RZ 3381 (# 93427) 11/26/01
Computer Science 14 pages

Research Report

Allocating Resources to Unknown or Non-Linear Systems Using Feedback

Sean Rooney

IBM Research
Zurich Research Laboratory
8803 Rüschlikon
Switzerland
sro@zurich.ibm.com

Allocating Resources to Unknown or Non-Linear Systems Using Feedback

Sean Rooney

IBM Research, Zurich Research Laboratory, 8803 Rüschlikon, Switzerland,
sro@zurich.ibm.com

Abstract

Recent work has attempted to achieve better resource allocation in complex computer systems by using feedback control. Such work typically assumes that the behavior of the underlying system can be modeled using a set of linear difference equations and that those equations are known at the time of the controller's design. Determining a model for an arbitrary set of interdependent applications is non trivial and there is no guarantee that it will be linear. We describe a feedback controller capable of allocating just enough resource to permit a required system behavior without knowledge of the system's model. We prove that the algorithm the controller uses guarantees convergence given certain assumptions and we compare the transient behavior of our general purpose controller with a classic proportional integral controller for a variety of different underlying systems.

I. Introduction

Control is: *the process of causing a system variable to conform to some desired value, called a reference value* [1]. Typically a feedback controller measures the output from the system, calculates the divergence from the desired output and changes the input using knowledge of the system model.

The problem of computer application resource allocation can be framed in terms of control theory: the set of computer applications is the system; the allocated resources (cpu, memory, bandwidth, etc.) are the system input, while some measurable aspect of the application's performance, e.g. the number of video frames produced per second, delay in returning a web page, is the system output. Ideally the amount of resource allocated to an application can be maintained at a level just sufficient to guarantee a desired application behavior. This is of particular interest in an environment in which the owner of infrastructure is leasing resources to third parties and is contractually obliged to provide a minimum level of service to those third parties, but is anxious not to over allocate resources.

Recent work [2], [3], [4] has proposed using feedback controller for resource allocation within computer systems. This work typically assumes that: a model of the computer system is available to the designer of the controller and that this model is linear[1].

Using classic control theory to calculate computer system resource allocation presents some challenges:

- The behavior of non trivial computer applications is not linear with resource allocation. For example, every threshold in the application, in the operating system and in the hardware creates a point of non linearity.
- Even if a good enough linear approximation exists for the application behavior, it is still difficult to include all important parameters, for example, with a given resource allocation a http daemon serving static html pages may have a very different behavior when the pages have to be dynamically generated. [5] points out that normal linearization techniques may not be applicable to computer real-time systems as the system inputs and states are not clearly defined.
- Assuming an adequate linear model exists for two applications separately, there is no guarantee that their joint behavior will be a simple combination of their separate ones.
- Finally, while an adequate linear may exist, it may not be available to the environment in which the applications are executing. For example, when the application is being hosted by a third party.

[6] describes an architecture which delegates the control of resource allocation to clients within an ASP infrastructure. It uses a simple feedback loopback with which a client can indicate the need for more, less or the same amount of resources at regular time interval, allowing adequate resources to be allocated to an application without the application knowing about the nature of the infrastructure, or the infrastructure about that of the application. This paper gives a much more detailed description of that architecture's controller.

First we explain the basic design of the generic controller; we prove that convergence of the resource allocation is guaranteed under certain conditions. We compare its behavior with that of a Proportional Integral (PI) controller for the same system and reference value. We discuss the dynamic response of the controller in tracking fluctuating reference values and we conclude by discussing the use of the controller within the architecture.

II. Design of the Generic Controller

The controller assumes that some entity is capable of determining if the application has enough, too much or too few resources to perform its function. This entity could be the application itself, a dedicated resource monitor for the application or a human; the actual use in practise is explained in Section IV.

Assume that there is only one resource type u and that u_k is the amount of resource allocated to the application during the time period k. From the point of view of the monitor there are two thresholds for the application: an acceptable behavior and an acceptable cost. Both of these are some function of u. Assuming that an acceptable behavior is achievable at an acceptable cost then there are three possible outputs from the resource monitor:

$$y_k = \begin{cases} +1 & \text{too few } (unhappy^-), \\ 0 & \text{enough } (happy), \\ -1 & \text{too much } (unhappy^+) \ . \end{cases}$$

[1] A linear system has the property that: $if\ o_1 = f(u_1),\ o_2 = f(u_2)\ then\ o_1 + o_2 = f(u_1 + u_2)$.

In effect the application specific monitor applies a simple filter to the actual output from the application to produce y_k an application independent output; this is then fed into the application independent controller.

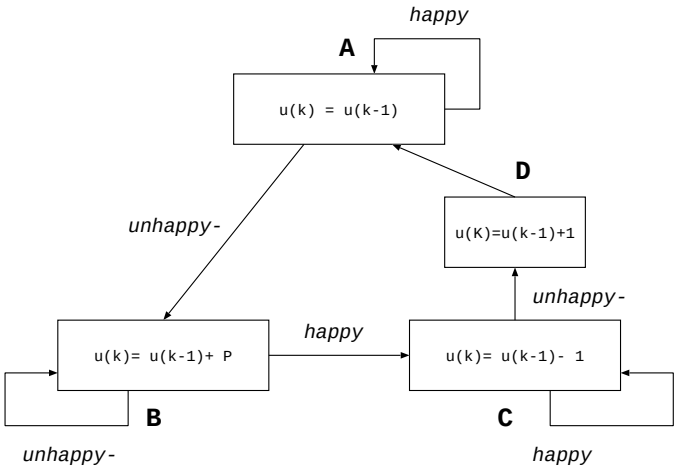


Fig. 1. State Machine for the Controller

The action of the controller during a resource increase, i.e. a change from $unhappy^-$ to *happy*, is described by the finite state machine in Figure 1. If an application was happy during time period $k-1$ it is allocated the same amount of resources in time period k. This is labeled state **A** in the diagram and is called the *stable* state. If has too few resources in time period $k-1$, then in time period k they are increased by amount P, this is labeled state **B**. When the application leaves state **B** it is *happy*, but not necessarily *minimally* happy, i.e. the application may be satisfied with the resource allocation, but could actually make do with less. In order to achieve a minimal resource allocation we reduce the allocation by unity while the application is still *happy* until it is $unhappy^-$, labeled state **C** and then add unity back to the resource allocation, state **D** and return to state **A**. State **C** and **D** are called the tuning states.

To write this algebraically we introduce four variables a_k, b_k, c_k and d_k, which are equal to one when in the corresponding states **A**, **B**, **C** and **D** and zero otherwise:

$$1 = a_k + b_k + c_k + d_k \quad \forall k \tag{1}$$

The state machine can then be described by the following equation:

$$u_k = a_k \cdot u_{k-1} + b_k \cdot [u_{k-1} + P] + c_k \cdot [u_{k-1} - 1] + d_k \cdot [u_{k-1} + 1] \tag{2}$$

The difference between the change of state from $unhappy^+$ to *happy* and from $unhappy^-$ to *happy* is simply the action in state **B**, i.e. P is subtracted rather than added. The stable state **A** is clearly identical, while regardless of whether the *happy* state is reached by adding or subtracting resources, the resource allocation still needs tuning to be *minimally happy*; so the states **C** and **D** are also identical. This observation gives the general equation:

$$u_k = a_k \cdot u_{k-1} + b_k \cdot y_k \cdot [u_{k-1} + P] + c_k \cdot [u_{k-1} - 1] + d_k \cdot [u_{k-1} + 1] \tag{3}$$

Simple manipulation and the use of Equation 1 gives:

$$u_k = u_{k-1} + b_k \cdot y_k \cdot P - c_k + d_k \tag{4}$$

The cyclical and non branching nature of the finite state machine shown in Figure 1 means that being in a given state depends at most on: whether the controller was in that state in the previous time period; whether the controller was in the only other state that can lead to that state in the previous time period; the value of y_k in the previous time period. For example, if b_{k-1} is equal to one then a_k must equal zero, as there is no possibility of going from **B** to **A**

in one time period. This observation allows the state variables to be expressed as follow:

$$a_k = (1 - abs(y_{k-1})) \cdot (a_{k-1} + d_{k-1}) \tag{5}$$
$$b_k = abs(y_{k-1}) \cdot (a_{k-1} + b_{k-1}) \tag{6}$$
$$c_k = (1 - abs(y_{k-1})) \cdot (b_{k-1} + c_{k-1}) \tag{7}$$
$$d_k = abs(y_{k-1}) \cdot (c_{k-1}) \tag{8}$$

From Equation 8, d_k can be written in terms c_{k-1}, and Equation 6 can be rewritten using Equation 1 such that b_k can be expressed in terms of c_{k-1} and d_{k-1}. Simple manipulation then allows Equation 4 to be rewritten only in terms of state variable c.

$$u_k = u_{k-1} + P \cdot y_{k-1} \cdot abs(y_{k-1}) \cdot [1 - c_{k-1} - abs(y_{k-2}) \cdot c_{k-2}] - c_k + abs(y_{k-1} \cdot c_{k-1}) \tag{9}$$

In Equation 9 the action of the controller is expressed in terms of the previous input to the application, the last two outputs from the application and the last three value of one state variable of the controller. The relationship between y_k and u_k is non linear, as y_k is a threshold on u_k. Moreover, this non linearity is a necessary criteria for the controller to work and is not something which can be linearized.

Comparison with classic controller is instructive; the general form of a discrete PI controller is:

$$u_k = u_{k-1} + K * [r_{k-1} - o_{k-1}] \tag{10}$$

where r_{k-1} is the reference output, o_{k-1} the actually output during time period $k-1$ and K is the controller gain. Knowing the linear relationship between u_k and o_k, the designer of the PI controller can choose an appropriate value of K, for example using the Root Locus approach, to achieve an appropriate dynamic response. In particular, K can be chosen such that the system converges to the reference value in a well bounded time; such a system is termed *stable*.

Inspection reveals similarities between Equation 9 and Equation 10. The role of the gain in Equation 10 is played by the additive factor P in Equation 9.

If P is a constant and the application does not fluctuate during states **B**, **C** and **D**, i.e. the controller acts quicker than the application reference behavior varies, then it is trivial to prove that Equation 5 will converge. However, if P is small relative to the values of u, it may take a long time to reach a *happy* state, if P is large relative to u, then it may take a long time to reach a *minimally happy* state or alternatively we may overshoot entirely, passing from $unhappy^-$ to $unhappy^+$ in one step.

Instead of a constant we define P by:

$$P = 2^i \tag{11}$$

where i is the number of time periods since the last time y_k changed, i.e. every time we move between the states $unhappy^-$, *happy* and $unhappy^+$ we reset i to zero and in consequence P to one. This means that the longer the system remains in the *same* undesirable state, the faster the increases or decrease in the allocated resources. We now prove that the system will converge:

Proof of stability

Suppose at time zero the controller is in state **A**, the application has resource allocation n and tells the controller it needs more resources[2], i.e. it produces output $y_k = 1$. Suppose the new resource allocation that would satisfy the application is m where $m > n$.

The controller adds an increasing power of two resource units to the applications resource allocation until at time k the new resource allocation, $n + \sum_{i=0..k} 2^i$, is either bigger or equal to m. Either it is close enough to m that the application declares itself happy and the system moves into the fine tuning state **C** or the application has now too much resource.

In order to prove that the system will not oscillate forever, it is necessary to prove that each time y_k changes that the resource allocation is closer to m. If the system has moved from $unhappy^-$ to *happy* this is implicit in the definition,

[2]If the application started with too many resources then $y_k = -1$ and $n > m$; exactly the same argument works in both cases by simply inversing the signs.

so we need only consider the case where the system has overshot, i.e. moved from $unhappy^-$ to $unhappy^+$ in one time period.

At time zero the application's resource deficit is $(m - n)$, while at time k the application's resource surplus is $(n + \sum_{i=0..k} 2^i) - m$

Suppose the surplus is greater or equal to the deficit, i.e we are now no closer to m than when we started. Then:

$$
\begin{aligned}
(m - n) &\leq (n + \sum_{i=0..k} 2^i) - m \\
2m - 2n &\leq \sum_{i=0..k} 2^i \\
(m - n) &\leq \sum_{i=-1..k-1} 2^i \\
m &\leq n + \sum_{i=0..k-1} 2^i + 1/2 \\
m &\leq n + \sum_{i=0..k-1} 2^i \quad \text{(m and n integers)}
\end{aligned}
$$

However the right hand side of the last equation is the resource allocation at time $k-1$, at which time the application said it didn't have enough resource, this is a contradiction and the initial assumption must be wrong. Therefore after each change in y_k the system is closer to the reference value and the system must converge.

Convergence is proved under the assumption that:

- the variation in application requirement is slow compared to the action of the controller;
- the application decision as to whether it is happy or not, depends only on the resource allocation at that moment rather than the time history of resource allocations.

The first assumption is true of all controllers; we will examine this point in more detail in the next section when we compare the behavior of the generic controller with a PI controller with a varying reference value.

The second assumption is unlikely to be true in practise, i.e. there will always be some delay in resources being allocated and the application reacting. If at time k the application makes its decision as to whether it is happy based on its reaction to the input at time $k-1$, then the stability criteria does not hold, the application realizes too late that it has enough resources and because of the exponential growth of P overshoots by more than it original distance from m.

The key difference with the PI controller is that successive PI controller inputs gets closer together as the output approaches the reference value, while the same is not true of the generic controller as the controller has no knowledge of the reference value and the inputs are simply informed guesses.

If after every change to u_k, in the next period u_{k+1} was held at the same value regardless of the value of y_k, then the same proof for stability described previously for application basing y_k on u_k alone would apply to an application observing some combination of u_k and u_{k-1}. If the controller held the input constant for n time periods after a change, then an application using some combination of the n previous time periods to determine y_k would be stable.

However, the longer the controller holds a value the more sluggish is its performance. This is exactly analogous to the case of the gain in a PI where higher gains lead to better performance, but a high enough gain causes the system to oscillate.

If the controller can recognize that it is not converging, then it can increase the hold period and reset the resource allocation. In this way the controller can adapt itself to applications whose outputs depend on arbitrary time histories of input. In Section III we will examine the performance of the controller in regard to such systems.

We have been assuming that the input u_k can be expressed as an integer, in order to make it more general, every time we reset P i.e. every time we overshoot or undershoot the reference value, we set P not to 1 but a random number in the range 0...1. This also has the advantage that when a periodic reference value is changing faster than the controller actions, the controller's action is not periodic and in consequence is not locked into following the reference value without ever reaching stability.

III. Comparison with Proportional Integral Controller

In this section we compare the behavior of the generic controller described in the previous section with a PI controller for a variety of different systems. It should be noted that although we will speak of the reference resource allocation for both controllers, the PI knows and uses the reference value, while it is unknown to the generic controller.

A. Time to stabilize

Figure 2 shows how the number of time steps to converge to stability varies with the reference value for the generic controller. The underlying system is defined simply by $o_k = u_k$, i.e. the input is simply the output, although this relationship is unknown to the generic controller. The error margin is $\pm 5\%$ and the initial value of u_k is always zero. A PI controller with gain 0.1 is shown for the same system and reference values.

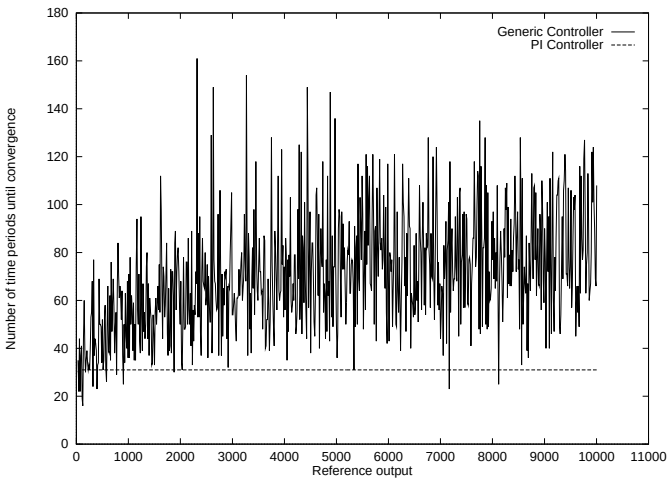


Fig. 2. Number of time units required to reach convergence for PI and generic controller, system is $o_k = u_k$, PI gain = 0.1, reference value = 0...10000, initial value = 0

The number of steps the generic controller requires to reach stability is higher for large reference values and more erratic than the PI controller, varying from 20 to 160 time steps. The PI controller always gets within $\pm 5\%$ of the reference value with exactly 31 steps; a higher gain would allow an even quicker convergence of the PI controller. Note that although the generic controllers convergence time increases as the difference between the initial and reference value, it is a logarithm function, this is due to the exponential growth of P. As the time the experiments measure is between leaving the stable state **A** and returning to it, some of the convergence time is actually spent in the tuning state, in which the application is actually happy but the controller is ensuring minimal happiness.

Figure 3 shows how the output evolves during the convergence for both the generic and PI controller (gain = 0.1) for a system defined by $o_k = u_k$, for two different reference values: 1000, 3000.

The PI controller is much smoother than the generic controller. The generic controller overshoots by the difference between the reference value and biggest sum of powers of two greater than the reference value. So the overshoot for a 1000 is quite small, while than for 3000 the overshoot is more than a 1/3 of the reference value. In classic control theory undershoot and overshoot can be treated symmetrically, e.g. a robot arm 2 cm to the left of the reference position is much like a robot arm 2 cm to the right. This is not true of the environment in which the generic controller is intended to be used. An undershoot means the applications hasn't enough resources while an overshoot means that its costs are unnecessarily high. Not having enough computer resources to accomplish a given task is a physical property of the system, while the cost the controller charges to the application is a policy set by the owner of the infrastructure. In consequence, the policy may be set to tolerate a limited period of overshoot, for example by not charging the client until they have announced themselves *happy*.

The paper [2] describes a PI controller for a set of differentiated web servers. The paper assumes the most important resource for the server is the number of kernel threads allocated to it. The PI controller redistributes the kernel threads amongst the set of web servers in order to maintain some ratio of performance between web servers with

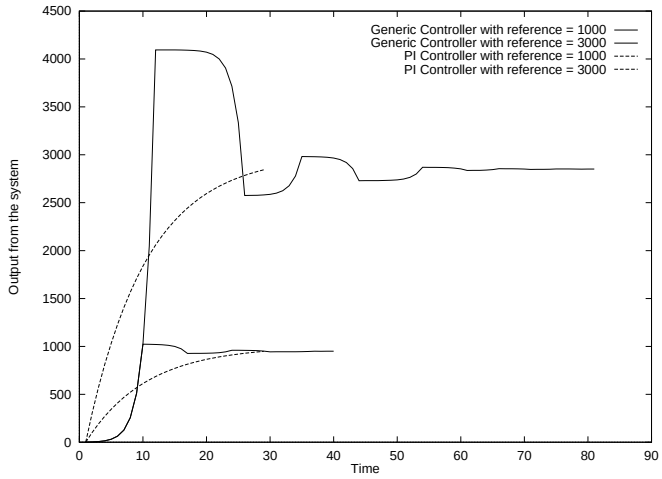


Fig. 3. Output during convergence for PI and generic controller, system is $o_k = u_k$, PI gain = 0.1, reference value = 1000 & 3000

different relative priorities independently of the nature of the content they server which is unknown and can be expected to vary over time.

[2] uses system identification — i.e. the iterative calibration of a linear model against actual response — to determine how the output from the web server is related to the number of process allocated to it. The following linear approximation is given:

$$o_k = (0.74 \cdot o_{k-1} - 0.37 \cdot o_{k-2} + 0.95 \cdot u_k - 0.12 \cdot u_{k-1}) \tag{12}$$

where u_k is relative number of kernel threads and o_k is a relative delay.

Figure 4 shows the behavior of the generic controller and the PI controller for this model. The lag in the effect on output in changing the input, as described by the u_{k-1} term in Equation 12 results in an even greater overshoot than that for the previous system.

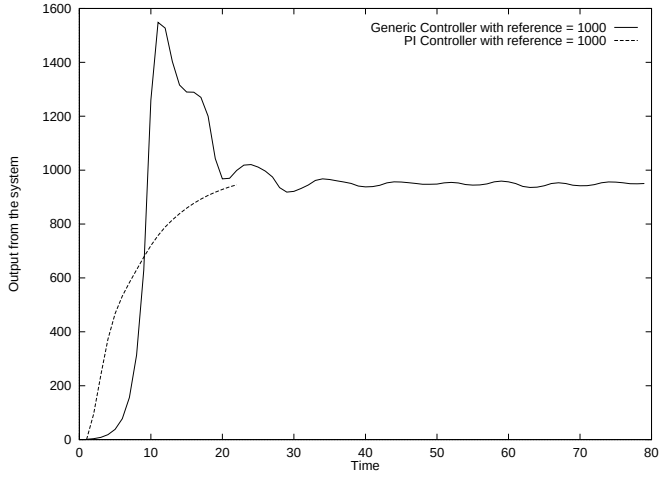


Fig. 4. Output during convergence for PI and generic controller, system is model of a web server ($o_k = (0.74 \cdot o_{k-1} - 0.37 \cdot o_{k-2} + 0.95 \cdot u_k - 0.12 \cdot u_{k-1})$, PI gain = 0.1, reference value = 1000

B. Effect of time delay

In order to further investigate the effect of time delays on the generic controller, Figure 5 shows how the output from the application converges for a system described by: $o_k = u_k + u_{k-1} + u_{k-2} + u_{k-3}$, i.e. the output is related to the input by three separate time delays. As P is dependent on a random number each time the controller over or undershoots, three different data sets are shown. They all have the same shape, although the degree to which they fluctuates varies. Note that negative resources are plotted just to show the pattern of the curves; they do not have any physical meaning, and would correspond to an allocation of zero resources.

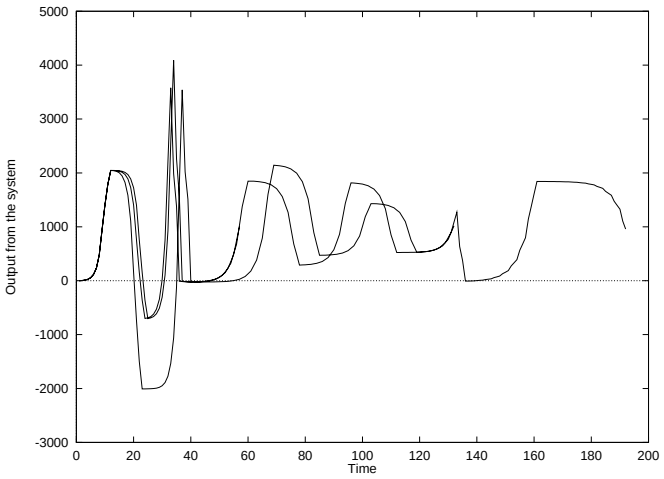


Fig. 5. Output during convergence for generic controller, system has three time delays ($o_k = u_k + u_{k-1} + u_{k-2} + u_{k-3}$), reference value = 1000

The controller is initially started with no time hold and realizes that its not converging. It does this by checking at each change of value of y_k to $unhappy^+$ whether it had already been in the same state previously with a lower input u_k, if so the system is not converging.

When the controller realizes the system is unstable it then increases the time hold by one and resets the system to its original resource allocation, i.e. the resource allocation it had when it was last happy. The data sets chosen are to show the range of behaviors rather than as a random sample, the trace which converges at more than 200 steps is atypical, what it does shows is a trace requiring the hold period to be twice reset; as the hold increases the rise time of the system decreases, this can be seen in the graphs as the gradient of the curves decreases after every reset (i.e. after u_k is set to its original value zero).

Note that if $o_k = u_{k-n},\ n > 0$, then using the method of recognizing instability described here, the system will have a maximum overshoot of: $the\ reference\ value\ \cdot 2^{(n-1)}$. In practise upper bounds imposed by the resources at the infrastructures disposal would prevent huge overshoots.

C. Nonlinear functions

Figure 6 shows the behavior of the generic controller for a system with a non linear relationship between input and output: $o_k = u_k^2$, with reference value 2000. Three data samples are shown again demonstrating that the starting factor after the initial overshoot is chosen randomly. The generic controller actually converges faster for $o_k = u_k^2$, than its does for $o_k = u_k$.

A simple PI controller cannot handle non linear systems directly; by way of example consider the behavior of the controller when given a reference value of 200: at time zero the controller calculates the input as the gain times the difference in reference and zero, this equals 20. The output from the system is 400. The controller then calculates the new input as the old input plus the gain times the error. The new input is again zero and the controller oscillates between these value and never converges. In order to apply control theory to non linear system, a linear approximation must be found using techniques such as Lyapunov theory.

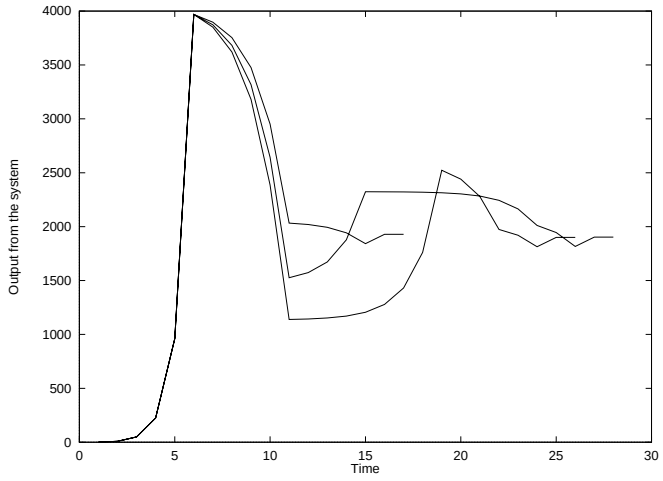


Fig. 6. Output during convergence of generic controller, system is $o_k = u_k^2$, reference value = 20000

The ability of the generic controller to handle non linear relationships between resource allocation and application behavior is one of its most attractive features.

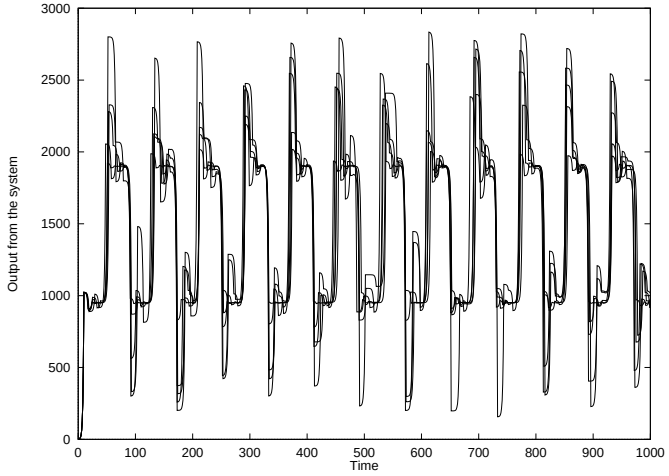
D. Dynamic Response

A controller must not only be able to converge the output of the system to a reference value but also to track that reference value as it changes over time. The resources that an application requires change overtime either because the work it performs is periodic, e.g. a web server is more loaded at lunch time that late at night, or because the client changes the cost function, for example is willing to pay more to have a better service.

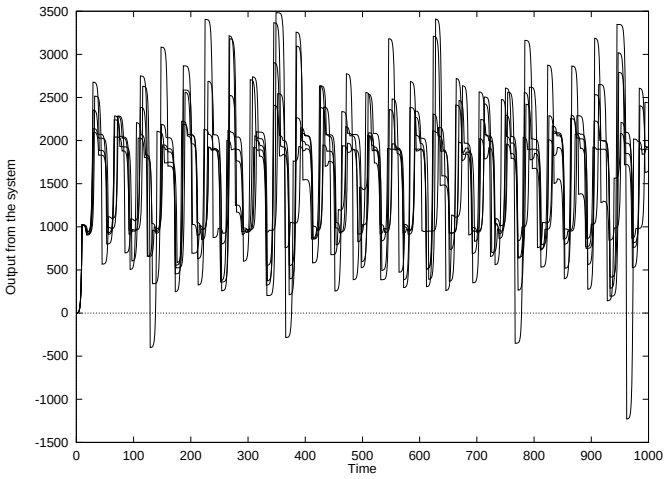
Figure 7 shows the response of the generic controller, to an underlying system $o_k = u_k$ whose reference values doubles, then halves after a fixed number of time periods, i.e. the reference value is a step function.

Figure 7(a), shows that the generic controller is capable of tracking the step function at frequency 40. Although the controller overshoots when the reference value steps up and undershoots when it steps down, the majority of the time the output is at the appropriate reference value. Figure 7(b) shows what happens when the frequency of the step function is doubled, the generic controller follows the reference value, but a much larger portion of its time is spent recovering from over and undershoots.

Figure 8 shows how the PI controller (with gain 0.1) deals with the step function with frequency 20 and 40. At frequency 40 (Figure 8(a)) the PI controller just has time to reach the reference value before it changes, at frequency 20, the PI controller never reaches either of the two reference values and is always caught between them. Better tracking could be achieved with a higher gain, but normally a PI controller is designed for a particular system with a particular reference input. A PI controller designed to optimally follow a step input, will not follow a parabolic one.

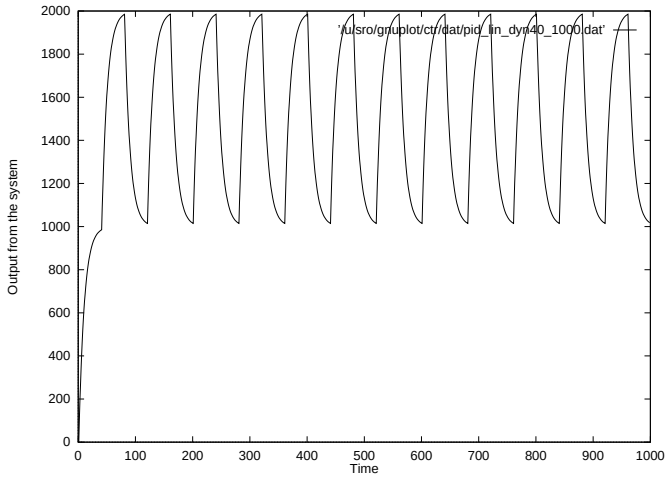


(a) Step function with frequency 40

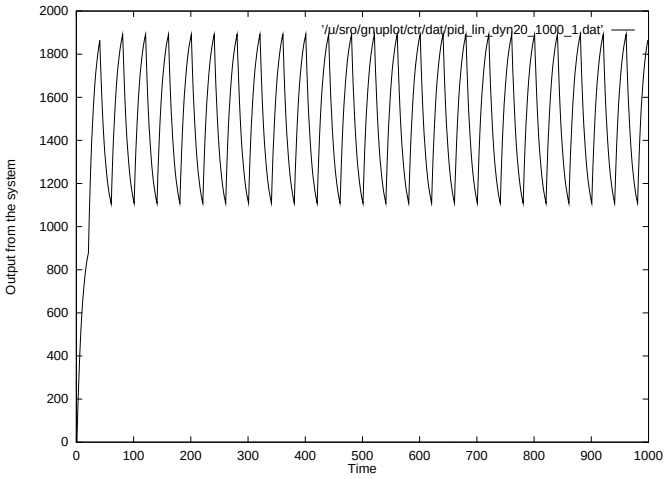


(b) Step function with frequency 20

Fig. 7. Output during convergence for generic controller, system is $o_k = u_k$, reference value = step function(1000,2000)



(a) Step function with frequency 40



(b) Step function with frequency 20

Fig. 8. Output during convergence for PI controller, system is $o_k = u_k$, gain = 0.1, reference value = step function(1000,2000)

IV. Use of the Controller

[6] describes the general architecture in which the generic controller is used and is the context in which it was developed. The objective of this architecture is to permit very large number of different client applications to be hosted on the same physical infrastructure.

In order to reduce the number of physical devices required, multiple different clients are hosted on the same devices. Clients are allocated some partition of one or more physical servers in which to run a set of potentially dynamically loaded applications. The server partition appears to the application as if it were a normal server and in this paper we will refer to it as *a virtual server*.

The set of virtual servers are interlinked via a partition of the owner's network and made available to the public internet via a partition of a proxying device, also running client chosen proxying code. The unified set of partitions on the various different devices is called an *iCorp*.

In order to limit the amount of human operator time required to manage the infrastructure, the process of adding, removing and modifying *iCorps* is entirely automated and under the client's control. Initially a client is given one from a set of predefined resource allocations but the client may modify this either because it is inappropriate or the client's needs may vary over time.

Within each virtual server is a client accessible interface which allows the client to indicate to the infrastructure whether it wants more, less or the same amount of resource. Invocations on this interface are sent to the architectural entity running on the physical server responsible for managing the virtual servers. How and under what circumstances the client invokes the interface is entirely up to the client. The client can manually telnet into the virtual server, observe the behavior of their application and make the appropriate call, or have a piece of software running inside the virtual server perform the same action on their behalf.

A set of generic controllers is run on the physical server for each virtual server. For a given virtual server, each controller controls one resource; in the current implementation there are two managed resources: the fraction of the servers physical cpu allocated to the client and the amount of network bandwidth that it may use.

A client is free to send messages to the controller as many times as it wishes, but the messages are written to a mailbox which the controller reads at a rate it determines. The mailbox is capable of buffering only a small number of messages, new messages overwrite old ones when the buffer is full. In this way a badly behaving client cannot force the controller to spends its entire time resource updating.

Our initial tests have involved trying to adjust the resource allocation of a video server. Both the virtual server and physical servers run the Linux OS; the virtual server is created using the VMWare [7] product. The virtual server appears to the physical server as a Linux process and its cpu can be modified by adjusting the process priority using the *renice* option. Note that there is not a simple relationship between the priority allocated to a process and the number of processing cycles it actually receives, but the generic controller does not require there to be one. The amount of bandwidth that the virtual server can produce is controlled by the Linux traffic shaper.

Although it would be possible to write applications that directly communicate with the generic controller, this would inhibit the use of existing applications. Instead a small client supplied monitor runs on the virtual server. It observes the number of users the video server is serving and the number of frames being produced per seconds. When the number of frames per second per user falls below a certain threshold it asks for more resources. To what extent the client should or can be able to identify which resource it is lacking is an open question, currently all resources are increased and decreased independently, i.e. it is allocated both more cpu and more bandwidth. Figure 9 shows the relationship between the entities in the architecture, only one generic controller is shown for reasons of simplicity.

The video server is a freeware version of the commercial Real Server from Real Network, this version does not have much in the way of performance monitoring tools so we observe the performance of the video server indirectly examining the totality of the traffic being produced by the Linux virtual server (via the pseudo */proc* file system), and the number of end users by observing the number of open connections on the well known Real Server port 554. By assuming that the only traffic produced by the virtual server is video frames, we can calculate the average number of frames per user. This is step **(1)** in Figure 9.

The monitor periodically tests if the current performance is adequate by calculating the number of kbp per user per second. It attempts to maintain a minimum number of kbps to each user to permit an acceptable quality of playback at the user's video player.

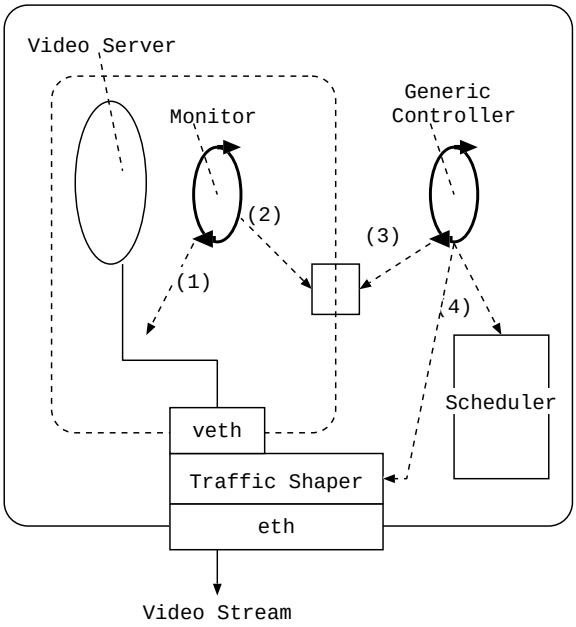


Fig. 9. Overview of controller for Video Server

The monitor cannot depend on operating system sleep instructions to be accurate, especially as the physical servers get more and more loaded. It asks for a nominal time to be set to sleep, but when woken checks using the Linux real time clock how much actual time it was a sleep for and it is this figure it uses to calculate the performance.

If the performance is not adequate, then the monitor puts a request for more resources into the mailbox, this is step **(2)** in Figure 9. The generic controllers periodically reads the mailbox **(3)**, and when instructed attempts to change the process priority of the virtual server and the amount of traffic it is allowed to send **(4)**.

Cost is not considered in the current implementation, not because it presents a particular technical challenge, but because more work is needed to determine a suitable charging model. Clients are simply assumed to be frugal in their resource use, for example when the number of video server users decreases the monitor asks for less resources.

Figure 10 shows the behavior of both an unmonitored and monitored video server initially running on a virtual server on an unloaded machine, whose load is suddenly increased by an intensive cpu consuming high priority process (calculating prime numbers) in the 20th time period. The monitor attempts to maintain the play out rate for each user at 128 Kbps. In Figure 10 there is only one user for each server, the transport protocol is RTSP/TCP and the number of TCP segments required to maintain an 128 Kbps video flow is about 25. Initially both video servers are started with p riority[3] +10, the process which creates the server congestion is started with priority -10.

The Real Player software running on the user's workstation itself attempts to react to what it perceives as network congestion by asking the Real server to play out at a reduced rate. However, as it requires almost as much cpu to play out a 64 Kbps film as a 128 Kbps, in the case where the bottleneck is server processing power, this is of no benefit. To give the monitor time to react to the reduced amount of allocated cpu, we intercept the client messages asking for the server to down shift to a lower play out rate and ignore them. This is easily done within the architecture described in [6] as users do not communicate directly with the virtual servers but via virtual proxies. RTSP requests from the real player to the real server are intercepted at the proxies and those requesting down-shifts are removed.

In the 20th time period both video servers suffer a steep drop in the play out rate, but the monitored video servers recognizes that the number of frames has fallen below the threshold and requests more cpu; after about 5 time periods it has fully recovered, raising its priority to -2.

This simple example demonstrates the advantages of the empirical approach described in previous section: to determine the relationship between the frame rate of a Real Server running with a VMWare instance and the priority allocated

[3]On Linux 2.4, the lowest possible priority is +20 and the highest is -20.

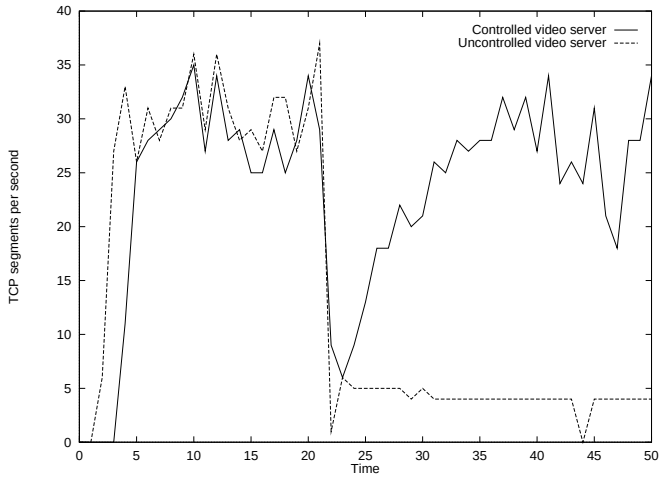


Fig. 10. A controller and uncontrolled video stream suffering server congestion at time 20

to that VMWare instance requires knowledge both of how the physical server and virtual servers map a given priority on a particular scheduling regime and how being scheduled at such a rate effects the frame rate of the video server. This is specific to those particular applications and might have to be changed if a different video server or a different version of the same video server were used. The mapping between priority and scheduling rate is dependent on the priorities and activities of the other processes running within both the physical and virtual servers. In consequence it is extremely difficult to give a general model.

V. Related Work

Some work has already been mentioned which uses feedback control for resource allocation in a computer system [2], [3], [4]. The approach described here is designed for an architecture in which arbitrary client applications are dynamically loaded onto an application service provider's infrastructure. So while, a pure control theoretical approach is more predictable for a given known application, e.g. HTTP daemon, running in a well controlled environment, it is not feasible in the target environment.

Modern operating System research has concentrated on making best use of the available resources rather than given bounded guarantees, this is logical as the various tasks on the OS tend to be executed by the same client or a group of trusted clients. Some real-time operating systems, for example [8], [9], [10] have allowed precise resource guarantees to be given to application, but these have to be specified by the client and associated with a single task or process.

The controller described in this paper supposes an allocation of resources not directly to a client process on a shared OS, but to a client's virtual server within an architecture capable of supporting and controlling many such servers. With the arrival of Linux on a mainframe [11] it is now possible to dynamically create many ten of thousands of Linux images, each with a very fine resource allocation. As client can run whatever they want inside these secure sandbox, they may run their own application specific monitors as described in Section IV

The client has no knowledge of how much resources there are available on the physical server, nor how much is currently allocated to it, but is capable of adjusting to its environment using the generic controller. This is reminiscent of the TCP/IP congestion control mechanism [12] whereby a TCP client attempts to probe the current state of the network to determine the rate at which it should send. The role of dropped packets in TCP is played by cost in the generic controller.

VI. Conclusion

The generic controller described in this paper allows the adaptive resource allocation to an arbitrary computer system. The algorithms that the generic controller use are simple, but permit system stability. Although better control could be achieved using classic control theory if the model of that system was known and was linear, we have argued that the behavior of computer systems is not linear with resource allocation and even if they were, obtaining the linear

model is complex and not really suitable for the dynamic environment for which the controller was designed.

References

[1] G. Franklin, D. Powell, and A. Emami-Naeini, *Feedback Control of Dynamic Systems*. Addison-Wesley, 1994. ISBN: 0-201-53487-8.

[2] C. Lu, T. Abdelzaher, J. Stankovic, and S. Son, "A Feeback Control Approach for Guranteeing Relative Delays in Web Servers," *IEEE Real-Time Technology Sysmposium, Tai Pei, Taiwan*, June 2001.

[3] T. Abdelzaher, K. Shin, and N. Bhatti, "Performance Gurantees for Web-Server End Servers: A Control Theoretical Approach," *Accepted for IEEE Transactions on Parallel and Distributed Systems*, 2001.

[4] S. Cen, C. Pu, and J. Walpole, "Flow and congestion control for internet streaming applications," *In Proceedings of Multimedia Computing and Networking (MMCN)*, 1998.

[5] C. L. *et al*, "Performance Specifications and Metrics for Adaptive Real-Time Systems," *IEEE Real-Time Systems Sysmposium, Sydney, Austrialia*, December 2000.

[6] S. Rooney and A. Bussani, "Client Delegated Control within an ASP Infrastructure," *Journal of Communications and Networks*, vol. 3, March 2001.

[7] VMWare, "Getting Started Guide, VMWare 2.0 for Linux," *VMWare Technical Support*, January 2000.

[8] I. Leslie, D. McAuley, R. Black, T. Roscoe, P. Barham, D. Evers, R. Fairbairns, and E. Hyden, "The Design and Implementation of an Operating System to Support Distributed Multimedia Applications," *IEEE JSAC*, vol. 14, pp. 1280–1297, September 1996.

[9] RTLinux.org, "Workshop on Real Time Operating Systems and Applications and second Real Time Linux Workshop, November 27-28 2000, Florida, www.rtlinux.org," November 2000.

[10] WindRiver, "VxWorks 5.4 Programmers Guide," *WindRiver Production Information*, vol. DOC-12629-ZD-01, May 1999.

[11] IBM, "S/390 Virtual Image Facility for Linux, Guide and Reference," *Version 1.0, Release 1.0 SL0500, GC24-5930-04*, June 1999.

[12] V. Jacobson, "Congestion Avoidance and Control," *Computer Communications Review*, vol. 18, pp. 314–329, August 1988.