POLICY SPECIFICATION AND ARCHITECTURE FOR QUALITY OF SERVICE MANAGEMENT

Nathan Muruganantha and Hanan Lutfiyya *
Department of Computer Science
The University of Western Ontario
hanan@csd.uwo.ca

Abstract: An application's Quality of Service (QoS) requirements refers to non-functional, run-time requirements. These requirements are usually soft in that the application is functionally correct even if the QoS requirement is not satisfied at run-time. QoS requirements are dynamic in that for a specific application, they change. The ability to satisfy an application's QoS requirement depends on the available resources. Since an application may have different QoS requirements in different sessions, the resources needed are different. A differentiated service must be supported. Since an application's QoS requirement is soft, it may not always be satisfied. It must be possible to dynamically allocate more resources. In an overloaded situation, it may be necessary to allocate resources to an application at the expense of other applications. Policies are used to express QoS requirements and actions to be taken when the QoS requirement is not satisfied. Policies are also used to specify actions to be taken in overloaded situations. Policies dynamically change. Supporting these policies is done through a set of distributed managed processes. It must be possible specify policies and have these policies distributed to managed processes. This paper describes how these policies can be formally specified and a management architecture (based on the IETF framework) that describes how the policies are distributed and used by the management system. We conclude with a discussion of our experiences with the management system developed.

Keywords: QoS Management, Policies, Application Management

1. Introduction

An application's Quality of Service (QoS) refers to non-functional, run-time requirements. A possible QoS requirement for an application receiving a video stream is the following: "The number of video frames per second displayed must be 25 plus or minus 2 frames". A QoS requirement is *soft* for an application, if the application is functionally correct even though the QoS requirement is not satisfied at run-time. Otherwise, the QoS requirement is said to be *hard* (e.g., flight control systems, patient monitoring systems).

The allocation and scheduling of computing resources is referred to as *QoS management*. QoS management techniques (e.g., [25]), such as resource reservation and admission control can be used to guarantee QoS requirements, since resource reserva-

*This work is supported by the National Sciences and Engineering Research Council (NSERC) of Canada, the IBM Centre of Advanced Studies in Toronto, Canada and Canadian Institute of Telecommunications Research (CITR).

tions are based on worst-case scenarios. This is useful for applications that have hard QoS requirements, but often leads to inefficient resource utilisation in environments that primarily have applications with soft QoS requirements.

Application QoS requirements are often dynamic in that they may vary for different users of the same application, for the same user of the application at different times, or even during a single session of the application. We developed a QoS management system that deals with soft and dynamic QoS requirements by providing management services (implemented by a set of management processes and resource managers) that support the following: (1) Detecting that an application's run-time behavior does not satisfy the application's QoS requirements. This is called a *symptom* and is considered a manifestation of a fault in the system; (2) Fault location, using a set of symptoms, determines a hypothesis identifying possible faults causing the violation of the QoS requirements; (3) Based on the fault and the system state, adaptation actions are chosen that may take either the form of resource allocation adjustments or application behavior adjustments. These adaptations depend not only on the cause of the violation, but also depend on the constraints (*administrative* requirements) imposed on how to achieve the QoS requirement. This is especially important in the case of overloaded conditions.

It should also be possible for the QoS management system to communicate to an application that it should adapt its behavior (e.g., change video resolution) if it is not possible to reallocate resources. The implication of different QoS requirements suggests that a differentiated allocation of resources should be allowed.

Policies are used to express requirements. A *policy* can be defined [20] as a rule that describes the action(s) to occur when specific conditions occur.

The QoS management system should be able to support the strategy described above as well as deal with changes in QoS requirements.

Our earlier work [11] described a policy-based QoS management framework and the initial results indicated that this approach is worth pursuing. This paper focusses on the definition and specification of different categories of policies and describes how they can be distributed to the appropriate entities in the QoS management system. This paper is organized as follows: Section 2 describes the different types of policies needed and the architecture needed to support the QoS management system to support the QoS management strategy described earlier. Section 3 briefly describes the implementation. Section 4 discusses the lessons learned. Section 5 discusses the related work. Finally a conclusion is presented.

2. Architecture

This section describes different categories of policies and architectural components (depicted in Figure 1) needed to support the strategy to QoS management described in Section 1. The management architecture used in this work is considered an adaptation of the IETF policy framework [14, 19, 22] and is a generalization of our architecture described in [11]. An example QoS requirement that will be used throughout the paper is the following:

EXAMPLE 1 *A video client is to receive video at a frame rate of 25 frames per second, plus or minus 2 frames.*

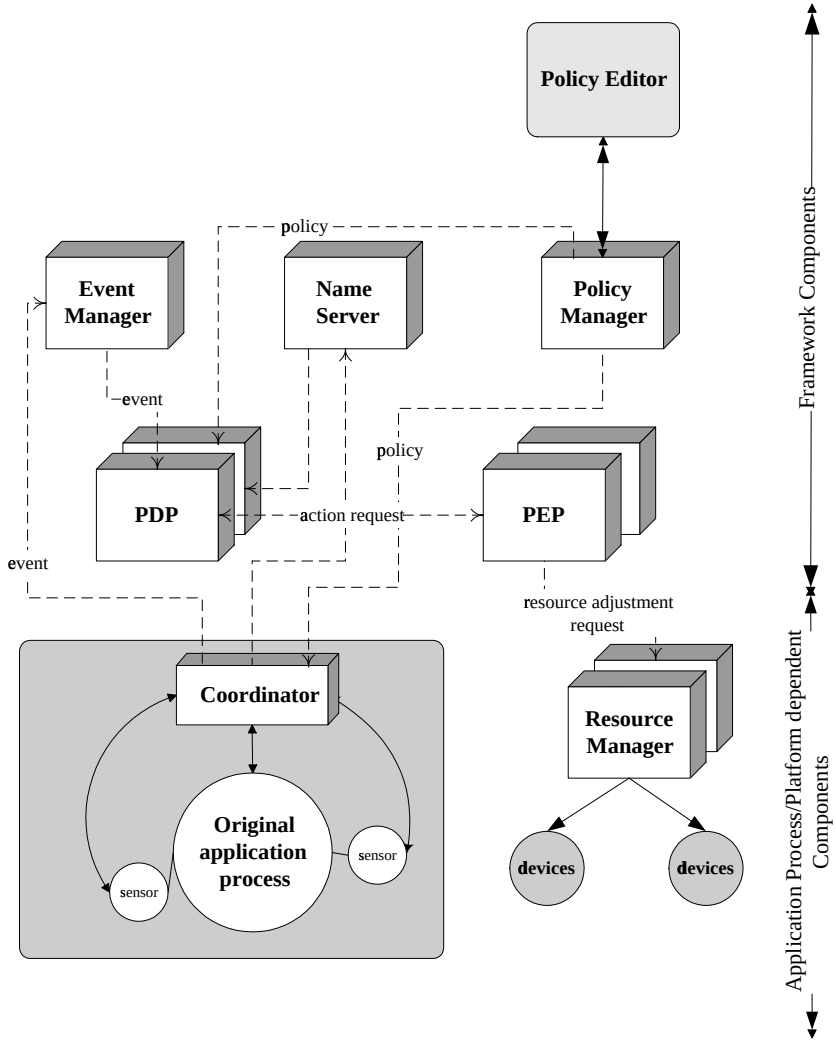

Figure 1. Policy-Based QoS Management Architecture

2.1 Expectation Policies

An *expectation* policy is used for stating how QoS requirements are determined and where to report the violation of a QoS requirement and any other possible actions. Expectation policies are specified using Ponder obligation policy formalism (for more information see [3]). In this formalism, the policy specifies the action that a subject must perform on a set of target objects when an event occurs. An expectation policy type associated with Example 1 is the following:

EXAMPLE 2

```
type oblig QoSreq_spec (target Coordinator, ScriptType FindQoSReqs){
  subject PolicyManager;
  on requestQoSRequirements(ProcessInfo)
  do FindQoSReqs(in ProcessInfo,
                 out EventIdentifier,
                 out AttributeConditionList,
                 out ActionList); -->
     Coordinator.Initialize(EventIdentifer,EventManagerIdentifier,
                            AttributeConditionList, ActionList);
}
```

This can be instantiated as follows:

```
inst VideoClientProcessReq=
    QoSreq_spec(VideoClientProcessCoordinatorSet,DetermineAllowedFrameRate);
```

It is assumed that each process has a management coordinator. The target domain for policy VideoClientProcessReq is the set of management coordinators of processes instantiated from the executable at .../syslab/rockyroad/videoClientExecutable (where ... refers to /ca/uwo/csd/syslab). It is not assumed that all instantiations are executing on rockyroad. DetermineAllowedFrameRate is a script that computes the frame rate per second that the process is allowed to have. Its input is process information (represented by ProcessInfo) that includes the user of the process and the time. The script returns the QoS requirements in AttributeConditionList. This is a list of attribute conditions. Application QoS requirements can be specified as a conjunction of the conditions specified in the attribute conditions. The general form of an attribute condition is the following: (anAttribute, comparisonOperator, aThreshold, EventIdentifier) where anAttribute denotes the attribute being monitored, aThreshold denotes a threshold that the value of anAttribute is being compared to, and comparisonOperator denotes the comparison operator by which the value of anAttribute and aThreshold are to be compared. EventIdentifier is the identifier of the event that is generated when this specific attribute condition is not satisfied (and hence the the QoS requirements are not satisfied). If an application process is to have the QoS requirements stated in Example 1, then the attribute condition list would be the following: ((current_fps, $\leq$, 27,fps_high),(current_fps, $\geq$,23, fps_low)). The attribute identifier, current_fps, represents frames per second. The event, fps_high, is generated when current frames per second is above 27 and fps_low is generated when the current frames per second is below 23.

The "calculation" or determination of QoS requirements can be guided by policies. A simple policy can be informally stated as follows: "if the process belongs to the GroupA users then it gets a target frames per second of 25 plus or minus two frames; if the process belongs to other users besides GroupA users then the target frames per second is 20 plus or minus two frames". This can be stated using Ponder formalism.

The ActionList denotes the action(s) the coordinator is to execute if the conjunction of the attribute conditions in the attribute condition list is found to be true. The general form of an action is the following: action = (targetObject, (actionMethod, actionMethodParameter)) A specific example is the following: (EventManagerHandle, (notify, current_fps, target_fps,...). Basically, this action specifies that an event handler process, that can be referenced using EventManagerHandler, is notified of the values of specific application attributes.

2.2 Monitoring

Example 2 illustrates that application QoS requirements are defined in terms of application-specific attributes. Monitoring of the application is needed to collect values of the attributes (e.g., current_fps). One monitoring mechanism is through instrumentation of the application which is briefly described here.

An attribute is associated with a sensor (see Figure 1). A sensor is a class with variables for representing threshold and target values. Sensors are used to collect, maintain, and process a wide variety of attribute information within the instrumented processes. The sensor's methods (*probes)* are used to initialise sensors with threshold and target values and collect values of attributes. Probes are embedded into process code to facilitate interactions with sensors.

As an example, consider the sample psuedo-code for a video playback application in Figure 2 that has the QoS requirement stated in Example 1 and sensor s_1. This QoS requirement suggests an upper threshold of 27 frames per second and a lower threshold of 23 frames per second. Sensor s_1 includes probes such as the following: (1) An initialisation probe (line 3 of Figure 2) that takes as a parameter the default threshold target value, and the lower and upper bounds. When the coordinator was instantiated (line 2 of Figure 2) it communicated with the QoS management system to get the application's QoS requirements in the form of the AttributeConditionList. Thus when the sensor requests this information, the coordinator will already have retrieved it. (2) A probe that (i) determines the elapsed time between frames and checks to see if this time falls within a particular range defined by the lower and upper acceptable thresholds. Unusual spikes are filtered out; and (ii) informs the coordinator if the frames per second fall below the lower threshold or is higher then the upper threshold.

When the coordinator retrieved the QoS requirements, it also retrieved the action list consisting of the actions in the form of the ActionList to be taken if the QoS requirement is not satisfied. In this example, this action is to notify the Event Manager process of the QoS management system whose handle is EventManagerHandle. The coordinator generates an event whose identifier is the event identifier found in the received action list.

We note that not all sensors measure attributes that are directly used in the specification of a QoS requirement. For example, a sensor may be used to measure the current size of the communications buffer.

2.3 QoS Management System

In the IETF framework, a Policy Decision Point (PDP) is used to retrieve stored policies (which is stored in the repository), and interpret and validate them. PDPs also make decisions on actions to be taken based on the receipt of an event which is generated from the monitoring of the environment. A Policy Enforcement Point (PEP) applies these actions. In this section, we will describe how PDPs and PEPs are used in this work and the additional components needed to support the QoS management strategy described in Section 1.

Name Server. When an applications starts up, it instantiates its coordinator. The coordinator then registers with the Name Server. The Name Server receives and maintains in a repository registration data from other components and application processes. It

Given: Video application *v*.
QOS expectations *e*.

---

1. Perform initialization for *v*.
2. Initialise coordinator *c*.
3. Execute $s_1 \rightarrow$ *init_probe(e)*
4. **while** (*v* **not done**) **do:**
5. Retrieve next video frame *f*.
6. Decode and display *f*.
7. Execute $s_1 \rightarrow$ *probe_framerate()*
8. **endwhile**

Figure 2. Instrumentation Example

assigns a unique instance identifier for each registered process. The Name Server coordinates the interaction between the QoS management components and the application's coordinator.

Policy Decision Points (Policy Manager). Within an administrative domain, there is one Policy Decision Point (PDP) that communicates with a repository where policies are stored and information about users is kept. This PDP is referred to as a *Policy Manager*. The Policy Manager is responsible for retrieving and mapping high-level policies to rules and communicating the rules to the appropriate management entities. The need for this should become clearer later in this section. The Policy Manager keeps track of where policies were deployed to.

After the application has registered with the Name Server, it requests its QoS requirements from the Policy Manager. It does this by generating an event requestQoS-Requirements(ProcessInfo). Upon receiving the request, the Policy Manager determines the policy that applies. If the policy that applies is Example 2 then the Policy Manager will execute the script DetermineAllowedFrameRate. The Policy Manager sends the QoS requirements and actions to be taken in the case of QoS violation to the application's coordinator as a condition list and action list which were described earlier.

The Policy Manager is the only type of PDP to receive events directly. The reason for this is the assumption that within an administrative domain there is only one Policy Manager and that the only type of event that it can receive is the requestQoS-Requirements event.

Event Manager. The Event Manager receives events and allows other management entities to register an interest in an event. The Event Manager can receive events from the coordinators that are generated in response to a violation of QoS requirements and it can receive events that represent a regular report of monitored data e.g., resource usage information.

Policy Decision Points for Violation Location. In Section 1, we identified the need for violation location. In this work, the location process is event-driven in the sense that the location process does not begin until an event indicating that a QoS require-

ment is violated is generated. Conditions on the state of the system are evaluated to help in diagnosis.

PDPs are used to respond to a violation of a QoS requirement. This violation is an event. The PDP uses this event and other monitored information to diagnose the problem. Diagnostics are guided by *diagnosis* policies. A diagnosis policy associated with Example 1 is the following:

EXAMPLE 3

```
inst oblig fps_low1
{
   subject s = /ca/uwo/csd/brown/pdpServer;
   target t = /ca/uwo/csd/syslab/rockyroad/PEP;
      on fps_low(VideoClientInfo, current_fps, low_target_fps
                 current_buffer_size, current_buffer_trend);
      do t.CPUIncrease(VideoClientInfo.ProcessIdentifier,current_fps),
      when registered(VideoClientInfo.ProcessIdentifier) and
                     (current_fps < low_target_fps) and
                     (buffer_size > bufferThresholdValue) and
                     (current_buffer_trend = UP);
};
```

This policy states that if the current frame rate (current_fps) is lower then the target frame rate (target_fps) and if the size of the client's communication buffer (current_buffer_size) is higher then a specific buffer threshold and if the client's buffer size (current_buffer_trend) tends to increase it should be requested that the resource manager at host *rockyroad.sylab.csd.uwo.ca* should increase the CPU priority of the video client. A policy fps_low2 is used to specify that if the event fps_low has occurred and the size of the buffer is relatively low and the CPU load of the machine that hosts the video server is high, then the manager process on the video server's host machine is to be instructed to increase the CPU priority of the video server process. These related policies are grouped into one composite policy.

There is a similar composite policy for when the current frame rate is higher then the higher bound allowed on the frame rate.

PDPs get the diagnosis policies associated with an application after the application registers with the Name Server. Upon receiving a registration request from an application process through its coordinator, the Name Server registers the application process with a PDP. There may be several PDPs. It is assumed that the administrator has determined how application processes are assigned to PDPs (which can be formalized as policies). The Policy Manager retrieves the diagnosis policies. However, Ponder policies are not executable. The user of Ponder makes it relatively easy for an administrator to use it, but it is too high-level to be executable by a PDP. An alternative formalism for specifying policies is needed. We refer to an executable policy as a rule. In this work, the alternative formalism was chosen to be JESS. An example of a JESS corresponding to the policy in Example 3 is the following:

EXAMPLE 4

```
( defrule fps_low_local
    ( registered process_id ?pid )
    ( fps low ?current_fps ?target_fps)
    ( and ( buffers high ?current_buffer )
          ( buffers up_trend ?current_buffer )
```

```
      )
=>
 bind ?amount ( resource_distance ?current_fps ?target_fps 10 )
( if ( > ( resource_conditions ?pid rtcpu ?amount ) 0.5 )
     then
     ( adjust_resource ?pid rtcpu ?amount ) )
  ( if ( < ( possibility_condition ) 0.1 )
     then
     ( assert ( service_diff rtcpu ) )
     )
  ( set_reset_delay 1 )
)
```

We will not go into details of what each specific part means. The point being made here is that formalisms that can be executable are usually more difficult to understand by administrators. Hence, the need for different formalisms.

JESS actually refers to a low-level rule engine and language. The Policy Manager will map the diagnosis policies specified using Ponder to JESS rules and then send the rules to the requesting PDP. Also sent to the PDP are the corresponding event identifiers that trigger the execution of rules. The PDP then registers its interest in the event identified by the event identifier with the Event Manager.

A PDP uses the set of rules to determine the corrective action(s). This involves determining the cause of the policy violation and then determining a corrective action(s). The process of determining the rules to be applied is called inferencing. Inferencing is used to formulate other facts or a hypothesis. Inferencing is performed by the Inference Engine component of the PDP. The inferencer chooses which rules can be applied based on the fact repository. The inferencing that can take place can either be as complex as backward chaining (working backwards from a goal to start), forward chaining (vice-versa) or as relatively simple as a lookup. In this work we used forward chaining.

A PDP is also used to authorize actions for PEPs on behalf of applications. This will be explained in more detail.

Policy Enforcement Point. A PEP is a management process that is responsible for monitoring the device that it is on and executing actions. A PEP receives requests from a PDP for a resource allocation. It verifies that it may execute the action, through the use of a PDP. This PDP uses *administrative* policies that formalize the administrative requirements.

Administration requirements can be specified using Ponder authorization policy constructs. In Ponder authorization policies, a policy is imposed by the target object, where the target can prohibit an action on itself based on different criteria such as subject, date and time. An authorization policy type associated with Example 1 is the following:

EXAMPLE 5

```
type auth+ authCPUIncreaseT (subject s, target t)
{
  action  CPUIncrease(ProcessIdentifier,normalizedvalue)
    when belongs(GroupA, ProcessIdentifier) and
    CPUResourcesAvailable(normalizedvalue);
}
```

This can be instantiated as follows:

```
inst auth+ allocateResourcePolicy=
  authCPUIncreaseT(/ca/uwo/csd/syslab/rockyroad/PEP,
                /ca/uwo/csd/syslab/rockyroad/ClientVideoProcess);
```

This policy states that PEP on .../syslab/rockyroad is allowed to increase the CPU priority for a video client process if the video client process belongs to GroupA and if there are enough CPU resources which is calculated by CPUResourcesAvailable. The parameters to this action include the process identifer (ProcessIdentifier and a normalized value (normalizedvalue) representing the difference between the attribute's current value and the expected value. The actual control of allocating CPU resources is encapsulated by a *resource manager*. There is a resource manager for each resource.

This is a simple policy and it is not application specific. It is host machine specific. A possible application-specific policy could be the following:

EXAMPLE 6

```
type auth+  authChangeApplicationResolution (subject s, target t)
{
  action ChangeResolution(ProcessIdentifier);
  when not(CPUResourcesAvailable(normalizedvalue));
}
```

This can be instantiated as follows:

```
inst auth+ authChangeApplicationResolution=
 authChangeApplicationResolution(/ca/uwo/csd/syslab/rockyroad/PEP,
                                 /ca/uwo/csd/syslab/rockyroad/ClientVideoProcess
```

This policy specifies that /ca/uwo/csd/syslab/rockyroad/PEP requests that the video client process's resolution is to be changed if there are not enough CPU resources available. This is an authorisation policy since this action should be prohibited except under specific circumstances.

Policy Editor. The Policy Editor provides a Graphical User Interface (GUI) for the administrators to compose and store high-level management policies (through the Policy Manager) in a policy repository.

QoS Management System Organization. It is assumed that each administrative domain has one policy server and at least one PDP. It is possible to have a configuration with one PDP on each device, a configuration with one centralized PDP or a configuration with more than one PDP, but not one on each device.

3. Implementation

We have built a C++ instrumentation library that implements a hierarchy of sensor classes. The base of the hierarchy provides a registration method that provides the ability to have all sensors register with the coordinator in a uniform manner. Other base methods are for enabling/disabling and read/report. C++ was used since we have an instrumentation library that has been developed and evolved over a period of time. There is a prototype of a Java instrumentation library that can be used for Java programs to interact with the QoS management system.

The repository used was LDAP. LDAP is used since this seems to be the choice of the IETF. The PDP, Event Manager, Name Server, and Policy Manager were all written

in Java. The rule engine used by the PDP is JESS. The PEP has two components. The first component, implemented in Java, interacts with PDPs. The second component has specific resource managers such as the CPU manager and the Memory Manager. These resource managers actually manipulate the resources. This is written in C and is specific for Solaris. Currently, the CPU manager is integrated into the prototype.

The Policy Editor has a GUI that allows the user to enter policies The GUI was designed so that the user could be guided through the development of the policy by selecting whether they want to define event names, events, actions, attribute conditions (through the constraints). The policies were put into XML and references to the XML file were placed in the LDAP directory. XML was the language syntax used to express the Ponder policies. The user is able to directly put Ponder policies (in the Ponder format) in files and ask that they get loaded into the LDAP repository.

Translation is based on the existence of templates for each type of Ponder policy. The Policy Manager has a JESS template of rules associated with each Ponder policy. The Policy Manager uses this template to guide the transformation of a parsed Ponder component to a JESS rule or part of a JESS rule.

We have deployed the prototype within our laboratory and instrumented a number of applications. Currently, the prototype works as expected.

4. Discussion

This section describes our observations and experiences.

1 Section 2 describes how an application can be instrumented. It will not always be possible for a user or administrator to instrument an application themselves. They may have to rely on the output of logfiles that many applications generate. It is still possible to use the architecture of the QoS management system described in Section 3. The coordinator refers to a process that reads the logfiles generated by an application or uses the operating system to retrieve information about the application behavior.

2 Currently, PEPs determine if they are authorized to carry out an action by sending a message to the PDP. Potentially, the PDP could become a bottleneck. The PEP could have a rule engine to directly determine if it can carry out the action. This would reduce the load of the PDP, but increase the management load on the device that the PEP is on. This is not a problem in the sense that our architecture can easily handle this. Alternatively, multiple PDPs might be necessary. One PDP focusses on diagnosis and the other PDP focusses on messages from the PEP.

3 In our earlier work [11], diagnosis was distributed. This assumed that there were management processes on each host machine as well as a 'global' management process for an administrative domain. Rules to diagnose the cause of a violation of a QoS requirement were found in the local management processes as well as the 'global' management process. These management processes are similar to the PDPs. We found the translation of diagnosis policies to rules and the deployment of those rules was more complicated. This is the result of the translation having to take into account that if management process determines that it cannot resolve the problem, it needs to send an event to another management process so that it might try to resolve the problem. On the otherhand,

the use of one PDP results in many rules being sent to that PDP and it could become a bottleneck. More work is needed in understanding how to optimize the configuration of PDPs.

4 It is possible to have the different management entities interested in a specific event to directly register with the component generating the event i.e., the generator of the event acts as its own event manager. The disadvantage is that this assumes that these management entities know *apriori* where the events are coming from. Another possiblity is to have the policy specifications explicitly specify the source and receivers of a specific event. However, if we change the configuration of PDPs by changing their location and/or number then the policy specification should be changed. By separating these out, we can continue to use the same policy specification even if the underlying management system changes.

5 It is assumed that there is agreement on attribute names. It is assumed that the attribute names of values that are sent as part of a notify message from the coordinator are the same as what is expected in the diagnosis policies. This requires that policies entered are checked for consistency.

6 The policy stated in Example 5 is quite detailed in that it assumes that the administrator will specify the normalized value (specified using normalizedvalue) is to be used to determine if there are enough CPU resources available. We believe that we can simplify this by assuming that this is part of the mapping of the policy to rules.

7 We find the specification of expectation and administrative policies relatively easy. Diagnostic policies are more difficult and require a good understanding of the applications and how they behave. The administrator must specify a good deal of information. There is a good deal of work on diagnostics (e.g., [10, 9, 7, 8]) that use a variety of techniques. There exists efficient software to support these techniques. We believe that the diagnostics process should be able to take advantage of these techniques and tools. This will likely require changes in the way diagnostic policies are specified. Potentially, this could reduce the number of rules needed by the rule engine.

8 JESS is a rule-based engine. Initially, we found that JESS had a lot of overhead and had an impact on the application processes on the executing on the same machine. However, this was optimized and we no longer have this problem. It seems to be fast enough, but we do not have experience with an environment with hundreds of applications and potentially thousands of policies. However, in this case there would likely be more PDPs. This would distribute the load and thus minimize the number of policies at a specific PDP.

5. Related Work

Our work involved the use of policy formalisms and the development an architecture for distribution and enforcement of policies. Relatively little of this work examined all of these issues in an integrated framework that addresses end-to-end QoS management. Our work does this. In this section, we describe the related work done in policy specifications and architectures. We describe how we differ and how our work relates.

Policy Specification.

IETF is currently developing a set of standards (current drafts found in [14, 19, 22]) that includes an information model to be used for specifying policies, a standard that extends the previous standard for specifying policies for specifying QoS policies and a standard for mapping the information model to LDAP schemas. Informally, speaking an IETF policy basically consists of a set of conditions and a set of actions. The standards focus on low-level details related to policy encoding, storage and retrieval. We have experience in using the IETF standards. We did examine the use of IETF rules for specifying policies. IETF rules are reasonable for expectation policies, but we found that it was much more difficult to specify the administrative and diagnostic policies. Generally, we found easier to use. It is possible to specify policies using Ponder and translate to the IETF format.

Other language standards by IETF, as well as other network policy languages (summarized in [21]), include RPSL for describing routing policy constraints, PAX for defining pattern matching criteria in policy-based networking devices and SRL for creating rule sets for real-time traffic flow measurement. These languages primarily focus on policies applied to a network device. None of this work addresses the use of policies applied to applications.

Other policy specification languages [2, 6, 5, 15] focus on features that enable the specification of security related policies. The Ponder Policy Specification language [3] has a broader scope than most of the other languages in that it not only was designed with specifying security as the primary objective, but also general management policy. Ponder allows the administrator to use declarative statements and is independent of underlying systems. This is the primary reason why we have chosen to use Ponder to describe a high-level specification representation of policies.

Policy languages vary in the level of abstraction. Some languages focus on the specification of policies that describe what is wanted, while others focus on how to achieve what is wanted. This has resulted in different levels of policies, which has led to several attempts at a policy classification. This includes a policy hierarchy and a formal definition of policies defined by [24]. Our work defines two levels and would seem to correspond to the bottom two levels of the hierarchy in [24].

Architectures, Policy Distribution and Enforcement.

There has been some recent work in service level management [17, 16] that describes an architecture and policies for management of a differentiated services network so that users receive good quality of service and fulfill the service level agreements. This work does not make use of administrative or diagnostic policies.

There has been quite a bit of work (e.g., [1, 18, 23, 12]) that has looked at the translation of policies to network device configurations which are then sent to the PEPs so that the PEPs can configure the network devices. The focus is on network devices. In most of this work, policy distribution is initiated by an administrator and focusses on one device. Our work differs in that it focusses on applications and it is the initiation of an application session that causes the distribution of polices to management components.

A deployment model was developed by [4] for Ponder. Each policy type is compiled into a policy class by the Ponder compiler. The instantiation of a policy class is a policy object. Each policy object has methods that allow a policy object to be loaded to an *enforcement* agent and unloaded from an agent. Additional methods include enable and disabling of objects. Agents register with the event service to receive relevant

events (as specified by the policies) generated from the managed objects of the system. There is no discussion on configuration of devices or applications. The roles of PDP and PEP is similar to that of an enforcement agent. We decided though that use of JESS would make certain tasks easier. For example, although Ponder assumes that the policies are independent in the sense that the order they are presented in does not matter. We assumed that order of policies mattered when translating to JESS rules. This meant that not every single JESS rule that is interested in a specific event has to be executed. This is important for the following reason: Example 3 described a policy called fps_low1 that is triggered when the event fps_low is generated. fps_low2 is also to be triggered when the event fps_low is generated. The difference is that that the action to be taken depends on different conditions of the state. If the conditional statement in the when clause is true in Example 3 this implies that the fault has been found and there is no reason to look at fps_low2. JESS's implementation allows for this. In the Ponder deployment model, all policy objects that register for a specific event will receive notification that the event has occurred and conditions are checked.

Generally, we found that very little work focusses on applying policies at the application level. The difficulty with the application level is that different policies will be needed for different sessions of the same application.

6. Conclusions

The initial deployment of the prototype has proven to be successful. The experimental results are similar to that reported in [13, 11]. The prototype allows a user to specify Ponder policies from a console and distribute them to the distributed components of the management system successfully. This work is one of the few that addresses the use of policies to application QoS management.

Future work includes addressing some of the issues brought up earlier in the Discussion section as well as the following:

1 We chose to specify our framework based on the IETF policy-based management framework. We are currently working on integrating network resource management.

2 We would like to do more work on ensuring that policy specifications are consistent. This requires many changes to the mapping approach used.

3 We are examining an environment with a mix of applications such as soft IP phones and web servers.

References

[1] M. Brunner and J. Quittek. MPLS Management using Policies. *Proceedings of the 7th IEEE/IFIP Symposium on Integrated Network Management (IM'01)*, Seattle USA, May 2001.

[2] F. Chen and R. Sandhu. Constraints for Role-Based Access Control. *Proceedings of 1st ACM/NIST Role Based Access Control Workshop*, Gaithersburg, Maryland, USA, ACM Press, 1995.

[3] N. Damianou, N. Dalay, E. Lupu, and M. Sloman. Ponder: A Language for Specifying Security and Management Policies for Distributed Systems: The Language Specification (Version 2.1). Technical Report Imperial College Research Report DOC 2000/01, Imperial College of Science, Technology and Medicine, London, England, April 2000.

[4] N Dulay, E. Lupu, M. Sloman, and N. Damianou. A Policy Deployment Model for the Ponder Language. *Proceedings of the 7th IEEE/IFIP Symposium on Integrated Network Management (IM'01)*, Seattle USA, May 2001.

[5] J. Hoagland, R. Pandey, and K. Levitt. Security Policy Specification Using a Graphical Approach. Technical Report Technical Report CSE-98-3, UC Davis Computer Science Department, UC Davis Computer Science Department, July 22, 1998 2002.

[6] S. Jajodia, P. Samarati, and V. Subrahmanian. A Logical Language for Expressing Authorisations. *Proceedings of the IEEE Symposium on Security and Privacy*, 1997.

[7] S. Katker. A Modelling Framework for Integrated Distributed Systems Fault Management. In *Proceedings IFIP/IEEE International Conference on Distributed Platforms*, pages 186–198, 1996.

[8] S. Katker and H Geihs. A Generic Model for Fault Isolation in Integrated Management Systems. *Journal of Network and Systems Management*, 1997.

[9] S. Katker and M. Paterok. Fault isolation and event correlation for integrated fault management. In *Proceedings of the 5th International Sypmposium on Integrated Network Management*, 1997.

[10] S. Kliger, S. Yemini, Y. Yemini, D. Ohsie, and S. Stolfo. A Coding Approach to Event Correlation. In *Proceedings of the 4th International Sypmposium on Integrated Network Management*, 1995.

[11] H. Lutfiyya, G. Molenkamp, M. Katchabaw, and M. Bauer. Issues in Managing Soft QoS Requirements in Distributed Systems Using a Policy-Based Framework. *Proceedings of the Policy 2001 Workshop:International Workshop on Policies for Distributed Systems and Networks, Bristol, UK, Springer-Verlag LNCS*, pages 185–201, January 2001.

[12] P. Martinez, M. Brunner, J. Quittek, F. Strauss, J. Schoenwaelder, S. Mertens, and T. Klie. Using the Script MIB for Policy-Based Configuration Management. *Proceedings of the IFIP/IEEE Symposium on Network Operations and Management Symposium (NOMS'02), Florence, Italy, 2002*, pages 461–468, January 2001.

[13] G. Molenkamp, H. Lutfiyya, M. Katchabaw, and M. Bauer. Resource Management to Support Application-Specific Quality of Service. *IEEE/IFIP Management of Multimedia Networks and Services (MMNS2001)*, October 2001.

[14] B. Moore, J. Strassmer, and E. Elleson. Policy Core Information Model – Version 1 Specification. Technical report, IETF, May 2000.

[15] R. Ortalo. A Flexible Method for Information System Security Policy Specificatio". *Proceedings of 5th European Symposium on Research in Computer Security (ESORICS 98)*, Louvain-laNeuve, Belgium, Springer-Verlag, 1998.

[16] P. Pereira, D. Dadok, and P. Pinto. Service Level Management of Differentiated Services Networks with Active Policies. *3rd Conferencia de Telecomunicacoes.*, Rio de Janeiro, Brazil, December 1999.

[17] P. Pereira and P. Pinto. Algorithms and Contracts for Network and Systems Management. *Proceedings of the 1st IEEE Latin American Network Operations and Management Symposium (LANOMS99)*, Rio de Janeiro, Brazil, December 1999.

[18] Alberto Gonzalez Prieto and Marcus Brunner. SLS to DiffServ Configuration Mappings. *Proceedings of the 12th International Workshop on Distributed Systems: Operations and Management DSOM'2001*, Nancy France, October 2001.

[19] Y. Snir, Y. Ramberg, J. Strassner, and R. Cohen. Policy Framework QoS Information Model. Technical report, IETF, April 2000.

[20] Startdust.com. Introduction to QoS Policies. Technical report, Stardust.com, Inc., July 1999.

[21] G. Stone, B. Lundy, and G. Xie. Network Policy Languages: A Survey and New Approaches. *IEEE Network*, 15(1):10–21, January 2001.

[22] J. Strassner, E. Ellesson, B. Moore, and Ryan Moats. Policy Framework LDAP Core Schema. Technical report, November 1999.

[23] P. Trimintzios, I. Andrikopoulos, G. Pavlou, and C. Cavalcanti. An Architectural Framework for Providing QoS in IP Differentiated Services Networks. *Proceedings of the 7th IEEE/IFIP Symposium on Integrated Network Management (IM'01)*, Seattle USA, May 2001.

[24] R. Wies. Polcies in Network and Systems Management – Formal Definition and Architecture. *Journal of Network and Systems Management*, 2(1):63–83, 1994.

[25] D. Yau and S. Lam. Adaptive Rate-Controlled Scheduling for Multimedia App lications. *Proceedings of the 1996 ACM Multimedia Conference*, Boston, Massachusetts, November 1996.