Self-Optimization in Autonomic Systems

Marko Kankaanniemi
Department of Computer Science
University of Helsinki
Email: marko.kankaanniemi@cs.helsinki.fi

***Abstract*—Autonomic computing is a research area that extends to numerous different fields of science. We describe how autonomic computing can be used to overcome many problems the IT industry is facing today. Autonomic computing systems are, by definition, self-configuring, self-healing, self-optimizing and self-protecting. We show some examples of existing systems that have these self-* capabilities. We look at what utility functions are and how they can be used in self-optimization. We describe a two-level architecture for self-optimization in a data center scenario. The architecture is flexible and suits very different kinds of applications. It supports fair resource allocation in a system where there can be very heterogeneous applications using shared resources.**

I. Introduction

Autonomic computing has become a significant research area since IBM launched its initiative in 2001 [1]. Computer systems are becoming so complex to manage that it is throttling their development. It seems clear that the only way to overcome this complexity is to make the systems manage themselves. Highly skilled human administrators usually cost a lot more than the systems themselves, and people with the right kind of expertise are often hard to find. Human error is the largest single source of failure in computer systems, accounting for more than 40% of failures according to some sources [2]. This may suggest that human administrators are often not as competent as they should be, and even those who are, will make errors. Errors can be very difficult for humans to locate and fix, and the process can be very time-consuming.

The way forward is to make computers do all the low-level management and to have humans only specify the high-level policies that represent the business requirements of the enterprise. When only the autonomic system itself has to know the fine details of its configuration, it is easier for human operators to learn how to administer it and configuration errors are less likely. Even in the case of an error, an autonomic system can heal itself.

The four major capabilities that an autonomic computing system must possess are self-configuration, self-healing, self-optimization and self-protection. Ganek and Corbi [1] list other self-* capabilities as well but some of them can be thought of as sub-capabilities of the four listed here. For example, a self-optimizing system has to be, in some ways, self-tuning and self-adaptive. In this paper we concentrate on the self-optimizing property.

Designing truly autonomic computing systems is a huge challenge to the research community and it will require expertise from many fields. E.g., by studying the existing research in the fields of biology and economics we can find ideas that can be used in the development of autonomic systems. The real world with its complex social systems can serve as an example of a huge autonomic system comprising a myriad of autonomic elements. Possible advances in the autonomic computing research might also contribute something valuable to other fields as well, so everyone may benefit in the end. The term autonomic system is actually motivated by the human autonomic nervous system that, e.g., makes the heart beat involuntarily. The human body is a great example of the kind of autonomicity we should strive for. Humans can breathe without conscious effort, wounds on their skin usually heal by themselves and their muscles work in coordination to produce many kinds of movement. Just like using human-like intelligence as a goal in artificial intelligence, we can use the whole human body as a near-perfect example of an autonomic system.

Simple examples of the self-* properties can be found in software that is already widely in use. For example, modern operating systems such as Ubuntu Linux [3] and Mac OS X [4] are able to configure themselves without almost any input from the user at installation time, which is a major improvement from the operating systems of the early 1990s. It used to be the case that the administrator had to specify detailed information about the computer to the operating system's installer, such as the geometry of the hard disk and the IRQs used by the devices attached to the computer. Likewise, when new devices are attached to a system that is already installed, the operating systems usually detect and configure them autonomically. Operating systems are also self-protective in some ways. For example, the OS kernel with the assistance of the CPU makes sure that low-level hardware access is protected from user-level processes and that the processes are protected from each other.

Internet routing is a good example of an existing self-optimizing system. IP packets usually find their way to the right destination via the optimal (or near-optimal) path. However, true self-optimization is mostly only found inside smaller network segments that use the Routing Information Protocol (RIP) [5] or the Open Shortest Path First protocol (OSPF) [6] for internal routing, since the Border Gateway Protocol (BGP) [7] (which is used for routing across larger network segments) requires a lot of configuration from human administrators. Routers using RIP and OSPF, on the other hand, can usually build routing tables entirely by themselves. In case of network outages, routers can relatively quickly update their routing

tables so that packets can go via a different route. This means that IP networks are also self-healing, which is no coindicence. Packet switching networks were largely motivated by the need for a communications system that could not be brought down by a nuclear war [8], which was a serious concern in the 1960s as the relationship between the United States and the Soviet Union was very tense at the time. This meant that the network had to be designed with a capability for self-healing. Although there are examples of systems that implement different kinds of self-* properties, the fully autonomic systems that the research aims for are still a thing of the future.

The remainder of the paper is organized as follows. In Section 2 we look at self-optimization in general and have a look at utility functions and other approaches that could be used for implementing self-optimization. In Section 3 we give a more detailed description of a specific kind of architecture that can be used in a self-optimizing system. In Section 4 we briefly describe a prototype implementation of the architecture presented in the previous section. Finally, in Section 5 we sum up the article.

II. Self-Optimization

Optimization means ”an act, process, or methodology of making something (as a design, system, or decision) as fully perfect, functional, or effective as possible” as defined by Merriam-Webster’s Online Dictionary [9]. In other words, optimization means finding the most effective ways to do things.

A. A General View

Usually when we talk about optimizing computer programs, we mean optimization at the source code level as done by the programmer. This is a very typical part of writing software, although it has become a bit less of an issue as computers have become faster. Optimizing programs in the small scale by writing certain operations in the Assembly language, for example, is rarely necessary these days but optimization in the large scale is just as important as always. This means that we still have to find and implement effective algorithms. Effective algorithms are important in stand-alone programs but they are even more important when we talk about the algorithms that components in distributed systems use to interact with each other. Ineffective algorithms in distributed systems may cause network congestion and they can disturb other users of the network, which can be a lot more serious concern than ineffective operation of the system itself.

A self-optimizing system is one that dynamically optimizes the operation of its own components while it is running. The optimizing component can be an agent that is separate from the component that is being optimized: the optimizer just continuously adjusts the control parameters that it passes to other components. Typically the optimizer has to have intimate knowledge of the components being optimized but optimization can be done on different levels as we will demonstrate in the following sections of this article. The higher-level optimizer does not need to know details of the components at the lowest level of the autonomic system.

B. Utility Functions

If we want computer systems to improve their performance by self-optimization, the systems need to have some kind of rules that they can follow. A naive solution would be to specify a set of situation-action rules for all situations that can occur. It would be a very bad strategy in the case of autonomic systems as it would require the human administrator to go into very low-level details, and, in many cases, the list of rules would probably never be quite complete. This kind of need for low-level configuration does not meet the objectives set for autonomic computing systems since one of the main requirements is that humans should only need to specify the appropriate high-level business policies that the autonomic systems follow [1].

Another approach would be to use goal policies that divide the states of the system into desirable and undesirable ones. That would definitely be a better approach than the one described above but it would not be very good for optimization because when the system has reached a desirable state, it would not try to improve its performance anymore. What we need is a method that continuously aims for better performance.

The third approach is the use of utility functions. With utility functions we can calculate the utility (i.e., business value) of even completely different kinds of systems in a common currency. In the modern society all goods and services are assigned some monetary value that can be used to compare them with each other in terms of how valuable they are to their owners. This is exactly what utility functions are used for in autonomic computing. We can specify high-level business rules that a utility function uses to evaluate the given state’s business value. When we have calculated the utility of different Autonomic Environments in a single autonomic system, we can calculate their sum, which is the utility of the entire autonomic system. The goal of self-optimization is to maximize the utility of the entire system at all times.

The use of utility-based resource allocation in computer systems goes all the way back to 1968 when Sutherland [10] presented a futures market in which the users could bid for computer time based on their own utility functions. Back then there was a PDP-1 server at Harvard University which was shared among students and faculty members. They were assigned differents amounts of virtual currency (called yen) according to the importance of their projects. The users would reserve computer time for the future and they would regain the yen as soon as they had used the reserved time. A user could not bid more than he could afford so the users with the most yen had the advantage. More recently utility functions have been chosen as the approach for self-optimization in many cases [11] [12] [13]. Traditionally utility functions have been used in the fields of microeconomics and artificial intelligence. In microeconomics utility functions are, e.g., used to model the

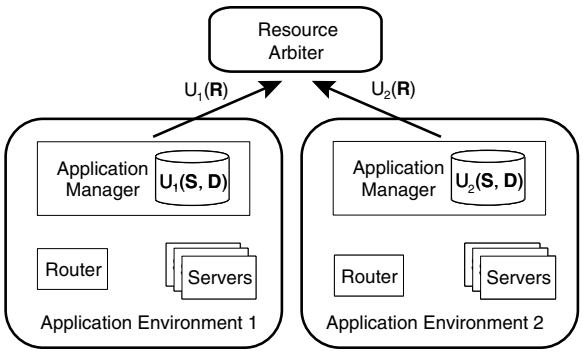

Fig. 1. Data Center Architecture [15, Figure 1].

TABLE I
AN EXAMPLE OF A RESOURCE-LEVEL UTILITY FUNCTION.

CPUs	RAM (GB)	Utility
1	1	12
2	1	25
3	1	44.2
1	2	19.4
2	2	39.1
3	2	65
1	3	40.3
2	3	44.7
3	3	78

way a single consumer tries to gain happiness by purchasing goods and services.

The use of action, goal and utility function policies in autonomic systems has been studied in more detail by Kephart and Walsh in a paper titled An Artificial Intelligence Perspective on Autonomic Computing Policies [14].

III. Two-Level System of Independent Autonomic Elements

Walsh et al. [15] show how utility functions can be used effectively in autonomic systems by means of a data center scenario. They present a two-level system of independent, interacting agents, which are generally called autonomic elements.

A. Overall Architecture

The architecture consists of a single Resource Arbiter and a number of Autonomic Environments. Each Autonomic Environment consists of an Application Manager, a router and servers as depicted in Figure 1. The Resource Arbiter and the Application Managers both do resource allocation at their own respective levels. The Resource Arbiter is not concerned about the internal workings of the Application Environments, it only allocates resources according to the data it receives from the Application Managers.

The Resource Arbiter is quite a general piece of software in this architecture. Even if we introduce entirely new kinds of Application Environments to the data center, we will not have to alter the Resource Arbiter component in any way because it only expects to receive resource-level utility functions from the Application Managers. The resource-level utility function $\hat{U}_i(\mathbf{R}_i)$, which is sent to the Resource Arbiter, is basically a table that maps the possible resource allocations to their utilities. $\mathbf{R}_i$ is a vector that specifies the resources allocated to the environment i. Its components specify each individual resource, such as the number of servers allocated etc. As an illustrative example, Table I presents a possible resource-level utility function in an autonomic system where CPUs and RAM are the only resources controlled by the Resource Arbiter. CPUs can only be allocated in whole units and memory can only be allocated in pieces of 1 GB. In this example, the resource vector $\mathbf{R}_i$ is of the form (N, M) where N is the number of CPUs and M is the amount of memory allocated. By sending the resource-level utility function the Application Manager basically tells the Resource Arbiter how important it is for it to gain resources. As time passes, the importance may change and, as a result, the resource allocations change accordingly. Traditionally many computer systems have been built with a lot of excessive power just to enable them to survive peaks in demand. However, dynamic resource allocation provides our system with great flexibility and helps us not to waste computing power.

E.g., a company can use a single cluster of computers for running a website for its customers and a database for its business administration unit so that the customers have a higher priority. Self-optimization guarantees that possible heavy database transactions made by the company's employees will not slow down the website if there are customers using it. Research shows that the typical user will tolerate at most 2–10 seconds of delay when loading a website [16]. After becoming frustrated with the delay, the user may well move to a competitor's site. Clearly it is very beneficial to have a flexible optimization architecture that both prevents customers from having to suffer long delays and is cost-effective so that the excessive computing power is used for something useful even when it is not needed for serving customers.

It is the job of the Resource Arbiter to maximize the global utility $\sum_i \hat{U}_i(\mathbf{R}_i)$ by distributing the system's resources in a way that produces the optimal result. In a system of n Autonomic Environments, the resource allocation is of the form $\mathbf{R}^* = (\mathbf{R}_1, \mathbf{R}_2, \ldots, \mathbf{R}_n)$ where $\sum_i(\mathbf{R}_i) = \bar{\mathbf{R}}$. $\bar{\mathbf{R}}$ indicates the total quantities of resources available. The Resource Arbiter periodically recomputes $\mathbf{R}^*$ so that $\sum_i \hat{U}_i(\mathbf{R}_i)$ yields a utility as high as possible. This is generally an NP-hard discrete resource allocation problem. There is a variety of standard optimization algorithms that can be used to solve it. One way is to use mixed-integer programming [17].

In order for the Resource Arbiter to dynamically allocate resources, the Application Managers send in new resource-level utility functions when they decide that there is enough reason to do so, i.e., when the utility function has changed considerably. The Resource Arbiter may also request the Application Managers send new data.

B. Application Manager

The Application Managers are at the lower level of the architecture. They have to be well aware of the details of their own Application Environments since they must be able to do low-level tuning of control parameters. The Application Environments in a single autonomic system can be very different from each other with respect to the kind of applications they are running. Their respective utility functions are used to map the Application Environment's state into a common currency.

The state of a system can be described as a vector of attributes, such as the number of CPUs, the amount of memory, and the amount of network bandwidth the system has been allocated.

The utility function for environment i is of the form $U_i(\mathbf{S}_i, \mathbf{D}_i)$ where $\mathbf{S}_i$ is the service level vector in i and $\mathbf{D}_i$ is the demand vector in i. The components of these vectors can be, e.g., the average response time and the average throughput for multiple different user classes. The goal of the entire autonomic system is to continually optimize $\sum_i U_i(\mathbf{S}_i, \mathbf{D}_i)$, i.e., the sum of the utility functions of all Autonomic Environments, so that most resources are given to those environments that need them the most. On the other hand, the goal of a single Application Manager i is to optimize $U_i(\mathbf{S}_i, \mathbf{D}_i)$ while it is given a fixed amount of resources. The Application Manager can only use the resources that the Resource Arbiter has allocated to it, so it will have to decide how to use those resources as effectively as possible. This is done by adjusting the control parameters of the application in question. In fact, the Application Manager can do its own small-scale resource-allocation for different transaction classes at this level. E.g., some users can be given priority in certain operations.

This is exactly why it is beneficial to have a two-level architecture instead of a centralized one where the Resource Arbiter would do all the work. The Resource Arbiter would have to be updated every time a new Application Environment is introduced to the system, and it would have to be aware of all the details. The resulting software would be quite bloated compared to the one needed in the two-level architecture. Different types of optimization require different time scales, which is also neatly handled by the two-level architecture. The Application Managers can do optimization on a time scale of seconds or anything that is suitable for them. The Resource Arbiter typically works on a time scale of minutes.

Figure 2 shows the inner composition of an Application Manager and some surrounding components. The figure illustrates how information flows inside the Application Manager and how it flows between the Application Manager and the external components. The area with a darker background represents the Application Manager. The rectangular objects inside it are its modules and the cylindrical objects represent the knowledge that the Manager maintains. Since we are concentrating on a single Application Manager here, we will abandon the i subscripts at this point.

The area inside the rounded rectangle represents the Application Environment, which contains the Application Manager, servers and a router. The Resource Arbiter is the only component outside the Application Environment. Of course there are other Application Environments as well but we do not have to pay attention to them here since they only interact with the Resource Arbiter and not directly with each other.

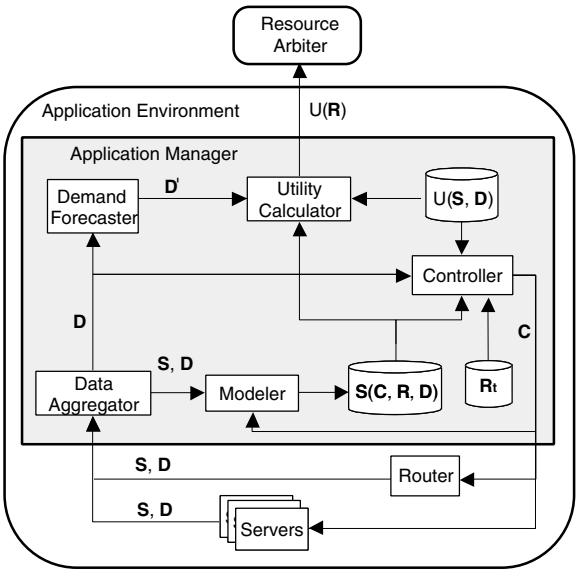

Fig. 2. The modules and data flow in an Application Manager. Symbols: $\mathbf{S}$ = service level/service model, $\mathbf{D}$ = demand, $\mathbf{D}'$ = predicted demand, $\mathbf{C}$ = control parameters, $\mathbf{R}_t$ = current resource level, U = utility function [15, Figure 2].

As we have already described earlier, the Application Manager sends the resource-level utility function to the Resource Arbiter. Next we will describe how the Application Manager is able to construct the function.

The Data Aggregator module receives a continual flow of raw measurement data from the servers and the router. This includes the service data $\mathbf{S}$ and the demand data $\mathbf{D}$. The Data Aggregator uses some method to aggregate the data into a more suitable form. It can, e.g., calculate their average values over a suitable time window.

The Data Aggregator sends the refined demand $\mathbf{D}$ to the Demand Forecaster whose job is to provide an estimate of the average demand in the future. The Demand Forecaster, in turn, passes the predicted demand $\mathbf{D}'$ on to the Utility Calculator. The forecasting is necessary because the demand can suddenly peak for a relatively short period of time. It is hardly a good idea to do reallocation of resources among different Application Environments just because the demand rises (or lowers, for that matter) considerably for a brief moment in time. The shifting of resources can be a time-consuming operation and doing it unnecessarily may only lower the system's performance. The Demand Forecaster has to take historical observed demand $\mathbf{D}$ into account when it decides on the estimated future demand $\mathbf{D}'$.

In addition to sending the demand $\mathbf{D}$ to the Demand Forecaster, the Data Aggregator also sends the refined demand $\mathbf{D}$ and the refined service level $\mathbf{S}$ to the Modeler module.

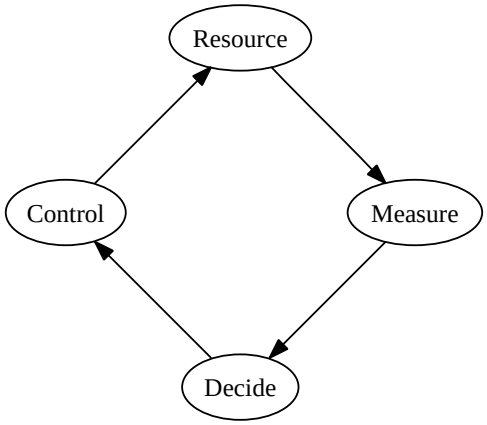

Fig. 3. Control loop.

The Modeler's task is to build a model $\mathbf{S}(\mathbf{C}, \mathbf{R}, \mathbf{D})$, which is basically a function that maps a set of control parameters $\mathbf{C}$, a resource level $\mathbf{R}$ and a demand level $\mathbf{D}$ into the service level that will be acquired with the specified conditions.

The Modeler gets the control parameters $\mathbf{C}$ from the Controller module. The Controller continually adjusts the control parameters that it also sends to the router and the servers. As can be seen from Figure 2, the Controller receives input from many directions. It receives the demand $\mathbf{D}$, the service model $\mathbf{S}$, the current resource level $\mathbf{R}_t$ and the service-level utility function $U(\mathbf{S}, \mathbf{D})$ as input and uses the data to determine the suitable control parameters $\mathbf{C}$ for the current situation. More formally we can say that the Controller aims to find the control parameters $\mathbf{C}$ so that $U(\mathbf{S}(\mathbf{C}, \mathbf{R}_t, \mathbf{D}), \mathbf{D})$, where $\mathbf{R}_t$ is the current resource level, yields its maximum value. The control parameters may include configuration options for the actual application the servers are running or different kinds of settings for the servers' operating systems. The control parameters can contain basically anything because they are application-specific and not restricted by the architecture.

The Utility Calculator is the module that actually generates the resource-level utility function $\hat{U}(\mathbf{R})$ that is sent to the Resource Arbiter. It receives the predicted demand $\mathbf{D}'$, the service-level utility function $U(\mathbf{S}, \mathbf{D})$ and the service model $\mathbf{S}(\mathbf{C}, \mathbf{R}, \mathbf{D})$ as input. The resource-level utility function is calculated with the formula

$$\hat{U}(\mathbf{R}) = U(\mathbf{S}(\mathbf{C}^*, \mathbf{R}, \mathbf{D}'), \mathbf{D}') \quad (1)$$

for all possible resource levels $\mathbf{R}$ where $\mathbf{C}^*$ is the optimal set of control parameters for the resource level $\mathbf{R}$. Note that the optimal control parameters may be different for different resource levels, which means that the optimal control parameters must be recomputed for all possible resource levels. Also we must use the predicted demand $\mathbf{D}'$, which was received from the Demand Forecaster, instead of the current demand $\mathbf{D}$.

The behavior of an Application Manager fits well into a general concept of autonomic computing called the control loop [1], which is illustrated in Figure 3. If we compare it to Figure 2, we can see the similarities. The measuring phase is represented by the router and the servers sending measured service and demand data to the Data Aggregator and the Aggregator passing the refined measurement data on to the other modules. The Controller and the Utility Calculator make decisions based on the measured data by computing the appropriate control parameters and the resource-level utility function. Resource allocation is partly done by the Controller as it sends the control parameters to the router and the servers, and partly by the Resource Arbiter when it makes decisions based on the resource-level utility functions it receives from the Application Managers.

IV. Prototype System

Walsh et al. present a prototype implementation of their architecture running on a cluster of four servers [15]. One of the servers is dedicated to the Resource Arbiter and two Application Managers, and the other three are used as resources. The servers are running Red Hat Enterprise Linux Advanced Server and the resource servers have WebSphere and DB2 installed. First one of the two different services that the system is running is a simulation of a web-based electronic trading platform requiring high availability and the second one is a batch process with no need for fast responsiveness. The three servers are the only resources that the system has and they are allocated in whole units to the different Application Environments. The main idea is that when the web server is getting only few hits per time unit, it is allocated maybe only one server and the batch process gets two, but when the demand rises the Resource Arbiter allocates more servers for the web server.

The prototype system is implemented using Unity [18], a general software architecture for autonomic computing systems developed at IBM Research. The whole framework is written in Java.

We call the Application Environment running the web server **A1**, and the Application Environment running the batch process **A2**. In this system $\mathbf{S}$, $\mathbf{D}$ and $\mathbf{R}$ are all single-valued, so we can use a scalar notation instead: S, D and R. The service-level utility function for **A1** is defined solely in terms of the average response time S_1 of the customer requests. This means that the demand is ignored by U_1 in this implementation for the purpose of simplification, i.e., $U_1(S_1, D_1) = U_1(S_1)$.

The customer demand D_1 is generated by repeated requests for the login web page at a variable rate. Each time the page is accessed, the application makes some random database queries and displays some information to the client to simulate a real trading platform. To realistically simulate periodic and bursty web traffic, a time-series model developed by Squillante et al. [19] is used to reset the demand generated by the transactional workload every ~ 5 seconds.

A1 uses a simple system performance model $S_1(\mathbf{C}_1, R_1, D_1)$ and the service-level utility function $U_1(S_1)$ to estimate the resource-level utility function $\hat{U}_1(R_1)$ for each possible number of servers R_1, i.e., 1, 2 and 3. We

are able to simplify $S_1(\mathbf{C}_1, R_1, D_1)$ to $S_1(R_1, D_1)$ because the control parameters of the servers are held constant.

Before doing the main experiment, a performance model was obtained by measuring the average response time at each of several values of D_1 for 1, 2, and 3 servers. Each data point was sampled for 15 minutes so that there were enough material to calculate averages over several hundred to a few thousand transactions, and all non-sampled points were generated by linear interpolation. Again, for simplification, the Demand Forecaster simply returns the current demand.

The Application Environment **A2**, which is running the batch process, is really simple. The service level S_2 is measured solely in terms of the number of servers R_2 allocated to **A2**. The utility function is a simple increasing linear function. The values returned by the utility function for the Application Environment **A2** are notably lower than those returned by the utility function for **A1**, which means that the batch process is given lower priority.

The test runs of the prototype system have been promising. A detailed walk-through of a single test run can be found in the original paper [15] with a number of graphs describing the state of the system as it operates.

V. Conclusion

Utility functions provide a good way to hide the internal complexities of different applications when comparing their utility with each other. There needs to be an easy way for human administrators to specify high-level instructions for the construction of utility functions. Human administrators usually know which systems are the most business-critical under certain circumstances. If an autonomic computing system has similar information on the importance of different applications it is running, it can to well-informed decisions of resource allocation under varying conditions on behalf of humans.

The two-level optimization architecture described here seems to be usable but it will still require further studying. The prototype implementation is quite simple compared to what the architecture is capable of. The architecture has been further studied by the same group that introduced it in a more recent paper [20]. In the paper they compare a queueing-theoretic performance model and model-free reinforcement learning as methodologies for estimating the utility of resources.

References

[1] A. Ganek and T. Corbi, "The dawning of the autonomic computing era," *IBM Systems Journal*, vol. 42, no. 1, pp. 5–18, 2003.

[2] D. Patterson, "A new focus for a new century: availability and maintainability performance," *Keynote speech at USENIX FAST, January*, 2002.

[3] Ubuntu Linux. Canonical Ltd. [Online]. Available: http://www.ubuntu.com/

[4] Mac OS X. Apple Inc. [Online]. Available: http://www.apple.com/macosx/

[5] G. Malkin, "Routing Information Protocol RIP version 2. Internet Engineering Task Force," November 1998. RFC-2453, Tech. Rep.

[6] J. Moy, "RFC2328: OSPF Version 2," *Internet RFCs*, 1998.

[7] Y. Rekhter, T. Li, and S. Hares, "RFC 4271, A Border Gateway Protocol 4 (BGP-4)," 2006.

[8] J. Abbate, *Inventing the Internet*. MIT Press, 1999.

[9] Optimization. Merriam-Webster's Online Dictionary. [Online]. Available: http://www.m-w.com/dictionary/optimization

[10] I. Sutherland, "A futures market in computer time," *Communications of the ACM*, vol. 11, no. 6, pp. 449–451, 1968.

[11] W. Wang and B. Li, "Market-based self-optimization for autonomic service overlay networks," *Selected Areas in Communications, IEEE Journal on*, vol. 23, no. 12, pp. 2320–2332, 2005.

[12] T. Kelly, "Utility-directed allocation," *First Workshop on Algorithms and Architectures for Self-Managing Systems*, pp. 2003–115, 2003.

[13] R. Das, I. Whalley, and J. Kephart, "Utility-based collaboration among autonomous agents for resource allocation in data centers," *Proceedings of the fifth international joint conference on Autonomous agents and multiagent systems*, pp. 1572–1579, 2006.

[14] J. Kephart and W. Walsh, "An artificial intelligence perspective on autonomic computing policies," *Policies for Distributed Systems and Networks, 2004. POLICY 2004. Proceedings. Fifth IEEE International Workshop on*, pp. 3–12, 2004.

[15] W. Walsh, G. Tesauro, J. Kephart, and R. Das, "Utility functions in autonomic systems," *Autonomic Computing, 2004. Proceedings. International Conference on*, pp. 70–77, 2004.

[16] D. Galletta, R. Henry, S. McCoy, and P. Polak, "Web site delays: How tolerant are users?" *Journal of the Association for Information Systems*, vol. 5, no. 1, pp. 1–28, 2004.

[17] G. Nemhauser and L. Wolsey, *Integer and combinatorial optimization*. Wiley-Interscience New York, NY, USA, 1988.

[18] D. Chess, A. Segal, I. Whalley, and S. White, "Unity: experiences with a prototype autonomic computing system," *Autonomic Computing, 2004. Proceedings. International Conference on*, pp. 140–147, 2004.

[19] M. Squillante, D. Yao, and L. Zhang, "Internet trac: Periodicity, tail behavior and performance implications," *System Performance Evaluation: Methodologies and Applications. CRC Press, August*, 1999.

[20] G. Tesauro, R. Das, W. Walsh, and J. Kephart, "Utility-Function-Driven Resource Allocation in Autonomic Systems," *Autonomic Computing, 2005. ICAC 2005. Proceedings. Second International Conference on*, pp. 342–343, 2005.