TECHNISCHE UNIVERSITÄT MÜNCHEN

FAKULTÄT FÜR INFORMATIK

Bachelorarbeit in Wirtschaftsinformatik

Automated Fault Recovery Planning in Cloud Computing

Pavlo Kerestey

Technische Universität München

Fakultät für Informatik

Bachelorarbeit in Wirtschaftsinformatik

Automated Fault Recovery Planning in Cloud Computing

Automatisierte Planung der Fehlerbehebung in Cloud Computing

Author:	Pavlo Kerestey
Supervisor:	Prof. Dr. Heinz-Gerd Hegering
Advisor:	Dipl.-Inf. Liu Feng
	Dipl.-Ing. Johannes Watzl
Date:	February 15, 2010

I assure the single handed composition of this bachelor thesis only supported by declared resources.

Garching, den 15. Februar 2010 Pavlo Kerestey

Abstract

This work investigates the applicability of the automated planning approaches to fault management in cloud computing implementations on the infrastructure as a service level. A decision support solution for the fault management in cloud computing is examined to identify the possibility of the automation of fault recovery in large scale cloud computing deployments.

Cloud computing is a fairly new topic with increased industrial interest. Cloud computing services are popular due to their flexible resource allocation and optimal economic usage. This allows to avoid under- and over-utilization of the computing resources and makes planning and management less cost-intensive task. At present, no good cloud computing management solution for fault recovery exists, which makes cloud computing services unattractive to many potential users. As mistakes do happen in every system it must be possible for a cloud service provider to guarantee that the terms of provisioning will not be breached even when faults happen. This can be achieved by automating error-prone and time-consuming tasks. Therefore the aim of the fault recovery solution examined in this work is the time minimization of complete service recovery.

To diminish the problem, an automated planning approach in the field of artificial intelligence is chosen as a solution. In addition, this work is based on operation research studies. The aim is to create a prototype of a decision support solution, which will help to lessen the complexity of fault recovery and also the expenses for the whole fault management. A system and its services should recover from different kinds of faults using fast and a systematic composition of recovery plans. A scenario will be created in cooperation with internet and computing provider Global Access GmbH and cloud computing provider Zimory GmbH to prove the usefulness of the solution. The aim is a machine aided improvement of IT service availability.

This work explores existing approaches of automated planning and uses planning applications in grid computing. It targets the analysis of the applicability of automated planning approaches for the fault management in cloud computing. An automated planning algorithm is examined and a prototype is implemented for a scenario to prove that functionality of the planning system is given.

Contents

1. Introduction

1.1. Cloud Computing

The definition of cloud computing changes rapidly. New concepts are often called cloud computing and there is no commonly agreed definition of cloud computing at the moment. The ACM SIGCOMM article "A Break in Clouds: Towards a cloud definition" provides the most exact definition of this term which is also used in this work:

> "Clouds are a large pool of easily usable and accessible virtualized resources (such as hardware, development platforms and/or services). These resources can be dynamically reconfigured to adjust to a variable load (scale), allowing also for an optimum resource utilization. This pool of resources is typically exploited by a pay-per-use model in which guarantees are offered by the Infrastructure Provider by means of customized SLAs." [19]

There are three levels of abstraction in cloud computing such as infrastructure as a service, platform as a service and software as a service, which can be grouped together or examined as standalone paradigms. To provide an overview each of them is described separately.

1.1.1. Infrastructure as a Service

Infrastructure as a Service (IaaS) is the lowest level of abstraction that is provided as a service (figure 1.1). Infrastructure services are used to built higher level services, or bundled together by service providers.

> Infrastructure providers manage a large set of computing resources, such as storage and processing capacity. Through virtualization, they are able to split, assign and dynamically resize these resources to build ad-hoc systems as demanded by customers. They deploy the software stacks that run their services. This is the Infrastructure as a Service (IaaS) scenario[19].

The typical service is a provision of an environment for so called virtual machines[1]. An instance of such a virtual machine can host an operating system and is therefore a good instrument for building a customized infrastructure upon it. The economical difference

[1]A virtual machine is a tightly isolated software container that can run its own operating systems and applications as if it was a physical computer. A virtual machine behaves exactly like a physical computer with its own virtual (i.e., software-based) CPU, RAM hard disk and network interface card (NIC).[9].

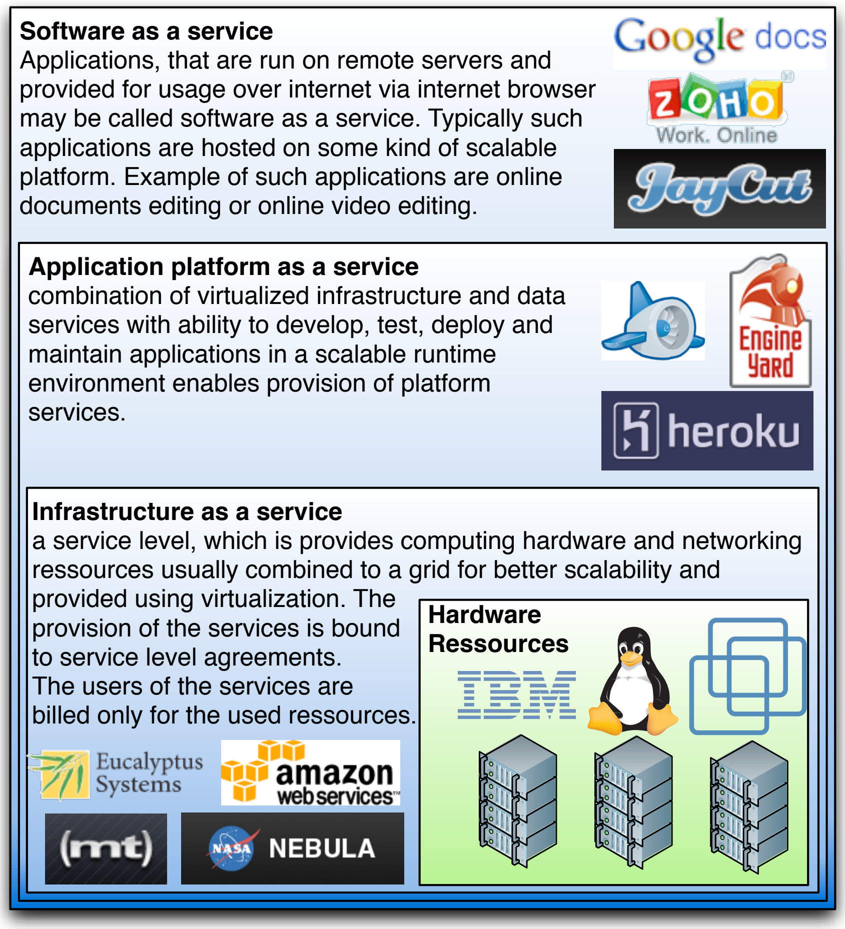

Software as a service
Applications, that are run on remote servers and provided for usage over internet via internet browser may be called software as a service. Typically such applications are hosted on some kind of scalable platform. Example of such applications are online documents editing or online video editing.
Google docs
ZOHO Work. Online
JayCut
Application platform as a service
combination of virtualized infrastructure and data services with ability to develop, test, deploy and maintain applications in a scalable runtime environment enables provision of platform services.
Engine Yard
heroku
Infrastructure as a service
a service level, which is provides computing hardware and networking ressources usually combined to a grid for better scalability and provided using virtualization. The provision of the services is bound to service level agreements.
The users of the services are billed only for the used ressources.
Eucalyptus Systems
amazon webservices
(mt)
NASA NEBULA
Hardware Ressources
IBM

Figure 1.1.: This image shows the hierarchy of three levels of cloud computing services - software as a service, platform as a service and infrastructure as a service.

between the usage of an ordinary infrastructure and the virtual infrastructure in the cloud is that the payment is made only for the used resources.

Well-known IaaS Providers are 3Tera with the AppLogic Grid OS [1], Amazon with Elastic Compute Cloud [2], and RightScale [6]. Software solutions which are used to create a cloud computing infrastructure VMWare vCloud [8], Eucalyptus open source and enterprise editions [3], and Xen Cloud Platform [10] can be pointed out as some well-known examples.

1.1.2. Platform as a Service

Some applications can be grouped together by common execution environment properties. Performance of such applications can be optimized by building a unified platform, which would run the applications. This platform can then be optimized to handle such performance bottlenecks as a database access, limited CPU or memory resources. Instead of supplying a virtualized infrastructure, providers can offer a software platform where applications run on. The sizing of the hardware resources demanded by the execution of the services is made in a transparent manner. This is denoted as Platform as a Service (PaaS) figure 1.1. [19]

To optimize the maintenance of such platforms, they are usually built by using flexible infrastructure. Usage of IaaS for this cause provides all benefits of virtual infrastructure like flexible resource allocation or scalable resource usage. Creating infrastructure for such platforms is the main topic of this work, therefore existing examples are examined.

A well-known platform service is the Google App Engine. It is a platform for python and java applications. These applications run inside dedicated environments consisting of runtime environments and a database. The environments are also capable of resource scaling to specific needs.

Another example of PaaS is Heroku. It provides a runtime environment to run Ruby applications. An overview of their platform is depicted on figure 1.2.

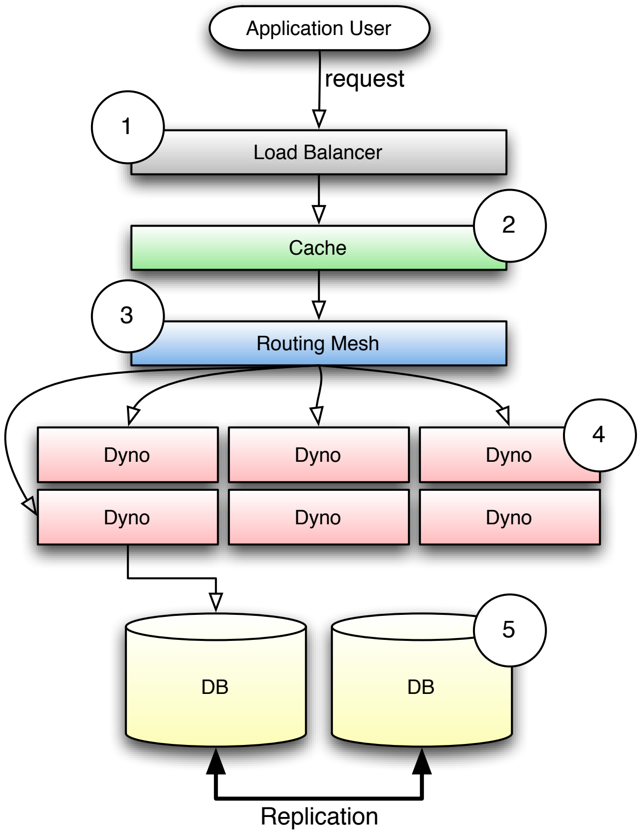


Figure 1.2.: Heroku platform architecture. [4]

Platforms are built to optimize the performance of a runtime environment for applications. Heroku is built specifically for web applications which deal with users requesting applications' content. It uses different machines to handle these requests during different phases of their processing. The first phase of a request is to find out by which application should the request be processed. This is handled by servers which receive these requests first. (figure 1.2 1), managing DNS, load balancing and fail-over scenarios. This layer is tuned for high performance and handles only requests, encryption and compression.

The next phase in request processing is querying the cache (figure 1.2 2). Every request passes through it and if requested content is available in the cache, which results in cache providing the content immediately to the user. Such a request is thus never processed by an application.

In order to run the application, Heroku uses dynos (figure 1.2 4). These are runtime environments for applications and are created as delimited processes. They are spread across a grid and an application maintainer may use several to increase performance of her or his application. The number of used dynos can be increased or decreased, it takes 2 seconds to start one.

Before being processed by a dyno, a request routing is handled by a routing mesh (figure 1.2 3) which enables load balancing of the requests between dynos that belong to the same application. In case a dyno becomes unstable a new dyno is launched. The routing mesh will then route all requests to the new dyno.

Heroku provides a SQL database for every application to store persistent data (figure 1.2 5). It is replicated to provide better access performance and stability. The application maintainer is not forced to use it. A database hosted by another provider can be used as well.

PaaS uses different machines to perform certain tasks and these machines can be dynamically deployed in great numbers to handle higher loads. To avoid high costs of underutilization of the infrastructure these machines are running on, they must be constantly monitored. Idle machines must be detected and stopped and faulty or hung-up machines should be replaced. In case the system receives higher load, new machines must be deployed to handle this load. Using virtual infrastructure with an automated management system would make such platforms maintainable.

1.1.3. Software as a Service

There are services hosted in cloud systems which are of potential interest to a wide variety of users. These services are alternatives to local run applications such as word processors, picture viewers and video players to name a few. Provision of such applications is called Software as a Service (SaaS) figure 1.1. [19]

Many modern web applications are forms of SaaS. They do not need to be installed, are configurable to personal needs of the users and are provided over internet. Most of such applications are built similarly and can therefore be deployed on an application platform. This allows to take benefit of all the qualities of a platform including flexible scalability and stability.

1.2. Automated Planning

The management of large scale infrastructures will at some point require that certain tasks will be automated. The aim of this work is to find a solution for automating the search for actions which have to be executed to recover a faulty infrastructure. This task will be achieved by examining automated planning.

> Planning is the reasoning side of acting. It is an abstract, explicit deliberation process that chooses and organizes actions by anticipating their expected outcomes. This deliberation aims at achieving as best as possible some pre-stated objectives. Automated planning is an area of Artificial Intelligence (AI) that studies this deliberation process computationally.[16]

A motivation for automated planning is designing information processing tools that give access to planning resources. For example an operation of recovering from an outage in a large infrastructure may involve a large number of actors. Deployment of the right amount of machines in the right environment and managing the availability of services at the same time relies on careful planning. This is time constrained and it demands immediate decisions which must be supported by a planning tool. A planning resource that is seamlessly integrated with management tools used by a system administrator could be of great support in handling constraints and offering alternate recovery plans not yet considered. The system is able to point out critical actions and show constraints that are needed to be relaxed to achieve a best possible solution.

Since there are various types of actions in general, there are various forms of planning as well. Examples include path and motion planning, perception planning and information gathering, navigation planning, manipulation planning, communication planning, and several other forms of social and economic planning. Certain aspects of these planning forms will be used to create a planning domain useful in this work.

1.2.1. Types of automated planning

Path and motion planning is concerned with the synthesis of a geometric path from a starting position in space to a goal and of a control trajectory along that path that specifies the state variables in the configuration space of a mobile system, such as a truck, a mechanical arm, a robot, or a virtual character.

Perception planning is concerned with plans involving sensing actions for gathering information. It arises in tasks such as modeling or identifying environments and objects, localizing through sensing a mobile system, or more generally identifying the current state of the environment. Data gathering is a particular form of perception planning which is concerned not with sensing but instead with querying a system: e.g., testing a faulty device in diagnosis or searching databases distributed over a network. The issues are which queries to send where and in what order.

Navigation planning combines the two previous problems of motion and perception planning in order to reach a goal or to explore an area. The purpose of navigation planning is to synthesize a policy that combines localization primitives and sensor-based motion primitives, e.g., visually following a road until reaching some landmark, moving along some heading while avoiding obstacles, and so forth.

Manipulation planning is concerned with handling objects, e.g., to build assemblies. A plan might involve picking up an object from its marked sides, returning it if needed, inserting it into an assembly, and pushing lightly till it clips mechanically into position.

Communication planning arises in dialog and in cooperation problems between several agents, human or artificial. It addresses issues such as when and how to query needed information and which feedback should be provided.

1.2.2. Representations of planning

To understand algorithms solving automated planning problems, we examine three different ways to represent them.

In a *set-theoretic representation*, each state of the world is a set of propositions, and each action is a syntactic expression specifying which propositions belong to the state in order for the action to be applicable and which propositions the action will add or remove in order to make a new state of the world.[16]

In a *classical representation*, the states and actions are like the ones described for set-theoretic representations except that first-order literals and logical connectives are used instead of propositions. This is the most popular choice for restricted state-transition systems.[16]

In a *state-variable representation*, each state is represented by a tuple of values of n state variables $(X1, ..., Xn)$, and each action is represented by a partial function that maps this tuple into another tuple of values of the n state variables. This approach is especially useful for representing domains in which a state is a set of attributes that range over finite domains and whose values change over time.[16]

In our solution attempt we will use the state-variable representation adopted to the planning domain.

2. Scenario

In this chapter a scenario based on PaaS is provided, which serves as a basis for further discussions. According to the presented scenario the research problem is derived. First a cloud computing infrastructure is examined which allows to provide a platform for development, deployment and maintenance of different distributed applications. We then identify the problems regarding stability and maintenance which arise in such infrastructure. In the end requirements for a solution of the problem are defined.

2.1. Model

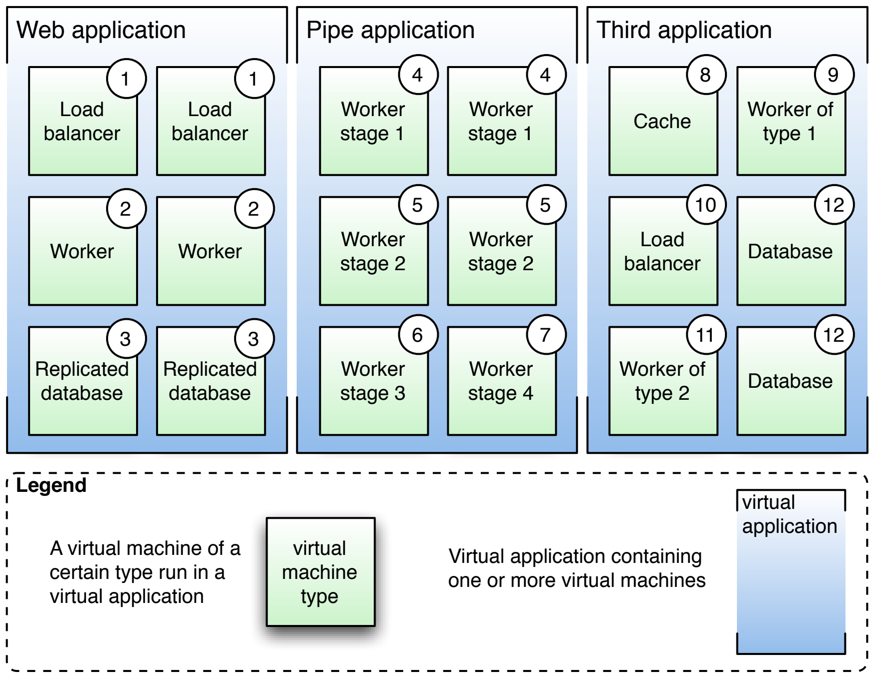


Figure 2.1.: A schematic overview of a virtual infrastructure, which hosts several virtual machines represented by squares. Virtual machines are grouped into three virtual applications: web application, pipe application and some third application

As shown on figure 2.1 we examine an infrastructure, which allows the deployment of different distributed applications with several virtual machines. To avoid cluttering of the overview, only 6 virtual machines per virtual application are shown, although the number of virtual machines per application is not limited to this amount. The depicted scheme shows different layers where the virtual machines can be placed.

The main purpose of a platform is to serve an application as stable as possible. When an application user requests computation or data retrieval, the virtual machines will have to evenly distribute the load of such requests. In a web application such requests will be handled by a load balancer first (fig 2.1 1). This request will then be routed to a least loaded worker instance (fig 2.1 3), and the latter will processes the request. It is likely that the worker instance has to access persistent data stored in a database to retrieve data or to perform computations (fig 2.1 3, 12).

Different applications may have several types of machines. These applications can be built using different architectures. They may for instance have certain types of data to deal with. Data, which should be stored securely and its loss must be avoided at all costs, have to be replicated throughout several database server instances. Big amounts of data, where consistency is not as important as access speed, can be stored in a sharded manner – i.e. split over different database servers and retrieved in parallel.

The infrastructure can also be built on top of different resource providers. They may have different terms of SLA, different capacities as well as different prices. These factors have to be considered when deploying virtual machine instances for various applications.

2.2. Problem Statement

The infrastructure is so complex, that faults may occur on several levels. These faults if not serious, may lead to sluggish response times or in severe cases even to unresponsiveness of the systems and violations of the SLAs'. An example of an outage in Amazon's elastic compute cloud[1] shows that any aspect of the application infrastructure, can cause a major failure if complex architecture is insufficiently planned.

On order to examine faults, which may occur to virtual machine instances and to the infrastructure as a whole, an example of a web application from the scenario (figure 2.1) is examined. The load balancer (figure 2.1 1) is the entry point for every incoming request. If it fails to route the request to the right worker instance (figure 2.1 2), one of the latter has to perform more computation tasks to handle the requests and therefore will be under higher load. The overall performance may therefore deteriorate and if a threshold is reached, the routing failure will lead to an overload of application instances which then results in unresponsiveness.

The applications may also contain software bugs. If such faults are hidden they are difficult to correct. They may lead to unpredictable behaviors and application unresponsiveness and finally deterioration of the overall performance of the system. The problem could be solved by replacing the unhealthy instances with new ones, but for a customer this process can be expensive in terms of time and cost.

The availability of connectivity to the services is an issue as well. It is always possible, that a cloud computing providers has technical difficulties. The virtual machine instances of the affected applications have to be migrated to other resource providers of the same system, whereas each provider supplying different levels of availability and SLA's. Therefore there are several target choices for the migration.

[1]On January 2. 2010, all of the specialized, high-capacity Amazon EC2 instances that run Heroku service became unavailable. Amazon routing device in its Virginia data center experienced failure. The service was recovered in an hour.[7]

The growing scale and the complexity of such infrastructures makes it difficult to manually construct resource allocation plans. Even if such plan can be found, in case of a failure, different users and customers have different requirements on provided resources and system uptime. The problem arises in making deployment decisions, which optimally distribute the resource usage, is a time-expensive management task that requires significant amounts of expertise knowledge. Even if the knowledge is available, it is difficult to foresee and evaluate all possibilities and behaviors of the system and manually plan an optimal provisioning scenario. It is also difficult to deal with constantly changing number of virtual resource providers. This may be the case when the demand for the resources is rapidly changing. The deployment decisions have to be calculated precisely where not only the price, but also the availability and performance of the provided resources has to be taken into account.

In particular, it is even more difficult to react to a fault in a constantly changing infrastructure. If a failure occurs, the mean time of recovery depends on the system administrator's knowledge. If the knowledge is insufficient on a certain level, or there are no experience on particular topic, the recovery process may take more time than needed. Thus leads to a breach of SLA's.

In summary the problem with virtual infrastructure provision can be derived from the following observations:

- A system is complex and error prone.
- A fault of one single component in the system may lead to a ripple effect[2], by which the whole system becomes unstable in an unpredictable amount of time and may eventually lead to unresponsiveness and SLA violation.
- The availability of a system depends on how prompt a service fault is identified and is largely related to expert knowledge of the system's administrator performing system repair. The general formula of calculating availability is given by: $availability = \frac{MTBF}{MTBF+MTTR}$[20] Where *MTBF* is *Mean Time Between Failure* and *MTTR* is *Mean Time To Repair.*
- The dynamic nature and the possible multi organizational deployment of the infrastructure makes it complicated to create an optimal solution which would include interests of all parties - i.e. provider, customer and user.
- The same dynamic nature causes an unpredictable recovery process where the reliability of the recovered system is questionable.
- The knowledge needed for recovery is often spread between different parties or even unavailable. This is critical for decision-making during a fault recovery process.
- The knowledge management is a very time- and cost-intensive task.
- The planning of a fault recovery process in a timely- and cost-effective manner is an complicated management activity.

2.3. Requirement analysis

To effectively provide virtual infrastructure services, a decision support system is needed, which quickly and dynamically determines a failure recovery plan. The system must be aware of the provided resource landscape, the reliability as well as the costs, the used amount of resources by the deployed applications and of the application load.

[2]The ripple effect measures impact, or how likely it is that a change to a particular module may cause problems in the rest of a program.[12]

As mentioned before, the mean time to a system recovery relies largely on human expertise. To provide reliable services, efficiency gains on fast root cause determination and on automatic recovery plan creation as a suggestion for the system administrator or automated actor have to be achieved. This way he would be able to promptly react to a system outage.

To tackle the problem of finding reasonable provision scenario in this environment, an automated decision support system [DSS] can be developed, which will compute the factors of different recovery scenarios.

After a fault event occurred in the given infrastructure, a complete description of what fault happened should be available to the DSS. It must then be able to quickly determine the actions, which must be performed by a system administrator or some automated system to recover from the error in the shortest period of time.

The DSS should include following aspects:

- determination of a reasonable recovery plan for an outage.
- ability to monitor the recovery process executed by an external agent and adopt the proposed recovery plan to the changing requirements.
- saving the knowledge from the recovery into a centralized knowledge repository for reuse.
- ability to retrieve the knowledge in a human readable format, to use, edit and improve it.

There are several reasons to develop a decision support system. The problem of quick communication of the information concerning outages could be targeted by a central knowledge base specifically created for this problem type.

Another reason for a decision support system is because the recoveries from failures are often error prone due to tight time frames. This makes such recovered systems unstable and their latter behavior unpredictable resulting in further outages and even causing instability of the whole system. It is difficult to foresee all complications of an outage and all effects of a recovery given only a small time frame to act.

Additionally a decision support system would be a good provider knowledge management and would perform in predictable amount of time. The more knowledge contributors will fill up the knowledge database, the more reliable the system will prove itself. The algorithms should allow better utilization of existing management knowledge and fast retrieval of the data as well as recovery plan generation for infrastructures of any scale.

3. Automated planning as a solution

3.1. Planning Techniques

The task of coming up with a sequence of actions that will achieve a goal is called planning. [18]

The main part of the required system, is to find a sequence of actions, an actor will execute on a system to recover from failure. Since we face a planning problem, we have to take a look for the right planning algorithm, which would meet our needs. Here, we will discuss the planning techniques and choose the best suited approach of solving the problem.

The simplest planning algorithm, would be solving a problem by using a search-based or logical planning agent. These approaches, though, perform insufficiently, when facing large and complex real-world planning problems[18]. Therefore the topic of this section discusses some improvements and alternative search concepts to the standard search algorithms like $A*$, which would allow to scale up and perform in environments that are beyond the limits of determinism and being fully observable, finite, static (changes happen only when agent act) and discrete. The problem, to be solved is non-static (The system constantly changes it's load independently from the agent). This means that advanced search techniques have to be used.

To use any of the algorithms, some definitions are presented. Every system can be represented using states. A *state* is a set of true propositions. The propositions, which are not in the set are considered false. These are so called close world assumptions used to solve the problem. A state can be changed to some other state, when some logical actions are performed on the assumptions of a state. The actions may be performed only if action preconditions are met. The preconditions are also a set of propositions which hold. The actions produce effects, which are taken over to the state description. Consider an infrastructure with 2 running virtual machines as an example. The state of such system would be represented as a set of propositions $[running1\&running2]$. This state may be changed by applying stop operation to the machine no 2. The operation can be similar to $stop(2, precondition : [running2], effect : [])$. The resulting status after the operation would be $[running1]$ since the effect set is empty.

Once the current state of the system and the goal are described, there are two ways of performing search. *Forward state-space search*, which is also called progression planning starts off from an initial state and tries to find a sequence of actions which would lead to a goal state, also defined at the beginning of the search. A *Backward state-space search* would do the opposite, searching from the goal state back to the initial state. This is sometimes called regression planning.

A distinction has to be made between totally ordered planning and partially ordered planning. The forward and backward searched are forms of *totally ordered plan search*. In a *partial ordered planning*, the planner would work on the “obvious” or “important” decisions first and not chronologically as it is done by the totally ordered planner.

3.1.1. Graphplan

For better results on partial and total space-search planning one would usually have an idea to use heuristics. These are usually inaccurate and therefore a special data structure is used to improve heuristics estimates called planning graph. The planning graph has to be spanned before the search for the plan is made.

> The graph consists of a sequence of levels that correspond to time steps in the plan, where level 0 is the initial state. Each level contains a set of literals and a set of actions. Roughly speaking, the literals are all those that *could* be true at that time step, depending on the actions executed at preceding time steps. Also roughly speaking, the actions are all those actions that *could* have their preconditions satisfied at that time step, depending on which of the literals[1] actually hold. [...] The number of steps on the planning graph provides a good estimate of how difficult it is to achieve a given literal from the initial state.[18]

On one hand planning graphs are used for heuristic estimation. On the other, they can also be used to directly extract a plan from. This is done using a so called *Graphplan* algorithm. It includes two main steps which alternate within a loop. It checks if all goal propositions are present in the current level with no mutual exclusion links between any pair of them. Mutual exclusion links mean, that one proposition negates the possibility of existence of another proposition. This may happen if one action negates an effect of another action, one of the effects of one action is the negation of a precondition of the other and one of the preconditions of one action is mutually exclusive with a precondition of the other. So if there are no mutex links between the actions, then a solution might exist within the current graph and the algorithm tries to extract that solution. If the previous is not the case, it expands the graph by adding the actions for the current level and the state propositions for the next level. The process continues until either a solution is found or it is learned that no solution exists. This way the Graphplan algorithm processes the planning graph, using a backward search to extract a plan. It allows some partial ordering among actions.

The Graphplan algorithm, although being fast at finding optimal solution does not suit our problem well, because it is still slow compared to finding some plan using HTN method.

3.1.2. Hierarchical Task Network - Informed Search

Aforementioned basic representations show what actions do, but they cannot show how long an action takes or even when an action occurs, except that it is before or after another action. To solve our problem of constantly changing nature, however, the planner needs to know when the actions begin and end. Here we discuss hierarchical task network planning. It is another approach to the planning, which is based on dealing with complexity using hierarchical decomposition.

[1]literals - here propositions

In HTN planning, the initial plan, which describes the problem, is viewed as a very high level description of what is to be done. Plans are refined by applying action decompositions. Each action decomposition reduces a high level action to a partially ordered set of lower-level actions. Action decomposition, therefore, embodies knowledge about how to implement actions. The process continues until only primitive actions remain in the plan. General descriptions of action decomposition methods are stored in a plan repository, from which they are extracted and instantiated to fit the needs of the plan being constructed. [18]

One may notice, that HTN plan creation is only as good as the quality of the knowledge base is. So it may happen that the planner won't find any solution if the problem didn't appeared before. Also the writing of the knowledge base may be more complicated than just defining classical propositions.

On the other hand the availability of consistent knowledge base allows implementations of the HTN planning algorithm to produce first usable actions much faster compared to the the Graphplan, where the Graph must be expanded completely to know if a solution exists at all.

The problem of complicated repository creation may be targeted using smart crowd sourcing strategies and automated tools, which can convert actions performed on a system to recover from failure into the methods for the HTN planner.

3.2. System design

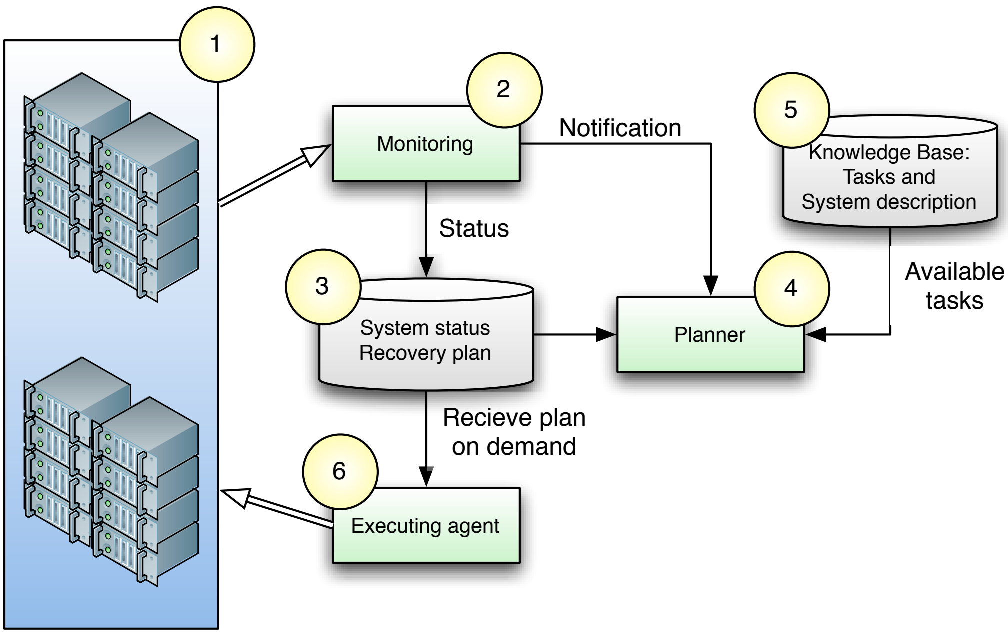


Figure 3.1.: Deployment view of the architecture of the planner. The monitoring system (2) receives the status of the infrastructure (1) and writes the data into the status database (3). In case of a system failure identification by the monitoring agent, the planner (4) is immediately notified. This triggers plan generation. Planner receives the system description and available tasks on the system as well as the current system status from the knowledge base (5) and the status database (3) respectively. It then produces a plan used by automated or human actors (6).

The overall structure of the application is fairly simple. As one can see on Figure 3.1, there are only two components, which define the whole software solution. One shows the

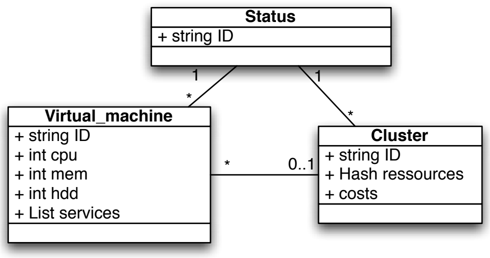


Figure 3.2.: Status of the virtual infrastructure environment. Provided by monitoring system.

status and the solution to the application user and the second does the planning. The application architecture is service oriented, so that other applications or components can be easily implemented. Knowledge, status and definition data of the system are kept separately in files using JSON[2] format.

3.3. Data types and data structure

The data, which our application depends on, can be classified by its source into four sets. First set includes the data provided by the monitoring system (figure 2.1 - Monitoring) represents the current status of the system (figure 3.2). An evaluation of this data can reveal faults in the system. If it indicates to a flaw, the data is used as a starting point for the search for a recovery plan. The evaluation of the system contains different attributes, which may indicate a fault in a virtual machine or the infrastructure. It shows idle, overloaded, hung-up, missing and stopped virtual machines as well as an evaluation value used in the planning algorithm.

The second and the third set includes the data provided by the administrator of the system. It is the description of the system and the set of tasks available to the system (figure 3.3). It consists of a set of parameters and thresholds, which depict a healthy system. One part of the description is information about sets of virtual machines, which are deployed together for performing some task. Here we will refer to such sets as virtual applications. The description of virtual applications can be further refined by assigning different types to different virtual machines that act in different roles. For example a virtual application, which is running to provide a blogging system needs some virtual machines, which will handle load balancing of the incoming requests, it needs some virtual machines to process the requests, and it needs some machines to store the data so it is available to every user. This virtual application would consist of three types of virtual machines - load balancer, worker and a database. The description of virtual applications and the virtual machine types within consists of the threshold for the maximum mean load of cpu and memory. and the maximum amount of used space on the hard disks. It also contains the information about services, which must be running on a virtual machine. If all of the above criteria are met, then the virtual machine of a particular type is considered healthy. The description also consists of the information on available virtualized hardware resources, which can host the virtual machines. These resources can be grouped together to define a set of clusters. It may be useful when dealing with multitenant resources.

Another part of the description provided by the administrator is set of tasks, which can be performed on a system and the operations which can be executed to complete the tasks

[2]JSON (JavaScript Object Notation) is a lightweight data-interchange format. It is based on a subset of the JavaScript Programming Language, Standard ECMA-262 3rd Edition - December 1999. JSON is a text format that is completely language independent but uses conventions that are familiar to programmers of the C-family of languages, including C, C++, C#, Java, JavaScript, Perl, Python, and many others. These properties make JSON an ideal data-interchange language. [5]

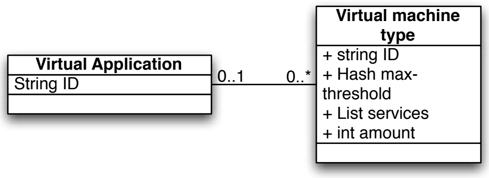


Figure 3.3.: Description of the system. Available clusters and normal state of running applications. Provided by administrator.

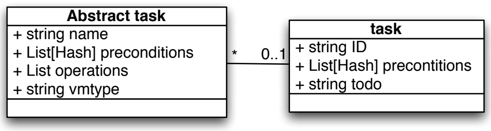


Figure 3.4.: Tasks, which can be executed in the environment. Provided by administrator.

(figure 3.4). The tasks can be of two types - decomposable and atomic tasks. Decomposable tasks consist of atomic tasks or other decomposable tasks. Recursive tasks are to be avoided. Both types of tasks contain a set of preconditions which have to be met to be able to perform the task. The preconditions of atomic tasks may only refer to the parameters of the virtual machines whereas the preconditions of decomposable tasks may only refer to the attributes of the evaluation of the system.

The last data set is the plan, created by the planner as a result of the planning operation (figure 3.5). It consists of a set of tasks which should be performed to recover from an error and the estimate of resulting system. The tasks are the same as the one available to the system, but they are ordered by the time of execution. The estimate is a set of virtual machines and their states by the moment of recovery.

3.4. Algorithm

The main part of the proposed solution is the search algorithm. It is a combination of the first-best search and HTN planning algorithms. The HTN planning algorithm would find a solution if the tasks which would solve the problem are defined in the knowledge base. If the tasks are not found in the knowledge base, the first-best search is run to try to find a recovery solution. The data flow overview of the system can be examined on figure 3.6 for better understanding of the context.

We will implement the search algorithm in a recursive function dig, which takes two parameters - a status, for which the solution has to be found and the list of tasks already performed to reach the status. The initial value of the tasks list should be an empty list (Algorithm 1line 1).

The Algorithm starts off by defining an empty list for all the states which can be generated from current status by applying tasks from the knowledge base on the infrastructure and on the virtual machines (Algorithm 1 lines 2). On the line 3 the algorithm gets the evaluation of the current status, so it can compare it with evaluations of the upcoming states and prune the ones, which evaluation value is greater, meaning they contain more failures. This way the search space will be reduced. Two nested loops find all the possible states, which can be derived from the current state along with the tasks needed to get to the derived status, and add them to the *statuslist* defined on line 2. The outer loop iterates over virtual machines in the current status (line 4) and the inner loop iterates over available tasks for every virtual machine (line 6). Derived states are produced by applying every

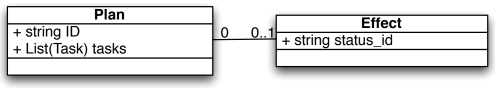


Figure 3.5.: Recovery plan and predicted resulting system after a plan is found. Provided by planner.

Algorithm 1 first best search planner with heuristic estimate evaluation.

```
1: function $dig(status, tasklist)$
2:     $statuslist \leftarrow []$
3:     $currenteval \leftarrow$ EVALUATE($status$)
4:     for $vm$ in $status$ do
5:         $tasks \leftarrow$ GETTASKS($vm$)
6:         for $task$ in $tasks$ do
7:             if $task$ is decomposable then
8:                 refine task
9:                 for $t$ in refined task do
10:                    $newstatus \leftarrow$ APPLY($t$, $vm$)
11:                end for                              ▷ new status with the changed vm
12:            else
13:                $newstatus \leftarrow apply(task, vm)$          ▷ new status with the changed vm
14:            end if
15:            $neweval \leftarrow$ EVALUATE($newstatus$)
16:            $appended \leftarrow tasklist + task$
17:            if $neweval = 0$ then
18:                return $appended$
19:            else
20:                $statuslist \leftarrow statuslist + (newstatus, appended)$
21:            end if
22:        end for
23:    end for
24:    filter $statuslist$ for $evaluation \leq currenteval$
25:    if $statuslist$ is empty then
26:        rase SolutionNotFoundError
27:    end if
28:    sort $statuslist$ ascending on value of $evaluation$
29:    try:
30:    return $dig(state, tasklist[0])$
31:    if S thenolutionNotFoundError catched
32:        process next possible solution
33:    end if
34: end function
```

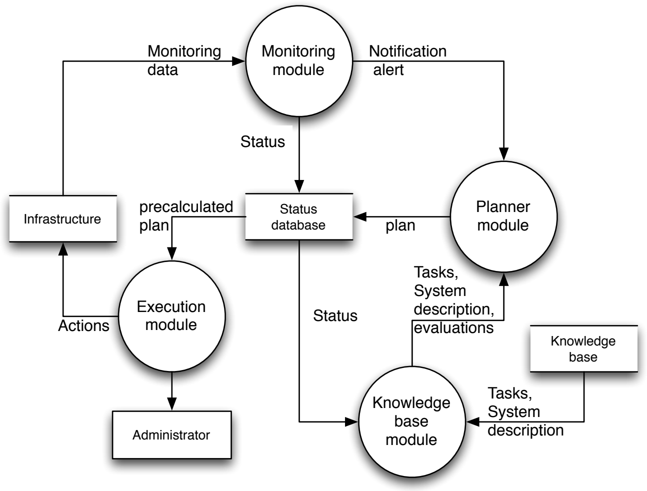


Figure 3.6.: Data flow scheme showing types of data required and produced by different parts of the planning system.

task that meets its preconditions from the knowledge base onto every virtual machine. The decomposable tasks have to be refined first before they can be applied (line 8). If in the course of the loop execution a solution is found, it is immediately returned (line 18). When the generation of the derived states is complete, the algorithm prunes the states, which evaluation is greater than that of the current status (line 24). The list with the states is then sorted ascending on the value of each state (line 28). The dig function is then called recursively with the first item of the status list as the best option and the tasks, which lead to the status, as parameters.

The evaluation algorithm which is used in the search is implemented as weighted estimation. It is computed as a weighted sum of different indicators of the faulty system and can be expanded by the administrator at any time. The indicators are missing, hung-up, idle and stopped virtual machines per virtual application as well as virtual applications with high load. The number of missing virtual machines is calculated from the number of the virtual machines in the application description. Machines where one of the mean performance metrics of CPU or memory are over the threshold set in the application description are marked as hung-up. Idle and stopped virtual machines are found by their performance metrics and the running status. Overloaded applications are the ones, where one of the mean metrics calculated over all of the machines within the application is greater than the threshold in the application description. The weight is a multiple of the number of available tasks. The hung-up virtual machines have the highest weight followed by the number of overloaded virtual applications. The missing virtual machines in an applications have the third highest weight and all the other indicators count with the weight of 1. The result of the evaluation is a structure, which contains all the parameters as well as the evaluation value (figure 3.7).

Algorithm 2 first best search planner with heuristic estimate evaluation.

```
 1: function evaluate(status)
 2:     for vm in status do
 3:         if vm is running then
 4:             add cpu, mem, hdd values to mean counters
 5:             if cpu too high or mem too high or hdd too high then
 6:                 $value \leftarrow value + 50$                    ▷ hung-up machine identified
 7:             end if
 8:             if cpu too low or mem too low or hdd too low then
 9:                 $value \leftarrow value + 1$                     ▷ idle machine identified
10:             end if
11:             if not all services on a vm are running then
12:                 $value \leftarrow value + 50$                    ▷ hung-up machine identified
13:             end if
14:             $deployed \leftarrow deployed + 1$
15:         else
16:             $value \leftarrow value + 1$                         ▷ stopped machine identified
17:         end if
18:     end for
19:     calculate mean counters
20:     if mean counters over threshold then
21:         $value \leftarrow value + 20$                         ▷ overloaded application identified
22:     end if
23:     $value \leftarrow$ ( amount needed $- deployed) * 10$    ▷ identified missing machines
24:     return value
25: end function
```

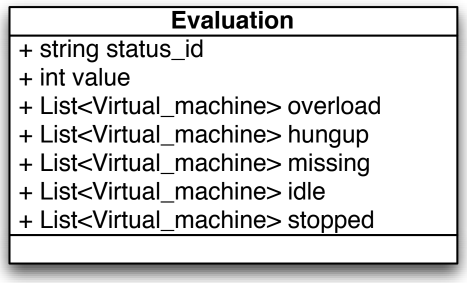

Evaluation
+ string status_id
+ int value
+ List<Virtual_machine> overload
+ List<Virtual_machine> hungup
+ List<Virtual_machine> missing
+ List<Virtual_machine> idle
+ List<Virtual_machine> stopped

Figure 3.7.: Evaluation object, returned by the *evaluate*(*status*) function.

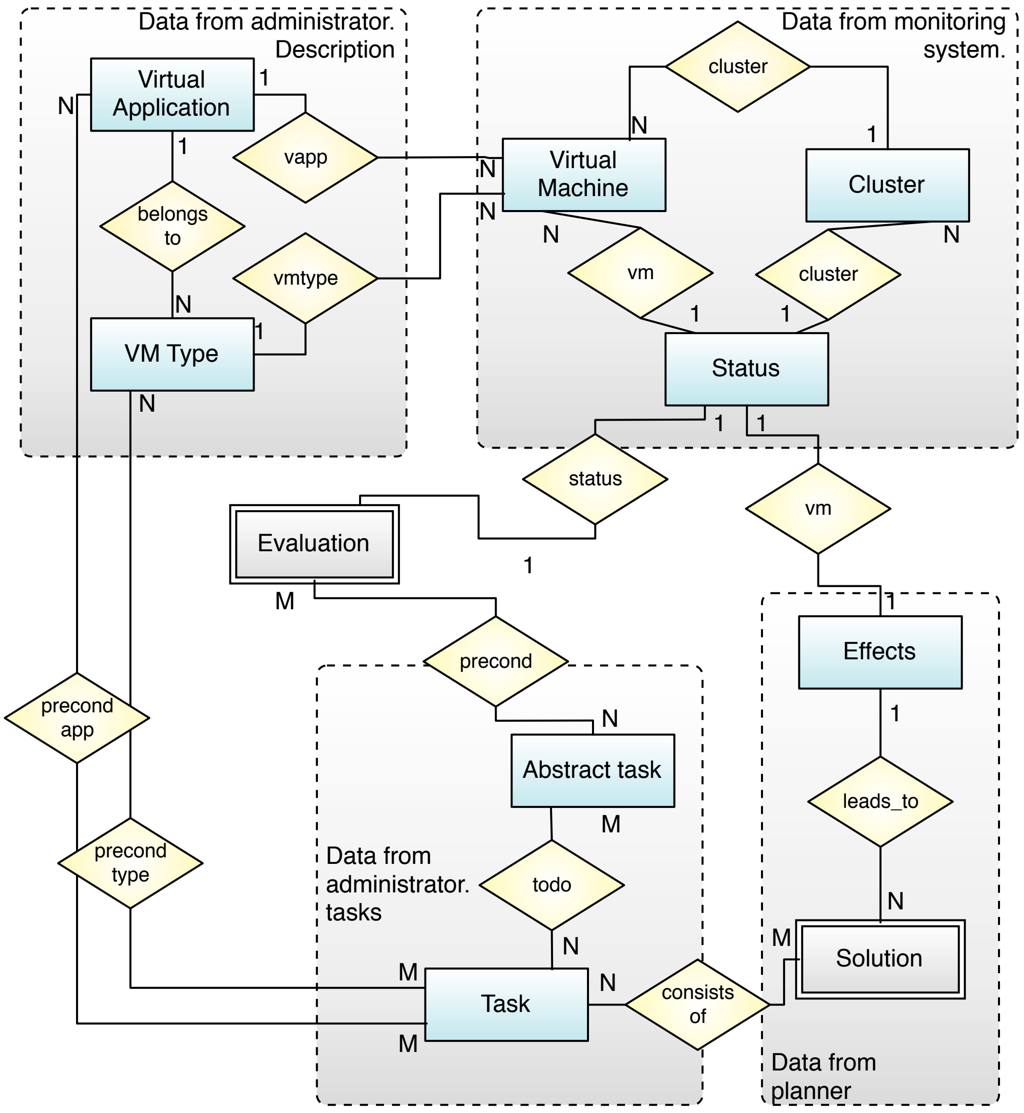


Figure 3.8.: Data structure representing the dependencies and connections between the entities that represent different aspects of the system. Application and type of the virtual machines are part of the cloud description. The tasks are part of the solution knowledge base and they have to be defined by an administrator together with the cloud description. The status of virtual machines and clusters are provided by the monitoring agent. Solution and the estimated effects are created by the planner.

4. Implementation

4.1. Implementation

The implementation is done using python programming language. The benefit of this is that the programming of the algorithm is done quickly. The language allows usage of functional programming methods, which makes the writing of optimized code for multiprocessor environment fairly simple. The implementation uses psyco module to improve performance [1]. Psyco makes the interpreter compile the code before the program execution. Usually python would compile the statements when it reaches them. Alone the usage of the psyco module makes the application runtime up to five times faster.

The result of the implementation is a framework, which can be extended to the needs of a concrete scenario. This shows possibility of interoperability of different modules and the ease of the extensibility of the implementation. The whole implementation was created with support of unit testing tools and we followed the test-driven development method. This produces good code quality and allows its secure alternation. A short overview of implementation of critical system parts is given by examining following examples.

The beginning of the planning cycle is an identification of a failure by the monitoring module. It constantly updates its knowledge about the status of the infrastructure. For example the status of a virtual machine in the infrastructure is of following structure:

```
{
    "some unique string id":{
        "cluster": "1",
        "type": "application",
        "vapp": "Lsc10w",
        "cpu": 34,
        "mem": 25,
        "hdd": 70,
        "running": 1,
        "services": ["nginx", "app"]
    }
}
```

[1]http://psyco.sourceforge.net

It is important to denote, that the unique id can be any string which identifies the virtual machine. An evaluation of this status can identify faults. For example determination of all the needed services of the virtual application on a virtual machine is implemented in following way:

```
if (not len(
      set(params[vmtype]['services']) - set(metr['services'])
      ) == 0):
    hungup[vapp][vmtype].append(vm)
    evalue += 5 * 10
```

The query is performed on a set subtraction operation. The result is the set of services which are not running. If the result set is not empty, the value will be increased and the hungup variable gets a mark that a machine is hanging. If in the course of evaluation a fault has been determined, the monitoring module writes the status of the monitored infrastructure into the status database and notifies the planner.

The planner gets all the needed information - already mentioned status, tasks and descriptions. Tasks are implemented as string which point to operations in the code. An illustration of stopping a machine:

```
def stop(self, **kwargs):
    params = kwargs['params']
    env = kwargs['env']
    machine = env[params['id']]
    machine['running'] = 0
    return env
```

Parameters is are represented as a generic hash *params*. It must contain an ID of the machine that should be stopped. The *env* variable represent the current status of the infrastructure.

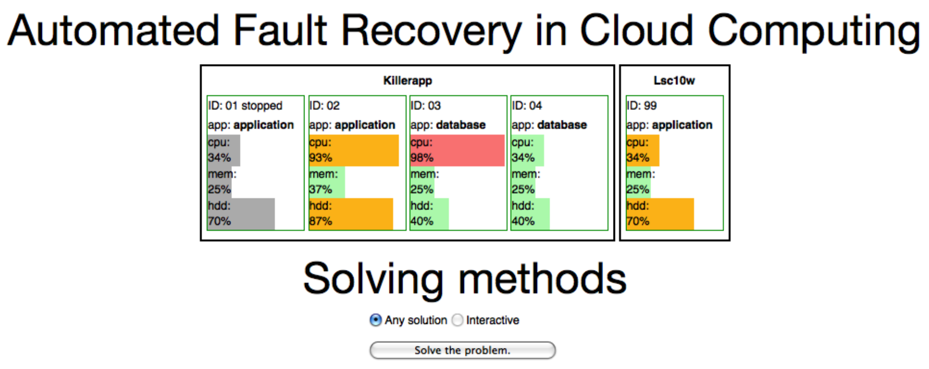
The knowledge module contains a mapping function of operation names to operation functions:

```
self._operations_ = {
    "start": self.start,
    "stop": self.stop,
    ...
    "delete": self.delete
}
```

The *apply*() function in the knowledge deals with transforming the virtual machine by an operation function. It copies the environment before applying the operation function so that it does not alter the data provided in the parameters.

```
def apply(self, operation, **kwargs):
    params = kwargs['params']
    env = copy.deepcopy(kwargs['env'])
    return self._operations_[operation](params = params, env = env)
```

The Search algorithm then applies the operations recursively on each new environment and searches for a first solution which can be found and returns it.



Automated Fault Recovery in Cloud Computing

Killerapp

ID: 01 stopped
app: **application**
cpu: 34%
mem: 25%
hdd: 70%

ID: 02
app: **application**
cpu: 93%
mem: 37%
hdd: 87%

ID: 03
app: **database**
cpu: 98%
mem: 25%
hdd: 40%

ID: 04
app: **database**
cpu: 34%
mem: 25%
hdd: 40%

Lsc10w

ID: 99
app: **application**
cpu: 34%
mem: 25%
hdd: 70%

Solving methods

[✓ Any solution] Interactive

Figure 4.1.: Initial screen of the user interface showing the current status of the infrastructure with 5 virtual machines. The red bar shows overloaded value of the cpu. grey bars indicate that the virtual machine is shut off.

```
newenv = self.knowhow.apply(task['todo'],
            params = params,
            env = env)
evaluation = self.knowhow.evaluate(newenv)['value']
...
if evaluation <= 0: return todos, newenv
```

The result contains a set of tasks to be performed on the infrastructure to get the estimated result.

All the modules were implemented separately. Moreover the user interface module was implemented in different programming language to show the independence of the solution to particular languages and possibility of integration into existing software. User interface module is implemented using Ruby programming language and represents a web UI accessible via browser. The initial screen shows the current status of the virtual machines and provides a possibility to search for a solution if a problem exists(figure 4.1). The result screen shows a found plan and estimated resulting infrastructure (figure 4.2).

4.2. Evaluation

To test scalability of the planner performance two evaluation scenarios were created. One is evaluating the impact of failure number increases, and the other is evaluating the impact of virtual machines number on the planning implementation. Each evaluation scenario consists of test cases, which provide different parameters for the planner. The results of the impact of failure number are presented on figure 4.3, and the results of the second scenario – on figure 4.4. Every test case in both evaluations was run 5 times to ensure the accuracy of the results. The results deviation of the test cases are so small, that only mean value is presented on the depictions.

The results show, that the execution time of the algorithm concerning the number of virtual machines is scaling in exponential time. However the performance concerning the number of failures scale in linear time. Performance of the planner, although implemented in a relatively slow programming language, is fast enough to be used in real-life situations.

The implemented solution is limited by hardware boundaries only at the cpu performance level. Being programmed sequentially, it does not utilize the possibilities of multiple processors yet. The amount of data held in memory and used for computations is reasonable. Calculating a plan for 250 virtual machines with 5 faults uses around 800MB of RAM.

The implementation solves following problems:

- Determination of a recovery plan for an outage is performed by planner in reasonable time .
- Monitoring of the recovery process can be performed by a monitoring module which will interact with the planning agent when changes of the infrastructure occur.
- Knowledge is stored in a human-readable format and can be used and altered by other applications.

4.2.1. Hardware and software setup for evaluation

Testing hardware is a virtual machine in a VMSphere environment of VMWare. It has following parameters:

CPU speed	2.2 GHz
memory amount	3 GB

Testing machine runs Ubuntu linux OS version 9.10 with the following relevant packages:

package	version
kernel	2.6.31
gcc	4.4.1
python	2.6.4
ruby	1.8.7

4.2.2. Evaluation procedure

The evaluation scenarios were created by manually altering the input which usually would be provided by a monitoring agent.

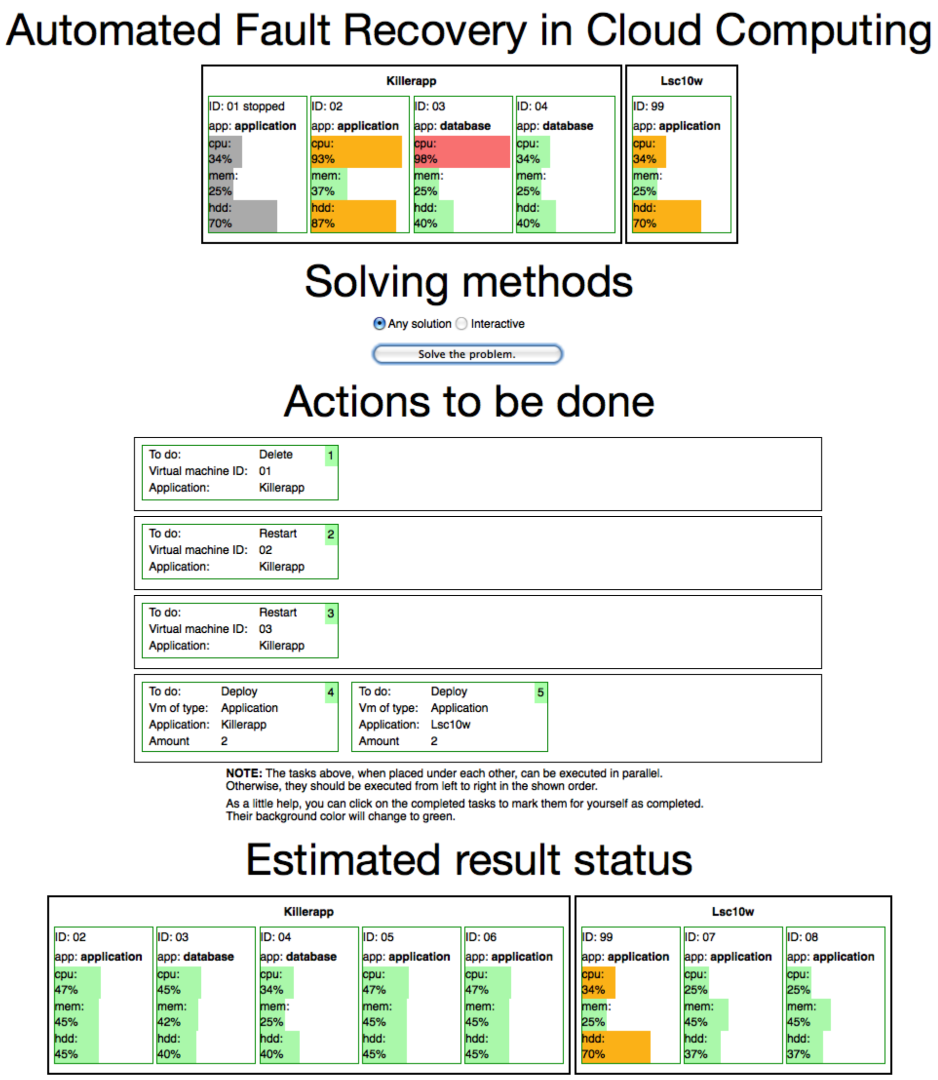
Each test case in the scenarios was run 5 times in a row. The results were recorded and mean value of spent time was calculated. It is then taken as result representation.

Evaluation of planner performance for generating a plan for 50 virtual machines and different amounts of faults produce following results:

5 faults	10 faults	20 faults	30 faults	40 faults	50 faults
2.169	5.026	12.112	18.344	33.854	47.292
2.125	5.040	12.102	18.461	33.731	48.013
2.116	5.062	12.264	18.431	33.014	45.223
2.174	4.926	12.149	18.472	34.292	46.053
2.242	4.968	12.262	18.427	33.966	45.994
mean results					
2.16520	5.00440	12.17780	18.42700	33.77140	46.51500

Evaluation of planner performance for generating a plan for an infrastructure with 5 faults and different amounts of virtual machines produce following results:

5 machines	20 machines	50 machines	100 machines	250 machines
0.044	0.350	2.313	9.479	112.671
0.046	0.339	2.298	9.599	104.138
0.044	0.361	2.303	9.640	97.604
0.044	0.341	2.383	9.452	97.645
0.045	0.337	2.296	9.449	97.301
		mean results		
0.04460	0.34560	2.31860	9.52380	101.87180



Automated Fault Recovery in Cloud Computing

Killerapp

ID: 01 stopped
app: **application**
cpu: 34%
mem: 25%
hdd: 70%

ID: 02
app: **application**
cpu: 93%
mem: 37%
hdd: 87%

ID: 03
app: **database**
cpu: 98%
mem: 25%
hdd: 40%

ID: 04
app: **database**
cpu: 34%
mem: 25%
hdd: 40%

Lsc10w

ID: 99
app: **application**
cpu: 34%
mem: 25%
hdd: 70%

Solving methods

[● Any solution] ○ Interactive

Solve the problem.

Actions to be done

To do:	Delete	1
Virtual machine ID:	01	
Application:	Killerapp	

To do:	Restart	2
Virtual machine ID:	02	
Application:	Killerapp	

To do:	Restart	3
Virtual machine ID:	03	
Application:	Killerapp	

To do:	Deploy	4	To do:	Deploy	5
Vm of type:	Application		Vm of type:	Application	
Application:	Killerapp		Application:	Lsc10w	
Amount	2		Amount	2	

NOTE: The tasks above, when placed under each other, can be executed in parallel.
Otherwise, they should be executed from left to right in the shown order.
As a little help, you can click on the completed tasks to mark them for yourself as completed.
Their background color will change to green.

Estimated result status

Killerapp

ID: 02
app: **application**
cpu: 47%
mem: 45%
hdd: 45%

ID: 03
app: **database**
cpu: 45%
mem: 42%
hdd: 40%

ID: 04
app: **database**
cpu: 34%
mem: 25%
hdd: 40%

ID: 05
app: **application**
cpu: 47%
mem: 45%
hdd: 45%

ID: 06
app: **application**
cpu: 47%
mem: 45%
hdd: 45%

Lsc10w

ID: 99
app: **application**
cpu: 34%
mem: 25%
hdd: 70%

ID: 07
app: **application**
cpu: 25%
mem: 45%
hdd: 37%

ID: 08
app: **application**
cpu: 25%
mem: 45%
hdd: 37%

Figure 4.2.: A proposed recovery plan consisting of five actions grouped by virtual machines they have to be performed on. Last two tasks are deployment tasks and these will be executed on a cluster. At the end an estimated resulting infrastructure is presented.

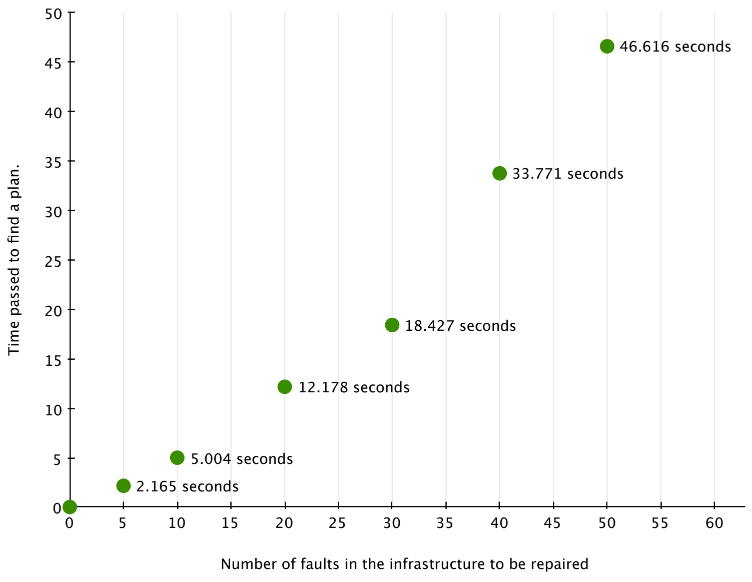


Figure 4.3.: This picture shows relation between amount of faults in a system and the time spent on finding a recovery plan. The amount of virtual machines remains the same during the test

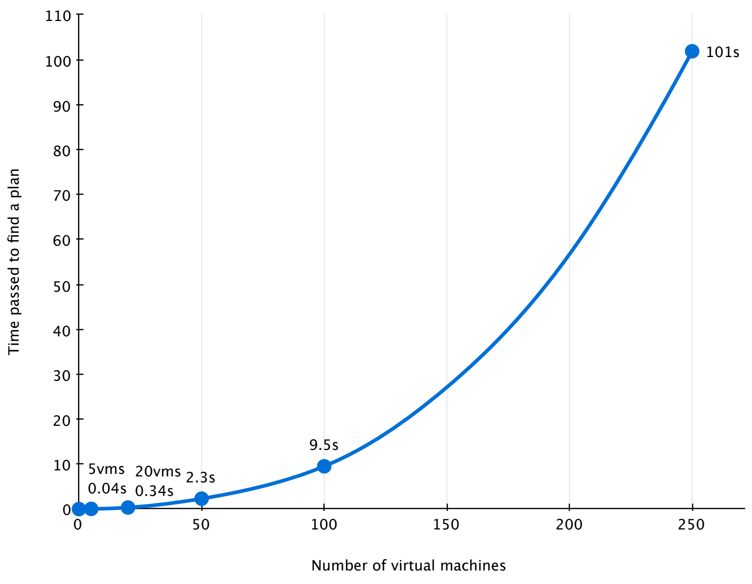


Figure 4.4.: This picture shows relation between amount of virtual machines and the time spent on finding a recovery plan. The amount of faults remains the same during the test.

5. Related work

There are several examples, where artificial intelligence and in particular - the automated planning was implemented in the similar large scale computing systems like grid or cluster computing.

The performance of the system is highly related to the representation of states, actions and goals. Here a similar approach was taken on dealing with constantly changing infrastructure metrics, goal and operation representation as Jim Blythe et. al. in [13]. However concerning plan generation our implementation aims at fast plan generation contrary to best plan generation because the latter is very time expensive task. Terry Zimmerman and Subbarao Kambhampati take here a Learning-assisted Automated Planning approach for dynamic environments [21]. William K. Cheung et. al. in their work[14] argue that the search for plan in their domain should be targeted at services composition and not at resource allocation, because the latter is already handled by grids. The presumption is not necessarily true for domain of cloud computing although their bidding-like approach for service composition is also considered in resource allocation in cloud computing.

An interesting idea concerning software stability is proposed by Paul Robertson and Brian Williams in [17]. Contrary to our approach of creating a solution to a generic system, they propose models, which, when used, create systems that will be able to recognize that it has failed and to recover from the failure. They try to increase system stability using automated planning algorithms by adding dynamic intelligent fault awareness and recovery to running systems. This enables the identification of unanticipated failures and the construction of novel workarounds to these failures. Their approach aims to minimize the cost, in terms of hand-coded specifications with respect to how to isolate and recover from failures.

Concerning plan execution there are several attempts to build a solution for grid computing environments. One of them is Pegasus - a system to generate executable grid workflows given a high-level specification of desired results. Pegasus uses Artificial Intelligence planning techniques to compose valid workflows, and has been used in several scientific applications. [15] Pegasus uses search to optimize the execution of a plan whereas our approach predefines the order of task execution. While pegasus can reduce plan execution time by finding tasks which can be executed in parallel, the search of the optimal execution scenario can also be a time-intensive task. The approach of search for optimal solution is also considered by CHAMPS - a prototype under development at IBM Research for CHAnge Management with Planning and Scheduling.[11] Although the last decision of action execution is made by a system administrator.

6. Outlook and future work

There is much room for further work of the current implementation and the addition of new features and modules. An implementation of the algorithm to utilize distributed computation mode may further shrink the time of planning phase. It may be possible by using functional programming methods $Map()$ and $reduce()$ provided by python. Another idea, mentioned by Jim Blythe is to make the planner reuse the plans which were found earlier[13]. The knowledge base may be refined by adding improved tasks which would fit better to concrete examples. The evaluation function may also be improved if such improvement is possible.

New features can be implemented such as improved search by constraints during the tasks application phase. Interactive search can also be an interesting idea to have a look at. Current implementation does not search for partial solutions if no solution can be found. It also does not include cost or time constraints in the search. These issues may be addressed in the future implementations.

Current implementation also assumes the fact, that the system react in the same way to the performed actions, as defined in the tasks operations. Although this static approach is good for finding a quick solution, it would be more realistic, if the algorithm would be adopted to search under the given uncertainty.

Nowadays the industry understands that it needs to automate its processes to be able to be competitive on the market. Therefore all the system automation research appears to be very useful. Current work opens new insights on complexity of such solutions especially because the topic which the work is targeted at is dealing with developing and changing technology. And as the technology changes and gains its usage shape, it will be easier to determine best practices of its use. Moreover determining problem places in its early development stages can be very beneficial later.

List of Figures

Bibliography

[1] 3tera. http://www.3tera.com/AppLogic/, visited on 4. October 2009.

[2] Amazon elastic compute cloud. http://aws.amazon.com/ec2/, visited on 4. October 2009.

[3] Eucalyptus systems. http://www.eucalyptus.com/, visited on 4. October 2009.

[4] Heroku, ruby cloud platform as a service, an architecture overview. http://heroku.com/how/architecture, visited on 4. October 2009.

[5] Json (javascript object notation). http://www.json.org, visited on 1. Dec 2009.

[6] Rightscale. http://www.rightscale.com/, visited on 4. October 2009.

[7] Techtarget.com: Heroku learns the hard way from amazon ec2 outage. http://searchcloudcomputing.techtarget.com/news/article/0,289142,sid201_gci1378426,00.html, visited on 5. February 2010.

[8] Vmware vcloud. http://www.vmware.com/solutions/cloud-computing/, visited on 4. October 2009.

[9] Vmware: Virtual machines, virtual server, virtual infrastructure. http://www.vmware.com/technology/virtual-machine.html, Visited on 7. June 2009.

[10] Xen cloud platform. http://xen.org/products/cloudxen.html, visited on 4. October 2009.

[11] J. L. Wolf K.-L. Wu V. Krishnan A. Keller, J. L. Hellerstein. The champs system: Change management with planning and scheduling, August 25, 2003.

[12] Sue Black. Deriving an approximation algorithm for automatic computation of ripple effect measures. *Inf. Softw. Technol.*, 50(7-8):723–736, 2008.

[13] Jim Blythe, Ewa Deelman, Yolanda Gil, Carl Kesselman, Amit Agarwal, Gaurang Mehta, and Karan Vahi. The role of planning in grid computing. In Enrico Giunchiglia, Nicola Muscettola, and Dana S. Nau, editors, *ICAPS*, pages 153–163. AAAI, 2003.

[14] William K. Cheung, Jiming Liu, Kevin H. Tsang, and Raymond K. Wong. Towards autonomous service composition in a grid environment. In *ICWS '04: Proceedings of the IEEE International Conference on Web Services*, page 550, Washington, DC, USA, 2004. IEEE Computer Society.

[15] Yolanda Gil, Ewa Deelman, Jim Blythe, Carl Kesselman, and Hongsuda Tangmunarunkit. Artificial intelligence and grids: Workflow planning and beyond. *IEEE Intelligent Systems*, 19(1):26–33, 2004.

[16] Dana Nau, Malik Ghallab, and Paolo Traverso. *Automated Planning: Theory & Practice.* Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2004.

[17] Paul Robertson and Brian Williams. Automatic recovery from software failure. *Commun. ACM*, 49(3):41–47, 2006.

[18] Stuart J. Russell and Peter Norvig. *Artificial Intelligence: A Modern Approach.* Pearson Education, 2003.

[19] Luis M. Vaquero, Luis Rodero-Merino, Juan Caceres, and Maik Lindner. A break in the clouds: towards a cloud definition. *SIGCOMM Comput. Commun. Rev.*, 39(1):50–55, 2009.

[20] Enrique Vargas. High availability fundamentals, November 2000.

[21] Terry Zimmerman and Subbarao Kambhampati. Learning-assisted automated planning: looking back, taking stock, going forward. *AI Mag.*, 24(2):73–96, 2003.

A. Appendix

The source code of the implementation of the planner is provided here. These sources as well as the web based user interface, test cases generator and the library dependencies can be found under http://home.in.tum.de/kerestey/BA/autofrc.tar.gz.

A.1. Planner: Planner.py

```
#!/usr/bin/env python
# encoding: utf-8

import sys
import unittest
import jsonlib2 as json
import copy

class SolutionNotFoundError(Exception): pass

class Planner:
  def __init__(self, knowledge_base):
    self.knowhow = knowledge_base
    self.solution_path = "../data/solution.json"

  def solve(self, constraints=None, method=None):
    env = self.knowhow.status()

    if not method: method = 'DFS'
    elif method not in ['DFS', 'Interactive']:
      return []

    effects = open(self.solution_path, "r+")
    effects.write(
      json.write(
        {"working": 1, "actions": [], "effects": {}}, indent = "    "
      )
    )
    effects.close()
    print ""
    print "machines in the infrastructure: ", len(env.keys())

    # expand the found tasks and get the lists of todo's
    if method == 'DFS':
      actions, result = self._dig_deep_(env)
    else:
      actions, result = self._dig_interactive_(([], [env]))

    print "amount of tasks in the recovery plan: ", len(actions)

    effects = open(self.solution_path, "r+")
    effects.write(json.write(
      {
        "working": 0,
        "actions": actions,
        "effects": result
      }, indent = "    ")
    )
    effects.close()

    return {
      "actions": actions,
      "effects": result
    }

  def _dig_deep_(self, env, level = 0, todos = None):
    """Retrieve the solution by applying the given set of todo's"""
    if not todos: todos = []

    curr_eval = self.knowhow.evaluate(env)

    envlist = []

    for vm, metr in env.iteritems():
      tasks = self.knowhow.tasks(vm, env = env, evaluation = curr_eval)

      for t_id, task in tasks.iteritems():
        params = {
          'id': vm,
          'app':  metr['vapp'],
          'type': metr['type'],
          'amount': curr_eval['missing'][metr['vapp']][metr['type']]
        }
        if task['type'] == 1:
          newenv = self.knowhow.apply(task['todo'],
                    params = params,
                    env = env)
          evaluation = self.knowhow.evaluate(newenv)['value']

          envlist.append((task, newenv, evaluation, vm, params))
          if evaluation <= 0:
            todos.append({
              'vm': vm,
              "task": task,
              'level': level,
              "params": params})
            return todos, newenv
        else:
          concr_task = map(lambda t: self.knowhow._tasks_[t] ,task['todo'])
          newenv = env
          for t in concr_task:
            newenv = self.knowhow.apply(t['todo'],
                    params = params,
                    env = newenv)

          evaluation = self.knowhow.evaluate(newenv)['value']

          envlist.append((task, newenv, evaluation, vm, params))

          if evaluation <= 0:
            todos.append({
              'vm': vm,
              "task": task,
              'level': level,
              "params": params})
            return todos, newenv


    # sort the new environment list to get the best tasks come first.
    envlist = filter(lambda a: a[2] <= curr_eval["value"], envlist)

    if envlist: envlist.sort(lambda a,b: cmp(a[2], b[2]))
    else: raise SolutionNotFoundError

    for e in envlist:
      newtodos = copy.copy(todos)
      newtodos.append({
        'vm': e[3],
        "task": e[0],
        'level': level,
        "params": e[4]
      })
      try:
        return self._dig_deep_(e[1], level=level+1, todos = newtodos)
      except SolutionNotFoundError:
        pass

  def _dig_wide_(self, params):
    todos, env = params

    return (todos, env)
  def _dig_interactive_(self):
    """Interactive search for the soultion"""
    todos, env = params

    return (todos, env)
```

A.2. Planner: Knowledge.py

```
#!/usr/bin/env python
# encoding: utf-8

import sys
import os
import unittest
import imp
import copy
import jsonlib2 as json
impott random

class KnowledgeException(Exception): pass
class TaskNotFoundException(KnowledgeException): pass

class Knowledge:
  def __init__(self):
    tasks_path = "../data/tasks.json"
    status_path = "../data/status.json"
    descr_path = "../data/descr.json"
    solution_path = "../data/solution.json"

    self._operations_ = {
      "start": self.start,
      "stop": self.stop,
      "pause": self.pause,
      "restart": self.restart,
      "deploy": self.deploy,
      "clone": self.clone,
      "migrate": self.migrate,
      "snapshot": self.snapshot,
      "restore": self.restore,
      "delete": self.delete
    }

    f = open(tasks_path, "r")
    self._tasks_ = json.read(f.read())
    f.close()

    f = open(status_path, "r")
    self._status_ = json.read(f.read())
    f.close

    f = open(descr_path, "r")
    self._descr_ = json.read(f.read())
    f.close

  #
  # Tasks on the machines
  #

  def tasks(self, vm, env=None, evaluation=None):
    """returns the tasks, which can be performed on a vm"""
    if not env: env = self.status()
    if not evaluation: evaluation = self.evaluate(env = env)

    vmtype = env[vm]['type']
    vapp = env[vm]['vapp']


    res = {}
    for t_id, t in self._tasks_.iteritems():
      if t['type'] == 0:
        fulfilled = True

        try:
          if vmtype not in t['vmtype']: fulfilled = False
        except KeyError:
          pass

        try:
          if fulfilled:
            for pre in t['preconditions']:
              fulfilled = True
              for premise, val in pre.iteritems():
                if val != evaluation[premise][vapp][vmtype]:
                  fulfilled = False
                  break
              if fulfilled: break
        except KeyError:
          fulfilled = False

        if fulfilled:
          res[t_id] = t
      elif t['type'] == 1:
        fulfilled = True

        try:
          if vmtype not in t['vmtype']: fulfilled = False
        except KeyError:
          pass

        try:
          if fulfilled:
            for pre in t['preconditions']:
              fulfilled = True
              for premise, val in pre.iteritems():
                if val != env[vm][premise]:
                  fulfilled = False
                  break
              if fulfilled: break
        except KeyError:
          fulfilled = False



        if fulfilled:
          res[t_id] = t

    return res

  #
  # Evaluation of the status
  #

  def evaluate(self, env = None):
    """evaluate the environment, set by env."""
    if not env:
      env = self._status_

    metrics = env
    vApps = self._descr_['vApps']

    cpu = {} # mean cpu - tells about mean cpu utilization of the vms
    mem = {} # mean mem - tells about mean mem utilization of the vms
    hdd = {} # mean hdd - tells about mean hdd utilization of the vms
    hungup = {}
    deployed = {}
    missing = {}
    stopped = {}
    overload = {}
    idle = {}
    evalue = 0
    multiplier = len(metrics.keys())

    for vapp, params in vApps.iteritems():
      # make a dictionary of crashed services in vm's in every vApp
      hungup[vapp] = dict(map(lambda x: (x, []), params.keys()))
      # make a dictionary of deployed vm's in every vApp
      deployed[vapp] = dict(map(lambda x: (x, 0), params.keys()))
      # make a dictionary of stopped vm's per application
      stopped[vapp] = dict(map(lambda x: (x, []), params.keys()))
      # make a dictionary of stopped vm's per application
      idle[vapp] = dict(map(lambda x: (x, []), params.keys()))
      # make a dictionary of the cpu, memory and hdd metrics grouped by
      # vapps
      cpu[vapp] = dict(map(lambda x: (x, 0), params.keys()))
      mem[vapp] = copy.copy(cpu[vapp])
```

```
hdd[vapp] = copy.copy(cpu[vapp])
overload[vapp] = dict(map(lambda x: (x, False), params.keys()))

missing[vapp] = dict(map(lambda x: (x, 0), params.keys()))

for vm, metr in metrics.iteritems():
  if not metr['vapp'] == vapp: continue
  vmtype = metr['type']

  if metr['running'] == 1:
    # adding metrics for mean calculations
    cpu[vapp][vmtype] += metr['cpu']
    mem[vapp][vmtype] += metr['mem']
    hdd[vapp][vmtype] += metr['hdd']

    if vm not in hungup[vapp][vmtype]:
      # too high metrics
      if (metr['cpu'] >= params[vmtype]['cpu']
        or metr['mem'] >= params[vmtype]['mem']
        or metr['hdd'] >= params[vmtype]['hdd']):
        hungup[vapp][vmtype].append(vm)
        evalue += 5 * 10
      # too low metrics
      if (metr['cpu'] <= 2 or metr['mem'] <= 2):
        idle[vapp][vmtype].append(vm)
        evalue += 1

    # finding out crashed services
    if (vm not in hungup[vapp][vmtype]
      and not len(
        set(params[vmtype]['services']) - set(metr['services'])
        ) == 0):
      hungup[vapp][vmtype].append(vm)
      evalue += 5 * 10

    # add to the number of deployed machines per vm type per app
    deployed[vapp][vmtype] += 1
  else:
    stopped[vapp][vmtype].append(vm)
    evalue += 1

for vmtype in deployed[vapp]:
  # calculate the mean metrics
  items = deployed[vapp][vmtype]

  try:
    cpu[vapp][vmtype] = float(cpu[vapp][vmtype]) / items
  except ZeroDivisionError:
    cpu[vapp][vmtype] = 0

  try:
    mem[vapp][vmtype] = float(mem[vapp][vmtype]) / items
```

```
      except ZeroDivisionError:
        mem[vapp][vmtype] = 0

      try:
        hdd[vapp][vmtype] = float(hdd[vapp][vmtype]) / items
      except ZeroDivisionError:
        mem[vapp][vmtype] = 0

      # relative levels to the app defaults
      if   ((cpu[vapp][vmtype]-params[vmtype]['cpu'])>=0
         or (mem[vapp][vmtype]-params[vmtype]['mem'])>=0
         or (hdd[vapp][vmtype]-params[vmtype]['hdd'])>=0):
        overload[vapp][vmtype] = True
        evalue += 2 * 10

      # calculate number of missing machines
      items = params[vmtype]['amount'] - items
      if items>0:
        missing[vapp][vmtype] = items
        evalue += items * 10

  result = {
    'hungup': hungup,
    'missing': missing,
    'overload': overload,
    'idle': idle,
    'stopped': stopped,
    'value': evalue
  }

  return result

#
# Status retriever
#

def status(self): return self._status_

#
# Operations on the machines
#

def apply(self, operation, **kwargs):
  params = kwargs['params']
  env = copy.deepcopy(kwargs['env'])
  return self._operations_[operation](params = params, env = env)


# Operations themselves
def start(self, **kwargs):
  params = kwargs['params']
  env = kwargs['env']
```

```
    descr = self._descr_
    machine = env[params['id']]
    mtype = machine['type']
    vApp = machine['vapp']
    machine['running'] = 1
    machine['cpu'] = descr['vApps'][vApp][mtype]['cpu'] / 2
    machine['mem'] = descr['vApps'][vApp][mtype]['mem'] / 2
    machine['hdd'] = descr['vApps'][vApp][mtype]['hdd'] / 2
    machine['services'] = descr['vApps'][vApp][mtype]['services']
    result = env
    return result

  def pause(self, **kwargs):
    params = kwargs['params']
    env = kwargs['env']
    machine = env[params['id']]
    machine['running'] = 2
    return env

  def stop(self, **kwargs):
    params = kwargs['params']
    env = kwargs['env']
    machine = env[params['id']]
    machine['running'] = 0
    return env

  def restart(self, **kwargs):
    params = kwargs['params']
    env = kwargs['env']
    descr = self._descr_
    machine = env[params['id']]
    mtype = machine['type']
    vApp = machine['vapp']
    machine['running'] = 1
    machine['cpu'] = descr['vApps'][vApp][mtype]['cpu'] / 2
    machine['mem'] = descr['vApps'][vApp][mtype]['mem'] / 2
    machine['hdd'] = descr['vApps'][vApp][mtype]['hdd'] / 2
    machine['services'] = descr['vApps'][vApp][mtype]['services']
    return env

  def deploy(self, **kwargs):
    try:
      params = kwargs['params']
      env = kwargs['env']
      descr = self._descr_
    except KeyError:
      return {}

    try:
      amount = params['amount']
    except KeyError:
      amount = 1
    # deploy the given amount of machines
    try:
      vapp = params['app']
      vmtype = params['type']
    except KeyError:
      return {}

    for i in range(0, amount):
      while True:
        m_id = str(random.randint(1, 100000))
        if m_id not in env.keys():
          break

      # find out the template for the new machine
      newvm = descr['vApps'][vapp][vmtype]

      # find out the cluster for the new machine
      if newvm['clustered'] == "deny":
        # find first machine of the same type and take it's cluster
        cluster = descr['clusters'].keys()[0]
        for metr in env.values():
          if metr['vapp'] != vapp: continue
          if metr['type'] == vmtype:
            cluster = metr['cluster']
            break
      else:
        cluster = dict(map(lambda x: (x, 0), descr['clusters'].keys()))
        for metr in env.values():
          if metr['vapp'] != vapp: continue
          if metr['type'] == vmtype:
            cluster[metr['cluster']] += 1

        cluster = min(cluster, key = lambda a: cluster.get(a))

      env[m_id] = {
        "cluster": cluster,
        "type": vmtype,
        "vapp": vapp,
        "cpu": descr['vApps'][vapp][vmtype]['cpu'] / 2,
        "mem": descr['vApps'][vapp][vmtype]['mem'] / 2,
        "hdd": descr['vApps'][vapp][vmtype]['hdd'] / 2,
        "running": 1,
        "services": newvm["services"]
      }

    return env

  def clone(self, **kwargs):
    try:
      params = kwargs['params']
      env = kwargs['env']
      vm_id = params['id']
    except KeyError:
      return {}

    while True:
      new_id = str(random.randint(1, 100000))
      if new_id not in env.keys():
        env[new_id] = env[vm_id]
        break

    return env

  def migrate(self, **kwargs):
    try:
      params = kwargs['params']
      env = kwargs['env']
      descr = self._descr_
      vm = env[params['id']]
    except KeyError:
      return {}

    vapp = vm['vapp']
    vmtype = vm['type']

    try:
      target = str(params['target'])
    except KeyError:
      if descr['vApps'][vapp][vmtype]['clustered'] == "deny":
        # find first machine of the same type and take it's cluster
        target = descr['clusters'].keys()[0]
        for metr in env.values():
          if not metr['vapp'] == vapp: continue
          if metr['type'] == vmtype:
            cluster = metr['cluster']
            break
      else:
        target = dict(map(lambda x: (x, 0), descr['clusters'].keys()))

        for metr in env.values():
          if not metr['vapp'] == vapp: continue
          if metr['type'] == vmtype:
            target[metr['cluster']] += 1

        target = min(target, key = lambda a: target.get(a))


    vm['cluster'] = target

    return env

  def snapshot(self, **kwargs):
    """Makes a snapshot of a VM"""
    try:
      params = kwargs['params']
      env = kwargs['env']
      vm_id = params['id']
    except KeyError:
      return {}

    state = copy.deepcopy(env[vm_id])
    env[vm_id]['snapshot'] = state

    return env

  def restore(self, **kwargs):
    """Restores a VM to it's previous state and deletes its snapshot"""
    try:
      params = kwargs['params']
      env = kwargs['env']
      vm_id = params['id']
    except KeyError:
      return {}

    try:
      state = copy.deepcopy(env[vm_id]['snapshot'])
      if state: env[vm_id] = state
    except KeyError:
      return env

    return env

  def delete(self, **kwargs):
    try:
      params = kwargs['params']
      env = kwargs['env']
      vm_id = params['id']
    except KeyError:
      return {}

    try:
      app = env[vm_id]['vapp']
      del env[vm_id]
    finally:
      return env
```

A.3. Planner: tests/test-planner.py

```
#!/usr/bin/env python
# encoding: utf-8

import sys
import unittest

import Planner
import Knowledge

class PlannerTests(unittest.TestCase):
    def setUp(self):
        self.knowhow = Knowledge.Knowledge()
        self.planner = Planner.Planner(self.knowhow)

    def test_planner_generates_a_plan(self):
        """
        Test if planner generates a correct plan on healing the system
        """
        constr = {
            "costs": {
                "cpu": 5.0,
                "mem": 4.0,
                "sto": 4.0,
                "traf": 0.4,
                "base": 25
            },
            "reliability": 99
        }
        start = self.knowhow.status()
        plan = self.planner.solve(constraints = constr)

        assert len(plan['actions']) == 5

if __name__ == '__main__':
    unittest.main()
```

A.4. Planner: runtests.py

```
#!/usr/bin/env python
# encoding: utf-8

import unittest
import psyco
from tests.test_planner import PlannerTests

psyco.full()

if __name__ == '__main__':
  unittest.main()
```

A.5. Data: tasks.json

```
  {
    "200": {
      "preconditions": [{"overload": true}],
      "todo": ["06", "07", "sync"],
      "type": 0,
      "name": "shard",
      "vmtype": ["database"]
    },
    "203": {
      "preconditions": [
        {
          "stopped": true,
          "overload": false
        },
        { "idle": true }
      ],
      "todo": ["10"],
      "type": 0,
      "name": "optimize utilization"
    },
    "202": {
      "preconditions": [{"overload": true}],
      "todo": ["05"],
      "type": 0,
      "name": "lower load",
      "vmtype": ["application"]
    },
    "205": {
      "preconditions": [{"missing": true}],
      "todo": ["200"],
      "type": 0,
      "name": "deploy missing",
      "vmtype": ["database"]
    },
    "204": {
      "preconditions": [{"missing": true}],
      "todo": ["05"],
      "type": 0,
      "name": "deploy missing",
      "vmtype": ["application"]
    },
    "02": {
      "preconditions": [{"running": 1}],
      "todo": "stop",
      "type": 1
    },
    "10": {
      "preconditions": [],
      "todo": "delete",
      "type": 1
    },
    "01": {
      "preconditions": [{"running": 0}, {"running": 2}],
      "todo": "start",
      "type": 1
    },
    "06": {
      "preconditions": [],
      "todo": "clone",
      "type": 1
    },
    "07": {
      "preconditions": [],
      "todo": "migrate",
      "type": 1
    },
    "04": {
      "preconditions": [{"running": 1}],
      "todo": "restart",
      "type": 1
    },
    "05": {
      "preconditions": [],
      "todo": "deploy",
      "type": 1
    }
  }
```

A.6. Data: description.json

```
{
"vApps":{
 "Killerapp": {
   "application":{
     "amount": 3,
     "clustered": "deny",
     "cpu": 95,
     "mem": 90,
     "hdd": 90,
     "services": ["app", "nginx"]
   },
   "database":{
     "amount": 2,
     "clustered": "force",
     "cpu": 90,
     "mem": 85,
     "hdd": 80,
     "services": ["postgres", "pgpool2"],
     "ensure": ["synchronised"]
   }
 },
 "Lsc10w": {
   "application":{
     "amount": 3,
     "clustered": "force",
     "cpu": 50,
     "mem": 90,
     "hdd": 75,
     "services": ["app", "nginx"]
   }
 }
},
"clusters": {
  "1": {
    "avail": {
      "cpu": 20,
      "mem": 24000,
      "hdd": 5000,
      "sla": 95
    },
    "price": {
      "cpu": 5,
      "mem": 5,
      "hdd": 1,
      "base": 0
    }
  },
  "2": {
    "avail": {
      "cpu": 10,
      "mem": 1000,
      "hdd": 5000,
      "sla": 100
    },
    "price": {
      "cpu": 15,
      "mem": 15,
      "hdd": 0.5,
      "base": 20
    }
  }
}
}
```

A.7. Data: status.json

```
{
    "01":{
        "cluster": "1",
        "type": "application",
        "vapp": "Killerapp",
        "cpu": 34,
        "mem": 25,
        "hdd": 70,
        "running": 0,
        "services": ["app", "nginx"]
    },
    "02":{
        "cluster": "2",
        "type": "application",
        "vapp": "Killerapp",
        "cpu": 93,
        "mem": 37,
        "hdd": 87,
        "running": 1,
        "services": ["nginx"]
    },
    "03":{
        "cluster": "1",
        "type": "database",
        "vapp": "Killerapp",
        "cpu": 98,
        "mem": 25,
        "hdd": 40,
        "running": 1,
        "services": ["postgres", "pgpool2"]
    },
    "04":{
        "cluster": "1",
        "type": "database",
        "vapp": "Killerapp",
        "cpu": 34,
        "mem": 25,
        "hdd": 40,
        "running": 1,
        "services": ["postgres", "pgpool2"]
    },
    "99":{
        "cluster": "1",
        "type": "application",
        "vapp": "Lsc10w",
        "cpu": 34,
        "mem": 25,
        "hdd": 70,
        "running": 1,
        "services": ["nginx", "app"]
    }
}
```