RC22637 (W0211-096) November 12, 2002
Computer Science

IBM Research Report

Anatomy of Autonomic Server Components

Kattamuri Ekanadham, Joefon Jann, Pratap Pattnaik,
Ramanjaneya Sarma Burugula, Donna Dillenberger
IBM Research Division
Thomas J. Watson Research Center
P.O. Box 218
Yorktown Heights, NY 10598

Research Division
Almaden - Austin - Beijing - Haifa - India - T. J. Watson - Tokyo - Zurich

Anatomy of Autonomic Server Components

Kattamuri Ekanadham
Joefon Jann
Pratap Pattnaik
Ramanjaneya Sarma Burugula
Donna Dillenberger

IBM Thomas J. Watson Research Center
Yorktown Heights, NewYork

{eknath, joefon, pratap, burugula, engd @ us.ibm.com}

Abstract

Autonomic systems are an essential part of modern server designs, and are key to the reduction of system-level complexity. Some of the challenges faced in such a design are: Increase in complexity, as each service is often a composition of many other services; Variability in the characteristics of inputs encountered by a service; and Infeasibility of a centralized control to mobilize resources in response to input variations. To face these challenges, the servers are being modularly designed, with each component providing a standard interface. Components are no longer expected to provide a rigid and deterministic functional behavior in all aspects of their interactions with other components. They are expected to efficiently deal with the varying behaviors of other components, including faults. Components can no longer rely on a centralized control to adjust their resources. Instead, they must be prepared to perceive changes and to negotiate the exchange of resources on a voluntary basis. In summary, the subsystems must be self-governing, self-organizing, self-stabilizing and self-healing in the surrounding world of unpredictable subsystems.
While there are many subsystems in the current servers that are self-organizing to varying degrees, it is necessary to (a) formulate a common framework for every autonomic component and (b) develop general principles for their stability. In this paper, we address the first part. We propose an operational definition of an autonomic component of a server, and identify the key features it must possess to be effective in the emerging environment. We discuss many examples from servers and cast them in this light. Examples range from simple software subsystems such as memory allocators, to more complex subsystems such as work-load managers invoking dynamic reconfiguration of resources in large servers, and also include hardware components like prefetch engines and branch predictors. The second part is an ongoing effort and we briefly summarize the direction we are pursuing. Often the constraints imposed by a system are nonlinear and the trick is to come up with linear approximations where some properties of the systems may be established under some simplified assumptions.

1 Introduction

The term, *autonomic*, is applied to computing systems [1] by taking the analogy from biological systems. A component of a system is like a cell or organism that survives in an environment by (a) receiving services rendered by others in the environment, (b) constantly evolving to fit in the environment in which it must survive, and (c) performing certain services to others in the environment. In the short term, the organism perseveres to perform its functions despite adverse circumstances, by readjusting itself within the degrees of freedom it has. In the long term, evolution of a species takes place, where environmental changes force permanent changes to the functionality and behavior. While there may be many ways to perform a function, an organism uses its local knowledge to adopt a method that economizes its resources.

Autonomic computing paradigm imparts this same viewpoint to the components of a computing system. The environment is the collection of components in a system. The services performed by a component are reflected in the advertised methods of the component that can be invoked by others. Likewise, a component receives the services of others by invoking their methods. The semantics of these methods constitute the behavior that the component attempts to preserve in the short term. In the long term, new resources and new methods may be introduced as technology progresses. Like organisms, the components are not perfect. They do not always exhibit the advertised behavior exactly. There can be errors, impreciseness or even cold failures. An autonomic component watches for these variations in the behavior of other components that it interacts with and adjusts to the variations.

In this paper we introduce a basic structure to a typical autonomic component. Section 3 delineates four basic characteristics of an autonomic component:

(i) a behavior specification that remains invariant
(ii) a set of alternative implementations, each of which preserves the behavior, but may use different resources and external services and hence has different implications on the quality of service
(iii) a set of environmental parameters that are not under the control of the component, but influence its performance, and
(iv) an adaptive nature, which includes a means to evaluate the quality of service from a client's perspective, a means to evaluate the efficiency from a server's perspective, a means to estimate the variations in the environmental parameters and choosing the right implementation for each input.

We illustrate two speculative methodologies that periodically correct the speculated information. While many system implementations may have these aspects buried in some detail, it is necessary to identify them and delineate them, so that the autonomic nature of the design can be improved in a systematic manner. We discuss several examples from server design and describe them in the light of this framework.

One would hope that a systematic design of this nature leads to the application of known control theory techniques, to show the stability characteristics of the algorithms. This is an ongoing study and we hope that this infrastructure facilitates this study.

2 Previous Efforts

Reduction of complexity is not a new goal. During the evolution of computing systems, several concepts emerged that help manage the complexity. Two notable concepts are particularly relevant here: Object oriented programming and Fault-tolerant computing. We briefly describe them and explain how autonomic systems extend them further.

Object oriented designs introduced the concept of abstraction, where the interface specification of an object is separated from its implementation. Thus implementation of an object can proceed independent of the implementation of dependent objects, since it uses only their interface specifications. While this approach has many advantages (such as hierarchical construction, inheritance, overloading etc.), an important advantage that is relevant to the present discussion is that the rest of the system is spared from knowing or dealing with the complexity of the internal details of the implementation of the object. One can easily change an implementation without affecting the rest of the system in any manner. Often an implementation is designed to optimize the performance for a given class of inputs. By building higher level abstractions, one can invoke different imple-

mentations (by overloading the method being invoked) depending upon the class into which a given input falls. This primitive notion of reacting to the input characteristic is further refined in autonomic systems. Each implementation requires a different set of resources. Changing from one implementation to another might involve some costs. An autonomic system balances the gains and losses involved by taking a goal-oriented approach. The goals are set at a higher level with global view of the system as a whole.

Fault-tolerant systems are designed with additional support that can detect and correct any fault out of a pre-determined set of faults. Usually the design of components and encoding of information is done with enough redundancy to make recovery possible. This notion of resilience to a set of known faults is further refined in autonomic systems. Autonomic systems do not expect that other components operate correctly according to stipulated behavior. The input-output responses of a component are constantly monitored and when a component's behavior deviates from the expectation, the autonomic system readjusts itself either by switching to an alternative component or altering its own input-output response suitably. Thus, failure of a component might gracefully degrade the performance of another component that uses it.

3 Autonomic Server Components

The basic structure of any Autonomic Server Component, C, is depicted in Figure 1, in which all agents that interact with C are lumped into one entity, called the *environment*, to include (i) clients that submit input requests to C, (ii) other components whose services can be invoked by C, and (iii) resource managers that control the resources for C. The basic structure of any autonomic component can be represented by the tuple: [Σ, Φ, β, Π, η, ψ, $\bar{\xi}$, α]. The functional behavior of C is captured by the first 3 symbols, Σ,Φ,β, where Σ is the input alphabet, Φ is the output alphabet, and β is a relation specifying valid input-output pairs. Thus if C receives an input uεΣ, it delivers

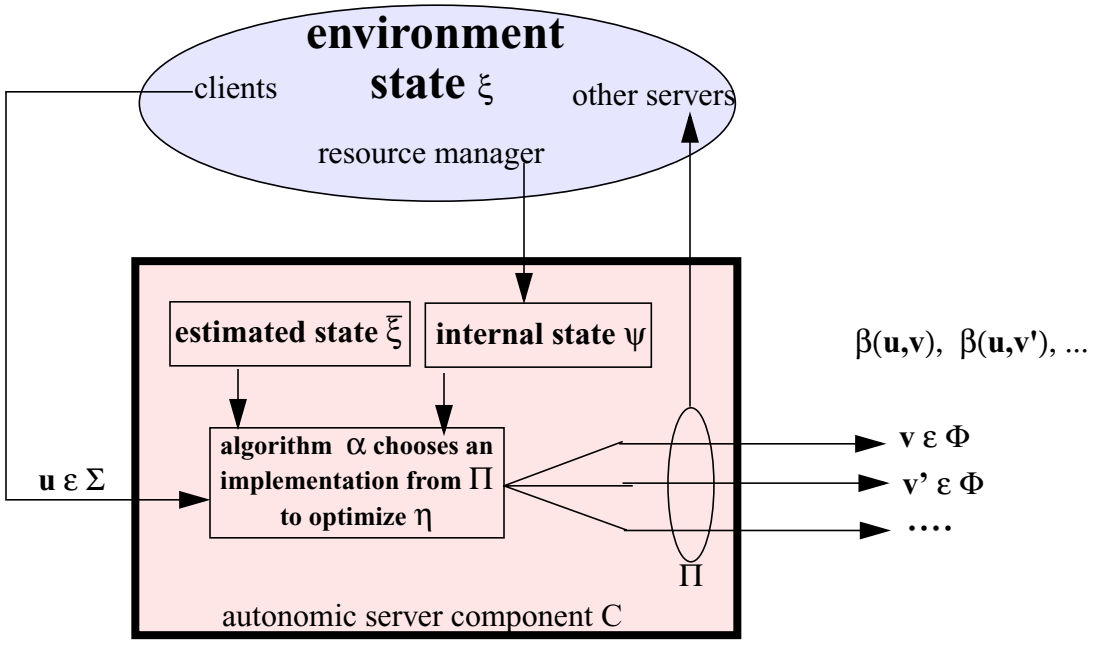

Figure 1: Schematic View of an Autonomic Server Component [Σ,Φ,β,Π,η,ψ,ξ,α]

an output $v \varepsilon \Phi$, satisfying the relation $\beta(u,v)$. The output variability permitted by the relation β (as opposed to a function) is very common to most systems. As illustrated in Figure 1, a client is satisfied to get any one of many possible outputs (v, v',...) for a given input u, as long as they satisfy some property specified β. An autonomic component usually has many alternative implementations, Π, available, to produce an acceptable output for a given input. Each of them may require different resources and may produce outputs of different quality. For each input, the algorithm α chooses a suitable implementation $\pi \varepsilon \Pi$ and executes it producing the output. An essential ingredient of an autonomic component is this choice of implementation. The choice needs to be made to optimize some objective function, η. The objective may vary from system to system. Typically it may include response time to the clients, utilization of available resources, employment of suitable resources/algorithms for anticipated input patterns etc. The sophistication of the autonomic component depends upon the intelligence embedded in the algorithm α.

Implementations maintain an internal state, ψ, which may contain the necessary data structures and book-keeping of the resources employed etc. The choice of an implementation certainly depends on input u as well as on the internal state ψ. In addition, the state of the external environment, ξ, plays a critical role for optimal performance. The state ξ is an abstraction that includes the input patterns arriving for C, the resources given to C (which are being dynamically adjusted by an external resource manager), and the performance level (including failures and degradation) of other components whose services are used by C. Thus the state information, ξ, is dynamically changing and is distributed throughout the system. C cannot have complete and accurate knowledge of ξ at any time. Hence, the best C can do is to keep an estimate, $\bar{\xi}$, of ξ at any time and periodically update it as and when it receives correct information from the appropriate sources. Thus an implementation is a transformation of the form $\pi: (u,\psi,\xi) \rightarrow (v,\psi',\xi')$. Resource changes independently change ψ and variations in the environment independently change ξ.

3.1 Approximation with Imperfect Knowledge

The crux of the intelligence in an autonomic component lies in the maintenance of accurate knowledge on the speculative state $\bar{\xi}$ and in reacting to perceived changes in the environment by choosing suitable implementations that optimize the objective η under those circumstances. Usually each implementation involves some arrangement of data upon which its algorithm operates. Changing an implementation will involve some restructuring costs. Hence one must balance these costs with the anticipated benefits from switching. Communication of environmental changes and the reaction to these changes take non-trivial amount of time. Hence the algorithm must take the hysteresis into account, to prevent oscillations in the decisions. We now focus on the maintenance of the speculative state, $\bar{\xi}$, of the environment. It can be done in two ways: by self-observation and by collective observation. We will now discuss each of these methods.

3.2 Self-Observation

Here a component operates completely autonomously and does not receive any information about its environment. The component deduces information on its environment solely from its own interactions with the environment. For instance, it can keep a log of the input-output history with its clients, to track both the quality that it is rendering to its clients as well as the pattern of input arrivals. Similarly, it can keep the history of its interaction with each external service that it uses and tracks its quality. Based on these observations, it adapts suitable method of implementation.

When conditions change and the switching costs are tolerable, it switches to better implementations whenever possible. This strategy results in a very independent component that can survive in any environment. However, the component cannot react to rapidly changing environment. It takes a few interactions before it can assess the change in its environment. Thus, it will have poor impulse response; but adapts very nicely to gradually changing circumstances. We illustrate this with the example of a memory allocator.

Example 1. Memory Allocator

This simple example illustrates how an autonomic server steers input requests with frequently observed characteristics to strategies that specialize in efficient handling of those requests. The allocator does not require any resources or external services. Hence the only environmental information, ξ, it speculates is the pattern of inputs.

The behavior, (Σ,Φ,β), of a memory allocator can be summarized as follows: The input set Σ has two kinds of inputs: alloc(n) and free(a); The output set Φ has three possible responses: null, error and an address. Alloc(n) is a request for a block of n bytes. The corresponding output is an address of a block or an error indicating inability to allocate. Free(a) returns a previously allocated block. The system checks that the block is indeed previously allocated and returns null or error accordingly.

The quality of service, η, must balance several considerations: A client expects quick response time and also that its request is never denied. A second criterion is locality of allocated blocks. If the addresses are spread out widely in the address space, the client is likely to incur more translation overheads and prefers that all the blocks to be within a compact region of addresses. Finally the system would like to minimize fragmentation and avoid keeping a large set of non-contiguous blocks that prevent it from satisfying requests for large blocks.

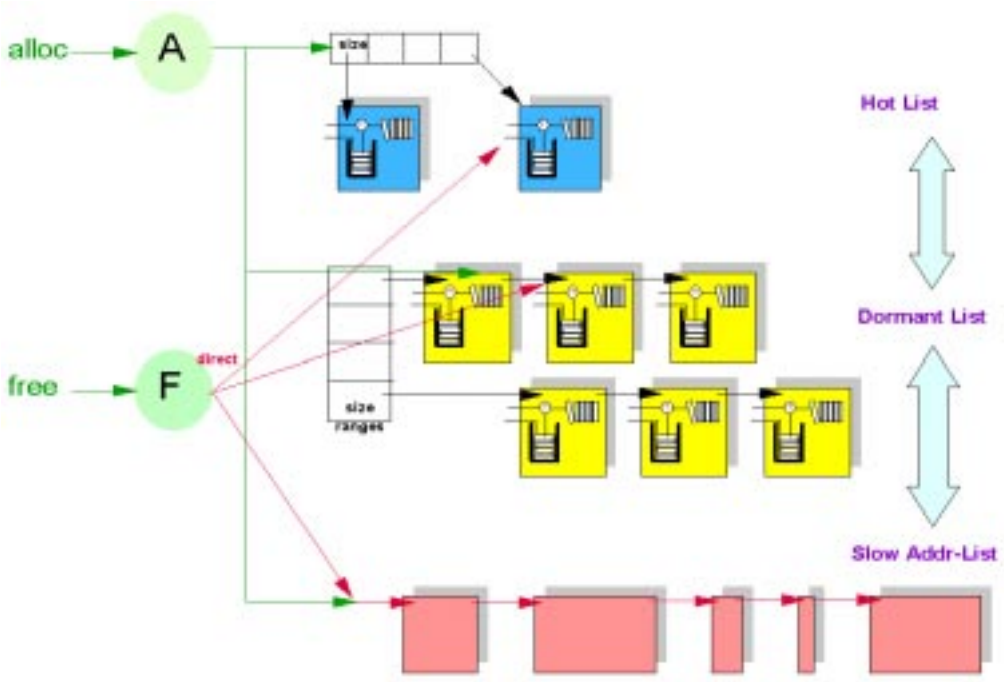

Figure 2: A Self-Organizing Memory Allocator that adapts to input characteristics

We illustrate a Π that has two implementations: The first is a linked-list allocator, shown at the bottom of Figure 2. Here the allocator keeps the list of the addresses and sizes of the free blocks that it has. To serve a new allocation request, it searches the list to find a block that is larger than (or equal to) the requested size. It divides the block if necessary and deletes the allocated block from the list and returns its address as the output. When the block is returned, it searches the list again and tries to merge the block with any free adjacent portions in the free list.

The second strategy is called slab allocation, shown at the top of Figure 2. It reserves a contiguous chunk of memory, called slab, for each size known to be frequently used. When a slab exists for the requested size, it peals off a block from that slab and returns it. When a block (allocated from a slab) is returned to it, it links it back to the slab. When no slab exists for a request, it fails to allocate.

The internal state, ψ, contains the data structures that handle the linked-list and slabs. The estimated environmental state, ξ, contains data structures to track the frequency at which blocks of each size are requested or released. The algorithm, α, always chooses the slab allocator when a slab exists for the requested size. Otherwise the linked-list allocator is used. When the frequency for a size (for which no slab exists) exceeds a threshold, a new slab is created for it, so that subsequent requests for that size are served faster. When a slab is unused for a long time, it is returned to the linked-list. The cost of allocating from a slab is usually smaller than the cost of allocating from a linked-list, which in turn, is smaller than the cost of creating a new slab. The allocator sets the thresholds based on these relative costs.

Figure 2 illustrates a strategy where the memory is organized into 3 layers. The top layer consists of a hot list of slabs for sizes which are most frequently accessed. These can be accessed very fast. The second tier is a list of slabs that are dormant and involve some searching before the right slab can be found. The last tier is the address-ordered linked list described above and has the highest latency on average. Slabs from the top tier are automatically aged out and pushed into the second tier. Similarly slabs are aged out from second tier and returned to the list in the bottom tier. Thus, the allocator autonomously reorganizes its data structures based on the pattern of sizes in the inputs. Note that the aging of slabs is not based on real time, but on the frequency of alloc/free events.

If input requests come with an attribute of the region from which the requests are made, the allocator can further improve locality, by maintaining a separate version of Π for each region and steering requests accordingly. The slab allocation algorithm can be further enhanced with information on the ratio of the total number of blocks in the linked list to the size of the largest block in it to minimize fragmentation. Another measure is the utilization of the memory pre-allocated as slabs.

3.3 Collective Observation

In general, a system consists of a collection of components that are interconnected by the services they offer to each other. As noted before, part of the environmental state, ξ, that is relevant to a component, C, is affected by the states of other components. For instance, if D is a component that provides services for C, then C can make more intelligent decisions if it has up-to-date knowledge of the state of D. If C is periodically updated about the state of D, the performance can be better than what can be accomplished by self-observation. To elaborate on this, consider a sys-

tem of n interacting components, and let $S_{ii}(t)$ denote the portion of the state of component i at time t, that is relevant to other components in the system. Every component $j \neq i$, keeps an estimate of $S_{ii}(t)$, which is denoted by $S_{ij}(t)$ and is used in the decision making algorithm, α, of component j. Thus, each component has an accurate value of its own state and an estimated value for the states of other components. Our objective is to come up with a communication strategy that minimizes the norm $\Sigma_{i,j=1,n} |S_{ij}(t) - S_{ii}(t)|$, for any time t. This problem is similar to the time-synchronization problem and the best solution is for all components to broadcast their states to everyone after every time step. But since the broadcasts are expensive, it is desirable to come up with a solution that minimizes the communication unless the error grows beyond certain chosen limits. For instance, let us assume that each component can estimate how its state is going to change in the near future. Let Δ^t_i be the estimated derivative of the state $S_{ii}(t)$, at time t - that is, the estimated value of $S_{ii}(t+dt)$ is given by $S_{ii}(t) + \Delta^t_i(dt)$. There can be two approaches to use this information:

Subscriber Approach: Let i be a component and its state, $S_{ii}(t)$, is of interest to a number of subscribers, j. Each subscriber's environmental state is initialized with $[\Delta^t_i, t, S_{ii}(t)]$, so that at any time, $t+dt$, the subscriber computes $S_{ii}(t) + \Delta^t_i(dt)$ and uses it as the estimated value for $S_{ii}(t+dt)$. Component i, which is the source of the information, monitors its own state and whenever the value $|S_{ii}(t) + \Delta^t_i(dt) - S_{ii}(t+dt)|$ exceeds a tolerance limit, it broadcasts the information $[\overline{\Delta}_i^{t+dt}, t+dt, S_{ii}(t+dt)]$ to each subscriber, where $\overline{\Delta}_i^{t+dt}$ is a new estimating function, derived from the current knowledge it has. The subscribers, reinitialize their environmental state with this latest information. In this scheme, the bandwidth of updates is proportional to the rate at which states change. Also depending upon the tolerance level, the system can have a rapid impulse response.

Enquirer Approach: This is a simple variation of the above approach, where an update is sent only upon explicit request from a subscriber. Each subscriber may set its own tolerance limits and monitor the variation. Having initialized with the information, $[\Delta^t_i, t, S_{ii}(t)]$, a subscriber requests for an update whenever the value $|\Delta^t_i(dt)|$ exceeds the chosen tolerance limit, relieving some burden on the source component. Since all information flow is by demand from a requestor, impulse response can be poor if the requestor chooses poor tolerance limit.

Example 2. Routing by Pressure Propagation

This example abstracts a common situation that occurs in web services. It illustrates how components communicate their state to each other, so that each component can make decisions to improve the overall quality of service. The behavior, b, can be summarized as follows: The system is a collection of components, each of which receives transactions from outside. Each component is capable of processing any transaction, regardless of where it enters the system. Each component maintains an input queue of transactions and processes them sequentially. When a new transaction arrives at a component, it is entered into the input queue of a *selected* component. This selection is the autonomic aspect here and the objective is to minimize the response time for each transaction.

The selection algorithm, α, in each component takes into account the distances between the components and estimated service times of all the components in the system. Each component i takes

a constant time, μ_i, to serve a transaction (determined by the level of resources it has). It takes τ_{ij} units of time to send a transaction from component *i* into the input queue of component *j*. Thus a transaction that arrived at component *i* and served at component *j* has the response time $\tau_{ij} + (1+ Q_j) \mu_j$, where Q_j is the length of the input queue at component *j*, when the transaction is queued there. In order to give best response to the transaction, component *i* chooses *j* which minimizes the above expression - that is, $\tau_{ij} + (1+ Q_j) \mu_j \leq \tau_{ik} + (1+ Q_k) \mu_k$, for all *k*. But component *i* has no precise knowledge of Q_j and hence must resort to speculation, using the collective observation scheme.

Thus, as described in the collective observation scheme, each component *i* maintains the speculative state $[\Delta_j^t, t, Q_j(t)]$, from which the queue size of component *j*, at time $t+dt$, can be estimated as $Q_j(t) + \Delta^t_j(dt)$. For a request arriving at time $t+dt$ at component *i*, the estimated response times, $\tau_{ij} + (1+ Q_j(t) + \Delta^t_j(dt)) \mu_j$, for all *j* are computed, sending the request to the target with minimal response time. When $|Q_i(t) + \Delta^t_i(dt) - Q_i(t+dt)|$ exceeds a tolerance limit, the component *i* will broadcast an updated value $Q_i(t+dt)$ to all the components.

In reality, a number of variations of this problem occur: components perform multiple functions, and a transaction is actually a series of requests for specific functions, which are determined dynamically after each function is performed. Also some functions may be generic so that they can be performed at any component and others may have to be performed at a designated component, because of state information resident at that component. Suitable modifications can be made to the model to deal with these variations. One can also extend this model to incorporate node/link failures and recoveries and the algorithm can adapt accordingly by changing the state values and thereby achieving self-reorganization and self-healing.

Example 3. Work Load Manager

This example illustrates how an enterprise system comprising of multiple servers and operating system instances can balance resources for varying and unpredictable workloads. This is a terse abstraction of the work load manager, whose details can be found in [3]. Once again, the problem reduces to maintaining a reasonable estimate of the global state, based on which any policy (such as the one illustrated here) can be implemented.

The behavior, β, can be summarized as follows: The system is a collection of components, called service classes. They are ordered by an attribute called, importance. Each input to the system, called transaction, has a designated service class and arrives at the respective component. Each component *i* has a goal, G_i, for its transactions, which is an index involving metrics such as response time, velocity *etc*. The actual performance achieved by component *i* at any time, P_i, can be computed from the input-output history of the component. The amount by which a component is missing its goals is characterized by the expression $M_i = (G_i < P_i) \; ? \; 0 : (G_i - P_i)$. The global objective, η, isb to minimize M_i for all components, in the order of their importance. The system is equipped with a set of resources, which are distributed among the components. Given that the current resource level at component *i* is R_i, the function, $\Delta_i(R_i)$, computes the rate at which the performance, P_i, can be improved by increasing the amount of resources. The autonomic algo-

rithm, α, redistributes the resources among the components to optimize η, using the following strategy.

The algorithm selects a *receiver* component, *i*, of highest importance having the largest positive M_i, a resource *R* that has the highest gradient, $\Delta_i(R_i)$, for the current resource level at *i*, and a *donor* component, *j*, of lowest importance having highest level of resource *R* and possibly doing well *(i.e. M_j=0)*. Based on $\Delta_i(R_i)$, certain number of resource *R* are transferred from component *j* to component *i*. Thus, in order to implement this algorithm, a component needs to know, $\Delta_i(R_i)$, M_i, R_i, for all components, *i*. The gradients $\Delta_i(R_i)$ are computed from real time plots. The pair, $[M_j, R_j]$, is the contribution of component *j* to the environmental state. Each component maintains an estimate of the environmental state. The subscriber method can be adopted to periodically improve the estimates.

Example 4. Global Resource Manager

Large SMPs can be divided into multiple partitions, where each partition runs a separate instance of the operating system. Partitions provide a flexible way of virtualizing different services that run in protected boundaries and can communicate in non-intrusive ways. At the same time, basic resources of the system can be shared by many partitions, dynamically rolling unused resources into needy partitions [4]. This dynamic resource movement is another instance of autonomic behavior, where each component must speculate on the global system state and react to changing needs of the components in the system. Websphere implementations on multiple partitions is an example of such a system.

The components in this system are the partitions. The inputs are the jobs entering at each partition. Each input in this system can be a dynamic sequence of functions, which are spread across the partitions. The series is dynamic in the sense that the next function in a job is determined by the outcome of the preceding function. The net effect is that the load at each partition is a random process. Given the resource requirements for functions and the rate at which resource levels improve their performance, one can design various policies to distribute the resources among the partitions. But the key ingredient is the dynamically changing global information, *viz*., the load and resource level at each component. Each component must speculate on the global state, make resource transfer decisions based on the speculation and update its estimates periodically. Since the formulation is similar to the previous example, we omit the details here.

Example 5. Affinity-based Thread Scheduling

This example is illustrative of many operating system functions where complete knowledge for optimal solutions is either very expensive to obtain or simply not possible. Hence one resorts to suboptimal solutions based on heuristics and constant effort is made to correct them as more information becomes available. The following is only illustrative of the techniques and it should not be construed as a recommended heuristic. Modern operating systems have much more sophisticated models for thread scheduling.

The system has a set of processors and a set of threads. A processor goes through the usual states of running and idle, while a thread goes through the states of running, ready (waiting for proces-

sor) or suspended (waiting for some other synchronization). A processor views the time as a sequence of quanta. After running a thread for a quantum, a processor returns the thread to either ready or suspended state and selects another thread to run during the next quantum. It idles when there are no threads to select. When a thread gets out of a suspended state, it selects an idle processor to run. If all processors are busy, it enters the ready state. Processors may fail or be taken away by external resource managers and may reappear later. The objective is to minimize the time spent by a thread in the ready queue and minimize the number of cache misses incurred by a thread. A thread incurs some penalty if it reruns on a processor whose cache may have been polluted by running other threads in the interim. Thus, one needs to balance the two objectives of keeping processors busy and at the same time try to run a thread on the same processor if possible. Since the behavior of the threads is not known apriori, one must resort to observation and some predictive strategies based on history and this is where autonomic design principles come into play.

A processor discretizes time into quanta and each quantum is scheduled to run a thread. Each processor, i, can maintain a sequence of recent history of, $[m^i_k]$, which is the number of misses it suffered in quantum k. Likewise, a thread, t, can maintain a sequence of recent history of, $[p^t_k]$, which indicates that it ran during quantum k of processor p. When a thread is to be scheduled, one can determine the best processor comparing its history $[p^t_k]$ with the histories $[m^i_k]$, of all processors. However, the process histories $[m^i_k]$ are private to a processor. When multiple processors attempt to access them, it causes contention and performance degradation. Each processor can speculate on the miss sequence of every other processor and make the decisions based on the estimate. Periodically, the estimates can be updated using the collective observation mechanism.

4 Hardware Aids for Autonomic Servers

Autonomic behavior equally applies to hardware components as well. However, since it is expensive to embed a variety of implementations in hardware, their focus takes a shift. Many times, hardware is designed to capture all possible outcomes of an operation and report them to a software facility to facilitate meaningful actions. At other times, when the alternatives are less expensive, multiple hardware components are incorporated along with the logic to select them based on statistical observations. Both these aspects are illustrated below with examples drawn from server systems.

4.1 Load-Store Errors in Memory Subsystems

As pointed out earlier, one of the aspects of autonomic servers is to detect faults and gracefully get around them, even if the faults may be unrecoverable. This means that faults must be containable and must have well-defined fault-isolation boundaries. When a fault cannot be recovered, the components interacting at the boundary must be able to recognize this and gracefully switch to alternative strategies. A practical example that enforced such a discipline is the load-recovery mechanism implemented in the IBM pSeries systems using IBM Power4 (and follow-on) processors, which is summarized below.

In conventional designs of processor-memory subsystems, failure detection is often based on time-outs. For instance, when a processor submits a load request to a memory subsystem (which, in these days, can be a complex maze of several levels of hierarchy), the processor expects the data within a time-out period. If data does not show up within that time, the processor usually "check-stops" which brings the entire system to a screeching halt, with very little prospect of any meaningful continuation. In Power4-based pSeries systems, data storage at all levels is equipped with an additional bit indicating whether data is corrupt. A data item along with this bit, whether it is corrupt or not, is moved along all the hierarchies of the memory subsystem, as per normal protocols. But when it reaches a processor (upon a load into some register), the processor is presented with a precise fault, indicating which load instruction failed in getting valid data and thereby providing an opportunity for the software to determine the fault-isolation boundary - which can be all the processes that might be sharing the address space containing that location. The software has the ability to terminate a subset of the processes and continue without disrupting the rest of the system. In the event a corrupt data is never read, no harm is done. Of course one could argue on the other hand, that it may have been better to detect which entity caused the error. A partial solution to this problem is achieved by reporting store-faults through an interrupt. Note that by the time a store fault is detected in the memory subsystem, the process that executed the store operation is no longer tractable (it may have terminated) and hence only an asynchronous fault can be presented to the software, which still provides an opportunity to isolate the boundary affected by that location.

In general, at every level of a design, an autonomic system must examine the following three choices, in that order: First, if the fault is correctable, then clearly one can proceed after correction. This is the standard technique of using ECC and Checksums in hardware. Secondly, if the fault cannot be corrected, (e.g., a multi-bit error or a time-out) the designer must incorporate any possible alternative strategies that can be employed. As described earlier, an autonomic server must switch to an alternative when possible. Finally, when no alternative strategy is possible, the server must indicate this in its response and propagate the fault upwards to other servers that depend upon this.

4.2 Branch Predictors

Branch predictor is a classic example of autonomic behavior in hardware. A number of strategies for predicting the outcome of a branch are in vogue - some use the branch history, some use the paths and so on. A unique quality of a branch predictor is that it is easy to determine how well a predictor is performing, since the actual outcome is known within a short time. A processor can employ multiple predictors, which are the alternative implementations, Π, in an autonomic system. Each alternative is constantly rated after the outcome is known. The algorithm, α, selects the predictor that has the highest rating at any time.

4.3 Support for Dynamic Resource Changes

The global resource manager discussed in Example 4 is intended to move resources such as processors, memory and devices between partitions. While the autonomic system makes the decision son resource movements, the actual movement requires extended support in hardware, firmware and software, which discussed in detail in a companion paper [4].

5 Conclusive Remarks

As systems get increasingly complex, natural forces will automatically eliminate interactions with components whose complexity has to be understood by an interactor. The only components that survive are those that hide the complexity, provide a simple and stable interface and possess the intelligence to perceive the environmental changes and struggle to fit into the environment. While facets of this principle are present in various degrees in extant designs, explicit recognition of the need for being autonomic can make a big difference and thrusts us toward designs that are robust, resilient and innovative. In the present era, where technological changes are so rapid, this takes even greater importance that adaptation to changes becomes paramount.

The first aspect of autonomic designs that we observe is the clear delineation of the interface - how a client perceives a server. Changes to the implementation of the service should not compromise this interface in any manner.

The second aspect of an autonomic server is the need for monitoring the varying input characteristics of the clientele as well as the varying response characteristics of the servers on which this server is dependent. In the present day environment, demands shift rapidly and cannot be anticipated all the time. Similarly components degrade and fail and one must move away from deterministic behavior to fuzzy behaviors, where perturbations do occur and must be observed and acted upon.

Finally, an autonomic server must be prepared to quickly adapt to the observed changes in inputs as well as dependent services. The perturbations are not only due to failures of components, but also performance degradations due to changing demands. Autonomic computing provides a unified approach to deal with both. A key aspect of autonomic behavior that we identified in all our examples is the timely knowledge of the global state, so that the system can respond to the changes. As we illustrated, a collection of autonomic components can collaborate on propagating local information to all the components, so that the community as a whole can benefit from the global knowledge.

We have tried to present a framework for designing autonomic components. While the costs and trade-offs are different from system to system, we envision a general structure emerging. Of particular use will be the formalization of some of these models, with suitable parameters, and bring in the concepts from control theory to establish optimality and stability of algorithms. Our ongoing work focusses on this aspect. Here we reviewed a few examples that we closely worked with in server designs and tried to cast them in this light.

6 References

[1] Paul Horn, “Autonomic Computing”, http://www.research.ibm.com/autonomic

[2] Irving Wladawsky-Berger, “Project Eliza”, http://www-1.ibm.com/servers/eserver/introducing/eliza

[3] J. Aman *et al*., “Adaptive Algorithms for managing a distributed data processing workload”, IBM Systems Journal, Vol 36, No 2, pp 242-283, 1997.

[4] Joefon Jann *et al*., “Dynamic Reconfiguration: Basic building blocks for Autonomic Computing for IBM pSeries” in this publication.