OPTIMAL POLICIES FOR MULTI-LEVEL ADAPTIVE DISTRIBUTED COMPUTER SYSTEMS

Mohammad Abdeen, Murray Woodside

Carleton University, Ottawa, Canada

{mabdeen|cmw}@sce.carleton.ca

Abstract

Modern distributed applications, such as distributed multi-media and mobile applications, face unpredictable operating conditions and load variations. Performance cannot be designed into such applications in advance; they have to be able to tune themselves into unexpected environments and to adapt to changes over time. We see many examples of single adaptations in applications and middleware, but the opportunities are even greater if many features of the system, at all levels, are adaptive. This paper proposes an architecture to support coordinated adaptive changes in all levels (application, middleware and operating system), with an optimal controller at its core. The controller uses optimal policies based on Markov Decision Processes (MDP) which seek to satisfy a set of system quality-of-service and resource-usage goals.

Key words: Adaptive systems, Distributed Systems, Adaptive architectures, Mobile applications, Multi-media Applications, Markov Decision Process.

1 Introduction

Modern distributed applications, such as e-commerce and enterprise computing, and many multimedia and mobile applications, face unpredictable environments due to user mobility, load variations, evolution of user access patterns, and varying resource availability. Figure 1 describes a typical example with a mobile user who moves from a radio LAN to an Infra-red sub-network with much lower bandwidth and a much higher error rate. To adapt to this move, the system must identify the need for a change, decide on the change and implement it in a timely way. Rapid

changes or disturbances are the most challenging, but slower disturbances, taking place over days or months, are also important. For slower disturbances the adaptation is a kind of self-tuning to track changes in the environment.

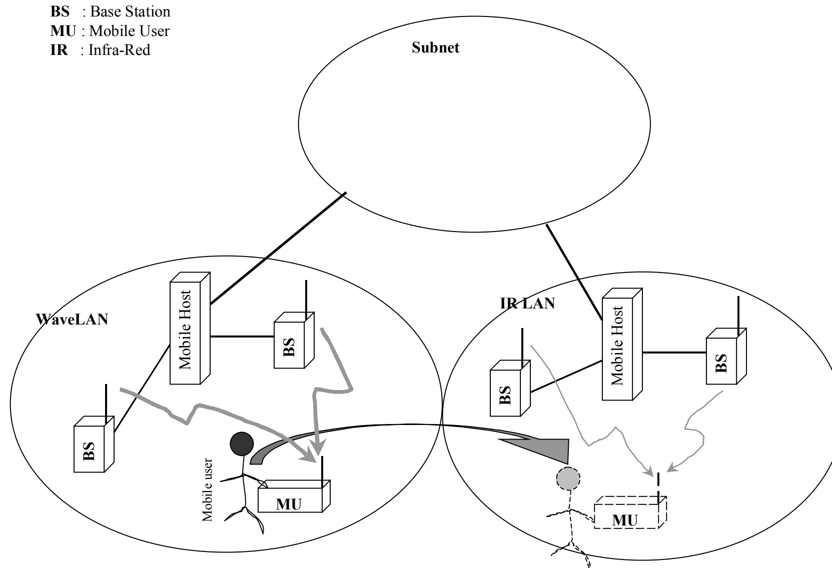

Figure 1 A mobile user roaming among multiple networks.

Adaptive features have been described, under names such as “reflection” [12] [14] [15] [16], “metaprogramming” [20], and “reliability architectures” [21]. We regard these systems as special kinds of feedback control system, with the elements of *sensor, decision-element* or *controller*, and *actuator*, as indicated in Figure 2a. Figure 2a show a single control loop, with one sensor controlling one actuator, which is the most common form in reported work. The loop controls one feature of the system, such as

- a degree of imprecise computation [23], driven by computing resource availability
- the number of threads of a server, driven by its queue size (in web servers)
- moving components between nodes of a distributed system, driven by their relative load (by a reflective ORB such as TAO [2])

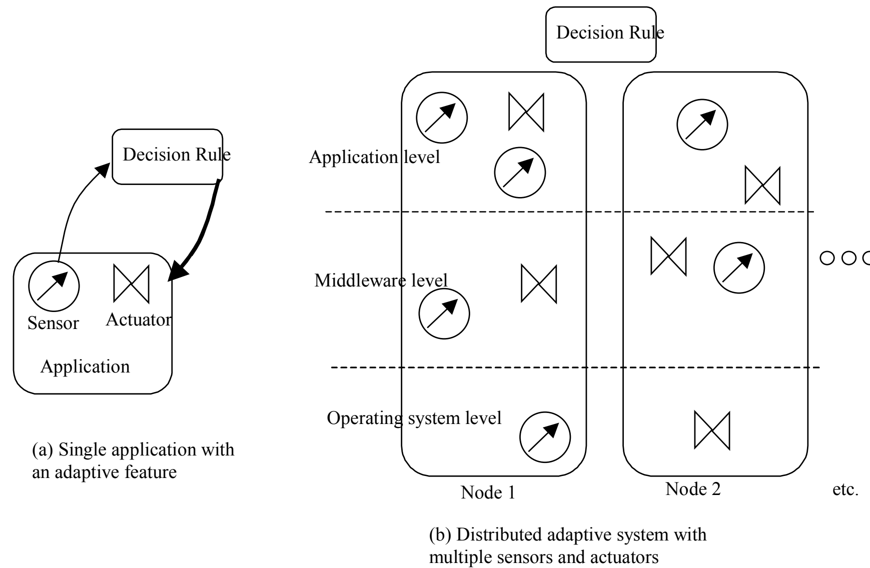

Figure 2 Adaptive systems, showing an isolated adaptive feature and a complex distributed system with multiple sensors and actuators

This paper considers techniques for controlling several system features simultaneously, in a coordinated manner, to react to changes detected by multiple sensors. Figure 2b shows many sensors and actuators in a distributed system.

For example, in the system of Figure 1, to adapt to lower bandwidth while transmitting a stream of images, one might increase the image compression, decrease the resolution, change from color to black-and-white, and reduce the frame rate. These are all application level changes. At the middleware level, the source of the images might be changed (for instance, to a server which stores pre-compressed frames), they might be routed through a proxy server attached to the new network. The protocol could implement frame filtering, packet header compression, data bundling of smaller data packets into a single larger packet for transmission, and packet filtering. At the operating system level, frames could be pre-fetched during periods of better connectivity to cover periods of reduced network capacity, and (in some terminals) power level may be controlled [22].

A system with many sensors and actuators needs a structure that could be called an *adaptation architecture*, with locations for the sensors, actuators and decision modules; in this work we adopt a centralized adaptation

architecture with:

- sensors and actuators embedded in the application components, middleware and operating system, with a standardized interface for communicating by messages with the decision module, and
- a single decision module, which periodically obtains data from the sensors and makes decisions

This work does not consider important aspects of the architecture such as the sensor and actuator messaging interfaces, the choice of the period or periods for decisions, the use of sporadic data sent by the sensors (event-driven adaptation), and the possible advantages of partitioning and distributing the decision module.

The adaptation architecture used here is described first. Then a discrete-state (Markovian) view of the controlled system and the control is defined, leading to a Markov Decision Process (MDP) problem, with a well-known optimal solution. This gives decision rules with discrete alternatives and threshold levels in the sensor measures, similar to previous work with single control loops. The optimization technique is applied to an example based on Figure 1, to investigate the complexity of the calculations, and to show the form of the solutions obtained.

2 An architecture for coordinated adaptation

A centralized architecture with a single decision module is based on the feedback control abstraction sketched out in Figure 2. All kinds of measures of program performance, and all kinds of adjustment mechanisms, are treated uniformly. This section considers the operations carried out by the components indicated in Figure 2, in greater depth.

2.1 The adaptation process

The adaptation process repeats a cycle of estimating (monitoring and tracking), deciding and acting.

Estimation of the operational state of the system requires functions (which we call sensors) to capture measures, which may take a wide variety of forms. This kind of self-awareness of a system goes under the general name of reflection [14], and the same name is also applied to programming techniques and language elements which can be used to implement some sensors at the application level [12] [24]. A more comprehensive view of system self-awareness calls it metaprogramming. These techniques can capture measures that are application specific, such

as message sizes, class or structure of data requests, relative importance of a request, and request contexts.

Sensor in this work also includes other awareness mechanisms such as monitoring by dedicated components in the middleware[4] [6][13], in the operating system [19] and in the application managers. These techniques are necessary to capture system-level performance measures such as throughput, CPU utilization, and average delay, and environment parameters such as subsystems used by a response, levels of competitive workloads. Failure detectors fall into the same group of sensors.

The *decision mechanism* derives new settings for the actuators from an analysis of the measures (feedback control). Knowledge of all the measures is important at this point, as they may identify under-used resources, or opportunities to change the application's behavior. In previous work on adaptation, the decision mechanism is sometimes a threshold-based decision to determine one of a set of pre-determined values of a parameter. For instance, in threshold queue mechanisms, when a queue of messages exceeds a threshold QUPR, additional software servers or server threads may be created. Later, if the queue length is less than another smaller threshold QLR, the additional servers may be removed. This is a rule with hysteresis, as illustrated in Figure 3.

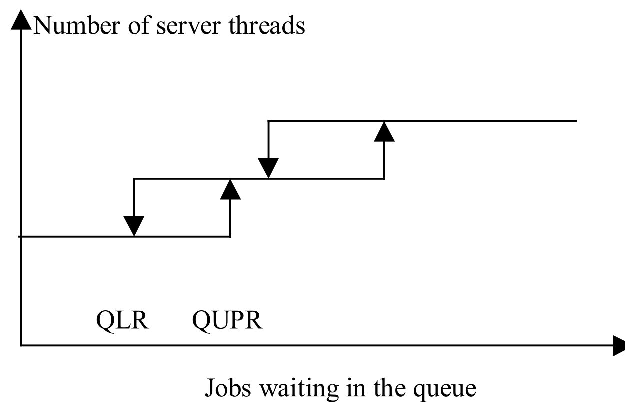

Figure 3 A threshold rule for adapting the number of servers threads for a queue

In this work there may be similar rules to determine a discrete level for a system parameter, but the rule in general depends on many variables instead of just one.

Software actuators are software components that implement the decision to change or tune the system, to help bring it back to the desirable range of operation. They may be built into the operating system, middleware, or

application, as described in the Introduction. Application level mechanisms described here can be combined with middleware such as CORBA, which already provides mechanisms for redirecting service requests, in order to balance load or to replace a failed server. The present work can tie these capabilities into a wider adaptive scheme.

2.2 Adaptation over Multiple levels

Figure 4 below depicts the centralized architecture considered in this paper, and its components.

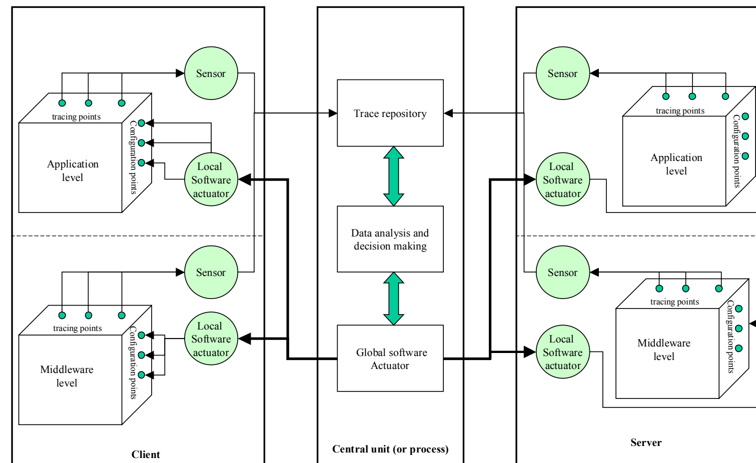

Figure 4 The Centralized multi-level adaptive architecture

The figure shows a distributed application with a client on the left and a server on the right, each with a middleware level and an application level equipped with sensor and actuator components. Sensor data are sent to a central Adaptation unit where a Decision Making Module (DMM) makes adaptation decisions. The DMM then instructs each level to take the required action(s). Software actuators in each level receive adaptation decisions made by the DMM and implement them.

The Decision Making Module

To construct a coordinated adaptive system with the architecture in Figure 4 one has to consider:

- how to choose the points to apply sensors and actuators
- The design of sensors and actuators
- The choice of the decision rules for the Decision Making Module.

In this work we address the third point, assuming that sensors and actuators are available.

In single loop systems it is not very difficult to construct a sensible feedback path with an ad hoc decision function. There is however almost no theory to guide the choice of the *best* feedback function (for example, to define the best values of the thresholds in Figure 3).

For more complex systems, with many control variables derived from many sensor values, intuition does not provide guidance and the lack of theory is even more serious. There is a potential for greatly improved performance, but some way must be found to construct reasonably good decision functions. The possibility of finding *optimal* rules is even more attractive, and here we will attempt to get them from Markov Decision Processes.

3 A Markov Decision Process (MDP) approach to optimal policies

Searching for a way to derive good policies to guide decisions, we consider a discrete-state model for the system, capturing the state information known to the DMM. Over time, state changes

- may be changes forced by the DMM, or
- may be observed but uncontrolled changes in the system, or
- may be the effects of *hidden* changes that are not otherwise known to the DMM

The last two categories of changes will be called *disturbances* and are modeled by random state transitions in a Markov Chain model of the system. The state space contains states Si, for i = 1 to I. In steady state, state Si has probability p_i. At each step, a transition from a state Si to another state Sj has a probability p_{ij}.

The system evolves randomly, but also can be controlled to some extent by the actuators, which can force (or, in general can influence) state changes to improve operation. Control is exerted by choosing values for control variables a, which influence the transition probabilities p_{ij} (i.e. the transition probabilities are actually given

functions $p_{ij}(a)$.

In Markov Decision Processes [1] the best choice of the control variables a, among the set D(i) of control values which are feasible in state Si, is determined. "Best" is intended in the sense of minimizing a given cost function which is the expected value of a function Cost(a, i) of the control and the state. It is possible to compute a rule or policy R which assigns an optimal choice of a to each state i. This is a versatile and powerful tool to analyze probabilistic sequential decision processes with infinite planning horizon. This model is an outgrowth of the Markov model and dynamic programming. It has many potential applications in inventory control, maintenance, resource allocation and others.

Optimal policy

A policy determines the actions to be taken at each decision epoch (moments which we will assume are equally spaced in time). A *stationary* policy R is a rule that always prescribes a single action R_i whenever the system is found in a state Si at a decision epoch.

In order to define an optimal policy, let's assume that the long-run average cost per unit time when using policy R is $g(R)$. A stationary policy R^* is said to be *average-cost optimal* if $g(R^*) \leq g(R)$ for each stationary policy R.

It is computationally infeasible to find the average-cost optimal policy by computing the associated average cost for all possible polices. However, some algorithms can be used to construct a sequence of improved policies until an optimal policy is found. Examples of such algorithms are: the policy iteration algorithm, and the value iteration algorithm.

3.1.1 Relative values

For a given policy R, the total expected cost over the first n decision epochs when starting with state Si, is denoted by $V_n(i,R)$. Starting with a different state other than Si, e.g. Sj, has the effect of changing the total expected cost function. This change in the total cost function is called the relative value $v_j(R)$. It can be shown, as in [1], that the average cost per unit time, $g(R)$, and the relative values, $v_i(R)$, can be calculated simultaneously by solving a

system of linear equations as follows:

$$v_i = c_i(R_i) - g(R_i) + \sum_{j \in I} p_{ij}(R_i) v_j \qquad \text{for each } i \in I \qquad (1)$$

where $c_i(R_i)$ is the cost of the decision $a = R_i$ made at state i, p_{ij} is the transition probability matrix, and is the transition probability from state Si to state Sj if the decision R_i is made.

3.1.2 Policy-iteration algorithm

The relative values associated with a given policy R provide a tool for constructing a new policy R' whose average cost is no more than that of the current policy R. It can be shown that if

$$c_i(R'_i) - g(R) + \sum_{j \in I} p_{ij}(R_i) v_j(R) \leq v_i(R) \qquad \text{for each} \qquad (2)$$

then

$$g(R') \leq g(R) \qquad (3)$$

Equations (2) and (3) suggests that an optimal policy R_{opt} can be obtained be recursively iterating with new values of R' until a minimum cost $g(R^*)$ is reached. The value of R^* is called the optimal policy.

The policy iteration algorithm always converges in a finite number of iterations [1], and empirically it is found to converge very fast on many problems. The number of iterations needed is insensitive to the number of states and of the starting policy, and varies typically between 3 and 15.

4 An Adaptive System

The construction of an optimal set of policies will be described in the context of a concrete example of a mobile multi-media application which could play a video clip, a movie, or support a video conferencing application from a mobile device (e.g. a PDA). Mobile users move among networks of various speeds, and hence of various delays, as illustrated in Figure 1. Competing traffic on each network may affect the quality of service obtained by the user.

If a user is faced with longer delays, more limited bandwidth, and diminished quality of service, two effective adaptation techniques can be employed to provide better service:

- The video frame rate can be reduced. This will cause a flickering effect but will send the essential information.

- The size of each video frame can be reduced by compression. Compressed frames have smaller sizes but the image quality within each frame is less.

The users move among four wireless networks, Net-1, Net-2, Net-3, and Net-4 with speeds of 10 Mbits/sec, 4 Mbits/sec, 1 Mbits/sec, and 200 Kbits/sec respectively. These movements are a random disturbance to the operation of the system, which affect the service quality. As controls, we have

- four different compression levels, level 1 (no compression), 2 (15 times compression), 3 (30 times) and level 4 (50 times).
- four frame rates, level 1 (normal), 2 (3/4 of normal), 3 (half normal) and 4 (1/4 normal).

In the following sections we show how we can build a Markov decision model to obtain an optimal decision policy.

4.1 The system model

A discrete state model is a simplification of a complex system, based on judgement of the key factors in the operation of the system. In this example the factors (already mostly described) are defined by:

- the network being accessed by the user (*N,* with four values 1 to 4),
- the current level of compression (*C,* with four values 1 to 4)
- the current frame rate level (*Fr*, with four values 1 to 4),
- an index of the available quality of service (*Q*, with three levels 1 to 3)

The value of the last factor is a function of the overall operation of the system, and a variety of measures could be used. Here, we consider an "available QoS" measure derived from the network bandwidth available to the user after accounting for contention from others. This could be found from network collision data or utilization data. The range of values that define level 2 of Q is to be chosen to represent "*typical"* expectations from the given network.

The QoS measure is affected by the other variables. We define Q*(*C,Fr,N*) as the most likely value of Q to result from a given the values (*C,Fr,N*). The central typical range of the QoS measure is modified by the

other variables (C,Fr,N). Therefore we can identify sets of values in the (C,Fr,N) space which map to the low range (Q* = 1, representing overloaded conditions), the middle range (Q* = 2, typical conditions) and the high range (Q* = 3, lots of capacity).

The second and third factors listed above are control variables. The decision levels are part of the state if the decisions to be taken naturally depend on the previous level of the same control variables, or if the next state depends on them.

The state S is thus a *tuple* of four values:

$$S = (C, Fr, N, QoS).$$

and the state space has 192 states (4 x 4 x 4 x 3).

The set of possible decision values is referred to as D, which can be expressed as:

$$D = (C_D, Fr_D) .$$

where C_D and Fr_D are the new values for C and Fr.

The transition probability matrix

Some transitions among states express the reaction of the system to control, and some express its reaction to disturbances, as described eariler. The probabilities of the transitions that are affected by disturbances are found by analyzing the behavior of the system, either by models or measurements.

In this example, the disturbances are the changes of network, and some changes of QoS. Suppose $P_{ij}(a)$ is the transsition probablity from $Si = (Ci, Fri, Ni, QoSi)$ to $Si = (Cj, Frj, Nj, QoSj)$. Then we define:

$P_{ij}(a) = P_{net}(i,j)P_{qos}(i,j)$ if S_j has C_j and Fr_j which correspond to the decision a.

$= 0$ otherwise

P_{net} is the transition probability of the user from Ni to Nj. P_{net} can be given by:

$P_{net} = p_n$ if $N_i = N_j$,

$= (1 - p_n)/3$ otherwise.

Similarly, $P_{qos}(i,j)$ relates to changes in quality of service. Suppose that Q*j is the value predicted by (Cj, Frj, Nj) then $P_{qos}(i,j)$ is given by the following table:

	Predicted QoS level Q*(*Cj, Frj, N_j*)			
Target QoS level, Qj		1	2	3
	1	p_q	$(1 - p_q)/2$	$(1 - p_q)/2 - \delta$
	2	$(1 - p_q)/2 + \delta$	p_q	$(1 - p_q)/2 + \delta$
	3	$(1 - p_q)/2 - \delta$	$(1 - p_q)/2$	p_q

where δ adjusts the terms to give smaller probability to larger changes.

The cost function and its formulation

A cost function associates a positive real value with every (state, decision) pair. It is the relative cost values in different states that are important, rather than the absolute cost values. Cost values can be in any units such as money, CPU cycles, delay, or memory utilization. In most real-time systems, various different kinds of costs (e.g. CPU, delay, memory, etc…) are incurred and a final combined cost function is desired. Finding the right formula for this function for a particular system can sometimes be challenging.

To evaluate a cost function we go through three steps.

1. Identify a set of cost factors, such as CPU cycles or delay.
2. For the mth factor, create a partial cost function $Cost_m(i,a)$ which evaluates the impact of that factor under decision a in state i.
3. Normalize $Cost_m$ to fall between 0 and 1, with the value of 1 being the highest cost.
4. Assign a weight value w_m for each cost function, to adjust for its importance in the overall cost to be minimized.

The overall cost function is the sum of the individual cost functions weighted by the weighting factors w:

$$Cost(a, i) = \sum_{m} w_m Cost_m(a, i)$$

where m indexes the partial cost functions.

In this example, there are three partial cost functions, one represents the CPU utilization, a second represents the delay cost, and a third representing the quality of service. Costs are assigned to levels in increments of 1. For example, the first compression level is assigned a CPU cost of 1, the second level is assigned a cost of 2, etc…. Higher frame rate created proportionately higher CPU cost. The un-normalized cost functions are given as follows:

$Cost_{cpu}$ = compression cost + frame rate cost.

$Cost_{delay}$ = (uncompressed frame size × compression level)/(network speed)

$Cost_{QoS}$ = $QoS_{typical} / QoS_{actual}$

The $QoS_{typical}$ is the value in a network under normal load conditions, as understood by the system designer. It depends on the network speed as well as the compression and the frame rate levels.

The weights, w_m, given to the cost functions were; 1 for the *CPU* cost, 2 for the *QoS* cost, and 3 for the delay cost. These values were chosen by judgement, and they do not have to be integers.

5 Results

The MDP calculations for this model were done using Matlab, with the values $p_n = 0.7$, $p_q = 0.5$, $\delta = 0.1$, $U = 4$.

The results shown in this section demonstrate how the DMM of the system reacts *optimally*, and in a collaborative fashion, to external environment changes (network choice and traffic contention) by making appropriate changes to the control variables (compression and frame rate).

There are two sets of results. The first set shows the changes in the control variables, as the network speed varies, for a constant available quality of service (i.e. invariant user traffic, traffic contention, and/or number of users). The second set shows the results as the available quality of service value changes. The decision values are

referred to as vectors (C, F_r), with integer values for C and Fr ($C = 1$ for low compression, $F_r = 1$ for low frame rate).

The minimum long-term average cost is 0.7585, compared to a worst case cost of 6. An ad hoc decision rule produced a cost that is roughly twice as large.

5.1 Effect of network speed on control variables

The optimal choice of the compression C and frame rate Fr will be displayed as one disturbance variable changes and the other is held constant.

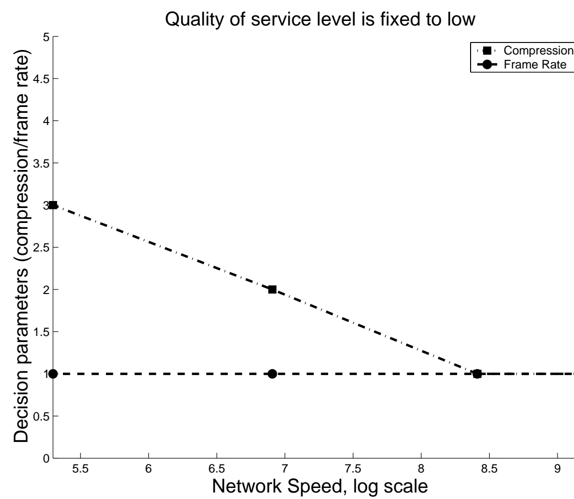

Figure 5 The variation of compression level and frame rate as a response to changing network speed with constant available quality of service (minimum value).

Figure 5 above shows how an optimal policy produces appropriate decisions at different network speeds in a collaborative way. It is also to be noted that the quality of service for the result shown in Figure 5 is fixed to its minimum value. At low network speeds, the optimal policy is to use a lower frame rate and a higher compression. Since the user contention is high (low available quality of service), the optimal policy minimizes the user traffic by using lower frame rates and higher compression values, especially for slower networks. When the network speed increases, the optimal policy responds by relaxing the compression level while keeping the frame rate at its minimum value. Relaxing compression is preferred over increasing the frame rate for a lower quality of service since the next step of the frame rate is double the current, while the next level of compression is more

than half of the previous compression thus producing less traffic.

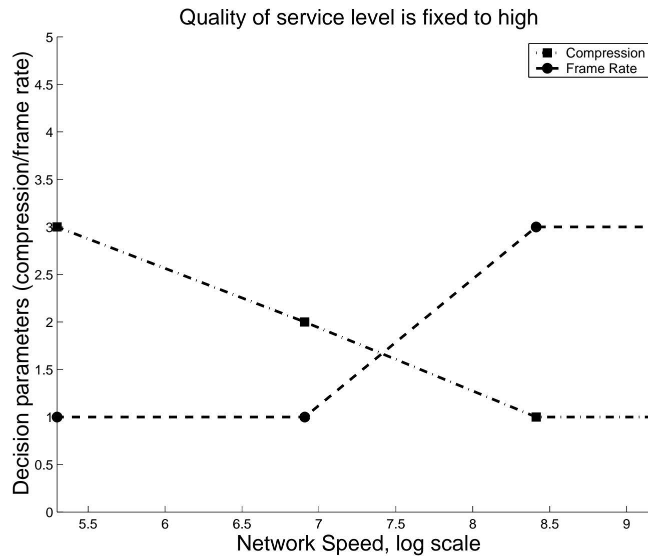

Figure 6 The variation of compression level and frame rate as a response to changing network speed with constant available quality of service (maximum value).

Figure 6 below, shows similar results but for the highest value of available quality of service (i.e. lower user traffic and user contention). For slow a network, the frame rate is set to a minimum and the compression to almost maximum compression (decision is (3,1)). Maximum compression consumes an extensive amount of CPU cycles and is therefore avoided. At high network speed, however, the increase in the available bandwidth is exploited by increasing the frame rate and by reducing the compression to a minimum (decision is (1, 3)).

5.2 Effect of different available quality of service values.

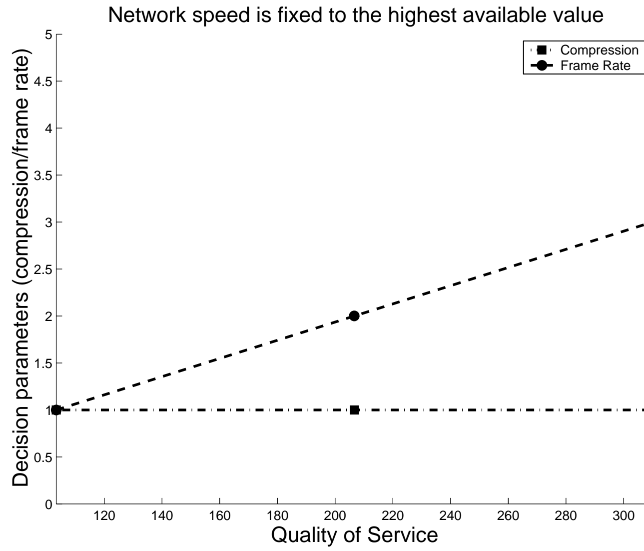

Figure 7 The variation of compression level and frame rate as a response to changing the available quality of service with constant network speed (maximum).

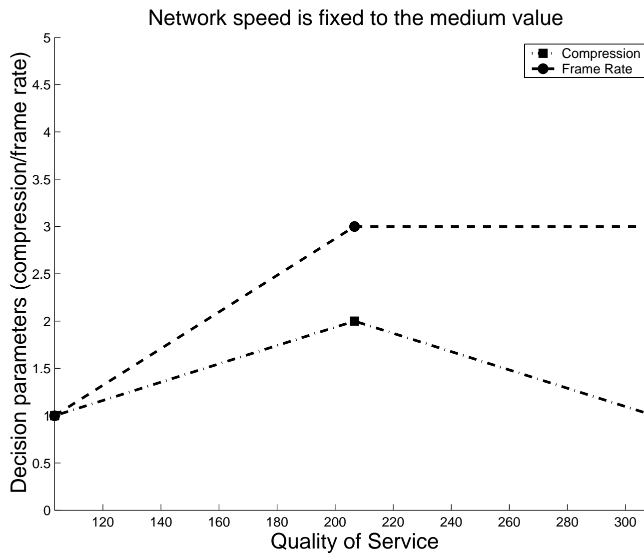

Figure 8 The variation of compression level and frame rate as a response to changing the available quality of service with constant network speed (medium speed).

Figure 7 and Figure 8 show the changes of the control variables with quality of service, for fixed network

speeds. For the high network speed (Figure 7), any additional available quality of service can always be exploited. This is shown by the increase in the frame rate while keeping the compression at its minimum. Decision (1, 3) is an example. For medium speed networks (Figure 8), however, the increase in the available quality of service (resulting from less user traffic/contention) is not always useful. The middle point in that figure (decision (2, 3)) shows that although the frame rate is increased due to the increase in the available quality of service, the compression is increased to try to ease the traffic on a relatively slow network.

5.3 Comparison with an ad hoc adaptation technique

The effectiveness of the MDP lies in that it takes into account the behavior of the system over a period of time. The optimization does not only consider the current situation of the system, but also the future overseen behavior. Ad hoc adaptation techniques lack that kind of ability. To give an example, we will assume a scenario in which an ad hoc adaptation rule is used. We consider the situation depicted by Figure 5. As an ad hoc rule suppose we just use the following:

IF (network speed is *high*) THEN (*increase* frame rate)

As an example, according this rule, the point in Figure 5 with high network speed should have a high frame rate which will be taken as level 2 (the level suggested by the MDP is level 1). When the rule above is applied, the steady state cost function is 1.4131. This is almost double the optimal cost.

6 Conclusions

This work has shown how system adaptation can be guided by decision rules found by the methods of Markov Decision Processes. This is still true when there are multiple controlled features, and multiple system measures used to drive the decisions. A model of a mobile multimedia system had 192 states, and optimal rules for it were found quickly by conventional calculation tools. The rules give reasonable-appearing coordinated changes in the frame rate and compression ratio, when a user changes network or when the local contention effects change. Using a cost function based on simple value relationships, the optimal operation was about twice as good as a reasonable ad hoc rule, in one case examined.

The core of the adaptation is a simplified discrete state model of the entire system, reflecting the designer's view of the granularity of decisions and measures. If the designer imposes a large granularity with just a few levels of each separate variable, the state space will be smaller and the calculations will be less complex. A problem can be attacked first this way, and then experiments can be made with finer granularity. The model must also capture the most essential causal relationships in the system.

Even when the Markov model is a drastic simplification of a complex system, this approach can be applied in practice, by using good judgement in the choice of model. It does not have to be a perfect predictive model, to select useful changes in a feedback mode. When the adaptation is used, the decisions are driven by actual system measurements and the model only evaluates the future effects of the decision, in a general way. Experience with feedback control in other industries shows clearly that an approximate model can contribute to good control.

This work has been restricted to centralized architectures, and to small models. If a larger model is needed, for instance to represent many decision variables, then some partitioning of the decision space would seem to be essential. This work has also not considered the practical issues of sensor and actuator implementation, sensor errors and statistical smoothing of measures, the choice of a time step, or the use of sporadic measures (rather than periodically collected data). These are topics for future works.

7 References

[1] Hennik C. Tijms, "Stochastic Modeling and Analysis: A computational approach", John Wiley & Sons 1986.

[2] F. Kon, M. Roman, P. Liu, J. Mao, T. Yamane, L. C. Magalhes, R. H. Campbell, "Monitoring, Security, and Dynamic Configuration with the dynamicTAO Reflective ORB", Proceedings of the IFIP International Conference on Distributed Systems Platform and Open Distributed Processing (Middleware2000).

[3] Bo Norregaard, Eddy Truyen, Frank Matthijs, and Wouter Joosen, "Customization of Object Request Broker by Application Specific Policies", Proceedings of the IFIP International Conference on Distributed Systems Platform and Open Distributed Processing (Middleware 2000).

[4] Hector Duran and Gordon S. Blair, "Configuring and Reconfiguring Resources in Middleware" IEEE Proc.

ISORC 2000.

[5] Richards, A. " DARTS - A Dynamically Adaptable Transport Service Suitable for High Speed Networks", 2nd International Symposium on High Performance Distributed Computing (Washington) 1993.

[6] Adrian Friday, Nigel Davies, Gordon Blair and Keith Cheverst "Developing Adaptive Applications: The MOST Experience " Journal of Integrated Computer-Aided Engineering, Volume 6, Number 2, 1999, pp143-157.

[7] Nigel Davies, Adrian Friday, Stephen Wade and Gordon Blair "L2imbo: A Distributed Systems Platform for Mobile Computing" ACM Mobile Networks and Applications (MONET), Special Issue on Protocols and Software Paradigms of Mobile Networks, Volume 3, Number 2, August 1998, pp143-156.

[8] Keith Cheverst, Nigel Davies, Keith Mitchell, Adrian Friday and Christos Efstratiou "Developing a Context-aware Electronic Tourist Guide: Some Issues and Experiences" Proceedings of CHI 2000, Netherlands, April 2000, pp 17-24.

[9] D.C. Feldmeier, A. J. Macauley and J. M. Smith, "Protocol Boosters," Technical report, U. Penn CIS Dept., 1996.

[10] A. Mallet, J.D. Chung and J.M. Smith, "Operating System Support for Protocol Boosters", Proceedings of the 3rd international Workshop on High Performance Protocol Architectures (HOPPARCH'97), June 1997.

[11] Bo Norregaard Jorgensen, Eddy Truyen, F. Matthijs, and Wouter Joosen, "Customization of Object Request Broker by Application Specific Policies", Middleware 2000.

[12] Jim Dowling, Tilman Schafer, Vinny Cahill, Peter Haraszti, Barry Redmond, "Using Reflection to Support Dynamic Adaptation of System Software: A case Study Driven Evaluation", OOPSLA Workshop on Object-Oriented Reflection and Software Engineering, Denver, Colorado, Nov. '99.

[13] Baochun Li, Won Jeon, William Kalter, Klara Nahrstedt, Jun-Hyuk Seo, "Adaptive Middlware Architecture for a Distributed Omni-Directional Visual Tracking System", Proceedings of SPIE Multimedia Computing and Networking 2000 (MMCN 2000), pp. 101- 112, January 25-27, 2000.

[14] Pattie Maes, "Concepts and experiments in computational reflection" OOPSLA'87, Sigplan Notices, Vol. 22 No 12. December 1987.

[15] Ian Welch and Robert Stroud, "Dalang: A reflective Extention for Java" Tech report, university of Newcastle-upon-Tyne, UK, Sept 1999.

[16] Ramana Rao, "Implementational reflection in Silica" In Proceedings of ECOOP'91, number 512 in Lecture Notes in Computer Science, pages 251--267. Springer-Verlag, July 1991.

[17] Pierre-Guillaume Raverdy, Robert Le Van Gong, Rodger Lea, "DART: A Reflective Middleware for Adaptive Applications", Proceedings of the ACM Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA), 1998.

[18] Raj Jain, "The Art of Computer Systems Performance Analysis", John Wiley & Sons, Inc. 1991.

[19] Jun-ichiro Itoh, Yasuhiko Yokoto, Mario Tokoro, "SCONE: Using Concurrent Objects for Low-level Operating System Programming". ACM OOPSLA'95

[20] K-Czarnecki and U.W. Eisenecker, "Generative Programming", Addison-Wesley, 2001.

[21] Paul Stelling. Ian Foster, Carl Kesselman, Criag Lee, Gregor von Laszewski, "A Fault Detection Service for Wide Area Distributed Computatiopns". In Proceedings of the 7th IEEE symposium on High Level Distributed Computaion. P 268-278, 1998.

[22] A Acquaviva, L. Benini, and B. Ricc. "An adaptive algorithm for low-power streaming multimedia processing". In Proceeding of the Conference on Design Automation and Test in Europe DATE'2001, 2001.

[23] J. W. S. Liu, W. –K Shik, K. –J. Lin, R. Bettati, J. –Y. Chung, "Imprecise Computations", Proceedings of the IEEE, Vol. 82, No. 1, Jan 1994, pp. 83-94.

[24] Hidehiko Masuhara, Satoshi Matsuoka, Akinori Yonezawa, "Implementing Parallel Language Constructs Using a Reflective Object-Oriented Language", In Reflection'96 conference, San Francisco, California, Apr., 1996.