in subsection 4.2.1 and are calculated as in equations 4.3, 4.4 and 4.5 respectively.

Calculating utilities of a service on all nodes in the system, best fitting RTE can be determined for a service to be placed. The weight vector provided by a service plays a major role in here. For example, a service providing the weight vector as $W = (80, 10, 10)$ means that this service is putting a lot of importance on CPU while memory and network are not that important for it. In this case, best fitting RTE for this service will be the one with the most free CPU in the system.

Smyrna utility functions are metrics for measuring available resources for consumption and service fitting values on nodes. In Section 2.3, some questions to be answered throughout this thesis are addressed. The utility functions are partial answer for the first subquestion about dealing with heterogeneity. Smyrna utility functions help to have a metric to treat all nodes in the system in a fair way.

4.3 Service Functionalities

We want to manage the services in our smart space by placing, migrating and replicating them. To achieve this, interaction with these services is required. This is possible via so called Application Programming Interface (API). Thus, certain operations that can be invoked on these services are defined. The services are called bundles. Bundles are applications that can be deployed in OSGi framework which is the adapted runtime environment as explained in details later in Section 4.6.

- *start*: the initiating operation that does the necessary initializing operations such as registering a service as a remote service. This operation calls the *run* method of a service after finishing the initialization. The implementation of this operation is provided and is not left for service developer. All a service developer needs to do is to call this operation after initializing its service.

- *stop*: the terminating operation that stops the service. This operation gives the possibility to finalize the service functionality and do the clean up such as closing connections and freeing resources.

- *suspend*: active replicas need to be migrated or replicated when it is necessary. This operation is used to signal an active replica to suspend its operation and save its state in the knowledge store. It can be called before migration or replication to perform necessary state saving operations.

- *resume*: this operation is used to signal a replica that it will be the primary replica which is going to handle the requests from now on.

- *run*: this operation is the main method of a service. This method does the actual work that a service is supposed to do. Thus, the implementation of the algorithm must be done within this method.

- *getBundleName*: we need to know the name of the bundle that each service is running. This operation will return the name of the bundle in the form

of [BundleSymbolicName_Version] (e.g. de.edu.tum.rte_1.0.0). BundleSymbolicName_Version constitutes a unique identifier in an OSGi framework.

- *getLastModified*: we need the information when a service was last modified. Last modified information denotes when a service was installed or updated. This information is necessary for making the decision about which service to migrate.
- *isActive*: we have primary replicas that are actively handling user requests and we also have their replicas. This state information states if this service is the primary service or it is a replica.
- *isAlive:* as we explain in Section 4.6, runtime environments are responsible for collecting information about services. This operation is invoked periodically by RTEs to monitor a service if it failed or not. If a service does not respond, a crash report is prepared for this service and a recovery protocol is applied.
- *getStateAddress:* we need to know about the address in the knowledge store of a service state. This address is necessary for state serialization in case of migration or replication. The serialized state is deserialized at the destination knowledge store. Thus, we transfer the data state of a service.

4.4 Failure Detection

Based on the analysis of the problem that a failure detection mechanism is required and the current state of the art solutions for this purpose, we decided to deploy eventually perfect failure detection mechanism. The main reason of this choice is that we have an asynchronous system. The system does not have a known time bound in which every operation is completed. It means that a message sent from node A to node B may take an arbitrary amount of time. It cannot be said that node A is crashed because node B didn't receive a heartbeat message within t milliseconds. There might be a congestion on the network route that the message travels which causes a delayed delivery.

The original algorithm for this type of failure detector is given in Algorithm 1. However, it cannot be adapt exactly the same mechanism and some modifications are required. The reason why it cannot be adapted entirely is the possibility of making false suspicion about a node's failure. False suspicion brings burden to the system because of the replication strategy. We need to keep a number of replicas of each service depending on the parameters described previously in the introduction of this chapter. When a failure is detected, new replicas are created and placed appropriately in the system. If there is a false suspicion, additional replicas of the services residing in the suspected node will be created. This causes more resource consumption, inconsistency and redundancy in the system.

A failure detector for another purpose is also required. Namely, to detect failure of the primary Smyrna instance. This failure detector is used by backup replicas of

Smyrna. Primary Smyrna instance periodically sends ping messages to its replicas. Replicas receiving the pings update their information of the primary instance as alive. All replicas use a modified version of the eventually perfect failure detector. The modification on the algorithm is at the same point as the one used by Smyrna where a node is suspected. The modified eventually perfect failure detector algorithm is given in Algorithm 2.

4.4.1 Eventually Perfect Failure Detector

It is based on maintaining a time delay and two sets of neighbours, namely *alive* and *suspected* sets. Initially, alive set is assigned all the known neighbours (Π), suspected set is empty ($\emptyset$) and the period is set to $TimeDelay$ value. Upon occurrence of heartbeat deliver event, the sender of the message is added to the alive set. Upon timeout for heartbeat event, a heartbeat message is sent to each neighbour and the heartbeat timer is reset. Upon timeout for check event, the alive set and suspected set are checked if they have a common entry. If so, it means that the algorithm has made a false suspicion and it increases the timer period by Δ. Then, each neighbour $p_i \subset \Pi$ is checked for two conditions. First, p_i is checked if it does not belong to both alive and suspected sets. If this condition is satisfied, p_i is added to suspected set and a suspect event is triggered. If not, the second condition is checked if p_i is in both sets. If it is, it means that the algorithm made a false suspicion and removes p_i from the suspected set and triggers a restore event. Finally, when all neighbours are checked, the alive set is set to be empty and the check timer is reset.

4.4.2 Modified Eventually Perfect Failure Detector

As it is described in the previous subsection, a modification the original algorithm is needed. The system is designed to operate on the overlay network provided by DS2OS. Thus, all communication is done using this overlay. New design of the failure detector works as follows: instead of sending ping messages to all nodes in the system, all nodes send periodic ping messages to primary Smyrna instance. Smyrna receives all ping messages and performs the steps as defined in the original algorithm (Algorithm 1) except for the suspicion event. When a node is suspected by Smyrna, it tries to establish a direct connection to this node using the overlay network and checks if it is alive or dear. If it does not respond positively, then this node is detected as dead and recovery protocol for the services deployed on that node is started. If the node is alive, then Smyrna evaluates this as a false suspicion and increases the timer period by Δ.

The primary Smyrna failure detector used by backup replicas is given in Algorithm 2. It works as follows: replicas maintain two flags indicating if the primary is alive or suspected. Upon timeout of check timer, replicas check if the primary is set both as alive and suspected. If this is the case, it means that they have made a false suspicion and they increase the check timer by Δ. Then they check if primary is not set nor as alive, neither as suspected. If it is the case, suspected flag is set to *true*

loa 1. Eventually Perfect Failure Detector [Gue06]

1: **upon event** $< Init >$ **do**
2: alive $\leftarrow \Pi$
3: suspected $\leftarrow \emptyset$
4: period $\leftarrow TimeDelay$
5: $startTimer(TimeDelay, HEARTBEAT)$
6: $startTimer(period, CHECK)$

7: **upon event** $< Timeout \mid HEARTBEAT >$ **do**
8: **for all** $p_i \subset \Pi$ **do**
9: **trigger** $< send \mid p_i, [HEARTBEAT] >$
10: **end for**
11: $startTimer(TimeDelay, HEARTBEAT)$

12: **upon event** $< Timeout \mid CHECK >$ **do**
13: **if** $(alive \cap suspected) \neq \emptyset$ **then**
14: $period \leftarrow period + \Delta$
15: **end if**
16: **for all** $p_i \subset \Pi$ **do**
17: **if not**$(p_i \subset alive) \wedge$ **not**$(p_i \subset suspected)$ **then**
18: $suspected \leftarrow suspected \cup \{p_i\}$
19: **trigger** $< suspect \mid p_i >$
20: **else** $\{(p_i \subset alive) \wedge (p_i \subset suspected)\}$
21: $suspected \leftarrow suspected \setminus \{p_i\}$
22: **trigger** $< restore \mid p_i >$
23: **end if**
24: **end for**
25: $alive \leftarrow \emptyset$
26: $startTimer(period, CHECK)$

27: **upon event** $< Deliver \mid src, [HEARTBEAT] >$ **do**
28: $alive \leftarrow alive \cup \{src\}$

and replicas try to establish a direct connection to the primary using the overlay network and check if it is alive or not. If it is not alive, then they start an election round among all replicas. The election process works as follows: each replica queries all replicas using the overlay if it can be the primary by sending its ID. When the query is received by any replica, the result is returned as a comparison of IDs. If the received ID is less than its own ID, then the requester can be new primary. The replica who collects all votes which is also the one with the lowest ID, becomes the new primary Smyrna instance.

Adapted failure detector helps to answer some of the subquestions listed in Section 2.3. This solution is a partial answer to the question of dealing with distribution and dynamism.

4.4.3 Worst Case Scenario

The worst case scenario for detecting a failure is given in Figure 4.3. Only heartbeat messages sent by Node 1 are shown to demonstrate clearly. Node 1 sends heartbeat messages to all its neighbours at time $t1$ and crashes right after that. All other nodes receive the heartbeat messages before their check periods at time $t2$ and they don't suspect Node 1. At time $t3$, when nodes check for received heartbeat messages, they realize that Node 1 is crashed. Then, they start a consensus among each other and exchange messages which takes c seconds as shown in the figure. The number of messages exchanged can be calculated in terms of nodes in a system. Let N be the number of nodes in a system. When a node fails, $N-1$ nodes start consensus and each of them sends $N-2$ messages and $N-2$ responses are generated to these messages. In total, $N-1*2(N-2) = 2N^2-6N+4$ messages are exchanged in a consensus round. This means that the worst case number of messages for consensus can be presented as $O(N^2)$. If period for checking alive neighbours is t seconds and a consensus round takes c seconds, then the worst case for detecting a crash would be $2t+c$ seconds.

4.5 SMYRNA

The special case on the left side of Figure 4.2 is the existence of a very special and critical bundle called SMYRNA. What makes this bundle so special is that it is the decision making mechanism. Previously we mentioned about the MAPE loop in Section 3.4. RTE bundles have informational functionalities that are described in Section 4.6. These functionalities form the monitoring phase of the MAPE loop. Collected information is stored in knowledge stores by RTEs. This information is then retrieved and analyzed by SMYRNA. The analyzed information results in knowledge which is used in the planning phase. Based on the knowledge gained from the analysis phase, SMYRNA can conclude into some decisions that involve services and runtime environments, planning phase. Putting these decisions into actions makes the execution phase. The logical view of the architecture is given in Figure 4.4. The system can be thought as a ring structure where all entities

loa 2. Modified Eventually Perfect Failure Detector

```
1: upon event < Init > do
2: alive ← false
3: suspected ← false
4: period ← TimeDelay
5: startTimer(period, CHECK)

6: upon event < Timeout | CHECK > do
7: if (alive and suspected) then
8:    period ← period + Δ
9: end if
10: if (!alive and !suspected) then
11:    suspected ← true
12:    alive ← primary.isAlive()
13:    if !alive then
14:       trigger < election >
15:    end if
16: else if (alive and suspected) then
17:    suspected ← false
18: end if
19: alive ← false
20: startTimer(period, CHECK)

21: upon event < deliver | [HEARTBEAT] >
22: alive ← true

23: upon event < election >
24: agreed ← 0
25: for all r_i ⊆ R do
26:    iAmPrimary ← r_i.amIPrimary(myId)
27:    if iAmPrimary then
28:       agreed ← agreed + 1
29:    end if
30: end for
31: if agreed = R.length then
32:    trigger < newLeader >
33: end if

34: upon event < amIPrimary | ID >
35: return  myId > ID
```

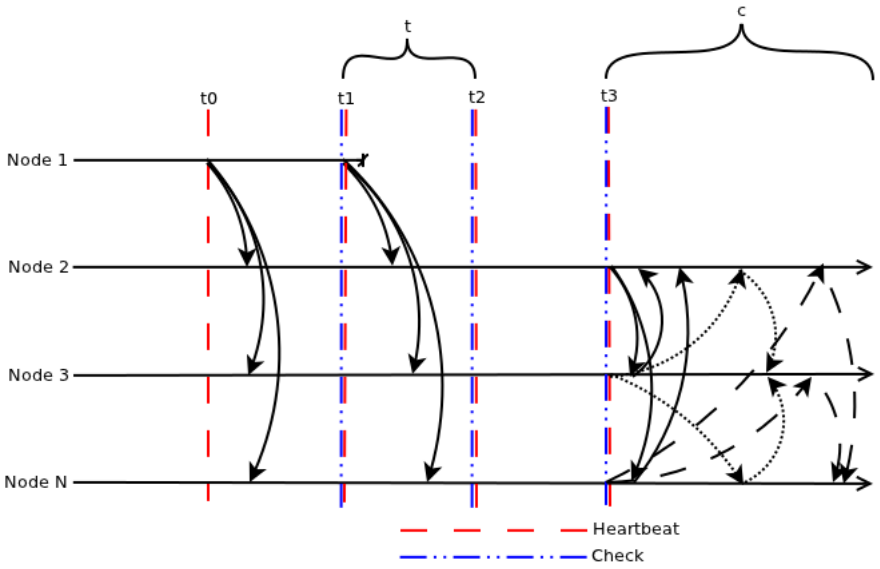


Figure 4.3. Worst Case Scenario

are connected over a common bus. Each RTE entity has the functionalities as are explained in Section 4.6. And of course the OSGi framework underneath each entity.

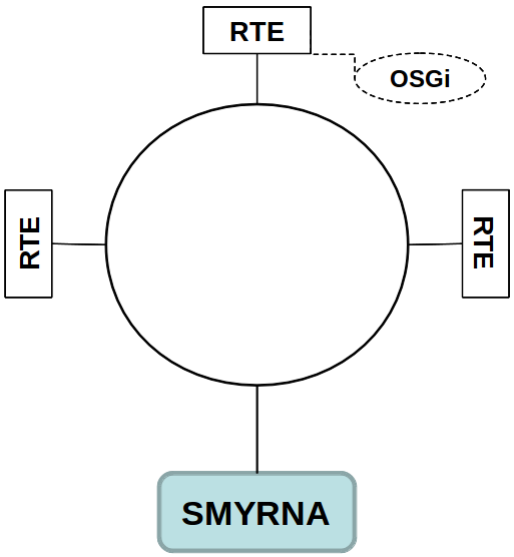


Figure 4.4. SMYRNA

The control of all services in the system is performed by SMYRNA. It has

to decide where to place a service if a user wants to install a new service. Not only where to place it but also it needs to decide how many replicas to maintain depending on the parameters provided by users. After calculating required number of replicas, then it has to decide where to place these replicas, too. Placing new services and maintaining their replicas was the first problem. The second problem raises during the operation of these services. We have a dynamic environment and SMYRNA has to adapt to the environmental changes.

Failure of a computing node is one of the factors affecting decision making mechanism. SMYRNA has to make sure that a node is for sure failed if it doesn't receive a heartbeat message. As described in Section 2.2.1, it might be the situation that connection between SMYRNA node and the other node is slow and heartbeat message is delayed. If it makes a false decision on the failure of a node, it has to manage all services that were running on that node. This false decision brings burden to the system by duplicating services on failed node and placing them on other nodes. Thus, detecting a failure is critical and it has to reach a consensus with other nodes in the system that the suspected node is for sure dead.

Beside failure of a node, changes on available resources is another important factor that affects the decision making mechanism. A user may start using one of the nodes in the system for any purpose which causes resource consumption. In this case, SMYRNA has to adapt the system so that services run on nodes with sufficient resources. This adaptation provides optimization and load balancing in the system. The decision to make in this situation is first to determine which service to migrate. Second, where to migrate this service. While deciding on where to migrate, it must take into account that service downtime must be minimum and also a service must never be in the same location with one of its replicas.

The proposed solutions about placement, migration and replication answer parts of all the questions listed in Section 2.3. Placement strategy answers the questions about where to place, migrate and replicate a service. Migration strategy answers the questions about making decisions on when and which service to migrate listed related to the questions about dealing with heterogeneity. Replication strategy answers the questions about dynamism in terms of recovery protocols to be applied in case of node or service failures.

Detailed explanations about these decisions are given in the following subsections.

4.5.1 Load Balancing

The environment is dynamic in terms of nodes, services and available resources for consumption. The necessity of load balancing is addressed in Section 2.2.3 and current state of the art solutions in similar domains in Section 3.5. *Many-to-many* scheme developed by [RLS+03] is adapted. The algorithm is composed of *unload*, *insert* and *dislodge* phases.

In the *unload* phase, an overloaded node transfers its lightest services into the global *pool* until it becomes underloaded. At the end of this phase, all nodes are

underloaded but the services in the pool are waiting to be assigned.

Insert phase aims to transfer all services from the pool to underloaded nodes without creating any new overloaded nodes. All services in the pool are tried to be transferred to an underloaded node starting from the heaviest service. A destination node d is selected from the underloaded nodes such that deploying service s will maximize the utilization of d. This is repeated for all services until all services are transferred in the pool or no more services can be transferred. If a service cannot be assigned, algorithm continues with dislodge phase. If all services are transferred, the algorithm halts. Otherwise, remaining services in the pool are assigned back to their original nodes.

Dislodge phase aims at swapping the heaviest service s from the pool with the lightest service s' from an underloaded node l. If no such service can be identified, the algorithm halts. Otherwise, swap operation is performed and the algorithm continues from the *insert* phase.

Smyrna is the decision making unit in the system where all nodes periodically report their capacities, current loads and services deployed to them. Thus, Smyrna is the directory where all information about overloaded and underloaded nodes is maintained in the system. Many-to-many scheme is applied only in the presence of both overloaded and underloaded nodes. It is not applied among nodes which are all overloaded or underloaded. If there are no overloaded nodes, the migration strategy explained in Section 4.5.3 is applied. The algorithm of many-to-many scheme is given in Algorithm 3.

loa 3. Many-to-many

1: N *is set of all nodes*
2: $underloaded \leftarrow \emptyset$
3: $pool \leftarrow \emptyset$
4: $unload()$ {all nodes are underloaded after this phase}
5: $insert(pool)$ {all possible services are transferred to underloaded nodes}
6: $redeployServices()$ {remaining unassigned services are deployed back to their nodes}

4.5.2 Placement Strategy

This decision is the most crucial decision that SMYRNA makes. Three different situations result in this decision. First, if a new service is being installed to the system, the service must be placed in an appropriate node. Second, when a decision about migrating a service is done and the service must be migrated to a node with sufficient resources. Third and last situation is when a decision to replicate a service is done and the replica must be placed in an appropriate node. The destination runtime environment is selected as the best node based on the utility histories of nodes in the system. The constraints of not placing an active service and its replica

loa 4. unload

```
1: underloaded ← ∅
2: pool ← ∅
3: for all n_i ⊆ N do
4:   if n_i is overloaded then
5:     while n_i is overloaded do
6:       lightest ← n_i.removeLightest()
7:       pool ← pool ∪ (n_i, lightest)
8:     end while
9:   else
10:    underloaded ← underloaded ∪ n_i
11:  end if
12: end for
```

loa 5. insert(pool)

```
pool.sort() {sort all services in ascending order according to their loads}
for i = pool.size() → 0 do
  i ← i − 1
  (n_i, service) ← pool(i)
  best = getBestFittingNode(service, underloaded)
  if best ≠ null then
    best.deploy(service)
    pool ← pool ∖ (n_i, service)
  else
    dislodge(i)
  end if
end for
```

and two replicas in the same node are also taken into consideration during this decision.

We have services with different resource demands and different weights provided for these resources. Thus, we need to calculate utility of a service for each RTE in order to determine the best fitting RTE. The selection of target RTE to place a service is done by choosing the best RTE with maximum utility value as calculated by equation 4.6 for the service.

There is a parameter and constraint that must be satisfied while choosing the best RTE in the system. This parameter is called $MinHistoryValues$ which denotes the number of utility values that must be available before making a decision. There is also another parameter which states the utility history length and is denoted as η. In our design, we decided to choose $\eta = 10$ and $MinHistoryValues = 2$. So, to make a decision about the best RTE, we need to have at least 2 values of the utility history to be filled. The reason for choosing $MinHistoryValues = 2$ is that

loa 6. getBestFittingNode(service, underloaded)

```
1: min ← MAX
2: best = null
3: for all n_i ⊆ underloaded do
4:    if n_i is not overloaded with service then
5:       remaining ← get free resources if service deployed on n_i
6:       if remaining < min then
7:          min ← remaining
8:          best ← n_i
9:       end if
10:   end if
11: end for
12: return best
```

we need at least 2 values to fit a line in order to characterize the state of a node. $MinHistoryValues$ is required to make the first decision about RTEs' states. This value can be set to a greater value, not greater than η, to make a more healthy first decision. η can also be set to a greater value in order to have more data about the past of RTEs.

4.5.3 Migration Strategy

The algorithm for making decision about migration is given in Algorithm 9. The idea is to migrate the oldest and most memory consuming service from the worst RTE to the best RTE by satisfying constraints about not placing replicas in same location. The algorithm works as follows: the worst RTE is selected according to utility histories. If no RTE satisfies the constraint of $MinHistoryValues$, algorithm halts. Then, a service from the worst RTE is selected with a priority to be an active service by excluding services in the candidate history. If no active service is selected, then a replica is selected. If no service is selected, algorithm halts and returns $NoOp$ which means to take no action. The selected service is added to the candidate history. Afterwards, locations of this service are discovered. This is done by searching locations of this service's active and replica locations. The reason for discovering service locations is to satisfy the constraint about not placing replicas in same location. Then, the best RTE is selected by providing the service locations to be excluded from the search. If no RTE is selected, algorithm returns to the point on selecting a new service. Otherwise, selected RTE is checked if it becomes overloaded if the candidate service is deployed on it. We do not want to create new overloaded RTEs because it increases the number of unsatisfied services. Afterwards a comparison is done with the worst RTE. If the best RTE satisfies the $MigrationThreshold$, then the selected service is migrated from the selected worst RTE to the best one. $MigrationThreshold = 0.90$ is selected as a design parameter. This threshold means that it is necessary for a best RTE to be at least

loa 7. dislodge(index)

```
1: modified ← false
2: heaviest ← pool.get(index).service
3: host ← pool.get(index).node
4: lightest ← null
5: node ← null
6: min ← 99999
7: for all n_i ⊆ underloaded do
8:    for all service ⊆ n_i.services do
9:       if host.services.contains(service) then
10:         continue
11:      end if
12:      if heaviest can be swapped with service without making n_i overloaded
         then
13:         load ← service.getLoad()
14:         if load < min then
15:            min ← load
16:            lightest ← service
17:            node ← n_i
18:            modified ← true
19:         end if
20:      end if
21:   end for
22: end for
23: if modified then
24:   node.swap(heaviest, lightest)
25:   pool.get(index).service ← lightest
26: end if
27: return modified
```

10% better than the selected worst RTE. If the threshold is not satisfied, algorithm returns to the point on selecting a new service. By keeping a candidate history, algorithm tries all services in the worst RTE to migrate. The algorithm eventually halts when all services are tried and no action is taken.

When to migrate a service?

Adapting the system to environmental changes is one of the tasks that SMYRNA must perform. Thus, deciding on when to migrate a service is a critical process. This decision is done in the planning phase of the MAPE loop after analyzing the collected information of the system. As it is described above in Section 4.5.3, there are some constraints to be satisfied to migrate a service. First of all, the worst RTE must be selected. And then, an active service is tried to be selected. If there is no suitable

loa 8. redeployservices

```
1: redeployServices()
2: while pool ≠ ∅ do
3:    {n_i, service} ← pool.remove()
4:    n_i.deploy(service)
5: end while
6: pool.clear()
```

active service, then a replica is tried to be selected. If there is no service satisfying these conditions, no action is taken. When a service is selected, then the best RTE is selected suitable for migrating the selected service by excluding service locations. If there is such an RTE, then it is checked if the $MigrationThreshold$ is satisfied. The selected service is migrated only if this condition is satisfied. Otherwise, a new service is selected as candidate and rest of the algorithm is repeated to select the best RTE. This is done until a service is migrated or no services are left that meet the conditions.

How Many Services to Migrate?

The number of services to be migrated at each round is a design parameter. We decided to migrate one service at a time. The main reason for this decision is to keep the decision mechanism as simple as possible. One can monitor resource usage of individual services in more details and can make more precise decisions. Monitored values can be considered while choosing services to migrate and how many services to migrate.

Currently, we can make a rough estimation about memory usage of a service. We do not have a mechanism to monitor resource usage of each service. Monitoring resources of individual services requires so much effort and resources. For example, if we want to monitor CPU usage of a service, we have to make analysis in thread level. Services can be multithreaded and we need a mechanism to track threads of each service which requires byte code instrumentation of services. This adds extra overhead to services and to the system to monitor threads and their CPU times. We have nodes with very low resources and we want to use them efficiently. Thus, we do not deploy a resource monitoring mechanism.

Which Service to Migrate?

When a decision is made to migrate a service from a runtime environment, a service must be selected to migrate. This selection function is based on weighted sum of two parameters. The service with the maximum value is selected for migration. These parameters are:

- last modification date l: The service that has been operating in the runtime environment for the longest time has more bias to be migrated. The main

loa 9. Migration Decision

```
1: result ← NoOp
2: candidateHistory ← ∅
3: done ← false
4: while !done do
5:    worst ← getWorstRTE()
6:    if worst = null then
7:       return  done ← true
8:    end if
9:    service ← getMigrationCandidate(worst, active, candidateHistory)
10:   if service = null then
11:      service ← getMigrationCandidate(worst, replica, candidateHistory)
12:   end if
13:   if service = null then
14:      return  done ← true
15:   end if
16:   candidateHistory ∪ service
17:   serviceLocations ← getServiceLocations(service)
18:   best ← getBestRTE(serviceLocations)
19:   if best ≠ null then
20:      if !best.isOverloadedWith(candidate) then
21:         if worst.utility / best.utility ≤ MigrationThreshold then
22:            result ← MigrationOp(service, worst, best)
23:            done ← true
24:         end if
25:      end if
26:   end if
27: end while
28: return  result
```

reason for this decision is to prevent a service from oscillating among different runtime environments. Another reason is the assumption that the longer a service runs, the more likely it is to fail or the more resources it consumes. This can also be thought as leasing runtime environment resources in a fair manner among services.

- memory usage m: The more memory a service requires, the more likely it is to be migrated in order to increase utility of its current RTE.

The selection function S for service i can then be formulated as

$$S_i = \frac{l_i + m_i}{2} \tag{4.7}$$

where l *and* m are normalized values. l is normalized based on the start time of calculation and subtracted from 1 to maximize the value of the oldest service. The

equation is as follows:

$$l_i = 1 - \frac{last_{mDate}}{currentDate} \tag{4.8}$$

m is normalized based on the total memory usage of the JVM. The equation is as follows:

$$m_i = \frac{m_{Si}}{m_{JVM}} \tag{4.9}$$

Having the set of selection functions $S = (S_1, S_2, ..., S_n)$, the service to be migrated is then selected as $Max(S)$. The goal of this selection function is to choose a service that has been operating for the longest time and consumes most memory.

4.5.4 Replication Strategy

Replication is one of the techniques for providing high software availability. However, it has the challenge of keeping states up to date. As it is pointed out previously, state of services are stored in the knowledge store. Any structural changes on the knowledge store are synchronized by DS2OS. Whenever a change on the structure of the knowledge store occurs, 3 update messages are broadcasted to all other DS2OS instances. If any other DS2OS instance does not receive these update messages, another strategy is followed. All DS2OS instances in the system broadcast periodic ping messages. Hash of the knowledge store is sent with these ping messages. Upon receiving ping messages, each instance compares its own hash and the received hash of the knowledge store. If they don't match, then requests for pulling the structure updates are sent. Nodes which do not receive the update messages update their knowledge stores via this mechanism of comparing hashes. However, DS2OS synchronizes only the structure but not the knowledge itself. For this purpose, we have designed a state update mechanism that periodically sends the states of primary services to their backup replicas. The consistency of service states is achieved by these updates.

How Many Replicas to Maintain?

The decision about how many replicas of a service to maintain is based on the parameters provided by users at service installation time. These parameters are described in the introduction of this chapter. These parameters indicate how critical a service is. For example, if we have a service that is responsible for building security and a service that is responsible for weather forecast, they will not have the same level of priority for replication. The number of replicas required to maintain is decided based on this priority value. Assume that the number of replicas to be maintained for a weather forecast service is $\tau = 1$. When this service is installed for the first time, immediately 1 replica is created and placed on an appropriate location. If the runtime environment that holds the replica of this service is detected as crashed, the required number of replicas are not satisfied any more. A new replica is created and placed on an appropriate runtime environment with sufficient resources.

Thus, the system ensures a constant τ number of replicas to be maintained for each service.

Which Service to Replicate?

Replication operation can take place only in two conditions. First one is when a service is installed for the first time to the system. The second one is when a runtime environment fails. In the first condition, required number of replicas of newly installed service are created and placed according to the placement strategy described above. In the second condition, all of active services and replicas operating in the failing node have to be compensated. Replication upon node failure is required to meet the constraints described in the previous subsection. When a node fails, replication of services has an optimization issue that has to be satisfied. This issue is that, replicas of crashed primary services are activated and they are replicated first. Then, the crashed replicas are replicated again either from other replicas or from the active one. Another optimization while replication is that, if a service to be replicated has a replica, this replica is replicated rather than the active one. Thus, the primary service is not interrupted. This optimization helps us to minimize service downtime.

Where to Place a Replica?

We have some constraints about how to decide where to place replicas of a service. First constraint is that no two replicas of a service are never placed in the same physical location. Second constraint is that an active service and its replica are never placed in the same physical location. A replica is placed as described in subsection 4.5.2 by taking into account these constraints.

4.5.5 Failure of SMYRNA

Smyrna is designed in the same way with all other services in the system. It also manages itself by migrating and creating backup replicas. A problem raises at this point. What happens when Smyrna fails due to failure of the node that it is deployed? At this point, replicas of Smyrna take action. Smyrna sends ping messages to its replicas as all RTEs send ping messages to Smyrna. When a replica detects the failure of the primary instance, it tries to query primary instance directly by using the overlay. It starts a new primary election among backup replicas if the primary is detected as failed. Decision about the new primary is based on replica IDs. Replica with the lowest ID becomes new primary instance and takes control. Failure detection is done with the modified eventually perfect failure detector.

4.6 Runtime Environment (RTE)

Taking into account the problems mentioned in Analysis chapter 2 and the current state of the art solutions, we decided to use the OSGi framework as the base of our runtime environment. Due to its modular structure and small footprint in terms of necessary resources, OSGi meets our needs in terms of dynamism and heterogeneity. One could ask why we didn't choose web services in an environment full of services. When we compare web services and OSGi framework, we can say that they both provide the same functionalities. Dynamic service discovery and event handling mechanism (publish-subscribe) can be listed among these functionalities. The main argument of choosing OSGi rather than web services is the small footprint requirements of OSGi framework. We have a heterogeneous environment where embedded computers with scarce resource can also operate. It would not be wise to run a web server on an embedded computer or on a mobile device.

As it was mentioned in subsection 2.1.1, the workspace of this thesis is the services layer of the autonomous control and management platform for home automation. OSGi allows mechanisms for providing, discovering and consuming services. It also provides a publish-subscribe mechanism where a service subscribes for another service and the framework notifies the subscribers of changes occurring on their interested services. When a service is registered to the framework, it notifies the subscribers that a service of interest has been added to the system. It also notifies when a service is removed or a change occurs on the subscribed services.

Our services are deployed as bundles in the OSGi framework. A special bundle called RTE, is deployed in every physical computing node. RTE is responsible for performing runtime environment specific functionalities. These functionalities can be classified into two main groups, *informational* and *operational*.

- **Informational** functionalities are as follows:
  - **Utility function:** as mentioned in Section 3.2, every runtime environment is represented by a utility function. Utility function calculates a utility value every Δ time interval and keeps a history of size η. This utility history is used for decision making. The values of Δ and η are a trade-off between quick reaction to changes in the environment and efficient resource consumption.
  - **Service tracking:** we need to know which services are running on which physical node. This functionality is a publish-subscribe service that notifies the subscribers of changes occurring on the services of interest.
  - **Service monitoring:** we need to collect some statistical information about services. Total time of operation, crash reports and resource consumption can be examples for information collected.
- **Operational** functionalities are as follows:
  - **Service transfer:** since we are talking about migration and replication, we need a mechanism to transfer services among different runtime en-

vironments. Every runtime environment has mechanisms to send and receive service. A service receiver service is waiting for incoming connections from other runtime environments. Service transfer is triggered by a notification received by the knowledge store when a migration or replication field is assigned a value by SMYRNA. When the notification is received, the runtime environment sets the required fields on the knowledge store of the destination RTE. These fields are the file name and state address. The destination RTE receiving the notification that a service is being transferred reads the file name and the state address of the service. Then, it requests the serialized state of the service from the knowledge store. When the file transfer is completed, the receiving RTE install this service and keeps the state in a buffer until this new service starts its operation and registers to the knowledge store. After installing the new service, a notification for the completion of transfer is sent to the sender. The sender uninstalls the service after receiving the transfer completion notification. OSGi framework provides a publish-subscribe mechanism which notifies when a service of interest is installed, updated or removed in the system. Upon receiving the installation notification for the new service, it gets the state address of the service through the API explained earlier in Section 4.3. Then, it deserializes the state of this service in the buffer and stores it to its new address in the knowledge store.

- **State synchronization:** different number of replicas are maintained for services. State consistency of replicas is provided by periodic state update messages triggered by Smyrna. Smyrna notifies RTEs to send the state of a service to other RTEs where replicas are located. Thus, RTEs are responsible for sending and receiving service state updates.
- **Updating:** users can update their current services with newer versions. Every runtime environment is responsible for the services deployed on its physical machine. Hence, updating services is in the responsibility of runtime environments.
- **Heartbeats:** every runtime environment needs to send heartbeat messages for failure detection. These messages are sent periodically every Δ time interval. The failure detection mechanism makes decisions based on the received heartbeat messages.

OSGi, as our runtime environment, handles many challenges and answers partially many questions listed in Section 2.3. It does not have high resource requirements which was a challenge caused by heterogeneity. It is service oriented and provides a plug-and-play mechanism where services can be added and removed to/from the system dynamically. Thus, deals with the dynamism of the environment.

Chapter 5

Implementation

This chapter introduces details about the prototype implementation of SMYRNA and RTE in 5.1 and 5.2 respectively. Then, our design argumentation on two main design choices are addressed. The first argumentation is about the adaptation of DS2OS to OSGi which is given in 5.3. The second argumentation is about the migration strategy on the decisions on the number of services to be migrated and the strategy on the service selection to be migrated and is given in 5.4. Finally, the reason for using simulations in our experiments instead of using the prototype is given in 5.5.

5.1 SMYRNA

A prototype of the designed solution of SMYRNA is implemented in Java as an OSGi bundle. At the time of writing this document (28 September 2012), it includes 3097 lines of code with some functionalities missing due to the reasons explained in section 5.5. List of classes of the prototype can be found in Appendix B.1.

SMYRNA treats itself also as any other service in the system which needs to implement the service API given in Appendix C. As soon as it starts operating, it subscribes for some special addresses in knowledge store for receiving requests from RTEs and its replicas. These addresses include notifications for RTE join request, RTE ping messages, alive-check requests by its replicas and removed service addresses. Then, it explores all RTEs in the system by querying DS2OS knowledge store. For each explored RTE, it subscribes for some addresses to communicate with the RTE. These addresses include list of services, generated crash reports and calculated utility values. It also informs the RTE for the address where to send ping messages and information about removed services. Explored RTEs are added to known nodes which are later monitored for failures.

When SMYRNA wants to manage a service, it simply sends a message to the RTE that hosts that service and responsible RTE performs desired operation on that service. This is possible again with publish-subscribe mechanism of DS2OS. RTEs subscribe for certain addresses of services which are managed by SMYRNA.

For example, if a service s is desired to be migrated, SMYRNA sets destination address to its corresponding address and responsible RTE is notified about this. An example address can be as

/knowledge/client_3/rte/services/de.edu.tum.service.impl_1.0.0/migrate. Value that is set by SMYRNA is a destination RTE which can be as */knowledge/client_7/rte*. When the RTE with ID *client_3* receives a notification about the change on the address, it gets the destination stored and performs necessary operations to migrate desired service to specified destination.

When the failure detector informs SMYRNA about a suspicion of possible failure of an RTE, it tries to directly query suspected RTE using the DS2OS overlay. This direct query works as a tunnel established by DS2OS between requester and listener ends. Listeners subscribe to special addresses indicating it as a tunnel address. When a get request is made to that address, DS2OS redirects the request to the listener and invokes the waiting method of which result is returned immediately to the requester synchronously. If the result returned by suspected RTE is positive indicating that it is alive, SMYRNA revises its decision and increments the interval for checking ping messages by Δ. If suspected RTE is dead, then a recovery protocol is started which first activates the backup replicas of active services deployed on the crashed RTE and then creates backup replicas of all dead services to maintain required number of replicas for each service.

A general flow of events is given shown in Figure A in Appendix A.

5.2 RTE

As it is shown in Figure 4.4, we have another entity that is part of our design. RTEs are responsible for the functionalities listed in Section 4.6. Thus, a prototype is implemented in Java as an OSGi bundle which performs desired functionalities. List of classes of the prototype can be found in Appendix B.2. RTE implementation currently includes 4380 lines of code.

As soon as RTE starts operating, it establishes connection to the DS2OS knowledge store and starts tracking for services in its local OSGi environment. Then it initiates service failure detector which periodically checks service states if they are alive or not. If a service is not alive, it is tried to be restarted. For each service failure, a crash report is generated. If a service keeps crashing, it is not restarted more than 3 times and a detailed crash report is generated indicating that the service is not operating any more. After starting service failure detector, RTE subscribes for some special addresses to receive notifications whenever a change occurs on state of these addresses. These addresses include where to send ping messages, where to send removed service notifications and utility calculation address which triggers the event of calculating a new utility. Values of these addresses are set by SMYRNA. Finally, RTE tries to explore a primary SMYRNA instance in the system and informs about joining the system if it finds any.

When a service registers itself to OSGi registry, RTE receives a notification about

this event via its service tracker and subscribes for special addresses for controlling that service. These special addresses include fields such as migration, replication, suspending, resuming and stopping of a service. Values of these services are set by SMYRNA and necessary actions are taken when notifications are received for state changes.

When a migration or replication notification is received for a service, RTE suspends desired service and retrieves its state from knowledge store. Then, state is transferred to destination RTE and service itself is transferred later. RTE that receives state of a service, puts it into a temporary buffer where it keeps for a period of time to be restored when the corresponding service is registered. If the service of which the state is stored registers to OSGi resgistry, its state is restored in the knowledge store. Thus, service can resume its operation from its previous state.

A general flow of events is given shown in Figure A in Appendix A.

5.3 Adapting DS2OS to OSGi

DS2OS project is originally implemented in pure Java and is deployed as a Java application. It provides both synchronous and asynchronous communication with the usage of TCP and UDP protocols respectively. It generally has two components, *agent* and *client* sides. The agent side is responsible for maintaining the middleware while the client side provides the communication of services with the agent side. We had two options, to have a single client and share it among all services or to have a client per service. If we use a single shared client, then the addressing mechanism would be complicated. So, we decided to have a client per service. However, this would lead to having a number of TCP and UDP connections equal to the number of services deployed on each node. Also the communication of services with the local agent would be performed over these protocols. This is costful in terms of resources such as ports and time. Thus, we decided to adapt the current DS2OS as an OSGi bundle and implemented an OSGi protocol additional to datagram and stream protocols.

The adapted version of DS2OS is a bundle deployed in the OSGi framework and it is registered to the OSGi registry as an agent. All other services that want to communicate with the agent, instantiate a client instance and register to the OSGi registry as clients. Thanks to the publish-subscribe mechanism provided by OSGi framework, the agent side tracks the clients registered to the framework. When a new client is registered, the agent side client tracker is notified about this. Upon receiving the notification, the agent adds the client to its repository of the known clients. The identification of clients is done with the combination of bundle name and bundle version which provides a unique naming mechanism in an OSGi framework. Instead of using ports for communication, the messages are sent and received via method calls. Thus, this mechanism eliminates the resource consumption of ports equal to the number of services and increases the speed of communication with the local agent side. The communication with the other agents

is done through the local agent with a single TCP and UDP port.

5.4 Service Migration Strategy

The current migration strategy migrates one service at a time and the service to be migrated is selected based on two parameters. The reason of this strategy is that we are not able to determine actual resource consumptions per service basis. Currently, only a rough estimation about the memory allocated by a service is done at service installation time. Monitoring the actual resource consumption requires either source code or byte-code instrumentation or the modification of the JVM in Java based applications [HK03]. However, source code instrumentation requires the source code to be available which is not the case all the time and service developers may not be willing to expose their source code. Byte-code instrumentation on the other hand, is possible either at class loading time or after source code compilation but this method introduces overhead to the applications which is not desired by our design because of the heterogeneity of node resources where nodes with low resources may exist. Modifiying the JVM however, does not introduce any overhead but it contradicts with the main purpose of the Java programming language which is "write once, run anywhere". The portability of applications gets restricted only to the modified JVM. Thus, none of these methods has been adapted to our design to monitor actual resource consumptions.

We decided to migrate only a single service at a time because the affect of the migration operation cannot be predicted beforehand. Hence, our migration operation aims at relaxing nodes with low resources in long terms to prevent possible overloaded nodes.

The last modification date of a service is used to decide on the service to migrate. Last modification date is the time at which a service has been installed or updated on an RTE. The main reason for this parameter to be used is to prevent the possibility that a service is migrated all the time. This situation can occur when a service consumes a lot of resources which causes the node that it is deployed to run out of resources. In this case, that node will be selected as the worst RTE where a service will be migrated. If the decision to select the service to be migrated is based only on the resource consumption, then this service would be the one to be migrated again. However, taking into account the last modification date prevents this from happening and lets this heavy service to operate on the node it is deployed for some more time. Another for choosing the last modification time for migrating a service is to prevent possible memory leaks Java which are caused by developers [DS99]. The longer a service runs, the more memory it may consume.

5.5 Prototype vs Simulation

It is mentioned earlier that SMYRNA depends on functionality of DS2OS. However, DS2OS has some functionalities missing at the time of writing this document (28

September 2012) which are essential for the functionality of SMYRNA. Thus, experiments on testing SMYRNA were conducted by simulations instead of using the prototype. Currently, the prototype implementation does not have load balancing fully functional. List of classes of the simulation can be found in Appendix B.3. The simulation has in total 3406 lines of code.

Chapter 6

Evaluation

We present the results of simulations performed to test our design under different circumstances. The placement protocol succeeds on dealing with the heterogeneity of the nodes. Load balancing algorithm performs well in the presence of both overloaded and underloaded nodes. The system is able to adapt to the dynamic nature of the problem space by detecting failures of nodes and balancing loads after resource demands on nodes change. Autonomic behaviour of the system is able to deal with changes and adapt to the environment except some cases where services maintain no replicas or all replicas crash together with the primary service. Resource utilization of nodes is not satisfying where more than one constraint are tried to be satisfied which raises the necessity of an improvement of utility functions.

This chapter introduces the simulations performed to test our design and the results obtained for different scenarios. Experimental setup is explained in 6.1 including the service placement strategy and performance metrics in 6.1.1 and 6.1.2 respectively. Then, it continues with the first scenario where node failures are simulated in 6.2. After that, second simulation scenario and its results are presented in 6.3 where nodes failures and recoveries are simulated. This chapter concludes with the last scenario in 6.4 where nodes run out of resources over time.

6.1 Experimental Setup

We used PeerSim for simulation purposes of our design. It is a scalable and dynamic peer-to-peer simulation framework [MJ09].

Our system makes a contract with the services deployed. This contract is of a kind that promises certain amount of resources to the services so that they can operate by a desired level of quality of service. Thus, our optimum case during any time interval is the case where all services are satisfied with the resources allocated to them. We have three different simulation scenarios to test our design to see how it performs under different circumstances. The first scenario is where nodes in the system keep failing with a certain probability. Second scenario is where nodes recover after failures. The last scenario is where the nodes in the system start to

run out of resources over time.

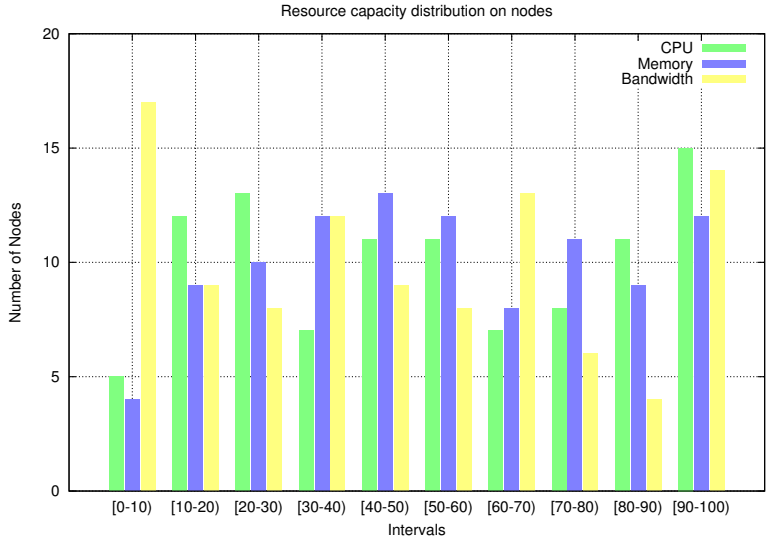
For all these scenarios, we have two different setups where node resources are generated uniformly and exponentially on an interval. Resources of nodes are CPU, memory and bandwidth capacities that are stated earlier in our system design 4. In the first setup, resources are generated uniformly on the interval $1 - 100$. However, it does not mean that a node with high CPU resource also has high resources for memory and bandwidth. In the second setup, resources are generated exponentially with $\lambda = 1$. The reason for this setup is to simulate where we have most of the nodes with small capacities and very few of them with large capacities. Figure 6.1(a) shows the histogram of uniformly generated resource capacity distribution of nodes. Figure 6.1(b) shows the histogram of exponentially generated resource capacity distribution of nodes.

In both cases, we have 100 nodes and 1000 primary services initially. Primary services have a number of replicas created. The maximum number of service replicas, R, is a parameter which is tested for different values. Replica services do not require any resources when deployed on a node because we assume that they do not consume any resources when suspended.

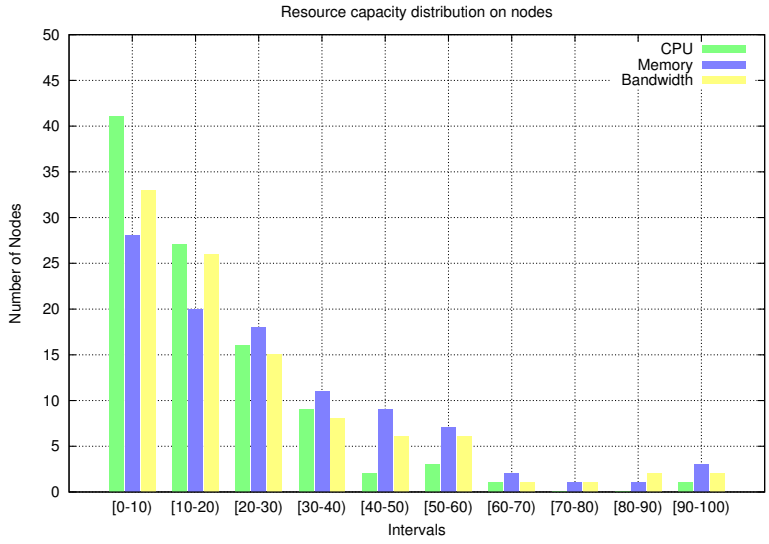
Service resource demands are generated uniformly on the interval $0-$*average node capacity*. This way of generating service demands guarantees that there will be some services with demands greater than node capacities. In such cases, service placement will cause troubles which we want to test.

An important thing about the experiments is that services that cannot be recovered after a node fails are not considered in the rest of the simulation. Failed nodes are recovered in some scenarios but the services are not recovered.

- *Number of nodes*: total number of nodes used in the system.
- *CPU capacity interval*: range of values that a node can have as CPU capacity.
- *Memory capacity interval*: range of values that a node can have as memory capacity.
- *Bandwidth capacity interval*: range of values that a node can have as bandwidth capacity.
- *CPU load factor*: ratio of total CPU demand to total CPU capacity.
- *Memory load factor*: ratio of total memory demand to total memory capacity.
- *Bandwidth load factor*: ratio of total bandwidth demand to total bandwidth capacity.
- *Node failure probability*: probability that a node can fail at each cycle.
- *Number of primary services*: total number of unique services in the system.



(a) Uniformly distributed



(b) Exponentially distributed

Figure 6.1. Node resource capacity distributions. Figure 6.1(a) depicts the case where node capacities are generated uniformly. Figure 6.1(b) shows the case where node capacities are generated exponentially.

- *Maximum number of replicas (R)*: maximum number of backup replicas that a primary service can have. I does not mean that every service this number of backup replicas. This is the maximum number of replicas a service can have. The number of replicas are generated uniformly on the interval $[0 - R]$.

Number of **nodes**	100
CPU capacity interval	$1 - 100$
Memory capacity interval	$1 - 100$
Bandwidth capacity interval	$1 - 100$
CPU load factor	$CLF = \%50$
Memory load factor	$MLF = \%50$
Bandwidth load factor	$BLF = \%50$
Node failure probability	$\%p$
Number of primary **services**	1000
Maximum number of replicas	$R = 0, 1, 2, 3, 5, 10$
Migration threshold	$M = 0.9$
Node utility history length	$\eta = 10$
Minimum non-null values	2
Number of migrations	1

Table 6.1. Experiment Parameters

- *Migration threshold*: ratio of the selected worst RTE's utility to the selected best RTE's utility. Used in case of migration when the system is totally overloaded or underloaded. For example, worst RTE's utility is 75 and best RTE's utility is 100. $\frac{75}{100}$ will lead to a value of 0.75 which is less than our migration threshold. Thus, it satisfies the constraint that the selected best RTE is at least %10 better than the selected worst RTE. So, migration will take place.

- *Node utility history length*: number of utility values to be stored.

- *Minimum non-null values*: minimum number of required utility values in order make calculate utility of a RTE's.

- *Number of migrations*: number of services to be migrated in each cycle when the system is totally overloaded or underloaded.

6.1.1 Service Placement

- Each service $s \subset S$ demands CPU resource $\omega_c(t)$, memory resource $\omega_m(t)$ and bandwidth resource $\omega_b(t)$ to operate and these demands vary over time t.

- The placement of services is defined in terms of the placement on each node $n \subset N$, denoted as $S_n(t)$. $S_n(t)$ represents the set of services placed to run on node n at time t. For a placement to be valid, each service $s \subset S$ must be placed on exactly one node.

- CPU capacity of each node is divided among the services $S_n(t)$ proportionally to their demands. This means that allocated CPU capacity $\psi_c(t)$ for a service

s on node n at time t is $\psi_c(t) = \frac{\omega_c(t)}{\sum_{x \subset S_n} \omega_x(t)} * \Omega_c$ where Ω_c is the CPU capacity of node n.

- Memory capacity of each node is divided among the services $S_n(t)$ proportionally to their demands. This means that allocated memory capacity $\psi_m(t)$ for a service s on node n at time t is $\psi_m(t) = \frac{\omega_m(t)}{\sum_{x \subset S_n} \omega_x(t)} * \Omega_m$ where Ω_m is the memory capacity of node n.

- Bandwidth capacity of each node is divided among the services $S_n(t)$ proportionally to their demands. This means that allocated bandwidth capacity $\psi_b(t)$ for a service s on node n at time t is $\psi_b(t) = \frac{\omega_b(t)}{\sum_{x \subset S_n} \omega_x(t)} * \Omega_b$ where Ω_b is the bandwidth capacity of node n.

6.1.2 Performance Metrics

We measure the satisfied services, alive services and coefficient of variation of CPU, memory and bandwidth loads. Coefficient of variation shows how good the loads are balanced. The less value, the better balancing. Detailed explanation about the metrics is given as:

- *Satisfied Demand*: out of all primary services, the fraction of services with satisfied demands. A service s is satisfied if and only if $\psi_c \geq \omega_c$, $\psi_m \geq \omega_m$ and $\psi_b \geq \omega_b$. This means that in order to satisfy a service s, all of its resource demands must be met. If one of these demands is not met, the service is said to be unsatisfied.

- *Alive Services*: out of all primary services, the number of alive services, may not be satisfied, which are operating in the system together with their required number of replicas.

- *Coefficient of variation of total resource loads*: CPU, memory and bandwidth loads for a node n are defined as $\tau_{c,m,b} = \frac{\sum_{s \subset S} \omega_{c,m,b}}{\Omega_{c,m,b}}$ which represents the fraction of the total resource demand to its capacity. The coefficient of variation is then defined as the ratio of the standard deviation to the average of the total CPU, memory and bandwidth loads over all nodes. The coefficient of variation for a single variable aims at describing the dispersion of the variable in a way that is independent of the variable's measurement unit. The coefficient of variation for a model aims at describing the model fit in terms of the relative sizes of the squared residuals and outcome values. The lower the coefficient of variation, the smaller the residuals relative to the predicted value. This is suggestive of a good model fit [cov].

6.2 Nodes Fail

In this scenario, we simulates the case where the nodes in the system keep failing with a constant probability, namely %2. Failed nodes are not recovered in this scenario. Resource demands of services are not changed. The main goal of this scenario is to test the influence of maximum number of replicas maintained per service in case of node failures where recoveries are not possible.

Figure 6.2 shows the fraction of satisfied service, overloaded and alive nodes in the system. In this scenario, maximum number of replicas per service is $R = 2$ which is also equal to $logN$. CPU, memory and bandwidth load factors are all set to %50 of the total system capacities. Node capacities are uniformly distributed over all nodes. As it is seen from the figure, the number of overloaded nodes increases as the fraction of satisfied services decreases together with the number of alive nodes. After cycle 60 where %70 of nodes have failed, we see significant jumps on the fraction of the overloaded nodes. The reason for this is that the remaining nodes host all of the alive services together with their replicas. Thus, whenever a node fails, a large number of active services also crash and their replicas are then activated on the nodes that host the replicas of these services. And this introduces new resource demands on a large number of nodes. This is something expected as a result of constantly failing nodes that are not recovered.

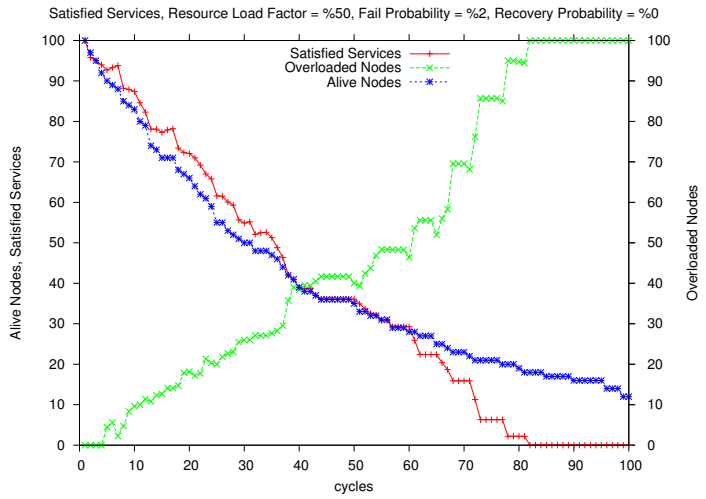


Figure 6.2. Satisfied services, overloaded and alive nodes where nodes fail with %2 probability. CPU, memory and bandwidth load factors are %50. Maximum number of replicas per service is $R = 2$.

Figure 6.3 shows the case of satisfied services for different maximum number of replicas maintained per service. The figure shows that the number of replicas maintained per service does not make any significant difference in terms of satisfied services. However, if we have a look at Figure 6.4 where the number of alive

services is shown, we see that the number of replicas makes a significant difference. Maintaining even a single replica per service enhances the resulting number of alive services by a factor 4. When we have services with maximum $R = 2 = logN$ replicas, we have 6 times more alive services compared to not maintaining any replicas. We see some spikes down towards the end of the simulation. This is the result of large number of services deployed on a node when we are running out of alive nodes. When a node fails, the primary services deployed on it are all considered to be dead until they are replaced by their replicas. Thus, there is a significant drop in the number of alive services at the end of the simulation but they are recovered by activating their backup replicas.

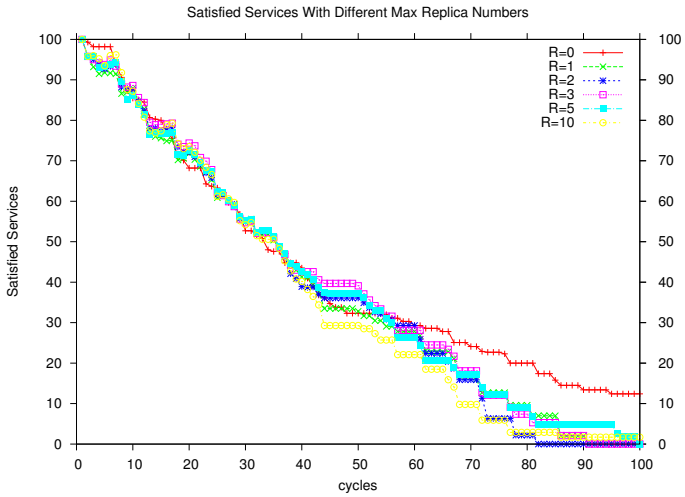


Figure 6.3. Satisfied services with different number of maximum replicas where nodes fail with %2 probability. CPU, memory and bandwidth load factors are %50.

Figure 6.5 shows that our load balancing algorithm works fine. Average CPU load increases because of the failing nodes and CoV of CPU load decreases inversely proportional to the average CPU load. When there are overloaded nodes in the system together with underloaded ones, our load balancing algorithm takes action and balances the load. The resource demands of services are changed every 10 cycles while keeping the overall CPU load factor stable. Thus, we see spikes in the CoV every 10 cycles where the demands are changed. However, these spikes are immediately healed by the load balancing algorithm. In cycle 45 where %65 of nodes have failed and the whole system becomes overloaded, CoV reaches its best point where it is almost 0. After that cycle, load balancing is not performed because there are no underloaded nodes in the system. Hence, CoV starts to increase again. Similarly, CoV for bandwidth and memory are given in Figure 6.6 and Figure 6.7 respectively.

We mentioned earlier that we have 2 different experimental setups where the node resources are generated uniformly and exponentially. Satisfied services with

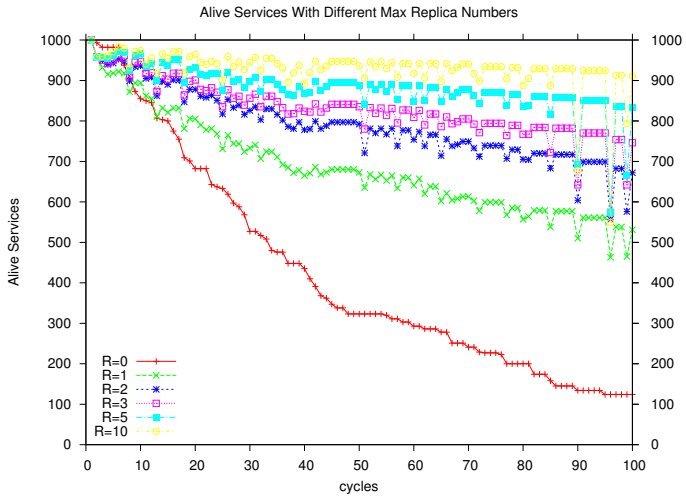


Figure 6.4. Alive services with different number of maximum replicas where nodes fail with %2 probability. CPU, memory and bandwidth load factors are %50.

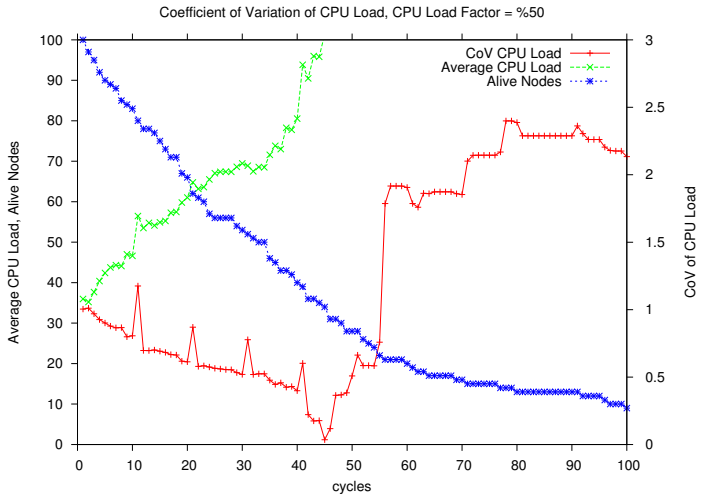


Figure 6.5. Coefficient of variation of CPU load where CPU load factor is %50, nodes fail with %2 and service demands change every 10 cycles.

different maximum number of replicas where node resources are generated exponentially is shown in Figure 6.8. In this case, we have plenty of nodes with low resources and only a few with high resources as it is seen from the histogram in Figure 6.1(b). We try to achieve satisfying service demands for CPU, memory and bandwidth at the same time in this scenario. The total demands on resources are %50 of total system capacities and nodes fail with %2 probability. Even from the beginning of

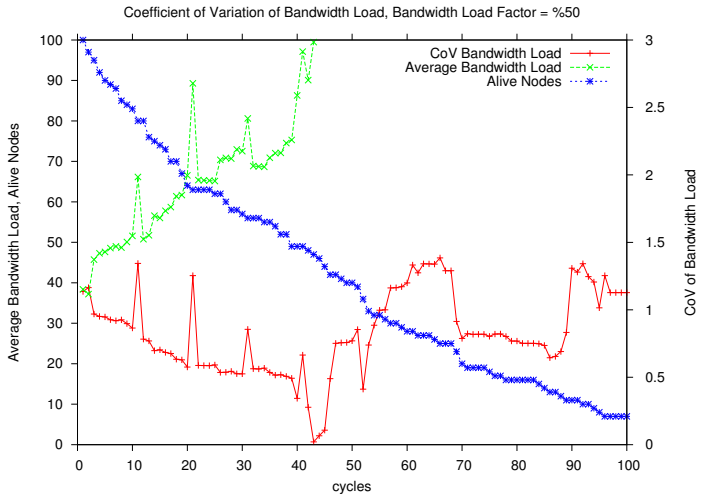


Figure 6.6. Coefficient of variation of bandwidth load where bandwidth load factor is %50, nodes fail with %2 and service demands change every 10 cycles.

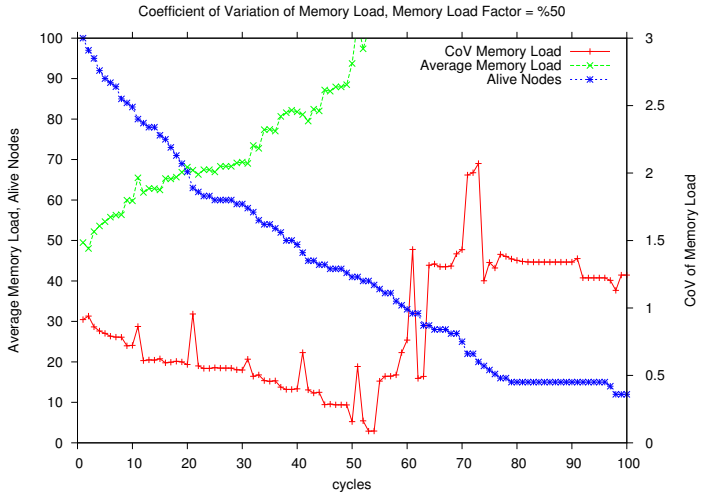


Figure 6.7. Coefficient of variation of memory load where memory load factor is %50, nodes fail with %2 and service demands change every 10 cycles.

the simulation, we see that around %70 of services are satisfied. We can say that having more backup replicas in such a case decreases the satisfied services. The reason for this is that when a node fails, all primary services are recovered from their replicas but this comes with its price. This price is that the more replicas, the more constraint on the number of nodes that a service can be placed. Thus, we see that our placement protocol performs worst with $R = 10$. However, if we

have a look at Figure 6.9 where the number of alive services is depicted, we can say that there is no significant difference compared to the case where node resources are generated uniformly in Figure 6.4.

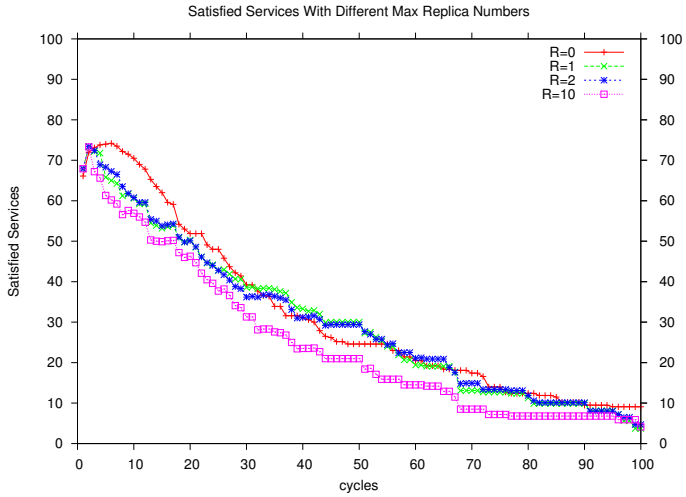


Figure 6.8. Satisfied services with different number of maximum replicas where nodes fail with %2 probability. CPU, memory and bandwidth load factors are %50. Node resources are generated exponentially.

Figure 6.9. Alive services with different number of maximum replicas where nodes fail with %2 probability. CPU, memory and bandwidth load factors are %50. Node resources are generated exponentially.

6.3 Nodes Fail and Recover

The second scenario that we tested our design is the case where nodes fail and then recover. However, services deployed on the failing nodes that do not have any replicas are not recovered. The main goal of this scenario is to test the influence of maximum number of replicas maintained per service in case of node failures where failed nodes recover.

Figure 6.10 shows the case where service resource demands change every 10 cycles. This change is the reason for the spikes downwards on the fraction of satisfied services. However, our load balancing algorithm performs well and within 1 or 2 cycles the load is balanced and services are satisfied again. The reason for this is that we have fresh nodes with full capacities available for service deployment and also the overall system load decreases because of the not recovered services that do not maintain any replicas. In Figure 6.11 where the service demands are kept constant, we do not have any significant spikes downwards.

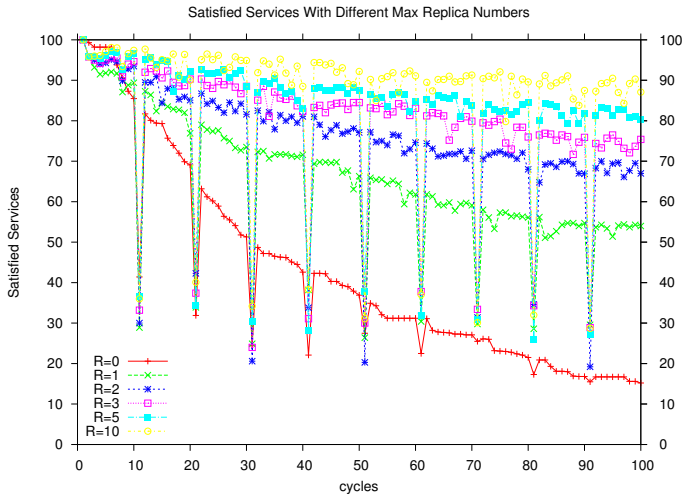


Figure 6.10. Failing and recovering nodes with different R where service demands are changed every 10 cycles.

Average CPU loads for both cases where service demands are kept constant and changed are given in Figure 6.12 and Figure 6.13 respectively. In Figure 6.12 we see the situation where the service demands are kept constant which results in a very stable load for the cases where services have backup replicas. This is not the case where no replicas are maintained because services are not recovered after a node crashed and the total resource demand keeps decreasing.

The results shown in Figure 6.13 however, are not that stable because the service demands change every 10 cycles. Thus, we have spikes upwards in the average CPU loads but our load balancing algorithm takes action in these cases and loads are stabilized as much as possible despite the constant node failures.

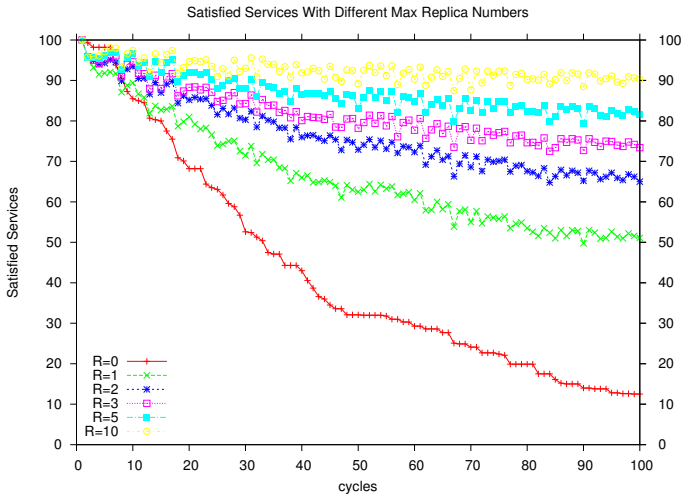


Figure 6.11. Failing and recovering nodes with different R where service demands are kept constant.

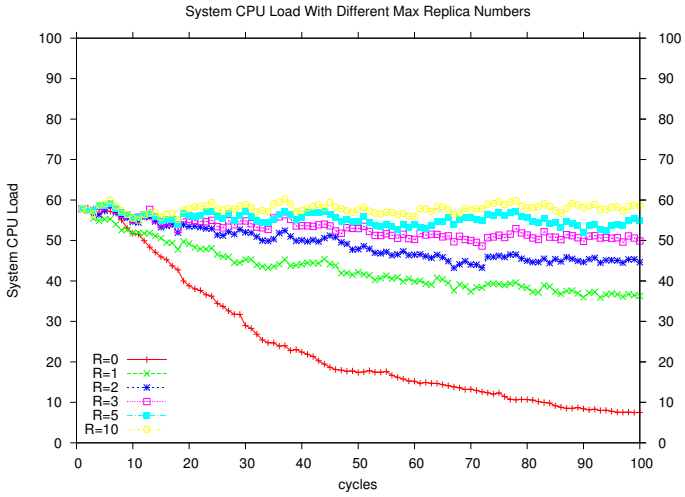


Figure 6.12. Average system CPU load for different R values where service demands are kept constant.

For both of the cases, we can say that the amount of satisfied services stabilizes around the end of the simulation for services which have backup replicas. We have around %65 of services satisfied when we have $R = 2 = logN$ and this number becomes %90 in the case of $R = 10$. This amount depends highly on the number of backup replicas. The more replicas, the more satisfied services.

For our second experimental setup where node resources are generated expo-

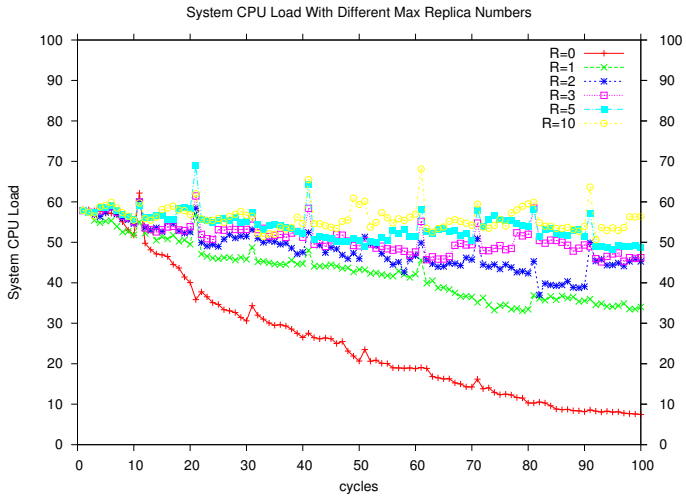


Figure 6.13. Average system CPU load for different R values where service demands change every 10 cycle.

nentially, we tested the fraction of satisfied services for this scenario. Figure 6.14 show the results of our simulation with different R values where nodes fail with %2 probability and resource load factors are all set to %50. Service demands do not change over time in this scenario. At the beginning of the simulation we see that only around %70 of services are satisfied as a result of our placement protocol. The reason for this is that there are plenty of nodes with low resources and only a few with high resources as shown on the histogram in Figure 6.1(b). On the other hand, we have services which demand resources up to the average capacity of the system. Thus, there are many services which demand resources that cannot be satisfied by most of the nodes in the system. Also we want to satisfy CPU, memory and bandwidth demands of services at the same time. Hence, we cannot satisfy the demands of all services. However, after a certain number of nodes have failed we have fresh recovered nodes with all their resources available for service deployment. This causes an increase in the amount of satisfied services which are the ones with backup replicas.

After cycle 70 is completed, the amount of satisfied services stabilizes for the cases where $R > 0$. For the case of $R = 10$, we see that there are some spikes downwards towards the end of the simulation. The reason for this situation is the failure of a node with high resource capacities and the constraint of not placing a service and its replica in the same location. Another reason is the large number of nodes with low resources. The more replicas, the more constraint on the number of suitable nodes for deploying a service. These spikes are not that significant for the cases when $R = 2, 3$ because there is less constraint on the available nodes to deploy services.

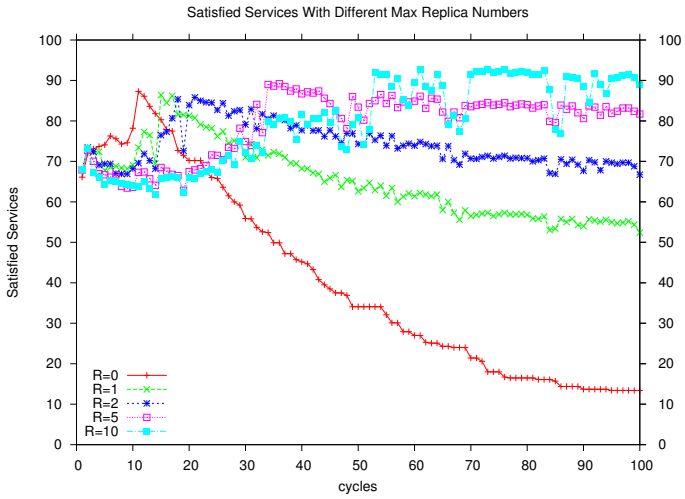


Figure 6.14. Satisfied services for different R values with exponentially generated node resources.

6.4 Decreasing Node Capacities

Our third scenario is the case where nodes run out of resources over time. This can be thought as clients of our smart environments start to use the available resources for their personal purposes. We maintain a threshold value indicating what percentage of a resource can be used by services on a node. The first scenario we tested is the case where we decrease this threshold for all resources on all nodes by %2 every 10 cycles. In this scenario, maximum number of replicas maintained is $R = 2$ and service demands do not change over time. Node resources are generated uniformly for this case and we want to satisfy CPU, memory and bandwidth demands where load factor for all of them is %50. Figure 6.15 depicts the results of this scenario. We see that decreasing the amount of available resources on the whole system even with %2 causes a large number of unsatisfied services and overloaded nodes. In the beginning of the simulation, all services are satisfied by the placement. However, after a decrement on all the nodes in the system, around %75 of services are unsatisfied but this is overcome by our load balancing algorithm within 2 to 3 cycles for the first three cases. It gets more and more difficult to satisfy service demands after the fourth decrement and we observe a falling wave model. The reason for this is the constraint of satisfying demands of CPU, memory and bandwidth at the same time and we are running out of resources.

We have another scenario where we decrement the capacity thresholds by %2.5 every 4 cycles. We test the cases where we want to satisfy only one of the resource demands -CPU-, two of the resources -CPU and memory- and all CPU, memory and bandwidth at the same time. In Figure 6.16, we try to satisfy only CPU demands of services and our placement protocol seems to be quite successful on this. All

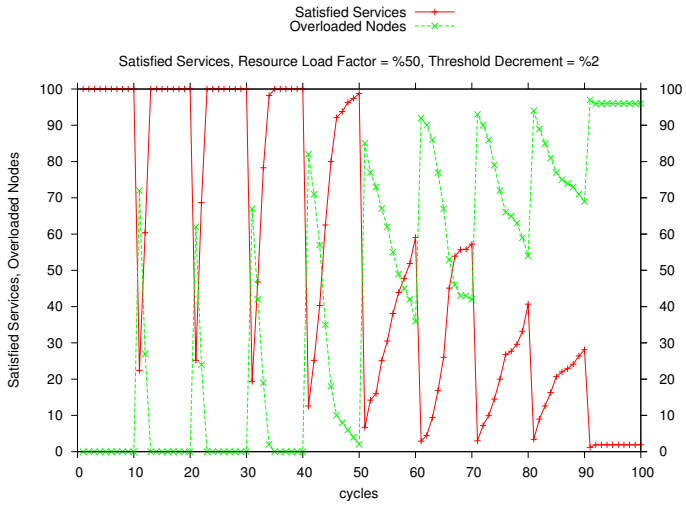


Figure 6.15. Satisfied services and overloaded nodes where CPU, memory and bandwidth load factors are %50. Node capacities decrease by %2 every 10 cycles.Maximum number of replicas per service is $R = 2$.

services are satisfied after load balancing is applied upon threshold decrement until cycle 84. At this cycle, the system capacity is decremented by %52.5 where we have %50 CPU demand. We can say that our placement protocol performs well in this case.

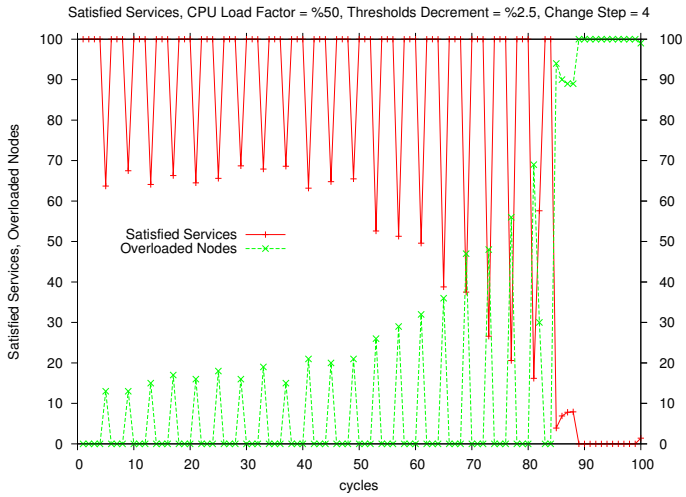


Figure 6.16. Satisfied services and overloaded nodes in case of decreasing node capacities. CPU load factor is %50. Capacities are decremented by %2.5 every 4 cycles. Maximum number of replicas per service is $R = 2$. We want to meet CPU demands of services.

In Figure 6.17, we have the results when we try to satisfy both CPU and memory demands of services. In this case, our placement protocol is able to satisfy all services after decreasing the overall capacity until %70. However, after subsequent decrements we are not able to satisfy all services at the same time. At this point we have around %80 service satisfaction and this number drops dramatically in the subsequent decrements. Even though the system has %70 of its capacity and the overall demand is %50, our placement protocol cannot satisfy all services. The reason for this is that we can have nodes with high CPU resources but with low memory resources or vice versa and it is not possible to satisfy both demands at the same time. Additionally, services also may demand high CPU and low memory or vice versa which causes some resources to be wasted and nodes which are not fully utilized.

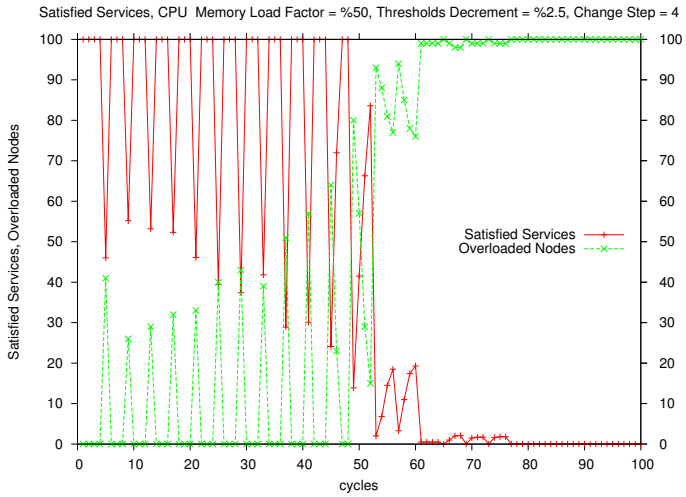


Figure 6.17. Satisfied services and overloaded nodes in case of decreasing node capacities. CPU and memory load factors are %50. Capacities are decremented by %2.5 every 4 cycles. Maximum number of replicas per service is $R = 2$. We want to meet CPU and memory demands of services.

In Figure 6.18 we have our most extreme case where we want to satisfy all CPU, memory and bandwidth at the same time. In this scenario, after cycle 16 where the overall system capacity is decremented to %90, we are not able to satisfy all services. We have %80 of services satisfied at this point and this value drops significantly in the subsequent decrements. Now we have one more constraint to satisfy and this results in a dramatic performance drop. The reason is the same explained for the previous scenario as we can not fully utilize all nodes because of the different node capacities and service demands.

We performed the above tests also for our second experimental setup where node resources are generated exponentially. Now we have many of the nodes with low resources and only a few nodes with high resources. Figure 6.19 depicts the case

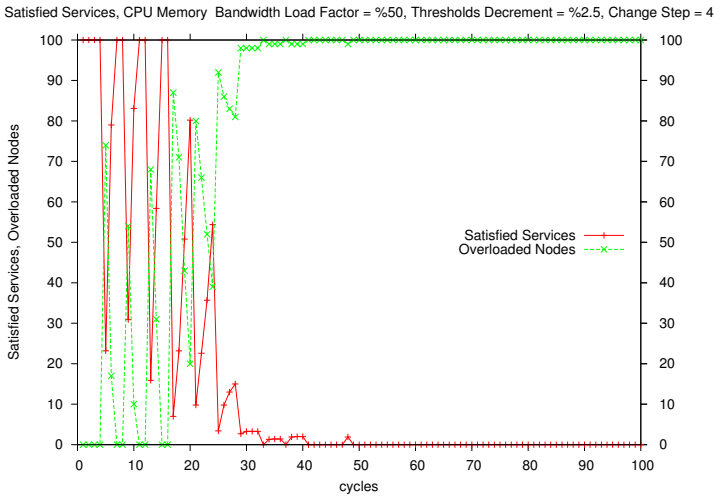


Figure 6.18. Satisfied services and overloaded nodes in case of decreasing node capacities. CPU, memory and bandwidth load factors are %50. Capacities are decremented by %2.5 every 4 cycles. Maximum number of replicas per service is $R = 2$. We want to meet CPU, memory and bandwidth demands of services.

where we want to satisfy only CPU demands of services and we can say that the results are almost the same with the case in uniformly generated node resources. The only difference is in the amount of satisfied services and overloaded nodes after decrements. In this case, we see longer spikes downwards in the number of satisfied services and longer spikes upwards in the number of overloaded nodes. And the reason for this is the large number of nodes with low resources that become overloaded each time the capacities are decreased. When a node becomes overloaded, all services in this node are unsatisfied because of our resource allocation strategy of sharing node resources proportional to service demands.

In Figure 6.20 we have the case where CPU and also memory demands are tried to be satisfied. In this case, we have a worse performance compared even to the case where node resources were uniformly generated and all resource demands were satisfied in 6.18. This is the result of having a large number of nodes with low resources and low number of nodes with high resources. And also nodes with high CPU resources may not have high memory resources at the same time or vice versa. This leads to a high number of nodes that are not fully utilized.

Our last scenario for this experimental setup is the case where we want to satisfy all resource demands at the same time. As it is shown in Figure 6.21, from the very beginning of the simulation even we can not satisfy all services. We start with %68 of services satisfied and this value is increased in the first 4 cycles to %77 by our load balancing algorithm. However, when the capacities are decreased for the first time, %98 of nodes become overloaded and almost all services are unsatisfied. After the capacities are decreased for the second time, all nodes are overloaded and in

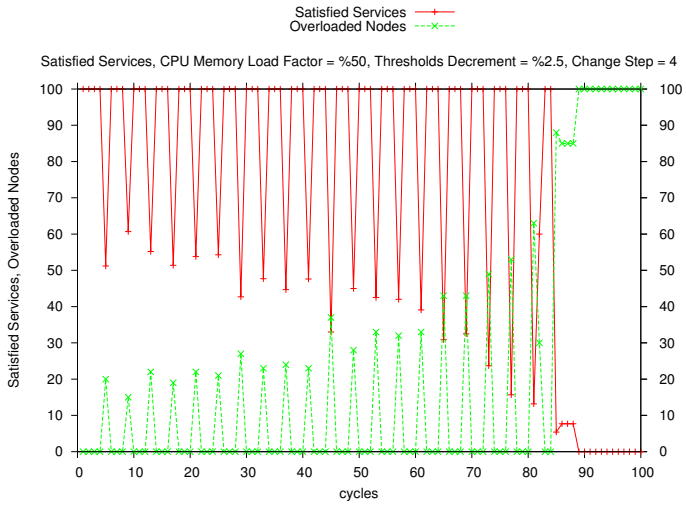


Figure 6.19. Satisfied services and overloaded nodes in case of decreasing node capacities where node resources are generated exponentially. CPU load factor is %50. Capacities are decremented by %2.5 every 4 cycles. Maximum number of replicas per service is $R = 2$. We want to meet CPU demands of services.

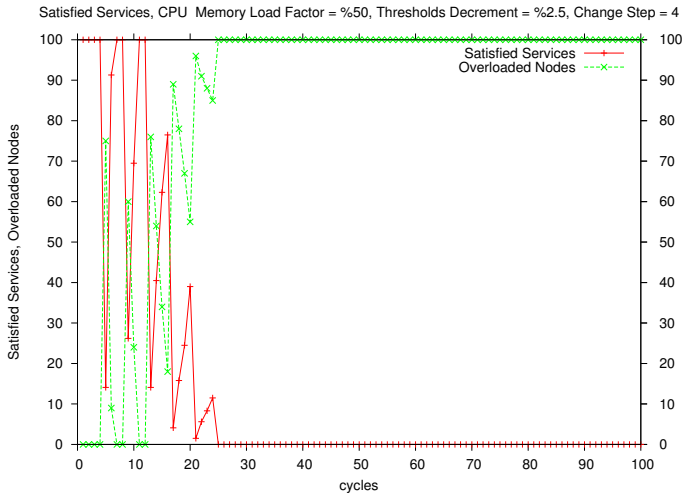


Figure 6.20. Satisfied services and overloaded nodes in case of decreasing node capacities where node resources are generated exponentially. CPU and memory load factors are %50. Capacities are decremented by %2.5 every 4 cycles. Maximum number of replicas per service is $R = 2$. We want to meet CPU and memory demands of services.

this case no load balancing is applied. As a result of this, we see the average loads of CPU, memory and bandwidth keep increasing as a pattern. Load balancing is applied only in the case where both overloaded and underloaded nodes are present.

Thus, we don't see any improvements in the rest of the simulation.

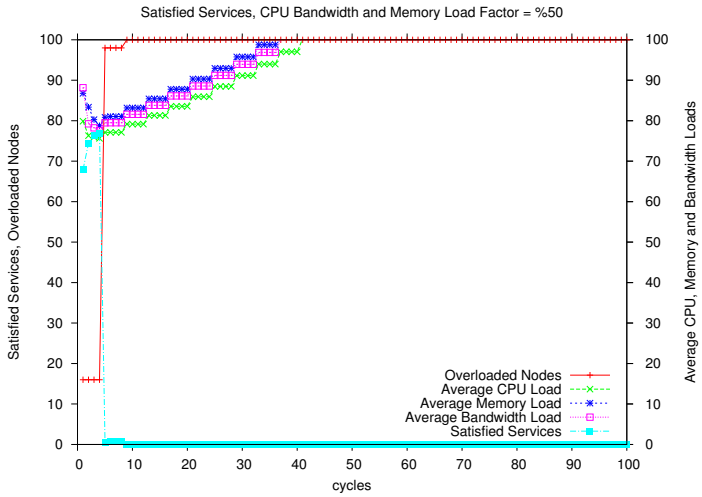


Figure 6.21. Satisfied services and overloaded nodes in case of decreasing node capacities where node resources are generated exponentially. CPU, memory and bandwidth load factors are %50. Capacities are decremented by %2.5 every 4 cycles. Maximum number of replicas per service is $R = 2$. We want to meet CPU, memory and bandwidth demands of services. Also average CPU, memory and bandwidth loads are shown.

Chapter 7

Conclusions and Future Work

7.1 Conclusions

In this thesis, we presented a reliable infrastructure for services in smart environments, SMYRNA. SMYRNA is an autonomous service placement protocol that is able to adapt to dynamic changes of heterogeneous environments. The presented design was developed by adapting solutions from different domains with similar problems. Research has been conducted on finding solutions in the domains of Grid Computing, Peer-to-Peer networks, Intelligent Agents, Ambient Intelligence Systems and Distributed Systems.

A prototype of the proposed system design was implemented and tested with different simulation scenarios. The results showed that the placement protocol succeeded on dealing with the heterogeneity of the nodes by using the introduced utility functions. However, utility functions need to be improved to perform better when more than one constraint is desired to be satisfied. It has been shown that the system load was balanced successfully in the presence of both overloaded and underloaded nodes. The loads were distributed proportional to node capacities which was among our goals to deal with heterogeneity of resources and use all system resources efficiently. However, node resource utilization was not satisfying when multiple resource demands were tried to be satisfied. Each additional constraint to be satisfied caused a significant drop on the overall system utilization. When the system was fully overloaded or underloaded, our novel migration strategy took action and migrated one service at a time from the worst node to the best node in order to prevent possible node overloading. It has been shown with extensive results that the system was able to adapt to the environmental changes by detecting node failures and balancing the load when the load on a node changed over time. This was also among our goals to deal with the dynamic nature of the environment. Our design has been tested that it was fully autonomous except some cases. These cases were the situations when a node failed and services on that node could not be recovered because there were no backup replicas maintained for these services. Another case was when multiple nodes failed at the same time and both primary

instances and replicas of services were deployed on these failing nodes. In this case also there were no backup replicas left in the system of these services and they were not recovered. Hence, we can say that external intervention is required to install the services to the system again only at such cases.

An autonomous service placement protocol that adapts to environmental changes and a reliable infrastructure for services in heterogeneous smart environments have been proposed and tested to be performing well under different circumstances.

7.2 Future Work

- **State Synchronization**: the consistency of replica states is currently performed by sending periodic state updates to service replicas. This scheme can create an overhead in the network traffic and needs improvement. A way of improvement can be achieved by sending these updates in an accumulated way where only the changed fields' states are sent. This can be done by calculating a hash for the states and periodically checking for the changes as it is currently done by DS2OS.

- **Load Balancing**: the current load balancing scheme is applied only in the presence of both overloaded and underloaded nodes. However, this can be further improved to be applied also in the cases where all nodes are overloaded or underloaded.

- **Service Resource Monitoring**: the current design is not able to monitor the exact resource consumption of individual services. Currently, we can only make a rough estimation about the memory used by a service when it is installed for the first time. However, we are not able to track its exact footprint of used memory, CPU and bandwidth. An improvement in this field can be applied by adding a mechanism for monitoring the exact resource consumption of a service. Based on these consumptions, more healthy decisions about migration of a service can be performed such as taking into account the bandwidth consumption while migration is possible on devices with limited connectivity. Another criteria for resource consumption can be the power consumption of each service so that this criteria can be taken into account while migration to devices with limited battery life.

- **Resource Allocation**: our current resource allocation mechanism allocates resources to services proportional to their demands. Currently, all services are treated equally. However, this can be further improved by introducing priorities to services and resource allocation can be performed by taking into account these priorities.

Bibliography

[age] Agentscape documentation, accessed 15.09.2012.

[Agl09] Aglets Development Group. *The Aglets 2.0.2 User's Manual*, March 2009.

[AH04] G. Attiya and Y. Hamam. Reliability oriented task allocation in heterogeneous distributed computing systems. In *Proceedings. ISCC 2004. Ninth International Symposium on Computers and Communications, 2004.*, volume 1, pages 68–73, June 2004.

[AmSS+09] K. Ahmed, M.A. m. Shohag, T. Shahriar, M.K. Hasan, and M.M. Rana. Strong thread migration in heterogeneous environment. In *Computer Engineering and Technology, 2009. ICCET '09. International Conference on*, volume 1, pages 205–209, jan. 2009.

[BCPR03] F. Bellifemine, G. Caire, A. Poggi, and G. Rimassa. Jade: A white paper. *EXP in search of innovation*, 3(3):6–19, 2003.

[BHD03] Sara Bouchenak, Daniel Hagimont, and Noël De Palma. Efficient java thread serialization. In *Proceedings of the 2nd international conference on Principles and practice of programming in Java*, PPPJ '03, pages 35–39, New York, NY, USA, 2003. Computer Science Press, Inc.

[BHR+02] J. Baumann, F. Hohl, K. Rothermel, M. Strasser, and W. Theilmann. Mole: A mobile agent system. *Software: Practice and Experience*, 32(6):575–603, 2002.

[BLC02] Eric Bruneton, Romain Lenglet, and Thierry Coupaye. Asm: A code manipulation tool to implement adaptable systems. In *In Adaptable and extensible component systems*, 2002.

[BOvSW02] F. M. T. Brazier, B. J. Overeinder, M. van Steen, and N. J. E. Wijngaards. Agent factory: generative migration of mobile agents in heterogeneous environments. In *Proceedings of the 2002 ACM symposium on Applied computing*, SAC '02, pages 101–106, New York, NY, USA, 2002. ACM.

[CFH+05] Christopher Clark, Keir Fraser, Steven Hand, Jacob Gorm Hansen, Eric Jul, Christian Limpach, Ian Pratt, and Andrew Warfield. Live migration of virtual machines. In *Proceedings of the 2nd conference on Symposium on Networked Systems Design & Implementation - Volume 2*, NSDI'05, pages 273–286, Berkeley, CA, USA, 2005. USENIX Association.

[cov] Ucla academic technology services, accessed 15.09.2012.

[Dah99] Markus Dahm. Byte code engineering with the javaclass api, 1999.

[DS99] Wim De Pauw and Gary Sevitsky. Visualizing reference patterns for solving memory leaks in java. In Rachid Guerraoui, editor, *ECOOPâ 99 â Object-Oriented Programming*, volume 1628 of *Lecture Notes in Computer Science*, pages 668–668. Springer Berlin / Heidelberg, 1999. 10.1007/3-540-48743-3_6.

[FD02] Alan Fedoruk and Ralph Deters. Improving fault-tolerance by replicating agents. In *Proceedings of the first international joint conference on Autonomous agents and multiagent systems: part 2*, AAMAS '02, pages 737–744, New York, NY, USA, 2002. ACM.

[FJK04] Ian Foster, Nicholas R. Jennings, and Carl Kesselman. Brain meets brawn: Why grid and agents need each other. In *Proceedings of the Third International Joint Conference on Autonomous Agents and Multiagent Systems - Volume 1*, AAMAS '04, pages 8–15, Washington, DC, USA, 2004. IEEE Computer Society.

[Fün98] Stefan Fünfrocken. Transparent migration of java-based mobile agents: Capturing and re-establishing the state of java programs. *Personal Technologies*, 2:109–116, 1998. 10.1007/BF01324941.

[Gre] Todd Greanier. Discover the secrets of the java serialization api.

[GS96] R. Guerraoui and A. Schiper. Fault-tolerance by replication in distributed systems. In *Reliable Software Technologies - Ada-Europe'96*, pages 38–57. Springer-Verlag, 1996.

[Gue06] Rachid Guerraoui. *Introduction to reliable distributed programming.* Springer, Berlin Heidelberg New York, 2006.

[Her10] Klaus Herrmann. Self-organized service placement in ambient intelligence environments. *ACM Trans. Auton. Adapt. Syst.*, 5:6:1–6:39, May 2010.

[HK03] Jarle G. Hulaas and Dimitri Kalas. Monitoring of resource consumption in java-based application servers, 2003.

[Huc] Gregory Huczynski. Forced migration in a mobile object system: Reducing fault tolerance costs and improving reliability in distributed systems.

[JB05] Márk Jelasity and Ozalp Babaoglu. T-Man: Gossip-based overlay topology management. In *3rd Int. Workshop on Engineering Self-Organising Applications (ESOA'05)*, pages 1–15, 2005.

[KC03] Jeffrey O. Kephart and David M. Chess. The vision of autonomic computing. *Computer*, 36(1):41–50, January 2003.

[KS97] S. Kartik and C. Siva Ram Murthy. Task allocation algorithms for maximizing reliability of distributed computing systems. *Computers, IEEE Transactions on*, 46(6):719–724, June 1997.

[LAB+] Jacob R. Lorch, Atul Adya, William J. Bolosky, Ronnie Chaiken, John R. Douceur, and Jon Howell. The smart way to migrate replicated stateful services. eurosys 2006.

[Lam01] Leslie Lamport. Paxos made simple. *ACM SIGACT News*, 32(4):18–25, December 2001.

[LYBB12] Tim Lindholm, Frank Yellin, Gilad Bracha, and Alex Buckley. The java virtual machine specification - java se 7 edition, June 2012.

[Mar08] Damianos Maragkos. Replication and migration of osgi bundles in the virtual osgi framework. Master's thesis, Swiss Federal Institute of Technology, 8092 Zurich, Switzerland, July 2008.

[MJ09] Alberto Montresor and Márk Jelasity. Peersim: A scalable p2p simulator. In *Proc. of the 9th Int. Conference on Peer-to-Peer (P2P'09)*, pages 99–100, Seattle, WA, September 2009.

[NS05] T Nakano and T Suda. Self-organizing network services with evolutionary adaptation. *Neural Networks, IEEE Transactions on*, 16(5):1269–1278, September 2005.

[OSG11] OSGi Alliance. *OSGi Service Platform Core Specification*, April 2011.

[PNS+09] Marc-Oliver Pahl, Christoph Niedermeier, Mario Schuster, Andreas Müller, and Georg Carle. Knowledge-based middleware for future home networks. In *IEEE IFIP Wireless Days conference Paris*, Paris, France, December 2009.

[PUC] Marc-Oliver Pahl, Deniz Ugurlu, and Georg Carle. Distributed smart space operating system.

[RLS+03] Ananth Rao, Karthik Lakshminarayanan, Sonesh Surana, Richard Karp, and Ion Stoica. Load balancing in structured p2p systems. In M. Kaashoek and Ion Stoica, editors, *Peer-to-Peer Systems II*, volume 2735 of *Lecture Notes in Computer Science*, pages 68–79. Springer Berlin / Heidelberg, 2003.

[SD07] Jan Sacha and Jim Dowling. A gradient topology for master-slave replication in peer-to-peer environments. In Gianluca Moro, Sonia Bergamaschi, Sam Joseph, Jean-Henry Morin, and Aris Ouksel, editors, *Databases, Information Systems, and Peer-to-Peer Computing*, volume 4125 of *Lecture Notes in Computer Science*, pages 86–97. Springer Berlin / Heidelberg, 2007. 10.1007/978-3-540-71661-7_8.

[SDCM06] Jan Sacha, Jim Dowling, Raymond Cunningham, and René Meier. Discovery of stable peers in a self-organising peer-to-peer gradient topology. In Frank Eliassen and Alberto Montresor, editors, *Distributed Applications and Interoperable Systems*, volume 4025 of *Lecture Notes in Computer Science*, pages 70–83. Springer Berlin / Heidelberg, 2006. 10.1007/11773887_6.

[SMK+01] Ion Stoica, Robert Morris, David Karger, M. Frans Kaashoek, and Hari Balakrishnan. Chord: A scalable peer-to-peer lookup service for internet applications. *SIGCOMM Comput. Commun. Rev.*, 31(4):149–160, August 2001.

[SSY00] Takahiro Sakamoto, Tatsurou Sekiguchi, and Akinori Yonezawa. Bytecode transformation for portable thread migration in java. In David Kotz and Friedemann Mattern, editors, *Agent Systems, Mobile Agents, and Applications*, volume 1882 of *Lecture Notes in Computer Science*, pages 443–481. Springer Berlin / Heidelberg, 2000. 10.1007/978-3-540-45347-5_3.

[Sun01] Sun Microsystems. *Java Object Serialization Specification*, August 2001.

[SVM07] R. Shah, B. Veeravalli, and M. Misra. On the design of adaptive and decentralized load balancing algorithms with load estimation for computational grid environments. *Parallel and Distributed Systems, IEEE Transactions on*, 18(12):1675–1686, dec. 2007.

[tec] Techterms, accessed 15.09.2012.

[TOT11] A. Takeda, T. Oide, and A. Takahashi. New structured p2p network with dynamic load balancing scheme. In *IEEE Workshops of International Conference on Advanced Information Networking and Applications (WAINA)*, pages 108–113, March 2011.

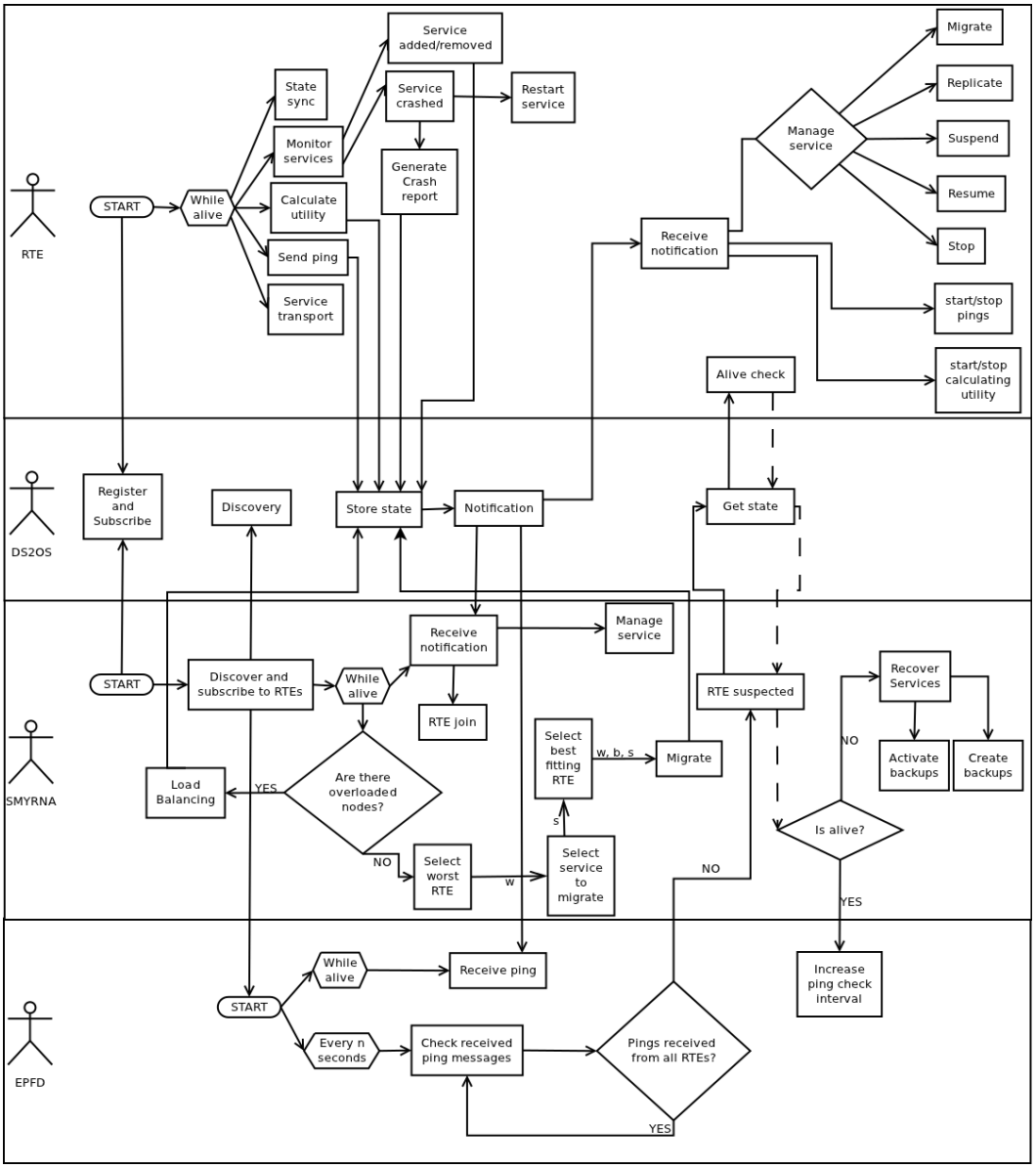
[Whi] Abe White. Serp - http://serp.sourceforge.net, accessed 15.09.2012.

[WS01] M. Wang and T. Suda. The bio-networking architecture: a biologically inspired approach to the design of scalable, adaptive, and survivable/available network applications. In *Applications and the Internet, 2001. Proceedings. 2001 Symposium on*, 2001.

Appendix A

General Flow of Events

The general flow of events is shown in Figure A. DS2OS, Smyrna, runtime environment (RTE) and Eventually Perfect Failure Detector (EPFD) are entities shown on the figure. EPFD actually functions as a part of Smyrna but it is shown as a separate entity for simplicity. Entities are separated by horizontal lanes where they can interact with other entities. Details are not shown in the figure. Only main functionalities are depicted.


RTE
START
While alive
State sync
Monitor services
Calculate utility
Send ping
Service transport
Service added/removed
Service crashed
Restart service
Generate Crash report
Receive notification
Manage service
Migrate
Replicate
Suspend
Resume
Stop
start/stop pings
start/stop calculating utility
Alive check
DS2OS
Register and Subscribe
Discovery
Store state
Notification
Get state
SMYRNA
START
Discover and subscribe to RTEs
While alive
Receive notification
Manage service
RTE join
Are there overloaded nodes?
YES
Load Balancing
NO
Select worst RTE
w
Select service to migrate
s
Select best fitting RTE
w, b, s
Migrate
RTE suspected
Is alive?
NO
YES
Recover Services
Activate backups
Create backups
Increase ping check interval
EPFD
START
While alive
Receive ping
Every n seconds
Check received ping messages
Pings received from all RTEs?
YES

Appendix B

List of Classes

B.1 SMYRNA

- de.edu.tum.smyrna.epfd.EPFD.java
- de.edu.tum.smyrna.epfd.EPFDSubscription.java
- de.edu.tum.smyrna.UtilityHistory.java
- de.edu.tum.smyrna.impl.ReplicaRecovery.java
- de.edu.tum.smyrna.impl.DecisionFunction.java
- de.edu.tum.smyrna.impl.Activator.java
- de.edu.tum.smyrna.impl.UtilityCalculationTask.java
- de.edu.tum.smyrna.impl.SmyrnaImpl.java
- de.edu.tum.smyrna.rep.PrimaryElection.java
- de.edu.tum.smyrna.rep.ReplicaPinger.java
- de.edu.tum.smyrna.rep.PrimaryEPFD.java
- de.edu.tum.smyrna.rep.PrimaryElectionVirtualNodeHandler.java
- de.edu.tum.smyrna.rep.ReplicaRepository.java
- de.edu.tum.smyrna.rep.AliveCheckVirtualNodeHandler.java
- de.edu.tum.smyrna.sub.ServiceRemovedCallback.java
- de.edu.tum.smyrna.sub.ServiceRegistrationCallback.java
- de.edu.tum.smyrna.sub.ServiceFailureCallback.java
- de.edu.tum.smyrna.sub.ReplicaPingCallback.java

- de.edu.tum.smyrna.sub.RTEJoinCallback.java
- de.edu.tum.smyrna.sub.ReplicaListCallback.java
- de.edu.tum.smyrna.sub.UtilityCallback.java
- de.edu.tum.smyrna.RTEEntry.java
- de.edu.tum.smyrna.Operation.java
- de.edu.tum.smyrna.OpType.java

B.2 RTE

- de.edu.tum.rte.Activator.java
- de.edu.tum.rte.ServiceEntry.java
- de.edu.tum.rte.Parameters.java
- de.edu.tum.rte.impl.RTE.java
- de.edu.tum.rte.impl.ServiceFailureDetector.java
- de.edu.tum.rte.impl.CircularBuffer.java
- de.edu.tum.rte.impl.ServiceManager.java
- de.edu.tum.rte.impl.SmyrnaUtility.java
- de.edu.tum.rte.sub.ResumeCallback.java
- de.edu.tum.rte.sub.SuspendCallback.java
- de.edu.tum.rte.sub.RemoveAddressCallback.java
- de.edu.tum.rte.sub.StopCallback.java
- de.edu.tum.rte.sub.TransferCompleteCallback.java
- de.edu.tum.rte.sub.MigrateCallback.java
- de.edu.tum.rte.sub.IsAlivePipe.java
- de.edu.tum.rte.sub.ReplicateCallback.java
- de.edu.tum.rte.sub.PingSubscription.java
- de.edu.tum.rte.sub.UtilityCallback.java
- de.edu.tum.rte.Utility.java

- de.edu.tum.rte.util.RTECommandProvider.java
- de.edu.tum.rte.util.RTEIServiceTrackerCustomizer.java
- de.edu.tum.rte.util.ServiceTransport.java
- de.edu.tum.rte.util.ServiceReceivePipe.java
- de.edu.tum.rte.util.StateRestoreTimer.java
- de.edu.tum.rte.util.EPFDPing.java
- de.edu.tum.rte.IRTE.java

B.3 Simulation

- de.edu.tum.smyrna.sim.main.Simulator.java
- de.edu.tum.smyrna.sim.DecisionFunction.java
- de.edu.tum.smyrna.sim.base.RTE.java
- de.edu.tum.smyrna.sim.base.ServiceMapping.java
- de.edu.tum.smyrna.sim.base.SortedList.java
- de.edu.tum.smyrna.sim.base.ServiceManager.java
- de.edu.tum.smyrna.sim.base.Service.java
- de.edu.tum.smyrna.sim.base.LCG.java
- de.edu.tum.smyrna.sim.base.RTEManager.java
- de.edu.tum.smyrna.sim.base.Operation.java
- de.edu.tum.smyrna.sim.base.OpType.java
- de.edu.tum.smyrna.sim.PerformanceObserver.java
- de.edu.tum.smyrna.sim.Smyrna.java
- de.edu.tum.smyrna.sim.controls.ServiceDemandGenerator.java
- de.edu.tum.smyrna.sim.controls.ServiceInitializer.java
- de.edu.tum.smyrna.sim.controls.SmyrnaPlacementInitializer.java
- de.edu.tum.smyrna.sim.controls.RTEInitializer.java
- de.edu.tum.smyrna.sim.controls.FailRecover.java
- de.edu.tum.smyrna.sim.controls.RTEThresholdGenerator.java

Appendix C

Service API

Service interface that defines the core functionalities required by Smyrna to manage services.

```
/**
 * This interface defines core functionalities required for management of services.
 * @author Cuneyt Caliskan
 */

public interface IService extends Runnable {
    /**
     * Starts the actual functionality of this service.
     */
    void start();

    /**
     * Resumes the functionality of the service which is waiting in
     * {@link Status#READY} state.
     */
    void resume();

    /**
     * Stops the functionality of the service and exits the system. Does the cleanup
     * like closing files, sockets, etc.
     */
    void stop();

    /**
     * Suspends the functionality of the service in case of preparation for migration
     * or replication. Also replicas of a service should be in suspended state.
     */
    void suspend();
```

```
/**
 * Returns the bundle name of this service in the form of
 * [bundleSymbolicName_Version].
 * @return the bundle name and version in the form of [bundleSymbolicName_Version]
 */
String getBundleName();

/**
 * Returns the last modified time of the bundle of this service in milliseconds.
 * @return last modified time in milliseconds
 */
long getLastModified();

/**
 * Returns the value representing if this service is the primary one or a replica.
 * @return {@code true} if this is the active service,
 * {@code false} if this is a replica.
 */
boolean isActive();

/**
 * Determines if this service is alive, for failure monitoring.
 * @return {@code true} if this method is accessible. Caller of this method will
 * determine that this service is dead if there is no response or cannot call
 * this method.
 */
boolean isAlive();

/**
 * Determines the amount of memory consumed by this service bundle.
 * @return amount of memory allocated for this bundle in bytes.
 */
long getMemoryUsage();

/**
 * Used to notify the failure of a runtime environment if this service is
 * subscribed for failures.
 * @param address address of the crashed RTE in KOR
 */
void notifyRTEFailure(String address);

/**
 * Used to add memory usage for this service. It is used by the RTE when a bundle
```

```
     * is installed and when a bundle is started.
     * @param amount memory in bytes
     */
    void addMemoryUsage(long amount);

    /**
     * Used to get the address where the service saves its state on the KOR.
     * @return the address where the service state is stored.
     */
    String getStateAddress();
}
```

Abstract class which includes some of the functionalities implemented. Service developers need to extend this class.

```
/**
 * This is the abstract class that must be extended by any service implementation.
 * It includes the implementations of core functionalities required for controlling
 * services such as getters of some methods and service registration and unregistering.
 * @author Cuneyt Caliskan
 */
public abstract class AService implements IService {
    /**
     * Time in milliseconds to sleep when the replica is suspended.
     */
    protected static final long REPLICA_SLEEP_INTERVAL = 1000;
    /**
     * Used to keep the registration of this service in order to use later for
     * unregistering this service.
     */
    private ServiceRegistration<?> serviceRegistration;
    /**
     * Bundle context used for OSGi specific operations such as registering
     * and unregistering.
     */
    private BundleContext context;
    /**
     * State of this service. Can be one of {@value Status#READY},
     * {@value Status#RUNNING} or {@value Status#DEAD}.
     */
    private volatile Status state;
    /**
     * Indicates if this service is the primary replica or a backup replica.
     * A value of <code>true</code> means it is the primary replica.
```

```
 */
private boolean active = false;
/**
 * Holds the bundle name in the form of bundleSymbolicaName_bundleVersion
 * which is a unique identifier in an OSGi framework.
 */
private String bundleName;
/**
 * Allocated amount of memory in bytes.
 */
private long memoryAllocated = 0;
/**
 * Connector to the knowledge agent.
 */
private Connector connector;
/**
 * The address that the {@link #connector} is registered.
 */
private String clientAddress;
/**
 * User id of the {@link #connector}.
 */
private String clientId;
/**
 * Logger for this class instance.
 */
private Logger logger;

/**
 * Constructs a new service by initiating the state to {@link Status#READY}
 * and connecting to the knowledge agent.
 * @param bundleContext bundle context
 */
public AService(final BundleContext bundleContext) {
    super();
    this.context = bundleContext;
    logger = Logger.getLogger(getClass());
    bundleName = bundleContext.getBundle().getSymbolicName() + "_"
            + bundleContext.getBundle().getVersion();
    state = Status.READY;
    active = false;
    initConnector();
    addPermissions();
}
```

```
/**
 * Tries to connect to the local agent instance by initiating the
 * {@link #connector}.
 */
private void initConnector() {
    try {
        connector = new Connector(context);
        clientId = connector.registerClient();
        if (clientId.isEmpty()) {
            logger.error("Could not register the service with the local agent.");
            throw new RuntimeException("Could not register the service" +
"with the local agent.");
        }
        clientAddress = connector.getKORSubtree();
    } catch (AgentCommunicationException e) {
        throw new RuntimeException(
                "Error while creating Connector, unable to register to " +
"local agent:" + e.getMessage());
    } catch (AgentErrorException e) {
        throw new RuntimeException(
                "Error while creating Connector, unable to register to " +
"local agent:" + e.getMessage());
    }
}

private void addPermissions() {
    try {
        connector.addReaderToSubTree(clientAddress, "*");
        connector.addWriterToSubTree(clientAddress, "*");
    } catch (AgentCommunicationException e) {
        logger.error(e.getMessage());
    } catch (AgentErrorException e) {
        logger.error(e.getMessage());
    }
}

@Override
public abstract void run();

@Override
public final void start() {
    registerService();
    Thread t = new Thread(this);
```

```
    t.start();
}

@Override
public abstract void resume();

@Override
public abstract void stop();

@Override
public abstract void suspend();

@Override
public final String getBundleName() {
    return bundleName;
}

@Override
public final long getLastModified() {
    return context.getBundle().getLastModified();
}

@Override
public final boolean isActive() {
    return active;
}

@Override
public final boolean isAlive() {
    return !state.equals(Status.DEAD);
}

@Override
public final long getMemoryUsage() {
    return memoryAllocated;
}

@Override
public final void addMemoryUsage(final long amount) {
    memoryAllocated += amount;
}

/**
 * Registers this service as {@link IService} instance to the OSGi registry.
```

```
 */
public final void registerService() {
    serviceRegistration = context.registerService(new String[] {
        IService.class.getName()
    }, this, null);
}

/**
 * Unregisters the service if it has been successfully registered previously.
 * Also unregisters the {@link #connector} and closes the connection with the
 * local knowledge agent.
 */
public final void unregisterService() {
    try {
        if (serviceRegistration != null) {
            serviceRegistration.unregister();
            serviceRegistration = null;
        }
        if (connector != null) {
            connector.removeKnowledge(clientAddress, clientId);
            connector.unregisterClient();
            connector.shutdown();
        }
    } catch (AgentErrorException e) {
        logger.warn(e.getMessage());
    } catch (AgentCommunicationException e) {
        logger.warn(e.getMessage());
    }
}

/**
 * Gets the sleep interval in case of suspended service.
 * @return the interval in milliseconds
 */
public static long getReplicaSleepInterval() {
    return REPLICA_SLEEP_INTERVAL;
}

/**
 * Gets the bundle context.
 * @return the bundle context
 */
public final BundleContext getContext() {
    return context;
```

```
}

/**
 * Gets the state of this service.
 * @return the state as one of the possible entries in {@link Status}
 */
public final Status getState() {
    return state;
}

/**
 * Gets the amount of memory allocated by this service.
 * @return memory in bytes
 */
public final long getMemoryAllocated() {
    return memoryAllocated;
}

/**
 * Gets the connector instance.
 * @return the connector
 */
public final Connector getConnector() {
    return connector;
}

/**
 * Gets the client address that the connector is registered to
 * the knowledge agent.
 * @return the address
 */
public final String getClientAddress() {
    return clientAddress;
}

/**
 * Gets the user id of the connector.
 * @return the id
 */
public final String getClientId() {
    return clientId;
}

/**
```

```
     * Sets the new value of the client address.
     * @param address new address
     */
    public final void setClientAddress(final String address) {
        this.clientAddress = address;
    }

    /**
     * Sets the new state of the service.
     * @param s new state
     */
    public final void setState(final Status s) {
        this.state = s;
    }

    /**
     * Sets the state as active or backup replica.
     * @param act new state
     */
    public final void setActive(final boolean act) {
        this.active = act;
    }

    /**
     * Sets the bundle name.
     * @param name bundle name
     */
    public final void setBundleName(final String name) {
        this.bundleName = name;
    }

    /**
     * Sets the amount of memory allocated by this service.
     * @param mamory memory amount
     */
    public final void setMemoryAllocated(final long mamory) {
        this.memoryAllocated = mamory;
    }

}
```

Appendix D

Sample Classes

D.1 Sample Notification Callback

PingSubscription class is the implementation in RTE for receiving notifications about the address where to send ping messages. Whenever an RTE receives a notification, it first tries to shutdown ping threads if they are alive. If the value is empty, threads are shutdown and no more action is taken because Smyrna sets this value in purpose to notify them that it is leaving the system and there is no need to send ping messages any more. If the value is not empty, a new thread for sending ping messages is created.

```
* Subscription for the ping address that is set by Smyrna.
 * @author Cuneyt Caliskan
 */
public class PingSubscription implements ISubscriber {
    /**
     * Connector instance to communicate with the local agent.
     */
    private final Connector connector;
    /**
     * Interval of sending ping messages.
     */
    private final long pingInterval;
    /**
     * Address of the RTE in the KOR which is set in the callback address of Smyrna.
     */
    private final String myAddress;
    /**
     * Thread that sends the ping messages periodically.
     */
    private EPFDPing pinger;
    /**
```

```
     * Logger instance for this class.
     */
    private final Logger logger;
    /**
     * Executor for the {@link #pinger}.
     */
    private Thread executor;
    /**
     * My bundle name.
     */
    private final String myName;

    /**
     * Subscription for the address where to send ping messages which is set by
     * Smyrna.When the address changes, the previous ping thread is stopped and
     * a new instance is started for the new address.
     * @param conn connector instance.
     * @param interval ping sending interval.
     * @param clientAddress my address in the KOR.
     * @param bundleName my bundle name.
     */
    public PingSubscription(Connector conn, long interval, String clientAddress,
    String bundleName) {
        super();
        connector = conn;
        pingInterval = interval;
        myAddress = clientAddress;
        myName = bundleName;
        logger = Logger.getLogger(getClass());

    }

    @Override
    public final void notificationCallback(String changedAddress) {

        try {
            String pingAddress = connector.get(changedAddress + "/value",
      connector.getId());
            shutDown();
            if (pingAddress.isEmpty()) {
                return;
            }
            pinger = new EPFDPing(pingAddress, connector, pingInterval,
myAddress, myName);
```

```
            executor = new Thread(pinger);
            executor.setName("EPFD Ping Thread");
            executor.start();
            logger.info("#PING# thread started for ping address " + pingAddress);
        } catch (AgentCommunicationException e) {
            logger.warn(e.getMessage());
        } catch (AgentErrorException e) {
            logger.warn(e.getMessage());
        }
    }

    /**
     * Shutdowns the pinger thread if it is alive.
     */
    public final void shutDown() {
        if (executor != null) {
            if (executor.isAlive()) {
                pinger.shutDown();
                executor.interrupt();
            }
        }
    }

}
```

D.2 Decision Function

DecisionFunction class is one of the most important classes in Smyrna implementation which is responsible for deciding where to migrate a service, where to place a replica, when to migrate a service and so on. It includes methods for determining the worst RTE in the system, service to be migrated, best RTE for a given service and necessary methods for calculating utility values.

```
public class DecisionFunction {
    /**
     * Connector instance used for communication with the local agent instance.
     */
    private final Connector connector;
    /**
     * Client ID used in the communication via {@link #connector}.
     */
    private final String clientId;
    /**
     * My address in the KOR.
```

```
 */
private final String servicesAddress;
/**
 * This parameter is required to make the first decision about a RTE if it shows
 * a decreasing or increasing characteristic. This value indicates what
 * percentage of RTEs utility history to be full in order to make a healthy
 * decision.
 */
private final double nonNullPercentage;
/**
 * Set of RTE entries in the system.
 */
private volatile ConcurrentMap<String, RTEEntry> rteEntries;
/**
 * Logger instance for this class.
 */
private final Logger logger;
/**
 * Migration threshold is a design parameter which indicates the fraction of
 * utility of a selected {@link #getWorstRTE() worst} RTE to a selected
 * {@link #getBestRTE(List) best} RTEs utility. If the utility of the worst RTE
 * is {@link #migrationThreshold} less or equal to best  RTEs utility, then
 * a migration takes place. It means that the selected best RTE must be at
 * least %10 better than the selected worst RTE. <p>
 * calculateUtility(worst)/calculateUtility(best) <=migrationThreshold
 * <b>then</b> migrate the selected candidate.
 */
private final double migrationThreshold = 0.9;

/**
 * Constructs a decision function that makes decisions about controlling the
 * services in the smart environment.
 * @param conn the connector used for communication with the local agent.
 * @param srvcsAddress address of the services stored in smyrna KOR
 * @param nonNullPer required percentage of utility values to make decisions.
 */
public DecisionFunction(final Connector conn, final String srvcsAddress,
        final double nonNullPer, ConcurrentMap<String, RTEEntry> rteMap) {
    super();
    connector = conn;
    clientId = connector.getId();
    servicesAddress = srvcsAddress;
    nonNullPercentage = nonNullPer;
    rteEntries = rteMap;
```

```
        logger = Logger.getLogger(getClass());
    }

    /**
     * Creates a replica of the given address in the best RTE. First, replicas of
     * this replica are searched within the KOR. If no replica is found, ends the
     * operation. Then, an appropriate RTE, namely the best RTE is selected to place
     * a replica by excluding the replica locations. If there is no such best RTE,
     * operation ends. Otherwise, replication signal is sent to the selected
     * destination.
     * @param serviceAddress address within the SMYRNA-KOR.
     */
    public final String replicate(String serviceAddress,
  List<String> excludeLocations) {
        String best = null;
        try {
            String bundleName = connector.get(serviceAddress + "/bundleName/value",
      clientId);
            List<String> replicaLocations = getServiceLocations(bundleName);
            if (excludeLocations != null) {
                for (String repLoc : replicaLocations) {
                    String rte = repLoc.substring(0, repLoc.lastIndexOf('/'));
                    rte = rte.substring(0, rte.lastIndexOf('/'));
                    if (!excludeLocations.contains(rte)) {
                        excludeLocations.add(repLoc);
                    }
                }
            }
            best = getBestRTE(excludeLocations);
            if (best == null) {
                logger.warn("Cannot locate a replica for service @ " +
    serviceAddress);
                return null;
            } else {
                connector.set(serviceAddress + "/replicate", best, clientId);
                logger.info("Replicating replica @ " + serviceAddress + " to " +
    best);
            }
        } catch (AgentCommunicationException e) {
            logger.warn(e.getMessage());
        } catch (AgentErrorException e) {
            logger.warn(e.getMessage());
        }
        return best;
```

```
    }

    /**
     * Determines a list of migrations that need to be performed by calculating
     * the worst and best RTEs. It might also result in a no operation if no
     * migration is necessary.
     * @param rteServices runtime entries including addresses and utility histories.
     * @param nonNullPercentage required percentage of utility values calculated
     * to make decisions.
     * @return {@link Operation operation} of type {@link OpType#MIGRATE} if any
     * migration decision is made, otherwise operation of type {@link OpType#NOOP}.
     */
    public final Operation migration() {
        Operation result = new Operation(OpType.NOOP);
        List<String> candidateHistory = new LinkedList<String>();
        String worst = getWorstRTE();
        boolean done = false;
        if (worst == null) {
            logger.debug("Worst RTE not selected.");
        } else {
            logger.debug(worst + " has been selected as the worst RTE.");
            while (!done) {
                String servicePath = getMigrationCandidate(worst, true,
  candidateHistory);
                if (servicePath == null) {
                    logger.debug("No active service has been selected from the worst
 RTE. Trying to select backup replica.");
                    servicePath = getMigrationCandidate(worst, false,
candidateHistory);
                }
                if (servicePath == null) {
                    logger.debug("No replica has been selected from the worst RTE.");
                    done = true;
                } else {
                    candidateHistory.add(servicePath);
                    logger.debug(servicePath + " is the candidate for migration.");
                    String serviceName = servicePath.substring(
servicePath.lastIndexOf("/") + 1);
                    List<String> excluded = getServiceLocations(serviceName);
                    String best = getBestRTE(excluded);
                    if (best == null) {
                        logger.debug("Best RTE not selected.");
                        done = false;
                    } else {
```

```
                        logger.debug(best + " is the candidate  destination to
    migrate " + serviceName);
                        double worstUtility = calculateUtility(
      rteEntries.get(worst));
                        double bestUtility = calculateUtility(
      rteEntries.get(best));
                        if (worstUtility / bestUtility <= migrationThreshold) {
                            result = new Operation(OpType.MIGRATE, servicePath,
    worst, best);
                            done = true;
                        } else {
                            logger.debug("There is no significant difference
between worst[" + worstUtility + "] and best[" + bestUtility
                                    + "] RTE utilities. Cancelling migraton...");
                            done = false;
                        }
                    }
                }
            }
        }
        return result;
    }

    /**
     * Selects the worst RTE among the given set of entries. The calculation is done
     * by fitting the utility values in a line by and getting its slope calculated
     * by least squares estimate. The slope contributes the 50% of the result, the
     * other half is contributed by the average utility value of the history.
     * @return the address of the worst RTE among the given entries or null if none
     * of the entries satisfies the given non null percentage
     */
    public final String getWorstRTE() {

        String worst = null;
        Double min = Double.POSITIVE_INFINITY;

        for (Map.Entry<String, RTEEntry> entry : rteEntries.entrySet()) {
            RTEEntry rte = entry.getValue();
            double utility = calculateUtility(rte);
            if (utility == 0) {
                continue;
            }
            if (min > utility) {
                min = utility;
```

```
            worst = rte.getRteAddress();
        }
    }
    return worst;
}

/**
 * Selects the best RTE among the given set of entries. The calculation is done
 * by fitting the utility values in a line by and getting its slope calculated
 * by least squares estimate. The slope contributes the 50% of the result, the
 * other half is contributed by the average utility value of the history.
 * @param rteEntries data structure containing the runtime environments and
 * their utilities
 * @param excludedAddresses list of RTEs to be excluded
 * @return the address of the best RTE among the given entries or {@code null}
 * if none of the entries satisfies the given non null percentage
 */
public final String getBestRTE(List<String> excludedAddresses) {
    String best = null;
    Double max = Double.NEGATIVE_INFINITY;

    for (Map.Entry<String, RTEEntry> entry : rteEntries.entrySet()) {
        RTEEntry rte = entry.getValue();
        if (excludedAddresses != null && excludedAddresses.contains(
rte.getRteAddress())) {
            continue;
        }
        double utility = calculateUtility(rte);
        if (utility == 0) {
            continue;
        }
        if (max <= utility) {
            max = utility;
            best = rte.getRteAddress();
        }
    }
    return best;
}

/**
 * Searches for the oldest and most memory consuming service in the given
 * runtime environment that is an active replica or not depending on
 * the parameter.
 * @param rteAddress address of the RTE to search for services
```

```
     * @param activeRequired indicates whether to consider the primary replicas
     * only or not.
     * @param excludedServices list of services to be excluded.
     * @return address of the oldest service in the given RTE address or null if
     * no service is found.
     */
    public final String getMigrationCandidate(String rteAddress,
boolean activeRequired, List<String> excludedServices) {
        String service = null;
        try {
            List<String> services = connector.getNodesOfType(rteAddress,
    "smyrna/service", clientId);
            if (services == null || services.isEmpty()) {
                return null;
            }
            services.removeAll(excludedServices);
            long currentTime = System.currentTimeMillis();
            long jvm = Runtime.getRuntime().totalMemory();
            double max = Double.NEGATIVE_INFINITY;
            for (String s : services) {
                boolean isActive = true;
                double l_i = 0;
                double m_i = 0;
                double S_i = 0;
                if (activeRequired) {
                    isActive = Boolean
                            .parseBoolean(connector.get(s + "/isActive/value",
  clientId));
                }
                if (isActive) {
                    long sLastModified = Long.parseLong(connector.get(s +
"/lastModified/value", clientId));
                    long sMemory = Long.parseLong(connector
                            .get(s + "/memoryUsage/value", clientId));

                    l_i = 1 - (sLastModified / currentTime);
                    sMemory = sMemory / jvm;
                    S_i = (l_i + m_i) / 2;
                    if (S_i > max) {
                        max = S_i;;
                        service = s;
                    }
                }
            }
```

```
        } catch (AgentCommunicationException e) {
            logger.warn(e.getMessage());
        } catch (AgentErrorException e) {
            logger.warn(e.getMessage());
        }
        return service;
    }

    /**
     * Searches for the replica locations of the given service.
     * @param serviceName the service of which replica locations to be searched.
     * @return list of locations or null if not found.
     */
    public List<String> getServiceLocations(String serviceName) {
        try {
            List<String> serviceLocations = connector.getNodesOfType(servicesAddress,
                    "smyrna/replica", clientId);
            if (serviceLocations == null || serviceLocations.isEmpty()) {
                return null;
            }
            List<String> result = new LinkedList<String>();
            for (String service : serviceLocations) {
                String bundleName = connector.get(service + "/bundleName/value",
clientId);
                if (serviceName.equals(bundleName)) {
                    String location = connector.get(service + "/location/value",
clientId);
                    location = location.substring(0, location.lastIndexOf('/'));
                    location = location.substring(0, location.lastIndexOf('/'));
                    result.add(location);
                }
            }
            return result;
        } catch (AgentCommunicationException e) {
            logger.warn(e.getMessage());
        } catch (AgentErrorException e) {
            logger.warn(e.getMessage());
        }
        return null;
    }

    /**
     * Returns the array of non null utility values of the given utility array
     * @param utilities data set to be examined
```

```
     * @return array of non null entries
     */
    public final Double[] getNonNullUtilities(Double[] utilities) {
        Double[] result = null;
        int dim = getNonNullUtilityCount(utilities);
        if (dim == 0)
            return null;
        result = new Double[dim];
        int index = 0;
        for (int i = 0; i < utilities.length; i++) {
            if (utilities[i] != null) {
                result[index] = utilities[i];
                index++;
            }
        }
        return result;
    }

    /**
     * Calculates the number of increments in the sequence of utility values
     * @param utilities data set to be examined
     * @return number of changes from low utility to a high utility
     */
    public final int countUps(Double[] utilities) {
        int ups = 0;
        double prev = 0;

        for (int i = 0; i < utilities.length; i++) {
            if (utilities[i] == null) {
                return ups;
            }

            if (utilities[i] > prev) {
                ups++;
            }
            prev = utilities[i];
        }
        return ups;
    }

    /**
     * Calculates the number of decrements in the sequence of utility values
     * @param utilities data set to be examined
     * @return number of changes from high utility to a low utility
```

```
 */
public final int countDowns(Double[] utilities) {
    int ups = countUps(utilities);
    return getNonNullUtilityCount(utilities) - ups;
}

/**
 * Calculates the average utility value of the given utility history
 * @param utilities data set of which the arithmetic average to be calculated
 * @return arithmetic average of the given data set
 */
public final double getAverageUtility(Double[] utilities) {
    int nonNull = getNonNullUtilityCount(utilities);
    if (nonNull == 0)
        return 0;

    return totalUtility(utilities) / nonNull;
}

/**
 * Calculates the total utility sum of the given array.
 * @param utilities data set of which the sum to be calculated
 * @return aggregated sum of the non null entries in the data set
 */
public final double totalUtility(Double[] utilities) {
    double total = 0;
    for (int i = 0; i < utilities.length; i++) {
        if (utilities[i] != null) {
            total += utilities[i];
        } else
            break;
    }
    return total;
}

/**
 * Returns the percentage of the non null entries in the given utility array
 * @param utilities array of data to be examined
 * @return percentage of the non null entries in the history
 */
public final double getNonNullPercentage(Double[] utilities) {
    double result = getNonNullUtilityCount(utilities);
    return result / utilities.length;
}
```

```
/**
 * Returns the non null entries in the given array.
 * @param utilities the array which to be examined
 * @return array of non null utility values
 */
public final int getNonNullUtilityCount(Double[] utilities) {
    int result = 0;
    for (int i = 0; i < utilities.length; i++) {
        if (utilities[i] != null)
            result++;
        else
            break;
    }
    return result;
}

/**
 * Calculates the average utility values of given entries and finds the maximum
 * average utility.
 * @return maximum average utility value or 0 if there is no utility value in
 * the given entry map.
 */
public final double getMaxAverageUtility() {
    double max = 0;
    for (Map.Entry<String, RTEEntry> entry : rteEntries.entrySet()) {
        double avg = getAverageUtility(entry.getValue().getUtilityHistory());
        if (avg > max) {
            max = avg;
        }
    }
    return max;
}

/**
 * Calculates the utility value for decision making for the given rte.
 * @param rte the RTE to calculate the utility of.
 * @return utility of the given rte or 0 if non-null percentage is not satisfied
 * or maximum average utility is 0.
 */
public final double calculateUtility(RTEEntry rte) {
    if (getNonNullPercentage(rte.getUtilityHistory()) >= nonNullPercentage) {
        Double[] data = getNonNullUtilities(rte.getUtilityHistory());
        SimpleRegression sr = new SimpleRegression();
```

```
            for (int i = 0; i < data.length; i++) {
                sr.addData(i, data[i]);
            }
            double maxAvg = getMaxAverageUtility();
            if (maxAvg == 0) {
                return 0;
            }
            double utility = getAverageUtility(rte.getUtilityHistory()) / maxAvg;
            double slope = sr.getSlope();
            slope = Math.max(-1, slope);
            slope = Math.min(1, slope);
            utility += sr.getSlope();
            return utility / 2;
        } else {
            return 0;
        }
    }
}
```

TRITA-ICT-EX-2012:266

www.kth.se