

Synthesis of application-level utility functions for autonomic self-assessment

Giuseppe Valetto, Paul deGrandis & Dale Seybold, Jr.

Drexel University, Department of Computer Science

Tel. +1 215 895-2669

Fax +1 215 895-0545

valetto@cs.drexel.edu paul.degrandis@drexel.edu drs66@drexel.edu

This is an extended version of the paper “*Elicitation and Utilization of Application-level Utility Functions*”, published in the Proceedings of the 6th International Conference on Autonomic Computing (ICAC 2009), Barcelona, Spain, June 2009.

Abstract We present a non-analytic approach to self-assessment for Autonomic Computing. Our approach leverages utility functions, at the level of an autonomic application, or even a single task or feature performed by that application. This paper describes the fundamental steps of our approach: instrumentation of the application; collection of exhaustive samples of runtime data about relevant quality attributes of the application, as well as characteristics of its runtime environment; synthesis of a utility function through statistical correlation over the collected data points; and embedding of code corresponding to the equation of the synthesized utility function within the application, which enables the computation of utility values at run time. We employ a number of case studies, with their results and implications, to motivate and discuss the significance of application-level utility, illustrate our statistical synthesis method, and describe our framework for instrumentation, monitoring, and utility function embedding/evaluation.

Keywords: *Application utility, utility functions, instrumentation, non-analytic self-assessment.*

Abbreviations: Aupy: Application utility in Python; BSD: Berkeley Software Distribution; COTS: Common Off-The-Shelf; CPU: Central Processing Unit; FTP: File Transfer Protocol; JDK: Java Development Kit; IP: Internet Protocol; IT: Information Technology; MAPE-K: Monitor, Analyze, Plan and Execute – Knowledge; NTP: Network Time Protocol; QoS: Quality of Service; RFC: Request For Contribution; RL: Reinforcement Learning; SSH: Secure Shell; VoIP: Voice over IP

Introduction

Autonomic software systems should be able to adapt to a wide range of conditions, including unforeseen and unexpected events that may occur as the system is operating. Such events may often cause the most critical in-the-field faults, since, by definition, they have not been accounted for, or discovered, during the various phases of the engineering process. It is therefore arguable that the significant additional cost and complexity incurred in designing and implementing autonomic mechanisms for a software system may at times be justifiable only in case they provide some level of protection from unpredictable conditions. However, self-adaptive facilities are often designed to cover only certain classes of expected or likely conditions, and *designing for the unexpected* remains one major challenge for the engineering of autonomic systems.

This paper presents an approach to attack that challenge, related to the area of self-assessment, a critical pre-requisite for being able to take decisions

upon any adaptation that may need to be enacted. Self-assessment requires that the autonomic system possesses means to recognize at least some salient properties of its operational state, and predicate upon that state, in order to evaluate whether it is satisfactory, or should rather be changed.

Many current techniques pursuing self-assessment are – in strict accord with the MAPE-K reference architecture for autonomic systems [13] - analytic in nature: for example, they may monitor the occurrence of specific runtime conditions within the system, as in [2], and try to diagnose their root cause, as in [19].

We propose a *non-analytic approach* that leverages the concept of application-level utility, an abstraction that captures in a single scalar value the salient properties of an application, such as health, value, or quality of service, with respect to its operational requirements. In practice, our approach for utility-based self-assessment is three-phased. Firstly, by means of instrumentation, we collect runtime data at two distinct levels: an array of measures sampling the environmental operating conditions for the application; and, in parallel, another set of quantitative characteristics of application-level quality and performance. We then synthesize a utility function for the application, through the statistical correlation of the data collected at those two levels. Finally, we turn utility into an intrinsic characteristic of the application itself, by using meta-programming techniques to embed the utility function in its runtime libraries. That enables the online evaluation of application utility, which lays an objective basis for self-assessment and, in turn, decision-making on the need and the scope of autonomic adaptations.

In the remainder of this paper, we first discuss the motivation and implications of application-level utility for autonomic self-assessment, also employing examples originating from our various experiments in this area. We then describe our statistical method for synthesizing utility functions on the basis of runtime monitoring of the application. We also describe a framework we have developed to enable the synthesis and embedding of utility functions in application code, called *Aupy*, and present some case studies in which we applied *Aupy* in different domains, with their results. We discuss the engineering lessons we have learned while building *Aupy*, with implications in particular to equipping legacy applications with utility functions. Finally, we outline further research opportunities on application-level utility for autonomic computing. But, first, we briefly review the concepts of utility and utility functions, and discuss their meaning and usage in Autonomic Computing and in our own work.

Utility in autonomic computing

The concept of utility has its roots in Utilitarian philosophy, and its earliest application perhaps in the field of microeconomics, where it indicates the satisfaction generated from owning or consuming a good or service. Computer Science has borrowed the term, and assigned similar semantics to it. For example, in Artificial Intelligence, *expected utility* has been indicated as a unifying principle for decision-making rules [6], thus highlighting how utility expresses the level of satisfaction of an agent's goals. Utility has been employed in many other computing areas, and even just an overview would

exceed our space limitations. To offer just a few examples, in communication systems utility can be used to optimize the consumption [3], or pricing [31], of networking resources. In distributed computing, it can for instance arbitrate the allocation of shared data and other resources among concurrent processes [15].

In all cases, it is essential to be able to make the utility abstraction operational. That translates into devising a utility function that can capture the definition - and compute values of - utility in the context at hand. A utility function can be expressed in the simple form:

$$U = f(V) = f[s_0, \dots, s_n]$$

which denotes how utility depends on a vector V of attributes (or *state variables*) that capture the state of the system and its environment; the function f maps that - possibly large - vector characterizing the running system to a single scalar value U , conventionally in the $[0,1]$ range.

In Autonomic Computing, as discussed in [14], utility functions become particularly attractive because of their ability to quantitatively attribute points in the system state space. They thus provide an objective and quantitative basis for sound, automated decision-making, and they can tie those decisions to high-level conceptual requirements, concerns and goals. A utility function can for example guide automated agents to devise or compare self-optimization strategies, in order to ensure certain levels of performance, or even assign monetary value to the operation of a system. A common self-optimization context in which utility has been applied is the automated management of large, multi-application computing infrastructures. Walsh et al. [29], and Das et al. [7], among others, have shown how utility functions can guide the allocation of resources in a data center, based on the evaluation of the (economic) value of the resources being provisioned for hosted applications, and of those applications for their owners or users. A different self-optimization scenario is proposed by Cheng et al [5], in which utility evaluation enables the choice among alternative strategies for adapting the configuration of the architecture of a software system at runtime.

Notice that in many current approaches, including the ones above, the definition of the utility functions is not a direct concern for the autonomic system itself. Utility functions are often manually elicited with the help of domain experts or application users [11]. In those cases, we talk about *subjective utility*, as it reflects the goals and the values of those human subjects. For instance, in previous work co-authored by the second author, the elicitation of subjective utility is used to characterize a communication application [25]. In Autonomic computing, however, it would be desirable to move towards *objective utility*, and from elicitation to (at least partially) automated synthesis, since that leads to utility functions that faithfully fit the recorded usage and intended purpose of the target system. In certain contexts, for instance the management of IT infrastructures, objective utility functions can be derived from contractual agreements, such as SLAs [24]. Also, some automated approaches based on negotiation among multiple parties have been proposed, limited to the resource allocation scenario [1] [20].

Our work on Aupy is different in purpose as well as scope. Its focus is upon the non-analytic self-assessment of a single application, or even a single

recognizable service or task executed by that application. Some other non-analytic approaches exist, based, e.g., on clustering [9] [22]. They are able to distinguish between anomalous and common behavior of an application; however, they cannot quantitatively attribute application states that correspond to either anomalies or normal operation. Our application-level utility functions can automatically compute a value for the current state of operation of the application or task, denoting, for instance, its health or performance. In that sense, our work is perhaps most similar to that of Sterritt and Bustard [26], which also establishes a framework for quantitative (although discrete) self-assessment, by means of a multi-level scalar, called a “pulse”. However, in that work the definition of pulse levels for an application, with their semantics and thresholds, is left to the implementer. We are interested, instead, also in the objective synthesis of utility functions that are tightly linked to observable runtime properties.

To that end, we employ a statistics-based approach that is predicated upon the identification of a set of observable characteristics of the application at hand, as well as its runtime environment. The selection of those characteristics is a critical step that currently relies upon expert knowledge. For that reason, we have found that our approach applies particularly well to certain classes of applications, which have some intrinsic coherence of purpose in their feature set, such as protocol-based communication systems, or infrastructure-level services and facilities. In the case of very heterogeneous systems, like enterprise applications, which may offer a large number of very diverse features to their clients, it becomes more complicated to conceive a uniform set of salient characteristics that is representative of the system’s utility. However, in those cases, our approach is fine-grained enough to enable the self-assessment of some given specific tasks of interest. The interested reader can find a more exhaustive discussion on applicability in a later Section of this paper, delineating our experience with our Aupy framework.

Application-level utility

Attributing utility to application tasks

As discussed above, the concept of utility within the domain of Autonomic Computing is applied largely at a “macro” level, that is, to guide policies for the automated management of large-scale computing infrastructures that may host many diverse applications. We maintain that it can be at least equally important when applied at a more fine-grained, “micro” level, which pertains to the assessment of how well a single application is able to perform the task (or tasks) requested by its users, or by other systems that behave as the application’s clients. In that respect, utility must reflect the value offered by the services made available by the application to those users/clients, and must thus be defined from their point of view.

In order to elucidate the potential of utility functions for application self-assessment, we describe two preliminary experiences we have had in this area. The first one has to do with assessing the utility of an interactive communication system, based on the perceived support it provides to its users in fulfilling various collaborative tasks. The second one, instead, discusses how a system for distributed parallel computation can benefit from

being aware of the utility of the various working subsystems it is composed of. Those experiences offer a rationale for self-assessment via application-level utility, and, more specifically, provided us with the motivation for developing a framework like Aupy.

Utility of IP teleconferencing

We briefly discuss here the set-up and outcome of a case study that is more thoroughly described in [25]. That case study involved a number of volunteer users of an IP teleconferencing system, and required them to try to perform various tasks that required pairwise remote communication, mediated through the IP conferencing system.

User tasks varied from simple communication exchanges, to time-sensitive exchanges, to more articulated collaborative tasks. Those tasks were carried out by pairs of volunteers on top of a framework that enabled throttling the basic parameters of the network channel upon which IP conferencing occurred, thus allowing to create a set of tests for each task. A test would correspond to a combination of bandwidth, latency, and packet loss conditions assigned to the network channel. For each test and task, the collection of application-level data was carried out by the users themselves, who subjectively rated and recorded their ability and satisfaction in carrying out that particular task with the support of the IP teleconferencing application. That data on *user-perceived utility* was then correlated to the metrics of the network channel used by the application, in order to obtain a mapping of how the utility for each given task depended on the environment-level conditions.

For each task, repetitions of test with different network conditions and different pairs of users yielded thousand of data points. A statistical analysis on the outcome of those experiments determined that the sensitivity of each task to the varying network conditions is different. For instance, increases in network latency have a bigger effect on the (lack of) user ability to perform satisfactorily time-constrained tasks, or collaborative tasks that require substantial amounts of interaction and back-and-forth communication between users. Bandwidth and packet loss, instead, are dominant in tasks that require simpler communication exchanges, such as asking a trivia question and getting an answer. As a result, different utility functions were statistically devised for our IP teleconferencing application, *on a per-task basis*, by using the data maps provided by the experiments, and curve-fitting the values of perceived utility to the three environment-level parameters considered.

This study not only highlighted the task-dependent nature of application-level utility, but also offered a method to devise operational utility functions for a given application task based on monitoring and statistic analysis.

Utility of MapReduce workers

MapReduce is a paradigm for distributed parallel processing of large amounts of data that has been pioneered by Google [8]. It allows the harnessing of the resources of a large number of commercial-grade machines (called *workers*, or *worker nodes*) in a very simple way, by segmenting the original data set in chunks that can be processed independently, assigning

those chunks to the worker nodes for the required processing, and then merging the partial results they produce back together. The segmentation, assignment and merging phases of a MapReduce job outlines above are orchestrated by a *master node*, and may take advantage of large-scale distributed storage systems for the input data set, the intermediate results and the final output, such as the Google File System [10]; however that does not represent a pre-requisite.

Since it does not require special-purpose hardware, or complex parallel system software, MapReduce has recently become quite popular for all kinds of data-intensive applications that do not require interaction among the processing tasks running on worker nodes. The main tenets of MapReduce are also in line with the vision of utility computing [23], and possibly amenable to be implemented as a service in a cloud, as discussed for instance by Jin and Buyya [12].

One known issue with this paradigm is its “endgame”, since a MapReduce job can be completed only as quickly as the completion time of its slowest worker node. Strategies for mitigating that issue exist: for example, [8] describes how, if some workers are unreasonably delaying job completion, whereas the rest of the worker nodes have already finished their part, the master can abort the processing of the late workers, and re-assign the corresponding chunks to some other available nodes. That helps in those situations in which some workers are “hanging”, have crashed, or have simply become excessively slow for some reason, and it has been shown to improve the endgame of MapReduce systems. However, strategies of this kind only react to a problem once it has already occurred, and their benefits are directly tied to the timeliness and accuracy of the problem detection mechanisms put in place.

We have thus started to explore how the concept of utility of worker nodes, as seen from the point of view of the master, can benefit the execution of MapReduce jobs. The basic idea is that a function continuously computing a value of utility for each worker node for each job – based on the worker’s past and current behavior - could become part of the assignment policy of the master node; it could then help at the same time to improve completion time for the various jobs, and optimize the usage of worker resources, based on their efficiency in satisfying the specific processing needs of different jobs. To examine this hypothesis we have developed a MapReduce simulator, and carried out a series of experiments on MapReduce execution.

Our in-house simulator encompasses the concepts of master, workers, and network links for data transfer, and its parameters include number of workers, data chunk size, total data set size, bandwidth for network links, processor speed, maximum number of simultaneous connections that the master can sustain, etc. The bandwidth of network links varies over time. Similarly, to simulate a set of workers with heterogeneous capabilities due for example to different hardware, each worker is set with a *processing speed* parameter for each job, which varies in a +/- 50% range with respect to a nominal speed. Each worker also has a variable load value, which represents the percentage of the processor that is unavailable because it is being used by other programs. Therefore, the computational power available to each worker for a given job is determined by:

$$\text{Computational power} = \text{processing speed} * (1-\text{load})$$

The utility function that we have used depends not only on the factor above, but also on the current bandwidth of the network link available to each node. This determines the time needed for transferring a chunk, or work unit, to the worker, as well as transferring the worker's results back (we have assumed in our simulations that the size of the work unit in input and the size of the processing results are similar). Therefore, the total time for a worker node to complete a work unit is:

$$\text{Completion time} = 2 * \text{Unit Size} / \text{bandwidth} + \text{Unit Size} / \text{Computational power}$$

We can then rearrange that equation to express the utility of a worker from the point of view of the MapReduce master, that is, in terms of the amount of data a worker can go through per unit of time. That yields the following utility function:

$$U_{\text{worker}} = \frac{\text{Unit size}}{\text{Completion time}} = \frac{1}{\frac{2}{\text{bandwidth}} + \frac{1}{\text{Computational power}}}$$

Workers calculate their utility periodically (for instance, every minute), and send that value to the master. The master records the utility of all of the workers involved in a job, as well as the state of all work units (i.e., whether they are unassigned, assigned, or finished), and which work units are assigned to each worker. Notice that the communication of utility values to the master also serves as a heartbeat signal. The master timestamps the last message it received from each worker, and if utility data is not refreshed for a certain amount of time, then the worker is considered "down", and its utility is set to 0 until further notice by that worker.

The MapReduce master exploits its knowledge of utility to rank worker nodes, and assign more work to those nodes with higher utility values. It can also re-assign chunks from workers whose utility has dropped, to other workers that rank better. Finally, since worker utility is expressed in terms of unit size and completion time, the total amount of data (i.e., the number of chunks) being assigned to each worker can be tuned in such a way to keep workers busy as much as possible.

We have compared such a utility-aware assignment policy, to a simple policy that assigns chunks uniformly to all available nodes, and implements the endgame re-assignment optimization we mentioned above, in case some workers hang or crash.

In a series of pairwise simulations, we have noticed significant improvements with regard to both job completion times, and the overall efficiency of the MapReduce system, expressed as the percentage of workers that remain busy throughout the duration of the job. Table 1 reports the comparison of completion times data for three different experimental set-ups: the first experiment was performed with 100 workers, a unit size of 15 MB, total data size of 72,000 MB, 600 KB/s average network speed, and 60 KB/s average processing speed; the second experiment was the same as the first, except that it had 300 workers, and a total data size of 300,000 MB. In the third experiment the total data size was changed to of 600,000 MB, and the average processing speed was changed to 40 KB/s.

Furthermore, Figures 1 and 2, provide a comparison of efficiency data related to workers utilization levels for the first experiment described above. In the

plots on the left-hand side, the green line shows the proportion of workers that are busy over the duration of the job, whereas on the right-hand side we show a graph of the distribution of worker utilization broken into 5% increments. As evident from those Figures, the utility-aware assignment policy is superior, and ensures very high, constant utilization of worker resources, whereas with the simple assignment policy the utilization of workers varies widely between the 45% and 95% percentiles. Also the graphs for our experiments # 2 and 3 present similar results.

We consider this kind of results particularly encouraging to continue exploring application-level utility in the context of MapReduce distributed computing. Therefore, we have started to look into ways in which we can confirm these results through a more extensive experimental plan: we are considering using a third-party MapReduce simulation platform for independent validation, such as MRPerf [30], as well as taking advantage of MapReduce capabilities made available by cloud infrastructures, such as the Amazon Elastic MapReduce¹.

Lessons learned

The two experiences described above offer a number of different insights on the issue of application-level utility. The latter suggests the potential of this concept for achieving self-assessment, and, in turn, being able to take informed adaptation decisions, without requiring deep analytic capabilities, even in the case of significantly complex, large-scale application environments. In the case at hand, the utility values of a worker are kept distinct for any different jobs running concurrently on the MapReduce infrastructure; the master is thus able to tune its policies on a per-job basis, without any job-specific knowledge, or a need to diagnose the workers' state. It is noticeable, however, that in that case study, we have used a pre-defined definition of worker utility, and its corresponding utility function.

The experience gained with the IP teleconferencing case study, instead, has improved our understanding of how utility can be linked to environment-level conditions that impact the ability of the application to perform its tasks. That insight, in turn, has enabled us to devise a method for defining utility functions starting from a set of observable runtime parameters, using statistical tools. That method is the basis of the automated process for the synthesis of utility functions we have adopted for Aupy, and is at the core of the design of the Aupy framework, as discussed in the following Sections.

Synthesizing application-level utility functions

In order to attribute utility semantics to a given application, we conduct a process of correlation and statistical fit between a vector of attributes of the application behavior, and another vector of characteristics of its runtime environment.

That process is based on the statistical utility elicitation technique used in [25] for IP teleconferencing, but we have worked to extend it to cases in which the utility information is not provided by users. Our goal is to move from elicitation to automated synthesis, and substitute user perception with

¹ See <http://aws.amazon.com/elasticmapreduce/>

objective measures of behavioral parameters of the application, as observed, for example, by any client system using that application's services. To achieve that, we instrument the application, and sample a chosen set of salient application-level parameters, in conjunction with environment characteristics. We then try to correlate application-level and environment-level measures: we calculate various moments of the multidimensional graph generated from the environment-level measures, and curve-fit it to the corresponding application-level graph. The equation that produces the best fit becomes our utility function, which allows us to assign a utility scalar to the running application, by simply monitoring the vector of environment- and application-level characteristics included in that equation.

Let us consider, for example, the utility function for downloading a file with an FTP service (FTP utility is discussed in detail as a case study further below). For such a task, the most significant application-level measure of utility, seen from the point of view of the FTP client requesting the transfer, is throughput. That effectively incorporates and abstracts a number of phenomena that occur within the FTP networking environment, such as protocol handshaking and control flow, transfer error, packet loss and retransmission, etc. It also depends on other environmental factors, such as the raw throughput and the latency provided by the network socket, and their fluctuations over the time of the transfer. We observe those environment metrics, as well as throughput as seen from the FTP client, for a large number of test runs. We employ a controlled testbed that enables us to collect at corresponding time intervals those tuples (our data points), as well as vary the running conditions from test run to test run, so that we can crawl systematically through the multi-dimensional operation space constituted by the metrics being observed. We then proceed to synthesize offline the equation for the application-level utility for FTP download, based upon the collected data points.

A critical aspect of our technique is the choice of parameters to be observed, which remains application- or even task-specific. However, an advantage of our approach over analytic techniques for self-assessment is that – once defined – utility functions do not need to monitor any application-specific events and patterns thereof, nor develop any diagnostic capability to prove/disprove hypotheses that are specific to the application and those patterns. A utility-based self-assessment approach may thus require significantly less domain knowledge than most analytic techniques, which is advantageous, since – as observed for example by Tesauro in [27] - the acquisition, modeling and maintenance of domain knowledge in Autonomic Computing is itself an open research issue, and tends to be time-consuming and hard to generalize. Another merit of our approach is that data collection can be fully automated through instrumentation of the application at hand, thus enabling collection in contexts that closely match its natural running conditions. Moreover, once our utility function are synthesized, they are already tied to measures that are inherently observable: therefore, the same instrumentation previously used for the synthesis process can be deployed within a utility-aware version of the application, to compute its utility on the fly, as it operates in the field.

Making software applications utility-aware

We have prototyped a system of code annotations that supports the data collection and synthesis process outlined above, as well as the embedding of code representing the resulting utility function within the runtime of the application, at the same points where the data was collected. Our annotations intercept the program execution upon entering marked functions in the target application. To the target application, those function calls seem semantically identical, however a background process is triggered by the annotation, which monitors and collects the parameters of interest, or calculates utility on the basis of those parameters. For our prototype, we selected Python as the target language, for its advanced introspection mechanisms and its ability to foster meta-level programming functionality. In fact, the name “Aupy” stands for “application utility with Python”.

In the first version of Aupy, our annotations were implemented using a series of Python decorators, associated to method definitions. Those decorators trigger code in the Aupy framework that is used either for monitoring or utility computation.

Table 2 provides a synopsis of those decorators. The `@monitor` decorator is the centerpiece of our system: it can be associated to any method in the application, and can take as an argument a list of functions that are to be executed in the same scope as the original “target” method. Those other functions must be marked with either `@pre_utility`, `@post_utility`, `@utility`, or `@concurrent_utility` annotations, and specify whether their code should be executed - respectively - before the target method, after the target method, before and after the target method (i.e. as a wrapper), or concurrently (that is, in the background by a separate thread, while the target method performs its operations).

These decorated functions do not take any arguments directly, but, since they are in the same programming scope as their target, they have visibility onto its data, and, specifically, its arguments. They can therefore be used to observe the work done by the target method, as part of the synthesis phase of our process, in which we gather runtime data on the operation of the system. Similarly, they can be used - once the synthesis of the utility function has been completed - for computing values of utility based on the observation of the same data.

In a second version of Aupy, we also introduced Python contexts. Annotations via method decorators remain very convenient to make legacy systems utility-aware, since Python makes possible to associate them to methods defined within - and imported from - libraries. Contexts, and the scoping constructs they enable, allow us associate utility not only to entire methods, but also to any block of application code. Therefore, the granularity of our monitoring and utility calculation facilities can become finer at will. The syntax of our utility contexts is as follows (boldface parts highlight keywords and other fixed syntactic elements, whereas parts in angle brackets denote parametric elements of the construct):

```

with Utility.contextUtility (
    <utility function>, <u.f. argument list>, evalFunc=<function>,
    utilityCallback=<callback function>
    utilityCallbacArgs=<array of argument for callback>)
    as status:
        # here follows original target code
    ...

```

By analyzing the construct above, we can discuss some further advantages introduced with contexts. First of all, it is worth noticing that contexts can be combined, by using multiple nested `with` statements, each associated with its own utility function. The ability for an outer “parent” context to decide the parameters of an inner context, and pass information to it, opens the door to sophisticated ways to define and compute utility. For example, it is possible to devise a stepwise method of utility calculation, in which the value returned by the utility function of the outer context becomes one of the arguments passed to the utility function of the inner context. It is also possible to code an escalation of diagnostic mechanisms: in such a case, the utility values computed by the outer context are used by the inner context in order to decide whether to activate a different method of utility calculation, which perhaps works at a different precision level, or takes into account additional data, or is more demanding in terms of computing resources. Also the `status` variable - which is used to denote whether a context and its associated utility function, has been executed correctly - can be leveraged by nested contexts: an inner context has access to that variable, and can decide what needs to be done based upon the information it carries.

Other noteworthy new features are the (optional) evaluation and callback functions. An evaluation function can be used to post-process the raw utility value returned by the utility function; it always returns a boolean value. Evaluation functions serve two main purposes. First of all, they allow a clean-cut separation between code that measures the utility of the application (which resides in the utility function associated with the context), and code that may use and interpret those utility measures, which may be situation-dependent, and should be placed in the evaluation function. That contributes to provide contexts with richer operational semantics.

For example an evaluation function like the one below:

```

def evalAgainstStdDev(u): # u is current utility value
    median = computeMedian(u)
    sd = computeStdDev(u)
    if (u > median + 2*sd) or (u < median - 2*sd):
        return False

    return True

```

can be used to compare the current value of utility to a distribution of utility values recorded in that past, and flag the current value, in case it represents a two-standard-deviations anomaly. In the simplest case, and as a default, the Aupy programmer can also define `bool` as the evaluation function, which represents a pass-through option: in that case, in practice, the utility function itself takes also the role of the evaluation function.

The other main purpose of evaluation functions is to work as a trigger for the corresponding callback function: whenever an evaluation function fails (that is, returns `False`), and in case the context includes the indication of a callback function, that function is invoked with the array of parameters provided in the context definition. The intent behind callback functions is to provide a flexible and practical way to connect the self-assessment capabilities of the Aupy framework with any available provisions for the proactive adaptation of the application. Callback functions could implement a set of adaptations, such as remedies and optimizations, customized for the application at hand, or could represent the entry point of some other comprehensive, generic framework that can be used to effect such adaptations, and can be – in this way – plugged in together with Aupy to construct a full-fledged autonomic solution.

Finally, contexts, analogously to decorators, can have different moments of execution, with respect to the execution of the target code. We have developed, besides `Utility.contextUtility`, which is depicted in the example above and denotes a wrapper, also the `Utility.preContext`, `Utility.postContext`, and `Utility.concurrentContext` constructs.

In the current version of Aupy, which has become an open source project under the MIT license to foster further enhancements and experimentation ², method decorators and contexts co-exist. Their combination supports the retrofitting of legacy systems with utility-aware behaviors, as well as the development of new systems with the same capabilities. That is illustrated in the two case studies that follow and describe our experience and results in working with Aupy.

Case studies

FTP download

The goal of this case study was to validate our approach to synthesis and embedding of application-level utility function, as well as the Aupy annotation framework, within a well-understood scenario. We chose the FTP protocol, since it is a well-known and widely studied system, with a clear principled parameter that can be used to represent the utility of the application, that is, data throughput.

This case study was conducted using a standard FTP client written in Python and conforming to the FTP RFC [21]. The FTP client was instrumented to record its application-level throughput, as well as CPU utilization, and RAM usage. From the networking environment, we recorded raw channel throughput and inter-arrival time, as measured on the socket of the client machine. The tasks we profiled with data collection were the download of one large multi-gigabyte file (in the following, *FTP-bulk* experiment), and the consecutive download of multiple one-megabyte files (in the following, *FTP-small* experiment).

First of all, an annotated version of Python FTP library was constructed. To accomplish that, a subclass of the FTP class in that library was developed, and we used Aupy annotations to mark the subclass methods implementing the FTP commands as code to be monitored. As shown in the example below, the

² See <http://www.aupy.org>

annotated methods simply wrap pass-through statements that invoke the same functionality in the original FTP library.

```
@monitor(None)
def ntransfercmd(self, command):
    return super(SafeFTP,
                 self).ntransfercmd(command)
```

Next, a concurrent utility function was developed, to capture and log data at the application as well as at the environment level.

```
@concurrent_utility
def ftp_utility(self):
    # monitoring code below
    ...
```

The effect of this set of annotations is that, as the monitored transfer command gets executed, Aupy automatically finds all the utility functions defined within the same language scope - in this example only `ftp_utility()` is available - and run them as prescribed by their `@utility` annotation.

Finally, we hijacked the original library and substituted it with the annotated version: that is easily accomplished due to the possibility in Python to import library classes under alternate names, to preserve and maintain control of the namespace. The Python code below overrides the namespace of the original FTP library.

```
# from ftplib import FTP
from aupy_ftp import SafeFTP as FTP
```

The instrumented FTP client application was then run on a test bed, comprised of two Linux endpoint machines and a BSD machine, which utilized the *dummynet* utility³ to simulate a variety of network conditions for our test runs. A single test run is a complete file transfer scenario for a *dummynet* network setup, which is comprised of given values for bandwidth, latency, and packet loss settings. The clocks on all the machines were synchronized before each test run using NTP [18]. A back channel was used for logging and administrative actions, allowing the network channel for the FTP experiments to be fully dedicated to transfers.

We carried out a set of 125 test runs and repeated it three times for both the FTP-bulk and FTP-small experiments. During each test run, measurements of environmental and application attributes were recorded each second. This process allowed us to establish quantitative relationships between those attributes, by aggregating the data and performing their interpolation. Finally, we obtained an equation that estimates client-side FTP application utility purely on the basis of environment-level metrics, i.e., packet inter-arrival time and channel throughput.

Since for this case study our utility definition directly corresponds to a specific principal QoS parameter of FTP (data throughput), the statistical process we employ can be seen as analogous to QoS profiling approaches adopted in adaptive or active middleware, such as QualProbes [17]. However, the goal there is to correlate application-level parameters to one another, as the active middleware is interested in evaluating how intervening upon some parameters under its control may impacts other facets of the application running on top of its platform. Our goal is instead to find out by observation

³ See <http://info.iet.unipi.it/~luigi/dummynet/>

what set of measurable parameters at environment level can be best used to accurately model the quality factors that are chosen as being representative of the application utility.

From the process above, we could clearly see how – like in our previous experience with IP conferencing – the FTP application utility is task-dependent: the equations we synthesized for the bulk-FTP and small-FTP experiments differ significantly, as shown in the formulas below and their coefficients, which are listed in Table 3.

$$U_{\text{bulk}} = a_1 X_1 + a_2 X_2 + a_4 X_4 + a_5 X_5 + b_1 Y_1 + b_2 Y_2 + b_5 Y_5$$

$$U_{\text{small}} = a_4 X_4 + a_5 X_5 + b_1 Y_1 + b_2 Y_2 + b_4 Y_4 + b_5 Y_5$$

Our empirical verification confirmed how estimated utility, i.e., the values of utility computed through the above equations, approximates well observed utility, that is, for this case study, the monitored values of throughput for the FTP client at corresponding data points. We show in Figure 3 and Figure 4 a comparison between observed and estimated utility, using the plots obtained with MATLAB from the FTP-bulk experiment. In those Figures, values of utility are color-coded, with red being highest utility, and deep blue being lowest utility. Those color-coded utility maps are situated within a tri-dimensional space of observable environmental characteristics of the network (bandwidth, packet loss and latency). For simplicity of visualization, the Figures display only some “slices” of that space.

Our next step was to embed the synthesized functions in the FTP client. For that, we simply had to enhance the function `ftp_utility`, which previously simply used to log our monitored metrics, with a call to Python code representing the utility equations. We also carried out further validation, since we continued to independently log the FTP client throughput, to keep the observed vs. estimated comparison active online. The results reported in Figures 3 and 4 were confirmed also in that validation phase.

Log explosion

The goal of this case study was to investigate application-level utility in a domain different from protocol-based applications (such as FTP or VoIP), and to experiment with our technique to solve a real-world problem related to the management of IT infrastructure. The problem at hand regards certain faults that can cause a system to consume a global resource rapidly and unexpectedly. As a consequence, other functioning parts of the system are starved, which usually results in complete system failure. Such a crippling problem can arise for many disparate reasons in a variety of computing systems, in the small as well as in the large.

We have been able to work with one specific form of that problem, occurring within a popular Python-powered Web site (more than 1 million unique views per day): a “log explosion”, which caused the log aggregation and rotation subsystem of the Web site infrastructure to be overwhelmed with hard disk writes, rapidly and at unpredictable junctures. System data on those events was collected and analyzed, to discover that a round-robin scheduler in charge of server-side cache replication within a caching cluster would at times fail and remain “stuck” on a given node. Every attempt the

scheduler would make to move to the next node would fail, resulting in a log entry. Very quickly the log would exhaust disk space, to the point that the site would no longer be able to accept new connections (which requires logging to be functional). Also attempts to log onto the nodes by other means, like an SSH session, would fail, since there would be no room on the disk to log the session information. Existing monitoring software for the site could not detect this problem quickly enough to avoid this complete system failure. We therefore decided to try to use our framework to recognize log explosions as they happen, without having to investigate or resolve the root cause of that faulty behavior. Our hypothesis was that by monitoring the rate and acceleration of log writes, we could come up with a utility function for the logging facility that would tell us when an explosion was about to occur. Our testbed for this experiment was simple. One machine was used to run a clean-room re-engineered version of the faulty round-robin scheduler. The system rotated through logical representations of nodes, one of which was injected to cause a fault. Additionally, in the experiment, log information was also piped to standard output, to watch the system live. It was pre-determined that on every twentieth rotation a fault would be triggered on one of the nodes, causing it to become unresponsive to all scheduler requests. We then embedded Aupy code within the scheduler, to capture the rate of log writes as well as their acceleration rate. Under normal operating conditions, the recorded log writes rate by the scheduler was consistently low, in the order of one write/sec. Our concurrent monitors, implemented as Aupy contexts, were set to monitor at about the same interval. During repeated experiments, when an explosion occurred, log write rates increased two orders of magnitude in the first second, to roughly 720-780 writes/sec, and again two orders of magnitude in the following second to about 15000 writes/sec. In our particular testing environment, hardware limits for disk write access rate were reached within the third second. Acceleration appears therefore to be exponential, which shows the extremity of the phenomenon. Consequently, its representation in terms of a utility function produced a step-wise function dropping from 1 to 0 in the event of an explosion. Whereas we had hoped for a utility profile, which could have possibly provided more preliminary evidence of the build-up of the phenomenon, the step-wise function, once embedded in the system allowed us to recognize log explosions within one second from the triggering of the log explosion phenomenon. We could thus easily put into place a provision to block the round-robin scheduler in time, through a customized callback function, stop its runaway logging, and avoid further clogging of the system and its total failure. We intend to carry out more work on this particular example, to gain further insight on the early assessment of explosive resource consumption problems.

Discussion of Aupy

Our work has provided us with a number of insights on application-level utility for autonomic self-assessment. First of all, we have gained an understanding of a process to synthesize utility functions by monitoring the behavior of applications, and their environment. That process is repeatable and requires only a modicum of application-specific knowledge. Although we

have experience only with carrying out that process in lab conditions, and in part (i.e. the delicate statistical analysis phase) offline, it seems feasible to consider it as a basis for working toward online, automated synthesis of utility functions. We expand on this matter as a possible direction for future research in the next Section.

Our case studies have confirmed that utility at the application level is task-dependent, and a number of consequences descend from that. Since an array of utility functions apply to the same system, and, in some cases, such as our case study on FTP download, even to the same application feature, having synthesized correctly those distinct utility functions is not per se sufficient: one must also have the means to select them opportunistically, based on the context of the task being performed. That has engineering implications, and has become one of the requirements driving our Aupy framework. Our latest prototype allows associating multiple (possibly concurrent) utility functions to the same code block. Moreover, mechanisms based on and enabled by our utility contexts, like their nesting, can also account for conditional selection and composition of our utility functions.

The task-dependent nature of application utility also helps understanding how our method fits naturally well certain specific classes of applications. As mentioned earlier, one prominent such class is constituted by communication-oriented systems, because they are usually based on protocols that formally codify and restrict what the application is intended for. That information is typically sufficient for the extraction of the principal features of the system, and to guide the synthesis of appropriate utility functions for the application tasks that reify those features. Similar considerations often apply to infrastructure-level software systems that - like in our case study on server logging - are characterized by a limited set of specific, self-contained services they provide to the computing environment as a whole. In the case of other classes of applications, e.g., enterprise systems, the synthesis of utility functions across the spectrum of all possible features and user tasks can quickly become unfeasible. In particular, in the absence of automated support in the synthesis process, our approach must necessarily be limited to a subset of those features, typically the ones that domain experts consider the most important or quality-critical.

An advantage of the fine-grained utility abstractions required and at the same time enabled by our framework is a viable approach to comparable, interoperable and composable definitions of “health” for different autonomic applications, which descends from the unit-less and summative properties of utility functions. It is well known how this applies at a macroscopic level, e.g., for multiple applications operating within the same provisioning infrastructure (leading to a view of utility for the whole computing environment, as discussed in [29]). It applies as well at the micro-level of different tasks being performed within the same application: for example, related to our FTP case study, the utility of the FTP download service as a whole could be expressed as a combination of the utility values recorded for bulk downloads and small download tasks as they occur. Things are analogous in the increasingly common case of component-based applications, or even systems-of-systems, in which different pieces of legacy, COTS or otherwise third-party functionality are integrated into a new application. If all components could be characterized with embedded utility functions, the

utility of the overall composite system could be derived by combining the utility values recorded for each of the various components.

Lastly, we have been able to test the limits of our hypothesis that the utility abstraction can lead to non-analytic self-assessment. Domain-specific knowledge plays a role in our approach, but only in the initial data collection phase, for the selection of the basic characteristics of the application or task that must be monitored. Past that phase, the whole process of function synthesis and embedding, as well as the computation of utility at run time, remain application-agnostic. We see this as a major factor enabling the assessment of the application under scrutiny, no matter how unexpected or unpredictable its state of operation can become.

Ongoing and future work

An effort that is currently undergoing aims at making available the self-assessment provisions described so far beyond the repertoire of software application developed in Aupy. We would like to be able to apply those provisions to prevalent examples of system software, middleware and application-level software. For that reason, we have begun the porting of Aupy to Java. So far, we have implemented and successfully validated functionality that is analogous to Python decorators, by leveraging a combination of Java annotations, which have become a standard feature of the JDK since its 1.5 release, and the JBoss Aspect Oriented Programming framework⁴, which offers an open-source library that enables to associate custom code to Java annotations. As our next step in this effort, we will attack the challenge of porting also the meta-programming semantics and features of our utility contexts to the Java platform.

A central part of our approach is the process for the synthesis of utility functions, which requires the collection of a large amount of data and its statistical analysis. Although data collection is automated, this is currently a time-consuming process that occurs in the lab, by means of a test bed environment that needs to be configured for each application. Moreover, the statistical analysis occurs offline, and can be rather labor-intensive. This is an issue that must be resolved, in order to adopt utility functions for self-assessment extensively (that is, for a diverse range of applications), and at the correct level of granularity (that is, for the distinct features and tasks enabled by a given application). We are now considering how to attack that issue.

First of all, we want to move from in-the-lab to in-the-field data collection. With Aupy, it is easy to deploy in the field an instrumented version of the application under scrutiny, so that its various features can be exercised in regular working conditions. However, in that case, it may be difficult to produce an exhaustive map of evenly distributed data points, to be fed to our statistical correlation analysis. It is likely that many of the collected data points would tend to gather in one or more dense areas within that space, with other data points constituting outliers or operational anomalies. Consequently, it might be unfeasible to synthesize an equation that is valid outside of the main operation areas, and which covers and accurately

⁴ See <http://www.jboss.org/jbossaop/>

attributes utility to anomalous data points. On the other hand, we could identify and tell apart different operation areas by analyzing in-the-field data with clustering techniques – as shown for instance by Quiroz et al. [22] - and possibly characterize those operation areas in terms of different application features being exercised. That would easily lead to the definition of distinct utility functions for each of them. Drawing an example from our FTP case study, tasks like downloading a single large multi-GB file, and downloading multiple 1 MB files would correspond to two distinct, recognizable data point regions, and could be treated independently from one another.

We also plan to leverage some AI-based techniques, to advance towards fuller automation of the utility function synthesis procedure. For example, cooperative negotiation has been already applied to Autonomic Computing contexts, in which overall utility is not known or cannot be expressed a priori, and must be elicited incrementally from multiple interested parties [1] [20]. To apply and adapt those earlier results to our context, we could employ multiple evaluating agents, each defining its utility in terms of a single axis of the multi-dimensional space being monitored. Each agent would compute its local utility values for a number of sample requests, and the aggregation of the various utility perspectives thus generated would be used to compute the overall utility function. Methods for constructing dependency graphs and profiling the relationships among multiple quality factors, analogous to those described in [16] [17] by Li et al, could be used to guide such an aggregation process, as well as the choice of the relevant dimensions. Further insight on this matter could come from the field of automated learning. We plan to experiment with supervised and unsupervised learning techniques, to help deriving utility functions from the potentially large and high-dimensional space monitored through Aupy.

Finally, since a utility function can be regarded as a multi-dimensional map of application value, in which the axes represent environmental as well as application attributes, and assuming a management infrastructure that is able to actuate some subset of those attributes (at either level), it is conceivable to elaborate actionable trajectories within that map, to move an application from a low-utility state towards a higher-utility state. A somewhat similar idea has been pursued for example in [4]. To efficiently calculate those trajectories, we could again leverage machine learning techniques, in particular Reinforcement Learning (RL). RL has been successfully used for Autonomic Computing in the recent past to synthesize policies for decision-making, aiming once again at self-optimization [27] [28]. A key concept in RL is “reward”, a scalar that values the observed consequence of a decision, and which the learning agent making decisions aims to maximize. The concept of utility naturally maps onto RL rewards; thus, having synthesized a utility function, it is possible to compute rewards for trajectories within the state space of the application, and segments within a trajectory.

With that development, utility-based methods could be applied in Autonomic Computing not only towards non-analytic self-assessment (i.e., within the reactive/introspective “Monitor+Analyze” portion of the MAPE control loop), but also to the equally important area of non-analytic, automated decision-making. Such a technique could thus enable the design of provisions for the

“Plan+Execute” half of the MAPE loop devoted to proactive adaptation, which could potentially cope with unexpected events and conditions.

Conclusions

This paper presents a method and a tool for the synthesis of application-level utility functions, which can be employed for autonomic self-assessment. Our work has provided us with an understanding of the potential of considering utility at the level of an application and its single, recognizable tasks or fetures. It has also led us to devise a synthesis process based on the statistical analysis of measurable data that is collected at two levels: the application under scrutiny, and the running environment hosting that application. We have also acquired know-how on the engineering of a framework for data monitoring and collection, as well as the embedding of synthesized utility functions in the runtime code, which leads to seamless computation of utility as the applications operates. To exploit that know-how, we have developed an open source prototype, Aupy, and experimented with it in two case studies from diverse application domains. Through those case studies we have acquired further insight on the task-based nature of application-level utility, which has prompted us to include in our framework fine-grained, block-wise support for utility functions. We are now involved to expand the functionality of our prototype to other platforms.

This work shows how application-level utility is a viable abstraction for the non-analytic self-assessment of autonomic systems, and can be made operational for newly developed as well as legacy systems.

Finally, this work can represent a first step that enables further research on application-level utility in Autonomic Computing, regarding the fully automated, in-the-field synthesis of utility functions, as well as the possibility to leverage utility information for automated and non-analytic decision-making on application adaptation.

Acknowledgements The authors would like to thank the other members of the Software Engineering Research Group (SERG) at Drexel University. In particular, we are grateful to Prof. Spiros Mancoridis, and to students Max Shevertalov and Ed Stehle, who have been instrumental in initiating and carrying out the work leading to the reported results. Sepcial thanks to Justin Hummel for his work on the porting of Aupy to the Java platform, his assistance in documenting Aupy features, and his help in the revision of this manuscript.

References

- [1] C. Boutilier, R. Das, J.O. Kephart, G. Tesauro, and W.E. Walsh, “Cooperative Negotiation in Autonomic Systems using Incremental Utility Elicitation,” In Proceedings of the 19th Conference on Uncertainty in Artificial Intelligence, Acapulco, Mexico, August 2003.
- [2] M. Brodie, S. Ma, G. Lohman, L. Mignet, M. Wilding, J. Champlin, and P. Sohn. “Quickly Finding Known Software Problems via Automated Symptom Matching”, in Proceedings of the 2nd International Conference on Autonomic Computing (ICAC 2005), Seattle, WA, USA, June 2005.
- [3] Z. Cao and E.W. Zegura, “Utility max-min: an application-oriented bandwidth allocation scheme,” in Proceedings of IEEE INFOCOM'99, New York, NY, USA, April 1999.
- [4] M. Karlsson, and M. Covell, “Dynamic Black-Box Performance Model Estimation for Self-Tuning Regulators”, in Proceedings of the 2nd International Conference on Autonomic Computing (ICAC 2005), Seattle, WA, USA, June 2005.

- [5] S.W. Cheng, D. Garlan, and B. Schmerl, "Architecture-based self-adaptation in the presence of multiple objectives," in Proceedings of the ICSE International Workshop on Self-adaptation and self-managing systems (SEAMS 2006), Shanghai, China, May 2006.
- [6] F.C. Chu, and J.Y. Halpern, "Great Expectations. Part II: generalized expected utility as a universal decision rule", *Artificial Intelligence*, Elsevier, 159(1-2):297-302, November 2004.
- [7] R. Das, I. Whalley, and J.O. Kephart, "Utility-based collaboration among autonomous agents for resource allocation in data centers", in Proceedings of the 5th Intl. Joint Conference on Autonomous Agents and Multiagent Systems, Hakodate, Japan, 2006.
- [8] J. Dean, and S. Ghemawat, "MapReduce: Simplified Data Processing on Large Clusters", in Proceedings of the 6th Symposium on Operating Systems Design and Implementation (OSDI 2004), San Francisco, CA, USA, December 2004.
- [9] W. Dickinson, D. Leon, and A. Podgurski, "Finding failures by cluster analysis of execution profiles", in Proceedings of the 23rd International Conference on Software Engineering (ICSE), Toronto, Ontario, Canada, May 2001
- [10] S. Ghemawat, H. Gobiuff, and S. Leung, "The Google File System", in Proceedings of the 19th ACM Symposium on Operating Systems Principles, Lake George, NY, USA, October 2003.
- [11] D.E. Irwin, L.E. Grit and J.S. Chase, "Balancing Risk and Reward in a Market-Based Task Service," in Proceedings of the 13th IEEE International Symposium on High Performance Distributed Computing (HPDC-13), Honolulu, HI, USA, June 2004.
- [12] C. Jin, and R. Buyya, "MapReduce Programming Model for .NET-Based Cloud Computing", *Lecture Notes in Computer Science* Vol. 5704, pp. 417-428, Springer, 2009.
- [13] J.O. Kephart, and D.M. Chess, "The Vision of Autonomic Computing", *IEEE Computer* 36(1):41-50, January 2003
- [14] J.O. Kephart, and R. Das "Achieving Self-Management via Utility Function", *IEEE Internet Computing* 11(1):40-48, January 2007.
- [15] J.F. Kurose, and R. Simha, "A Microeconomic Approach to Optimal Resource Allocation in Distributed Computer Systems", *IEEE Transactions on Computers*, May 1989, 38(5):pp705-717.
- [16] B. Li. "A Hierarchical Graph Model for Probing Multimedia Applications". In Proceedings of the IEEE International Conference on Multimedia and Expo (ICME 2001), Tokyo, Japan, August 2001.
- [17] B. Li and K. Nahrstedt. "QualProbes: Middleware QoS Profiling Services for Configuring Adaptive Applications". In Proceedings of Middleware 2000, Hudson River Valley, NY, USA, April 2000.
- [18] D.L. Mills, "Network Time Protocol (Version 3) Specification, Implementation and Analysis", IETF RFC 1305, March 1992, available at: <http://www.faqs.org/rfcs/rfc1305.html>
- [19] A.V. Mirgorodskiy, N. Maruyama, and B.P. Miller, "Problem diagnosis in large-scale computing environments", in Proceedings of the 2006 ACM/IEEE Conference on Supercomputing, Tampa, FL., USA, 2006.
- [20] R. Patrascu, C. Boutilier, R. Das, J.O. Kephart, G. Tesauro, and W.E. Walsh, "New Approaches to Optimization and Utility Elicitation in Autonomic Computing," In Proceedings of the National Conference on Artificial Intelligence, Pittsburgh, PA, USA, July 2005.
- [21] J. Postel, and J. Reynolds, "File Transfer Protocol (FTP)", IETF RFC 959, October 1985, available at: <http://www.ietf.org/rfc/rfc959.txt>
- [22] A. Quiroz, N. Gnanasambandam, M. Parashar, and N. Sharma, "Robust Clustering Analysis for the Management of Self-Monitoring Distributed Systems," *Journal of Cluster Computing*, 12(1):73-85, March 2009.
- [23] M.A. Rappa, "The utility business model and the future of computing services", *IBM Systems Journal*, 43(1):32-42, International Business Machine Corporation, 2004.
- [24] M. Salle and C. Bartolini, "Management by contract," In Proceedings of Network Operations and Management Symposium (NOMS 2004), Seoul, Korea, April 2004.
- [25] E. Stehle, M. Shevertalov, P. deGrandis, S. Mancoridis, and M. Kam, "Task Dependency of User Perceived Utility in Autonomic VoIP Systems", in Proceedings of the International Conference on Autonomic and Autonomous Systems, (ICAS 2008), Gosier, Guadeloupe, March 2008.

- [26] R. Sterrit, and D. Bustard. "A Health-Check Model for Autonomic Systems Based on a Pulse Monitor", *Knowledge Engineering Review*, 21(3):195-204, September 2006.
- [27] G. Tesauro, "Reinforcement Learning in Autonomic Computing. A Manifesto and Case Studies," *IEEE Internet Computing*, 11(1):22-30, January 2007.
- [28] G. Tesauro, N.K. Jong, R. Das, M.N. Bennani, "On the use of hybrid reinforcement learning for autonomic resource allocation," *Journal of Cluster Computing*, 10(3):287-299, September 2007.
- [29] W.E. Walsh, G. Tesauro, J.O. Kephart, and R. Das, "Utility Functions in Autonomic Systems", in *Proceedings of the International Conference on Autonomic Computing (ICAC 2004)*, New York, NY, USA, May 2004.
- [30] G. Wang, A. R. Butt, P. Pandey, and K. Gupta, "A Simulation Approach to Evaluating Design Decisions in MapReduce Setups", in *Proceedings of the 17th Annual IEEE/ACM International Symposium on Modelling, Analysis and Simulation of Computer and Telecommunication Systems (MASCOTS '09)*, London, UK. Sep. 2009.
- [31] X. Wang and H. Schulzrinne, "Pricing network resources for adaptive applications in a differentiated services network," in *Proceedings of INFOCOM 2001*, Anchorage, AK, USA, April 2001.



Figure 1: MapReduce worker utilization with utility-aware assignment policy.

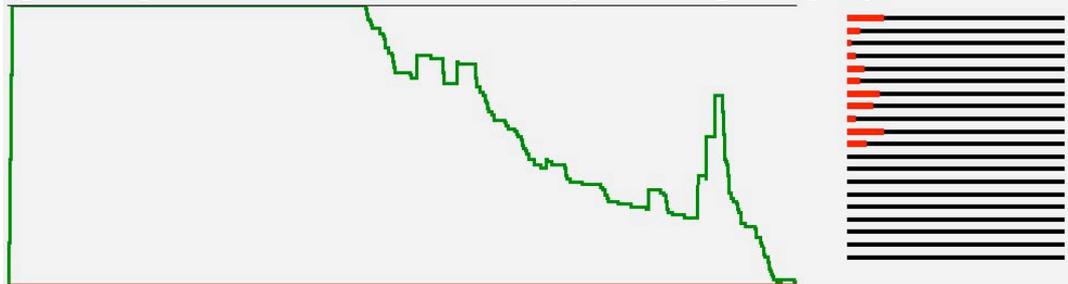


Figure 2: MapReduce worker utilization with simple assignment policy.

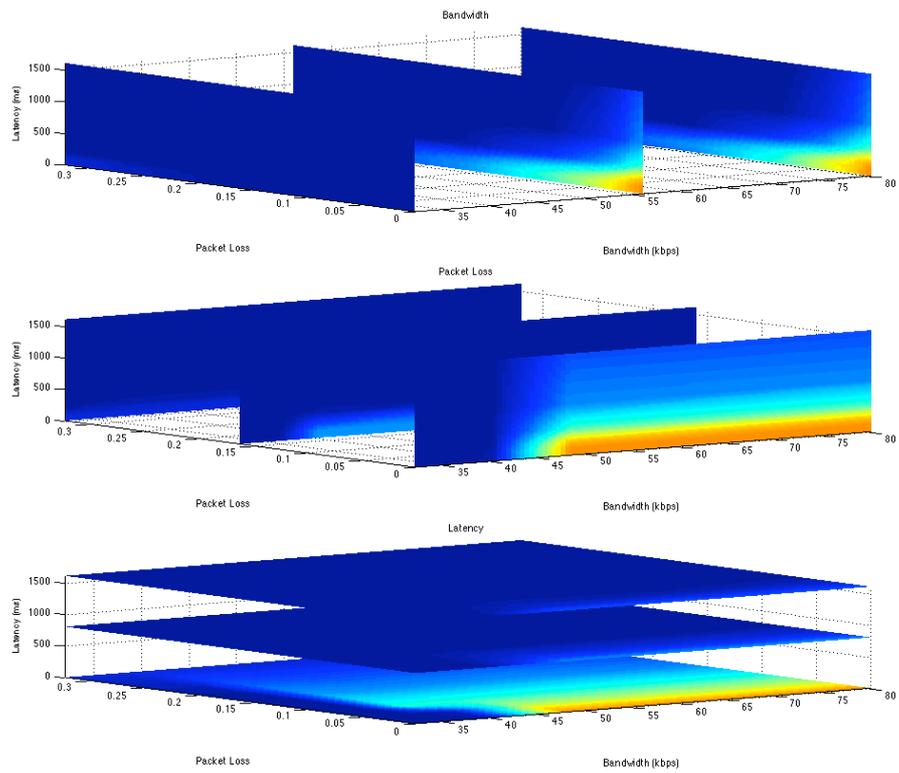


Figure 3: observed FTP utility.

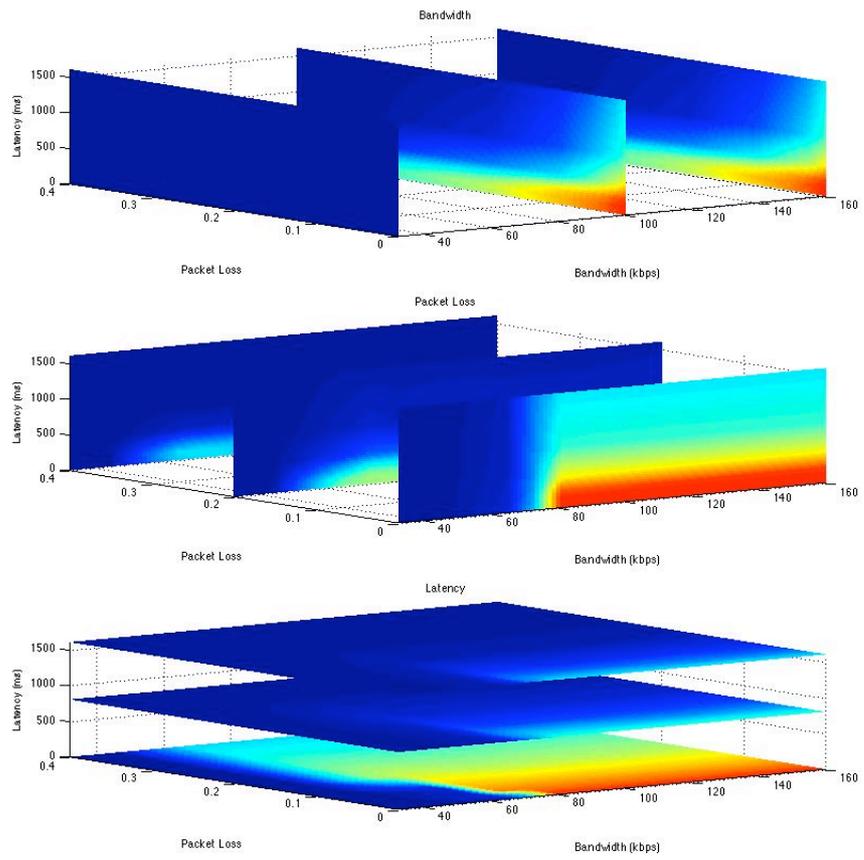


Figure 4: estimated FTP utility.

Table 1: comparison of MapReduce assignment policies – job completion time.

EXPERIMENT	UTILITY- AWARE POLICY	SIMPLE POLICY	IMPROVEMENT
# 1	25,939 s	38,926 s	33 %
# 2	34,226 s	47,368 s	28 %
# 3	120,671 s	188,425 s	36 %

Table 2: Aupy Annotations.

INTENT	SYNTAX	NOTES
Monitor operation of the annotated code	@monitor(*args)	*args is a dynamic list populated with utility functions. If None is passed, it will use all functions defined with in the same scope as the annotated code.
Pre-evaluated utility function	@pre_utility	Evaluation occurs upon entering the annotated code.
Post-evaluated utility function	@post_utility	Evaluation happens upon leaving the annotated code.
Enveloping utility function	@utility	Evaluation occurs upon entering, and again after leaving, the annotated code.
Concurrently evaluated utility function	@concurrent_utility	Evaluation occurs concurrently with the execution of the annotated code.

Table 3: utility equations for the FTP case study

Legend			
X_i : i^{th} statistical moment of packet inter-arrival time			
Y_i : i^{th} statistical moment of socket throughput			
<i>FTP-bulk</i>		<i>FTP-small</i>	
a_1	-175150.7079	a_1	0
a_2	-145163.8736	a_2	0
a_3	0	a_3	0
a_4	-579.7485	a_4	1028.3839
a_5	2.0136	a_5	-157.4976
b_1	0.9241	b_1	0.56368
b_2	-0.2341	b_2	-0.42717
b_3	0	b_3	0
b_4	0	b_4	612.90247
b_5	-0.17755	b_5	15.18621