Autonomic Computing

Nicola Desogus

May 7, 2010

1 Overview

The evolution of computing systems has gone through many generations, starting from a single process running on a single computer, to multiple processes running on very complex geographically distributed platforms consisting of heterogeneous systems which belong to different administrative domains and are connected by different kind of networks. This increase in complexity represents the main barrier limiting the future growth of computer systems, due to the fact that right now every single component of a system is entirely managed by hand by a concomitantly of skilled IT workers.

In mid-October 2001, IBM released a manifesto proposing a possible solution to this problem: the so called "*Autonomic Computing*", a new view of the software development based on concepts derived from the natural world, more precisely it is inspired by the autonomic nervous system of the human body. The aim of this new paradigm is to build computing systems that can manage themselves given high-level policies from administrators.

1.1 The current scenario

The scenario in which we were in 2001 and we are still today is an enormously complex computing platform (that, over time, is going to be increasingly complex) composed by completely heterogeneous systems that must interoperate.

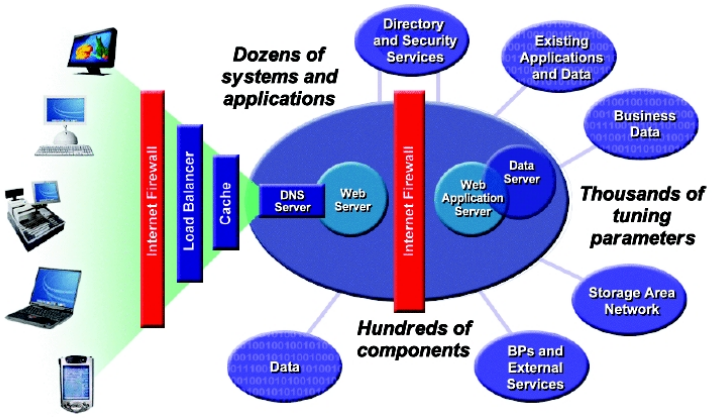

Figure 1: An example of a Web System.

Figure **??** shows an example of a Web System: we have a Web Application Server which provides computing services (Web Services), which accesses complex data storage systems, that can be connected to hundreds of Information Providers to retrieve data, that interacts with legacy applications and uses complex security protocols, etc. On the other side there is a Web Server, which in turn is a complex system that runs on heterogeneous platforms, that must connect to the Web Application Server (passing through firewalls and various security mechanisms) to provide to thousands of users, services offered by the Web Application Server.

Therefore, there are many elements, each one built with different hardware, different software, written in different programming languages, with different purposes, which must interoperate to provide a set of services to all system users. Each of these individual programs is configurable through an average of ten parameters, then when all these programs together interact to provide a service, we will have hundreds of programs with dozens of configuration parameters, so we will have thousands of configuration parameters.

How can it work? How do you make the tuning of all parameters so as to achieve the best possible result? What does it mean "the best possible result"?

1.2 Towards autonomic computing systems

To manage such a platform right now there are many problems:

- Administration of individual systems is increasingly difficult, we have people in charge of the database, that should talk to people who deal with the web server, with people who deal with the firewall, with people which manage security, etc; besides, all the configuration parameters of the system are managed by hand by human operators, who must communicate with each other to set up a structure that works;
- Heterogeneous systems are becoming increasingly connected, especially with the rise of web services, devices of all kinds interact between each other;
- Designers can't intricately plan interactions among all elements, we have increasingly complex systems, always more dynamic, whose dimensions have become unmanageable for an individual programmer; therefore, the continuous dialogue between the parties is the rule;
- Systems are constantly updated: what does the upgrade or the addition of a component imply? Everything still works? At worst is necessary to reconfigure the entire system;
- More of the burden must be assumed at run time, in fact is necessary that operators assume responsibility for monitoring of each element to verify the proper functioning and intervene promptly if problems, trying not affect the status of the other active elements;
- High cost of administrators: e.g. 6:1 cost ratio between storage admin and storage;
- *Errare humanum est*: 40% outages due to operator errors;

From all this comes the need for a self-managing computing system, that is an autonomic system capable of making decisions on its own, using high-level policies: a system that constantly checks its status and automatically reacts to changing conditions in order to achieve the optimal functioning; the only intervention of the human operators will be the definition of the high-level policies, this is the purpose of the autonomic computing.

2 Autonomic Computing

Autonomic Computing helps to address complexity by using technology to manage technology. The term autonomic is derived from human biology. The autonomic nervous system monitors, among other things, your heartbeat, without any conscious effort on your part. Your heartbeat automatically increases/decreases depending on what you want to perform. Certainly this is not the only example, in the human body there are many parts (subsystems) which manage themselves and work together towards a common goal, this collaboration represents the entire system, which in turn is an autonomous system that adapts itself to changing conditions. The common objective is the high-level policy that the body knows how to translate into incremental goals for each subsystem.

However, there is an important distinction between autonomic activity in the human body and autonomic activities in computing systems. Many of the decisions made by autonomic capabilities in the body are involuntary. In contrast, self-managing autonomic capabilities in

computing systems perform tasks that human operators choose to delegate to the technology according to high-level policies.

2.1 Self-Properties

There have been several efforts to characterize the main features that make a computing system or an application autonomic. IBM identified four aspects that an autonomic system must at least support, the so called *Self-Properties*:

- *Self-Configuration*: nowadays there are multi-platform systems sold by different companies, always more heterogeneous, that must be installed, configured, and integrated completely by hand; this activity, being human, is subject to frequent errors that require new interventions. An autonomic system should have the ability to dynamically adjust its status in response to changes in the environment, using policies provided by the human operators. Such changes could include the deployment of new components or the removal of existing ones;

- *Self-Optimization*: once it is defined the objective of the system, this must be achieved at best as possible, this involves the tuning of thousands of parameters to identify which combination can provide the best possible result. An autonomic system should be able to detect sub-optimal behaviors and intelligently perform self-optimization functions. The tuning actions could mean reallocating resources in response to dynamic workload changes, or to ensure that particular business transactions can be completed in a timely fashion; to ensure this there must be a mechanism for monitoring and tuning of each resource in the system in order to dynamically partition resources and/or manage workload.

- *Self-Protection*: detecting, isolating and protecting the system against possible attacks, currently are tasks carried out by a human operator, who occasionally goes to check the system log to verify for anomalies and take appropriate actions. An autonomic system should be capable of anticipating, detecting and protecting its resources from hostile behaviors and take corrective actions to make itself less vulnerable.

- *Self-Healing*: when there is a problem the best minds come together to find a solution, but only identifying the source of the problem in large, complex systems, can take a team of programmers working for weeks. An autonomic system should have the ability to discover, diagnose and isolate problems or potential problems, then find an alternate way of using resources or reconfiguring the system to keep functioning smoothly.

An autonomic system can be viewed as a collection of autonomic elements or building blocks in which each can support the four properties outlined above.

2.2 The architecture

The architecture proposed to implement such a system is shown in Figure **??**. We have a resource that should be managed automatically, but this resource per se is a passive entity which has no idea how to manage itself and it is not been designed to operate in an autonomic architecture, hence a "*Manageability Interface*" must be provided to allow the autonomic system to interact with this resource.

Self-managing capabilities in a system accomplish their functions by taking an appropriate action based on one or more informations that they pull out from the environment. Therefore, the interface should be able first to provide information of the resource status, whatever status means, it must be equipped with a set of *Sensors* that provide information about what happens to the resource (What is its throughput? Is it under attack? etc). On the other hand, once obtained status information from the sensors and made decisions on how to intervene, the interface should also be given with the *Effectors* to deploy and implement appropriate corrective operations.

At this point, is required an entity to analyze all the data emanating from sensors and drive the effectors on how to act and undertake operations from time to time to reconfigure, optimize

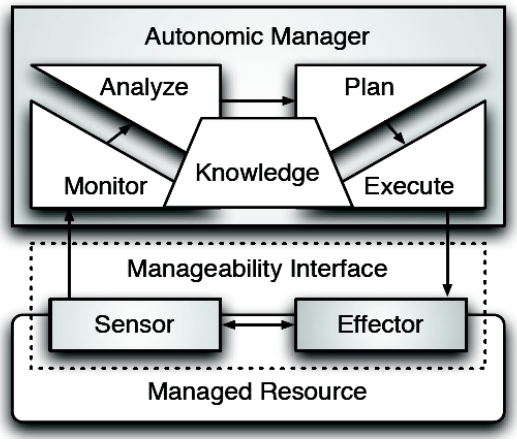

Figure 2: The autonomic computing system architecture.

and protect the resource: this entity is called "*Autonomic Manager*", which implements a control loop that collects details from the system and acts accordingly. The Autonomic Manager is organized as a ring and is divided into four parts:

- **Monitor**: the monitor function has within it all the mechanisms necessary to interact with the sensors of the Manageability Interface and its purpose is to gather data sent from these sensors; what kind of information is provided? For example, data on the percentage of resource utilization, data on the protection mechanisms, data on current performance, etc. *Monitor* can use a polling strategy, asking for data at predetermined moments, or the sensors can be programmed to send details of any significant change;

- **Analyze**: the analyze function takes all data from the monitor and processes it so as to check the status of the resource and identify any problem; which data should be analyzed? How to correlate data? At this stage there's a grouping, filtering and semantic transformation of data, which (in case of discovery of an anomaly that requires an intervention by the system) is then passed to the third component of the system;

- **Plan**: the plan function provides a sort of inference engine to select the actions needed in response to an abnormal behavior detected by the analyze mechanism and to achieve goals and objectives defined by the high-level policy;

- **Execute**: the execute function translates the decision made in the previous phase into invocations and commands for the effectors for modifying status and behavior of the resource and for achieving the result that had been planned previously.

The four parts communicate and collaborate with one another and exchange appropriate knowledge and data. In fact, **Knowledge** can be considered the fifth part of the Autonomic Manager, it represents a shared component that will contain the system model, the processing results of every function, the plans identified, etc.

These five parts work together to provide the control loop functionality; since any action taken by the effectors on the resource, has an impact on the status of the same, possibly producing new data that will be sent back by the sensors that again must be analyzed and processed to verify the result and eventually plan a new intervention, such a loop is called a *feedback loop*.

2.3 Some considerations

Autonomic elements will work at many levels, from individual computing components to the entire automated system. Each autonomic element will be responsible for managing its own

internal state and behavior and for managing its interactions with others elements of the environment. All the autonomic elements will work together towards achieving a common goal, which has been specified by designers.

At the lower levels, an autonomic element will have limited range of internal behaviors and relationships to manage; while at the higher levels, there will be increased dynamism and flexibility, the entire system must be running for a specific purpose given at higher level, whose policies must be translated into mechanisms of behavior for all individual components of the autonomic system.

At the autonomic element level, the resource from passive becomes managed with the intelligence of the autonomic manager, hence, all the infrastructure to make this resource per se independent must be provided. Moreover, all the mechanisms to enable communication and interaction among autonomic elements at various levels are needed, allowing "useful" interactions to "emerge".

2.4 Engineering challenges

From the engineering point of view there are several challenges that must be addressed:

- **Design, test and verification**: what does it mean? Have we to think of all possible error states? In large-scale systems it is not feasible.
- **Lifecycle**: what is the life cycle of an autonomic element? Can be broken? Can it make a break? While the manager performs a processing, what happens to the resource?
- **Upgrading**: when the resource is updated what should we do? Should we modify the autonomic manager?
- **Monitoring mechanisms**: how do we implement the data monitoring?
- **Adaptation mechanisms**: when adding new sensors/effectors how do we implement new action plans using the new components available?
- **Knowledge aggregation/distribution**: how do we put together the knowledge? How do we distribute it among all parts of the system?
- **Information Filtering**: how do we implement data filtering to detect anomalies in the system?
- **Interaction Specification**: how do we specify interaction between components of the system?
- **Interaction Implementation**: how do we implement interaction? How do we implement individual communications between autonomic elements?
- **Negotiation**: how do we provide stages of negotiation to choose between different options?
- **Hierarchical management**: how do we manage the various hierarchical levels of autonomic managers?

Then we have challenges that are not at the level of individual components, but are problems that concern the whole architecture of the autonomic system:

- **Authentication, encryption, signing**: security mechanisms must be shared between all parts of the system for authentication, authorization and encryption of information;
- **Introspection/Intercession**: what are the mechanisms for providing data? What are the mechanisms by which the autonomic manager can be informed of the sensors and effectors associated with a specific resource?

- **Robustness against attacks**: not only the single element, but the entire system must be robust to attacks, because if it fails one, and that failure was not anticipated, may fail the whole system.

Finally there is the problem of the goal specification:

- **Humans-provided goals and constraints**: goals and constraints of the system are provided by human operators and must be translated into any format that the entire system can understand, but... how?
- **Policies**: what are the policies that can be expressed? How to be expressed? Semantics and syntax must be defined. How to distribute policies to different levels of the system? How are translated from high level policies into goals for individual components?
- **Protection from input goals that are inconsistent, implausible, dangerous, or unrealizable**: needs for protection mechanisms to ensure consistency of objectives.

2.5 Scientific challenges

There are also issues from a scientific perspective, mostly related to research per se, which are:

- **Behavioral abstractions and models**: for each system, we look for an ad-hoc solution, but there are such patterns and general mechanisms that can be abstracted from a single implementation and provided as a set of tools that can be used by other developers of autonomic systems to solve the same type of problems, without necessarily starting from scratch;
- **Robustness and reliability**: how to verify that a system is robust? What if there was any error at design time? Must we restart from scratch or not?
- **Learning and optimization**: what are the rules that ensure the best possible result? What if we want a system that can automatically update and learn new rules? How can we guarantee that if we take a sequence of decisions the system will converge toward the goal? How fast?
- **Negotiation theory**: which are the most effective algorithms? How does overall system behavior depends on the mixture of negotiation algorithms?

3 Autonomic controller of the Apache Web Server

In this example will be shown a simple non-hierarchical autonomic controller.

A typical server, in the most simplified view, has a single Master process and a pool of Worker processes (or threads, depending on the circumstances). The Master process receives requests from users connected and select a Worker to handle the request. A Worker is a process with a TCP connection to the client; therefore, each process handles the interaction with a client, how many workers are there? We have a system configuration parameter called *MaxClients*, which defines the upper limit on the number of users that can be served simultaneously.

A process worker in a given moment can be in one of the following three states:

- *Idle*: is in the pool but is doing nothing;
- *Busy*: is connected to a client and is serving it;
- *Waiting*: is connected to a client and is waiting for receiving an HTTP request;

The Apache web server is an HTTP/1.1-compliant web server with persistent connections, that is TCP connections remain open between consecutive HTTP requests. Here a question arises, how long has a worker to wait for the next HTTP request? We have another parameter called *KeepAlive*, which defines the upper limit for connection inactivity time.

The goal of the autonomic controller in this case is to maintain the values of CPU and memory usage respectively around 50% and 60%. The optimal value of the control parameters depends in turn on the system workload, which is the number of users who require a connection to the server. The number of requests can not be controlled, but the system can operate on the maximum number of clients that can be served. Although there is a dynamic workload, the system must react to external events (connection requests from users) by modifying the control parameters in order to stay always as close as possible to the optimal values of CPU and memory.

How to codify the relationship between the change of control parameters and the variation of CPU and memory utilization? We need to define a model of the web server.

There are two possible approaches:

- build a mathematical model;
- build a data-based model ("blackbox" approach).

The first one requires perfect understanding of inner workings of server and the definition of the equations governing the evolution of the system. The second one is a more "wild" approach, that obtains a model by analyzing the result of a set of test which modify the control variables.

Applying the latter approach, in order to gain understanding of the dynamic behavior of the server, the modeling agent varies the tuning parameters (*MaxClients* and *KeepAlive*) of the server and records the resulting performance metrics (CPU and memory utilization). The collected time-series data provide knowledge about the dynamic relationship between the tuning parameters and the performance metrics. The model identified by IBM is as follows:

$$\begin{bmatrix} CPU_{k+1} \\ MEM_{k+1} \end{bmatrix} = A \cdot \begin{bmatrix} CPU_k \\ MEM_k \end{bmatrix} + B \cdot \begin{bmatrix} KeepAlive_k \\ MaxClients_k \end{bmatrix}$$

where

- A and B are matrices of constant elements defined by the modeling agent;
- CPU_k and MEM_k represent respectively the value of CPU and memory usage at time k;
- $KeepAlive_k$ and $MaxClients_k$ represent the selected values at time k for the control variables.

The user is only required to insert two inputs: the effective ranges of the tuning parameters and the maximum delay required for tuning parameters to take full effect on the performance metrics.

The implementation of the ring control is through the easier mechanism of feedback control, called **PID** (Proportional-Integral-Derivative). A PID controller attempts to minimize the error

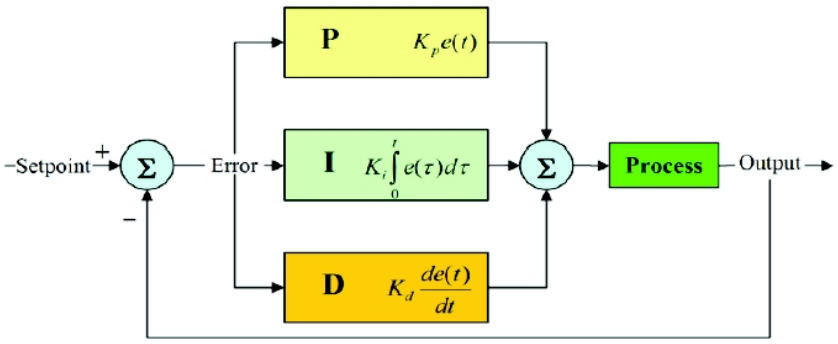

Figure 3: The PDI controller mechanisms.

between a measured process variable and a desired setpoint, by adjusting the control variables

of the system. In our case the process is the server, the setpoint is represented by the optimal CPU and memory usage values, while the process variable corresponds to the current CPU and memory usage values. To reduce the error, the control acts on the tuning parameters *MaxClients* and *KeepAlive* intelligently, exploiting the knowledge of the model seen above.

The PID controller involves the calculation of three separate parameters:

- the *Proportional value*, which is denoted by P and determines the reaction dependent on the present error; it produces a change to the output that is proportional to the current error value;
- the *Integral value*, which is denoted by I and determines the reaction based on the accumulation of past errors; it reduces the residual steady-state error that occurs with a proportional only controller and accelerates convergence towards setpoint;
- the *Derivative value*, which is denoted by D and is a sort of prediction of future errors based on the error rate of change; it improves the stability of the process by reducing the rate of change of the controller output.

The weighted sum of these three parameters is used to adjust the process via a control element. The output function is as follows:

$$Output(t) = K_p\, e(t) + K_i \int_0^t e(\tau)\, d\tau + K_d \frac{de}{dt}$$

where the tuning parameters are the weights K_p, K_i and K_d.

Some result of the autonomic controller of the Apache Web Server is shown in Figure **??**.

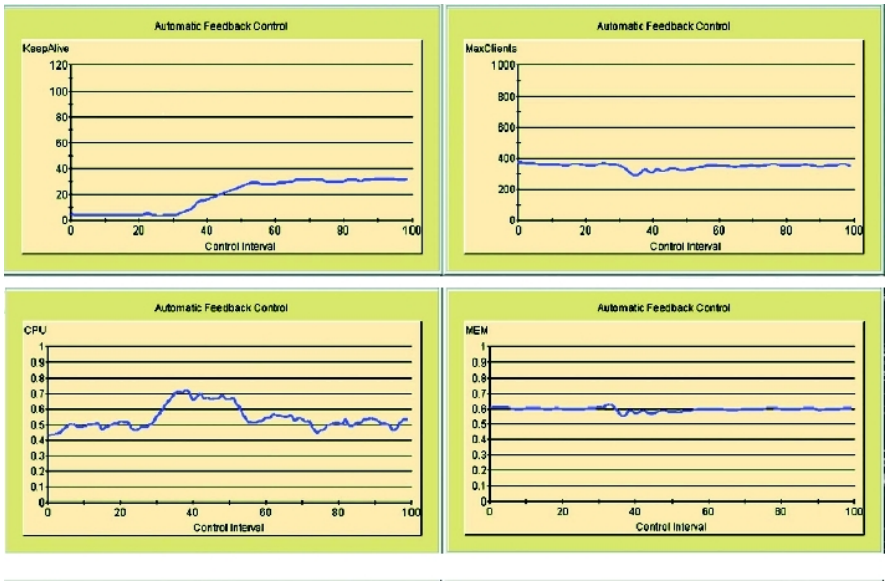

Figure 4: Some result.