Seeking Optimal Policies for Adaptive Distributed Computer Systems with Multiple Controls

Mohammad Abdeen and Murray Woodside
Department of Systems and Computer Engineering,
Carleton University,
1125 Colonel By Drive,
Ottawa, ON, Canada, K1S 5B6
{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. Single adaptations in applications and middleware are common, but the opportunities are greater if many features of the system, at all layers, are adaptive. This paper describes an architecture to support coordinated adaptive changes in all layers (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.*

Keywords: Configurable computing, Mobile applications, Multi-media applications, Modeling and simulation, 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 high bandwidth radio LAN to another wireless 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. Adaptation to slower disturbances is a kind of self-tuning to track changes in the environment.

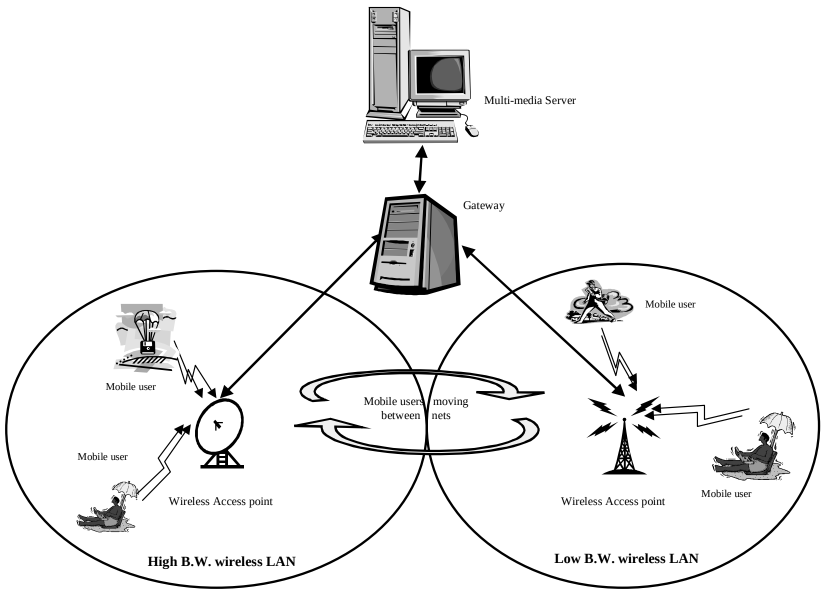

Figure 1 Mobile users roaming among multiple networks.

Adaptive features have been described, under names such as “reflection” [7][9], “metaprogramming”[11], and “reliability architectures” [12]. Reflection is a term applied to code or to a sub-system which is self-aware, in the sense that it has access to meta-information about its operational state. Metaprogramming deals with this information. Reliability architectures use operational information to adapt to failures.

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 shows 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[14], driven by computing resource availability

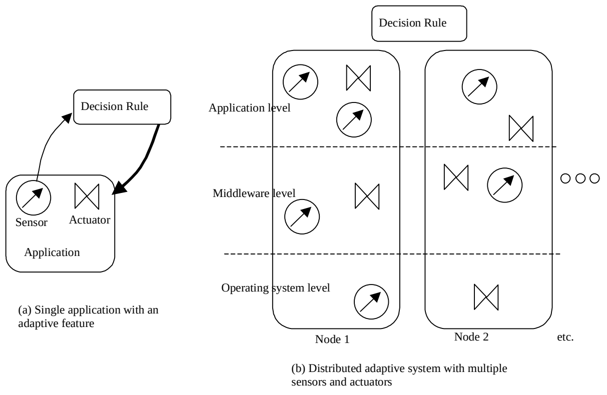

Figure 2 Adaptive systems, showing an isolated adaptive feature and a complex distributed system with multiple sensors and actuators

- 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])

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.

Adaptive features may be deployed at different system layers. The following list shows some of the features to adapt to lower bandwidth while transmitting a stream of images:

- at the application layer, increase the image compression, decrease image size, and change from color to black-and-white.
- at the middleware layer, change the source of the images (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 layer, 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 [13]. The performance of the protocol stack can be enhanced by the introduction of protocol boosters [5][6]. This is especially effective for wireless links with high bit error rates (BER of 10^{-5} or worse). In this case a forward error correction booster can produce a throughput increase of as high as 60 times [20].

A system with many sensors and actuators has a structure that will be called its *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 central 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 decision module can take a variety of forms. In [16] and [17], a feedback regulator approach is used, which forces a system to maintain or track a desired value of some system variables over time. In computer systems however, quality of service is usually specified as a bound on some measures, rather than a target value, and the acceptable range of delay often extends down to zero. It is then more natural to adapt by seeking either a feasible range or an optimal value (such as minimum delay). This viewpoint (adopted here) gives an adaptive optimization problem rather than a control problem.

An optimization approach used in some studies of decision making for adaptive and reconfigurable computer systems is Markov Decision Processes (MDP) [18][19]. Markov Decision Processes are well suited for discrete state space and for stochastic behavior. The optimization is simple and provides self-tuning. The studies in [18][19] considered only relatively small and one-dimensional state spaces.

The present paper adopts an MDP model, building it into a general and flexible architecture that considers multiple system layers, aspects and measures. State explosion remains a limitation, but useful systems can be built (for instance an example based on Figure 1 with two observation variables and three control variables described below gives 384 states). It is interesting that the optimal policy has a structure similar to various ad-hoc policies.

2. An architecture for coordinated adaptation

The architecture considered here has a central decision module connected to sensors and actuators distributed across components and layers, as shown in Figure 2. This section considers the operations carried out by the components indicated by the architecture in greater depth.

The adaptation process repeats a cycle of estimating (monitoring and tracking), deciding and acting. *Observation variables* capture the relevant aspects of the status of the system, and are provided by sensors.

Sensor in this work also includes other awareness mechanisms such as monitoring by dedicated components in the middleware [3][4][8], in the operating system [10] and in the application managers. These techniques are necessary to capture system-level 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.

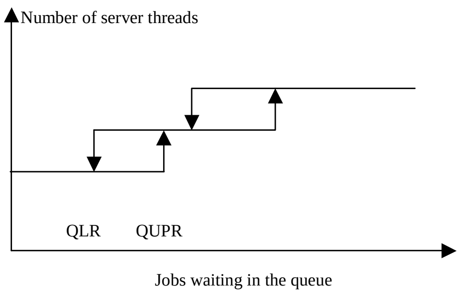

Figure 3 A threshold rule for adapting the number of server threads for a queue

The *decision mechanism* derives new settings for the actuators from an analysis of the measures (feedback control). Knowledge of all 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 an example of a decision rule with hysteresis, as shown in Figure 3.

In this work there may be similar decision rules to determine a discrete level for a system parameter, but in general the decision rule depends on many variables and not 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 layer 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.

The location of these components and its interaction in a three-layer centralized architecture is depicted in Figure 4.

The figure shows a distributed application with clients on the left and a server on the right, each with an operating system layer, a middleware layer, and an application layer 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 layer to take the required action(s). Software actuators in each layer receive adaptation decisions made by the DMM and implement them.

In constructing the coordinated adaptive 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 shown 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 obtain them from Markov Decision Processes.

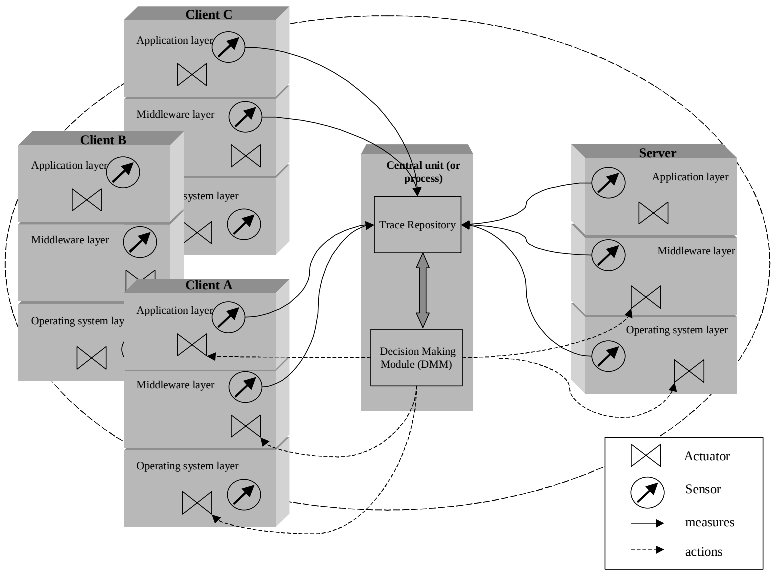

Figure 4 The centralized multi-layer adaptive architecture

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 S_i, for $i = 1$ to I. At steady state, state S_i has probability p_i. At each step, a transition from a state S_i to another state S_j has a transition probability p_{ij}.

The system evolves randomly, but can also 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 S_i, 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.

3.1. 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 S_i 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.

3.2. Relative values

For a given policy R, the total expected cost over the first n decision epochs when starting with state S_i, is denoted by $V_n(i,R)$. Starting with a different state other than S_i, e.g. S_j, has the effect of changing the total expected cost function. This change in the total cost function is called the relative value $v_{ij}(R)$. An arbitrary state, r, can be chosen to normalize relative values $v_{ij}(R)$’s. A relative value can then be denoted as $v_{ir}(R)$, or simply $v_i(R)$. 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:

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

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 S_i to state S_j if the decision R_i is made.

3.3. 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

$$c_i(R'_i) - g(R) + \sum_{j \in I} P_{ij}(R_i)\, v_j(R) \le v_i(R) \text{ for each } i \in I \quad (2)$$

then

$$g(R') \le g(R) \quad (3)$$

Equations (2) and (3) suggest that an optimal policy R^* can be obtained be recursively iterating the calculation in Eq (1) with new values of R' until a minimum cost $g(R^*)$ is reached. The value of R^* is called the optimal policy, and gives a value R^*_i for each state S_i.

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 [1].

4. Demonstration and Evaluation: 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, high bit error rates, and diminished quality of service, it could adapt in three ways 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.
- An error correction protocol booster can be installed and activated. This will increase the protocol overhead per packet and packet processing time but will produce an up to 60 times throughput improvement at bit error rates 10^{-5} or worse.

Mobile users move among four wireless networks, Alpha, Beta, Gamma, and Theta with speeds of 0.5 Mbits/sec, 2 Mbits/sec, 5 Mbits/sec, and 10 Mbits/sec and bit error rates of 10^{-3}, 10^{-4}, 10^{-5}, and 10^{-6} 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 (16 times compression), 3 (33 times) and level 4 (40 times).
- four frame rates, level 1 (2 frames/sec), 2 (10 frames/sec), 3 (20 frames/sec) and 4 (unfiltered rate of 30 frames/sec).
- two protocol booster levels, level 1 (protocol booster activated) and level 2 (protocol booster deactivated).

The adaptation policy will choose a set of control levels for each possible state of the system as described in the following section.

4.1. The system model

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

- the current level of compression (C, with four values 1 to 4)
- the current frame rate level (Fr, with four values 1 to 4),
- the current protocol booster level (B, with two values 1 and 2),
- the network being accessed by the user (N, with four values 1 to 4),
- an available network quality of service level (Q, with three levels 1 to 3, and high Q represent better service).

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 wireless channel fading and jitter effects as well as contention from others. This could be found from network collision data or utilization data.

The network *QoS* factor Q is defined relative to the range of capabilities of each network, with $Q = 2$ for a middle range of values of available *QoS*. A change in the decision variables can affect Q. For example, higher frame rates or lower compression levels may increase collision rates due to the higher traffic injected into the network, reducing Q. We define $Q = 1$ for overloaded conditions, Q = 2 for average conditions, and Q = 3 for lightly loaded and error-free conditions.

The first, 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 five values:

$$S = (C, Fr, B, N, Q).$$

and the state space has 384 states (4 x 4 x 2 x 4 x 3).

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

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

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

4.2. 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 earlier. 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 transition probability from $S_i = (C_i, Fr_i, B_i, N_i, Q_i)$ to $S_j = (C_j, Fr_j, B_j, N_j, Q_j)$. The transition probabilities express the influence of the current state on the evolution of the system. It is assumed that the choice of a new network, N_j, is made by the user. It is driven by factors outside this model, and so it is governed by its own probabilities and is independent of the policy choices. For this model we assumed that the probability of staying in a given network has a value of p_{net} (the same for all networks) and (1 - p_{net})/3 for each of the other networks.

Define:

$$P_{net}(i, j) = p_{net} \quad \text{if } N_i = N_j,$$
$$= (1 - p_{net})/3 \quad \text{otherwise.}$$

For each policy a there is a transition probability matrix $P_{ij}(a)$,

- which determines the change in the control variables to their new values,
- includes $P_{net}(i, j)$
- and also describes the evolution of Q, which is partly governed by changes in network traffic (which are largely determined outside this model) and partly influenced by a.

Here, the transition probability from a given *QoS* state Q_i to a new value Q_j has a baseline value $q(Q_{i,}, Q_j)$, plus an interaction term for the influence of the control variables C, Fr, B, due to the way they increase or decrease the total traffic. The interaction term is a function $\delta(Q_i, Q_j; a)(C_i, Fr_i, B_i)$,

The function, δ, depends on the current control variables (C_i, Fr_i, B_i) because they may influence the future changes in traffic levels due to other users, and on the decision a because it directly affects the traffic, as discussed above.

The baseline transition probabilities used for Q in this example are illustrated in Figure 5 below. When the network changes, the model assumes that Q for the new network is initially 2.

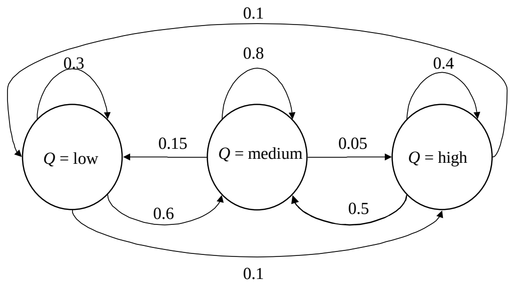

Figure 5 The Baseline QoS State Transition probabilities $q(Q_i,Q_j)$

Overall, the transition probability from state S_i to state S_j, where S_j has C_j, Fr_j, B_j corresponding to decision a, is given by:

$$P_{ij}(a) =$$
$$P_{net}(i, j) \; q(Q_{i,}, Q_j) \; \delta(Q_i, Q_j; a) \; (C_i, Fr_i, B_i) \text{ if } N_j = N_i \quad (4)$$
$$P_{ij}(a) = P_{net}(i, j) \qquad \text{if } N_j \neq N_i \text{ and } Q_j = 2$$
$$P_{ij}(a) = 0 \qquad \text{otherwise}$$

4.3. The cost function and its formulation

The cost function associates a positive 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. The optimal policy minimizes the cost function.

To construct a cost function we go through the following steps.

1. Identify a set of cost factors, such as CPU cycles or delay.
2. For the m^{th} factor, create a partial cost function $Cost_m(i,a)$ which evaluates the impact of that factor under decision a in state i.
3. Assign a weight value wm for each partial cost function, to adjust for its importance in the overall cost to be minimized.

The overall cost function is the sum of the partial cost functions weighted by the weighting factors w_m and is given as:

$$Cost(a, i) = \sum_m w_m \; Cost_m(a, i) \qquad (5)$$

where m indexes the partial cost functions.

Let us consider three partial cost functions, one represents the CPU utilization, a second represents the delay, and a third represents the quality of service. Costs are assigned to levels depending on the control variable under consideration as well as the value of the level relative to others, and are defined so the maximum value is 1.0. For example, compression levels are assigned a CPU cost that is proportional to the degree of compression performed. Higher frame rate creates proportionately higher CPU cost. The partial cost functions are given as follows:

$Cost_{cpu}$ = $Cost_{cpu}$(compression) + $Cost_{cpu}$(frame rate reduction) + $Cost_{cpu}$(protocol booster)

$Cost_{delay}$ = $Cost_{delay}$(video stream transmission) + $Cost_{delay}$(protocol booster)

$Cost_{QoS}$ = $Cost_{QoS}$(video stream quality) + $Cost_{QoS}$(wireless network jitter and other effects)

where,

$Cost_{delay}$(video stream transmission) is the transmission delay of the video stream in progress.

The approach adopted here is based on a 30 frames per second video stream (2 I-frames, 8 P-frames and 20 B-frames [21]). The $Cost_{delay}$(video stream transmission) can be given as:

$$Cost_{delay}(\text{video stream transmission}) = \frac{\text{size of uncompressed video stream } \textit{after} \text{ filtering}}{\text{compression level x network speed}}$$

The $Cost_{delay}$(protocol booster) represents the delay introduced by the protocol overhead after activating the protocol booster. This overhead can be as high as 3 times the size of the original video stream [20].

The $Cost_{QoS}$(video stream quality) function assigns a maximum cost for the worst video stream quality, i.e. the one with maximum compression and minimum frame rate (a normalized cost of 1 in our example). On the other hand, the best video quality is assigned a minimal cost (a cost of 1/16 in our example). The effects of the wireless environment as well as user contention are accounted for by assigning cost values to the $Cost_{QoS}$(network) that are inversely proportional to the available quality of service level. For example, a cost of 3 is assigned to the lowest *QoS* level.

All CPU costs are in CPU cycles based on a 300 MHz processor.

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

5. Results

The results shown in this section demonstrate how the decision-making module of the system reacts *optimally*, and in a collaborative fashion, to external environment changes (network choice and quality of service) by making appropriate changes to the control variables (compression, frame rate, and protocol booster).

The optimal policy calculations for this model were done using Matlab, with p_{net} = 0.7 to represent a relatively high user mobility (30% chance of changing networks). The results show that the minimum long-term average cost is 0.7585 (optimal policy cost), compared to a worst-case cost of 6 (maximum cost of 1 for all partial cost functions weighted by w_m = 1, 2, and 3). An ad hoc decision rule produced a cost that is roughly twice as large. This is shown in a later section.

The optimal choice of the compression level, *C*, frame rate level, *Fr*, and protocol booster level, *B*, are displayed as a function of the disturbances; the network speed, *N*, and the quality of service, *Q*.

Each set of results shows the optimal policy for one control variable as a function of *N* and *Q*, and is displayed as a set of 3x4 squares. The rows represent the quality of service and the columns represent the network speed.

The value of the compression decision, *C*, is represented by the shading level of the corresponding square. A darker square represents a higher image resolution and hence a lower compression. The decision numeric value (i.e. *C* = 1 - 4) is also written in the top-right corner of the same square. In a similar way, the value of the frame rate decision, *Fr*, is represented as the number of parallelograms inside the square. For example, *Fr* = 1 is the lowest frame rate (2 parallelograms). The frame rate numeric decision value (i.e. *Fr* = 1 - 4) is also written on the top-right corner of the square. The protocol booster decision is explicitly written as ON/OFF for activating and deactivating the protocol booster respectively.

Figure 6 shows the optimal decision policy for compression, *C*, at different network speeds and quality of service values. At low network speeds and low quality of service the optimal policy produces a maximum compression (*C* = 4). At high network speeds and high quality of service the optimal policy produces a minimal compression (*C* = 1). In the interim the optimal policy for compression is *C* = 2.

It is to be noted that a higher compression could be used for values of *N* and *Q* at the middle range. However, this will further degrade the video quality and consume more CPU cycles. The optimal policy optimizes both the CPU usage, especially for the CPU limited mobile devices, and the user quality of service.

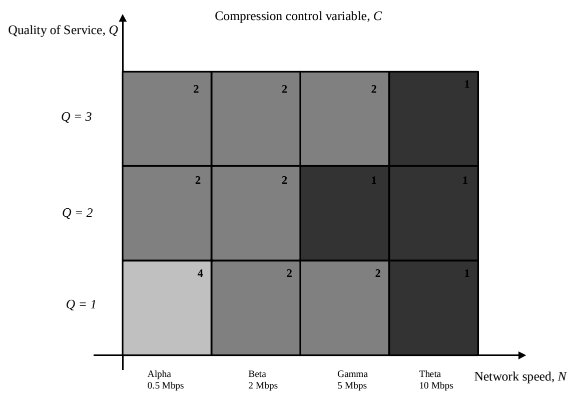

Figure 6 The optimal policy showing the variation of frame compression with N and Q

Figure 7 below shows the optimal decision policy for frame rate, *Fr*, at different network speeds and quality of service values. At low network speeds and low quality of service the optimal policy produces a low frame rate (*Fr* = 1). At high network speeds and high quality of service the optimal policy produces a high frame rate (*Fr* = 4). In the interim the optimal policy is *Fr* = 3 or 4.

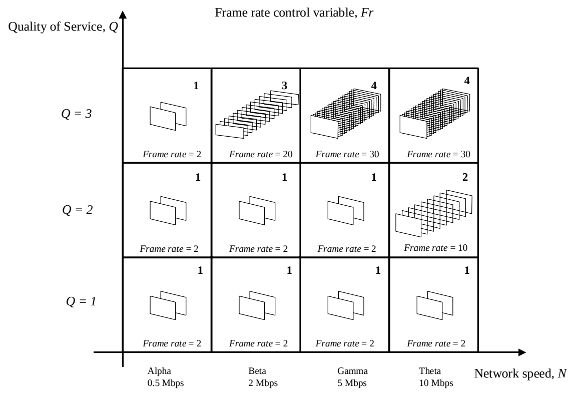

Figure 7 The optimal policy showing the variation of frame compression with N and Q

Two sets of values on Figure 7 are worthy of noting. The first is when (*N*, *Q*) = (Theta, 2) and the second is (*N*, *Q*) = (Beta, 3). The first value shows that although network speed is high, the frame rate is not that high (10 frames per second). This is due to the fact that the quality of service is intermediate and having enough bandwidth to send more frames might be hampered by the insufficient quality of service. On the other hand, the second set of values suggest that having a good quality of service is more "encouraging" to send more frames.

Figure 8 below shows the optimal decision policy for protocol booster, *B*, at different network speeds and quality of service values. The results suggest that activating protocol booster is largely dependent on the quality of the service. It is activated for low *Q* and deactivated for high *Q*.

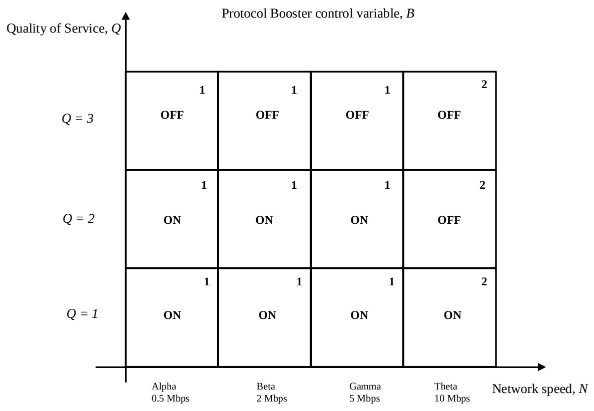

Figure 8 The optimal policy showing the variation of the protocol booster value with N and Q

5.1. Comparison with an ad hoc adaptation technique

The power of the MDP comes from incorporating a model for the behavior of the system over time. The optimization does not only consider the current situation of the system, but also the future foreseen behavior. Because of the feedback, the model does not even have to be particularly accurate. The MDP policy also incorporates all the observation variables.

Ad hoc adaptation techniques are much more limited. The rules they use are arbitrary and typically connect one observation variable to one decision variable. To give an example, we will assume a scenario in which an ad hoc adaptation rule is used. As an ad hoc rule for frame rate suppose we just use the following:

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

together with the optimal policies for C and B.

As an example, consider a situation with Q = 2, Fr = 1 and a change of networks from Beta to Gamma. This ad hoc rule would give a frame rate after the change of 2. The optimal policy would give a value which is still 1. When the ad hoc rule is applied, the steady state cost function is 1.4131. This is almost double the optimal cost (0.7585).

5.2. Discussion

The results presented here are obtained for a special case where the state transition probability matrix $P_{ij}(a)$ is partitioned into two non-interacting sub-matrices, one for the observation variables (N, Q) and one for the control variables (C, Fr, B) (if the interaction terms in Equation (4) were not zero, this would not be the case). As a result of this, the optimal policy is only a function of (N, Q), and it could be derived from a simpler model in which the state space only includes (N, Q). This explains why the optimal policies in Figure 6, 7, and 8 are function only of (N, Q).

In the more general case the transition probability matrix is not partitioned, and the full state definition is needed for the analysis. This would arise in our example if:

- the interaction terms in Equation (4) were non-zero,
- a cost is imposed for making changes in control variables, penalizing the effort of reconfiguring the actuator to a new value.

Then the optimal policy is a function of the state of the control. In that case it is multi-valued in the observation variables alone, and this for instance could generate an optimal policy with hysteresis, as illustrated in Figure 3. A large cost for control changes is likely to lead to hysteresis policies.

6. Conclusions

This work has shown how system adaptation can be guided by decision rules found by the methods of Markov Decision Processes, when there are multiple controlled features, and multiple system measures used to drive the decisions. An example model of a mobile multimedia system had 384 states, and optimal rules for it were found by conventional calculation tools. The rules give reasonable-appearing coordinated changes in the compression ratio, frame rate, and the protocol booster activation, 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. If the model is not very accurate, feedback from the observations counteracts any errors this might cause.

Even when the Markov model is a drastic simplification of a complex system, this approach can be applied in practice, by using good judgment 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 observations (rather than periodically collected data). These are topics for future work. The evaluation of this approach on simulations, as well as on a real system, is currently underway.

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] Hector Duran and Gordon S. Blair, "Configuring and Reconfiguring Resources in Middleware" IEEE Proc. ISORC 2000.

[4] 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.

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

[6] 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.

[7] 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.

[8] 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.

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

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

[11] K-Czarnecki and U.W. Eisenecker, “Generative Programming”, Addison-Wesley, 2001.

[12] 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.

[13] 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.

[14] 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.

[15] 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.

[16] Baochun Li and Klara Nahrstedt, “A control-Based Middleware Framework for Quality-of-Service Adaptations” IEEE Journal on Selected Areas in Communications, Vol. 17, No. 9, Sept. 1999.

[17] Tarek Abdelzaher, Kang G. Shin, Nina Bhatti, “Performance Guarantees for Web Server End-Systems: A Control-Theoretical Approach” *Transactions on Parallel and Distributed Systems* , Vol. 13, No. 1, Jan 2002.

[18] Kang G. Shin, C. M. Krishna, and Yann-Hang Lee, “Optimal Dynamic Control of Resources in a Distributed System” IEEE Transactions on Software Engineering, Vol. 15, No. 10, October 1989.

[19] Leonard. J.N. Franken and B.R. Haverkort, “Reconfiguring Distributed Systems using Markov-Decision Models” Trends in Distributed Systems (TreDS’96), Oct. 1996, Aachen, Germany, PP. 219-228.

[20] D. Bakin, M. Joa-Ng, and A. McAuley, “Quantifying TCP performance improvements in noisy environments using protocol boosters,” Proceedings: Fifth IEEE Symposium on Computer and Communications (ISCC 2000), July 2000.

[21] Armando Fox, Steven D. Gribble, Eric A. Brewer, and Elan Amir, “Adapting to Network and Client Variability via On-Demand Dynamic Distillation,” In Proceedings of the Seventh International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-VII), Cambridge, MA, October 1999.