Model and Architecture of a Timing Service for Adaptive Policy-Based Management Systems

Christoph Angerer
Computer Systems Institute
ETH Zurich
CH-8092 Zurich
Email: christoph.angerer@inf.ethz.ch

Thomas Gross
Computer Systems Institute
ETH Zurich
CH-8092 Zurich
Email: thomas.gross@inf.ethz.ch

***Abstract*— Policy-Based Management (PBM) is a promising approach to realize adaptivity in networks that would be hard — if not impossible — for administrators to configure manually. In PBM systems, the network configuration is not directly expressed in terms of low-level properties, such as IP or MAC addresses; rather, complex business entities, such as users or departments, can be defined and allow the administrators to specify high-level network policies.**

Even though PBM can reduce the effort for managing complex networks significantly, today's solutions lack a generic way for expressing adaptive policies. A PBM system can be called adaptive if the decision of when and in which order policies are activated is not only based on fixed predefined dates but the system also takes environmental and internal events into account.

In this paper we describe a model of adaptive timing constraints and introduce an architecture for executing timing specifications (sets of constraints). Within a timing specification, the administrator usually creates single constraints only implicitly by defining the lifetime of managed resources, sequences of events, and high-level temporal relationships between them. Because each timing constraint defines a $<$ relation between two single events, a timing specification can be described as a partial ordering of events over the event space. During execution, a Timing Service triggers PBM systems to activate, inactivate, or adapt management policies according to the timing specification.

I. Introduction

Today, specialized communication channels are introduced for various applications such that each of these channels requires a unique quality of service (QoS) from the underlying network. Some applications, such as video streaming or voice over IP (VoIP), demand certain guaranteed bandwidth and throughput; others have special security or management constraints, often summarized as authentication, authorization and accounting (AAA).

One effective way to provide an environment in which the various applications can coexist is to configure the network itself appropriately (i.e., its intermediate devices such as peers, routers, or switches). Virtual networks (VLANs) on top of the physical network, for example, are often used to segment the network into zones where each zone can offer a certain guaranteed quality of service. A segment may be expressed as a set of properties such as the used protocols, the IP- and/or MAC-address, or even on a per-user basis. While — in theory — this segmentation could be realized by software on the application layer, it is much more efficient to implement it on a lower level [16]. Switched Networks, for example, work on the MAC layer: Through dynamic configuration of the switches, the network topology itself can be adapted to the actual usage of the network.

Managing networks on such a fine-grained level is a complex and error-prone task, however. All network devices must be individually set up and tested in a complex environment. At the same time the requirements on the network are constantly changing: new hardware is introduced, old hardware is removed, devices join and leave the network, new applications require new QoS guarantees, and users move in physical as well as in organizational space.

Policy-Based Management (PBM) is a promising approach to simplify management of complex networks [9]. RFC 3198 defines a policy to be a "definite goal, course or method of action to guide and determine present and future decisions" that is realized as a set of rules to administer, manage, and control access to network resources [15]. Policies allow the administrators to specify a network in terms of business entities, like users or organizations, rather than in terms of low-level network features such as IP and MAC addresses. These policy rules usually follow IF, WHAT, WHEN, and THEN logic (example adopted from [4]):

> *If:* The user is CEO of the company
> *What:* the application is watching streaming video
> *When:* the time is between 9 a.m. and 5 p.m.
> *Then:* the user is entitled to a service level

[0]This work was funded, in part, by the NCCR "Mobile Information and Communication Systems", a research program of the Swiss National Science Foundation.

Premium that guarantees a throughput of 2Mbps and an end-end latency of no more than 150ms.

Even though policies can reduce the effort for network management significantly, they are too static for a lot of dynamic management problems. For managing networks in dynamic organizations policies must be adapted over time to respond to changes in the environment.

In fact, project managers face problems similar to network administrators in respect to planning and managing highly dynamic systems: Predicting the exact project end or joining and departure of project members, for example, can be difficult not only for long running projects. In such cases, managers often use project management tools to model the timing constraints they must deal with. Applications like MS Project [12] allow a manager to define fixed and definite events, such as the project start date or certain milestones. For elements where a definite time is not yet known, constraints, like minimal and maximal durations or known interdependencies of tasks, can be provided; the application then assures that all these constraints are met in the final plan.

One major drawback of the Policy Core Information Model (PIM) described in RFC 3060 [9] and extended by RFC 3460 [10] is the lack of a method for specifying such event-based timing constraints. In PIM, timing constraints are expressed as time periods either in local time or UTC time: A policy is said to be 'active' within its associated time period only. Various additions allow more fine-grained specifications by specifying masks for the months of a year, days of a month, days of a week, and times of a day.

In all these cases the exact time when the policy becomes active must be known in advance. But just like high-level business entities can be used in a network policy, one may analogously want to use higher level constructs for expressing temporal dependencies: Instead of providing *definite* times for each policy an administrator may also want to provide *logical* (or *indefinite*) times such as 'while our special offer is valid, the help-desk gets more bandwidth' or 'after user A has retired, policy B expires and policy C steps in'.

In this paper, we propose a framework for specifying such high-level timing constraints as well as an architecture to integrate these timing specifications with existing PBM systems. In the next section we present related work in the field of policy-based management. Section III presents a model that is capable of expressing adaptive timing constraints. A possible architecture of a Timing Service (TS) as well as our current implementation is described in Section IV. Finally, Section V concludes the paper.

II. Related Work

This work employs techniques developed for synchronizing media playback in multimedia systems to Policy-Based Management (PBM). PBM is currently attracting considerable attention as an enabling technology for managing distributed Multi-Protocol Label Switching (MPLS) environments, large scale IP networks, and heterogeneous information infrastructures [17]. PBM systems aim at methods for managing quality of service in enterprise networks and for controlling access to resources. Despite their common usage, however, PBMs are not restricted to network management only. Because the "policy classes and associations defined in this model are sufficiently generic to allow the representation of policies related to anything" [9] one can also use this framework to consider scenarios like session management in collaborative peer-to-peer applications or even controlling access to buildings

A. Policy Based Management

The IETF has published miscellaneous RFCs to specify the services and protocols for PBM systems [9], [10], [15]. Even though implementations and case studies exist that provide a proof of concept of these specifications, Law and Saxena [7] claim that the original design does not scale for large networks. Instead, they propose an alternative multitiered architecture in which middle-tier agents offer flexibility and scalability to the design and support load balancing mechanisms. The multi-tier architecture allows the system to scale even for large networks, but it does not address the problem of specifying temporal relationships between policies.

Access control models such as Role Based Access Control (RBAC) which are commonly associated with PBM systems require the administrators to model the human organization and roles in detail [2], [5], [13]. However, fluctuations in human communities, the heterogeneity of networks, and constantly changing resources demand more flexible approaches. Much work done to adapt RBAC for realizing dynamic scenarios concentrates on how the organization is modeled. Extensions to RBAC with a temporal component as done by Bertino et al. [1] rely on temporarily enabling or disabling the permissions of entire roles. This approach requires a detailed knowledge of the specific times, such as the exact dates or weekdays when roles should become active or inactive.

Large and complex networks do not ocurr only in controlled environments; rather, they can also be highly dynamic and unconstrained. The Internet is an example of a large community of resources that must be managed in an environment characterized by decentralized, non-hierarchical decision making in the

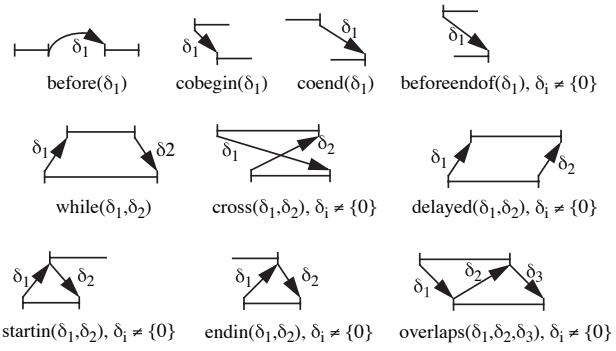

Fig. 1. Basic timing patterns (adopted from [14])

presence of distributed resource ownership. Feeney et al. [8] present a community-based model for the management of policies in the context of a large Internet community. Their central grouping construct, called a *community*, simplifies the specification of organizations whose internal structures are not well-defined. However, they focus on modeling *structural uncertainty* (hierarchies and groups) while we focus on how to deal with *temporal uncertainty*.

B. Representing Time in Multimedia Systems

Multimedia systems have led to the development of various models for expressing temporal relationships between single entities. Multimedia systems integrate a variety of time-based media items, such as audio, video, text, and images.

When playing back the media, these systems must assure the correct temporal appearance of the single items by a process called *synchronization*. The schedule for how the synchronization must be done is specified, for example, in the form of XML-files that define when, where, and how long a certain item should appear on the screen; such structures therefore play the role of a timing specification. One famous example of this approach is SMIL, the Synchronized Multimedia Integration Language [6].

Two approaches have been developed for expressing the temporal interdependencies of the media items: *point-based* models and *interval-based* models. While the elementary units in point-based models are discrete events in a one-dimensional time space (where an event can happen before, simultaneous, or after another event), interval-based models describe relationships of whole time intervals (for example, interval α while interval β). Wahl and Rothermel [14] present the ten basic time interval patterns depicted in Figure 1, which are capable of expressing all temporal relationships between two intervals. The work presented in this paper is based on these time patterns for modeling temporal relationships between policies or resources.

III. A Model for Specifying Adaptive Timing Constraints

Most multimedia systems model timing constraints as a set of events associated with scalar timestamps. The events tell the system when it is supposed to start or stop a piece of media during playback. The timestamps are defined in respect to one or more clocks (usually counting the number of samples that have been played until the current movie time at a given rate) that specify the speeds and durations of how the individual media chunks must be presented.

Due to their intended usage of playing *back* multimedia files, the time model, be it point-based or interval-based, allows arranging the elements on this one-dimensional timeline only. This arrangemetn requires a designer to associate definite clock-times with each of the events — an exact timestamp when a certain event is supposed to happen.

In current time-based systems, indefinite (logical) time definitions are often not considered because, the designers are aware of the detailed sequence of events. Thus a designer knows, in advance of timing the time events, the exact positions of decision points where a spectator may be allowed to interact with the application and therefore may influence the media presentation.

This concept is also common in current policy models that consider only relatively simple timing constraints (i.e., fixed and periodically recurring events). For expressing a policy like ‘grant the CEO higher bandwidth for watching a broadcast each Monday between 5:00 PM and 7:00 PM’ this model is sufficient (even though this constraint could be further refined by ‘but not after the end of the broadcast’ in case it ends early). However, simple constraints are not expressive enough to activate a policy at a *defined* event (i.e., an event that is well-known and can be identified) but at an *indefinite* time (i.e., the exact occurrence can at best be estimated); ‘after retirement’ would be an example for such an event.

The basic idea behind the model presented in this paper is to express a policy as a time interval with given start and end events. These events are logical events that are separated from the definition of the time when they actually happen. This decomposition adds the flexibility that is needed for adaptive PBM because it allows to associate an event not only with a scalar timestamp but also with higher-level temporal relationships to other events. The next section provides some general definitions and terms that will be used in the succeeding sections. The single model elements are then described in detail in Sections III-B and III-C.

A. Introductory Definitions

1) Clocks and Timestamps: Let $\mathbb{N}$ be the set of non-negative integer numbers. Any scalar $s \in \mathbb{N}$ that is used to refer to a time is called a *timestamp*. A $clock$ is a function: $\mathbb{N} \rightarrow \mathbb{N}$ that maps the current global time τ to a current clock-dependent timestamp (i.e., when called at any time, a clock returns its current scalar value). The current timestamp of any given clock $c \in C$ of the set of all possible clocks C is denoted by $c(\tau)$.

Clock functions do not need to be monotonically increasing (i.e., a clock does not necessarily count forward). An example for this is a 'countdown' clock.

2) Time Definition: We call a time definition (or simply time) $t \in T$, where T is the set of all possible time definitions, to be *definite* if the function of the form:

$$allOccurrences : T \times C \rightarrow \mathbb{N}^*$$

is defined for t. $allOccurrences$ returns a set of scalar timestamps for a given time definition t and a clock c and is already a way to express simple timing constraints. The definition:

$$allOccurrences(t, c) := \begin{cases} \{1, 8, 15, \ldots\} & \text{if } t = t_1 \\ & \wedge\ c = c_1 \\ \{\} & \text{else.} \end{cases}$$

for example constrains t_1 to a period of seven days (in respect to clock c_1 that counts days). There can be more than one timestamp for each time definition because it is possible to express recurring times, such as 'every Monday at 12:00'.

We call t to be *indefinite* if $allOccurrences$ is undefined for t. In all cases, being 'indefinite' is a property that is related to the future. Take, for example, an indefinite time definition t_2 = 'switch turned on'. As soon as t_2 happens (i.e., the switch is turned on), a scalar timestamp exists; but it is still indefinite for the future (i.e., it is unknown when the switch is turned on the next time).

Since $allOccurrences$ is the association between all the definite times in T with all clocks in C, indefinite times are not associated with clocks directly. Instead, they are associated with the occurrence of either system internal events or with detected external events. As a consequence, an implementation of the Timing Service must provide a mechanism to use (service external) events in the time definitions and to monitor the environment for their occurrences.

Furthermore, time definitions like t_2 require the Timing Service to first understand the syntax and semantics of this expression and then to monitor the environment (the state of the switch in this case). Note, that due to the linguistic expression of t_2, it may sound as being a high-level event; but in this example, the expression denotes only a simple trigger for a logical event like "light turned on" or "TV turned on", depending on the actual function of the switch. The meaning is carried in the event, not in the time. However, defining such a time definition language and monitoring mechanism is out of scope of this paper.

3) Next Occurrence and Time Comparison: The next occurrence of a definite time, that is, the smallest of all occurrences that is still equal or bigger than the current time of the clock is given by $nextOccurrence$ with the signature:

$$nextOccurrence : T \times C \rightarrow \mathbb{N}$$

and the definition:

$$nextOccurrence(t, c) := x \in \mathcal{O}$$

with $x \geq c(\tau) \wedge \nexists y \in \mathcal{O} : (y \neq x\ \wedge\ y \geq c(\tau)\ \wedge\ y < x)$ where $\mathcal{O} = allOccurrences(t, c)$. When no timestamp of a next occurrence of t can be computed, $nextOccurrence(t, c)$ is undefined.

Two times t_1 and t_2 can be compared with the operators $<$, $=$, $>$, and $||$. The comparison operators $<$, $=$, and $>$ are transitive, that is:

$$t_1\ R\ t_2 \wedge t_2\ R\ t_3 \Rightarrow t_1\ R\ t_3$$
$$\text{with } t_1, t_2, t_3 \in T\ \wedge\ R \in \{<, =, >\}$$

If $nextOccurrence(t_1, c_1)$ and $nextOccurrence(t_2, c_2)$ are defined for the same clock c, i.e., $c_1 = c_2$, the operators $<$, $=$, and $>$ are defined according to the corresponding operators in $\mathbb{N}$, and $||$ is always false.

If the next occurrences are undefined or the clocks differ, the times are said to be *potentially concurrent* (noted as $||$). These events are considered to happen in parallel as long as no further constraints (i.e., intervals, see Section III-A.7) are given.

4) Earliest and Latest Occurrence of Indefinite Times: By definition, it is not possible to tell the next occurrence of an indefinite time in terms of a single scalar value. However, it is possible to declare a time frame $(earliest(t), latest(t))$ in which an indefinite time t may occur. Here, $earliest(t)$ is defined as the latest time of all predecessors of t:

$$earliest(t) := x \in T \cup \{\vdash\}$$

with $x < t \wedge \nexists y \in T : (y \neq x \wedge y < t \wedge y > x)$ and the latest possible occurrence $latest(t)$ is the earliest time that directly succeeds t:

$$latest(t) := x \in T \cup \{\dashv\}$$

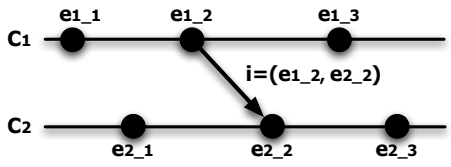

Fig. 2. Example of two clocks c_1 and c_2 with six event happenings and one interval i

with $x > t \wedge \nexists y \in T : (y \neq x \wedge y > t \wedge y < x)$. $\vdash$ denotes the time that compares smaller with every other time, i.e., $\forall t \in T : \vdash < t$, and $\dashv$ is the time that compares greater with every other time, i.e., $\forall t \in T : t < \dashv$.

5) Striking a Time: A clock $c \in C$ is said to *strike* a time $t \in T$ when the clock reaches the next occurrence of the time:

$$strikes(t, c) := \begin{cases} 1 & \text{if } nextOccurrence(t, c) = c(\tau) \\ 0 & \text{else.} \end{cases}$$

6) Events: Each event is associated with a certain time $time(e)$. When a clock strikes the time of an event $e \in E$ the event is said to *happen*:

$$happens(e) := \begin{cases} 1 & \text{if } \exists c \in C : strikes(time(e), c) = 1 \\ 0 & \text{else.} \end{cases}$$

If an event is associated with an indefinite time the event is said to be *armed* when any clock c strikes the earliest possible occurrence of $time(e)$:

$$armed(e) := \begin{cases} 1 & \text{if } \exists c \in C : earliest(time(e)) \leq c(\tau) \leq latest(time(e)) \\ 0 & \text{else.} \end{cases}$$

When an event is armed the system has to prepare for possible happenings of the event. If an event is not armed it is called *disarmed*.

7) Intervals: A meaningful period between two events is called an *interval*. An interval is bounded by exactly two events, a start event and an end event, and it is directed forward in time. It therefore constrains the times of its start and end events concerning the $<$ operation. When given two events e_1 and e_2 with $time(e_1)||time(e_2)$, the creation of an interval $i : (e_1, e_2)$ results in a new ordering relation $time(e_1) < time(e_2)$. If $time(e_1)$ is not concurrent to $time(e_2)$, $time(e_1) < time(e_2)$ must be true for all occurrences of the times before (e_1, e_2) can be created. Therefore, every creation of an interval preserves or increases the degree of order in the event-space.

By adding a ordering relation to the event space, intervals play an equivalent role as *messages* in models for synchronized clocks, such as vector clocks or dependency sequences [11].

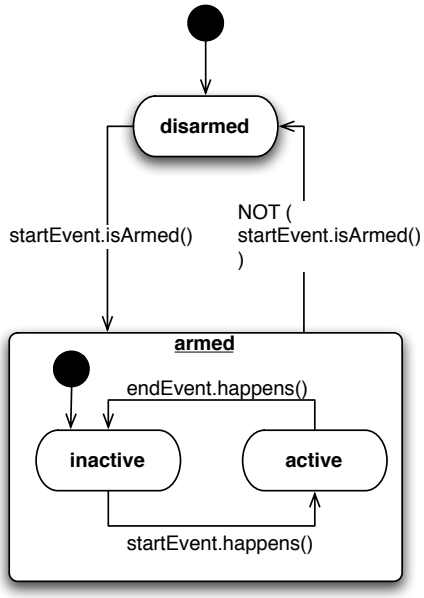

Fig. 3. State chart of interval activation

Figure 2 depicts an example of events whose times are defined in respect to two different clocks c_1 and c_2. Each clock defines a totally ordered space for its timestamps (since timestamps are elements of $\mathbb{N}$) and therefore it can be stated that $e_{1\_1} \leq e_{1\_2} \leq e_{1\_3}$. Furthermore, $e_{1\_1}||e_{2\_1}$ and $e_{1\_3}||e_{2\_3}$. However, since the interval $i(e_{1\_2}, e_{2\_2})$ implies $e_{1\_2} < e_{2\_2}$, it is further true that $e_{1\_1} < e_{1\_2} < e_{2\_2} < e_{2\_3}$.

The terms *armed* and *disarmed* can also be applied to intervals. Figure 3 depicts the internal states of an interval: When the start event of an interval gets armed, the interval gets armed, causing the system to prepare for a possible start of the interval. When the start event gets disarmed again, the whole interval gets disarmed. Being armed, the interval is said to become *active* when its start event happens. If the start event has not (yet) been happening or if the end event happens, the interval is said to be *inactive*.

8) Timing Specification: Definite and indefinite times, events, and intervals provide the basis upon which more complex timing specifications are built. We define a timing specification $spec$ as a tuple $(E, I, \mathcal{R})$ where:

- An event $e = (id, t)$ is a tuple that binds an identifier id to a time definition t. E then denotes the set of all events;
- $I \subseteq E \times E$ is a set of intervals where each interval $i = (e_1, e_2)$ is defined by two events: a start event e_1 and an end event e_2;
- and $\mathcal{R} : I \times I$ are binary timing relationships such as before or while between two intervals. In our model, we use the 10 basic patterns for timing

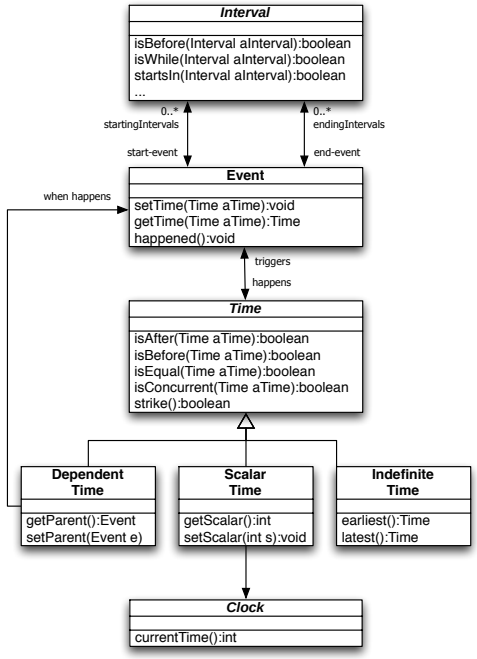

Fig. 4. Model of intervals, events, and time

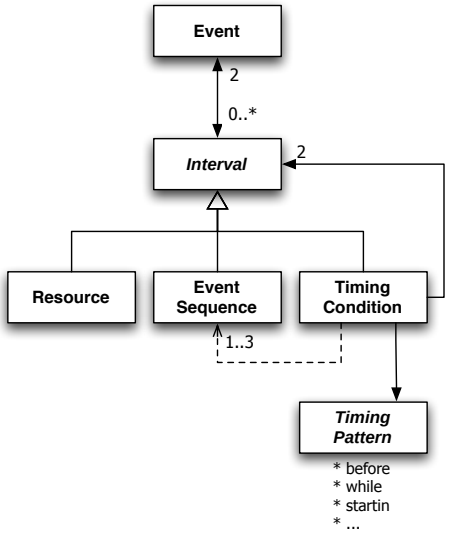

Fig. 5. Model for timing specifications

relationships adapted from [14] and depicted in Figure 1.

Compared to point-based models, interval-based models have the advantage to offer a higher level of abstraction that can facilitate a more intuitive way of specifying timing constraints. It is, for example, very straight forward to define 'apply policy A while the broadcast runs', while it is clearly harder to specify this constraint as 'policy A starts after the broadcast has started and ends before or together with the broadcast'. For this reason we propose an interval-based model for timing specifications.

B. Basic Timing Entities

As described in the previous section, an *Interval* marks a certain contiguous period in time. Figure 4 depicts the basic entities of the interval-based time model together with their relationships: *Interval*s, Events, and *Time*s. The *Interval* object defines comparison methods that correspond to the timing patterns shown in Figure 1. The boundaries of any *Interval* are given by exactly two Events: a start-event and an end-event. Each event is associated with a *Time* object that implements the comparison operators $<$, $=$, $>$, $||$. When the timing specification is executed, these time objects trigger the Event.happened() method when a clock strikes the time.

In this model, three concrete types of time exist. A DependentTime binds the time of an event e_2 to the happening of a parent event e_1. Therefore, a DependentTime is struck in exact the same moment as the parent event e_1 happens (i.e., the time of e_1 is struck).

The second type of time object is the ScalarTime. The ScalarTime is specified as a single scalar value in respect to a certain *Clock* object. Whenever *Clock*.currentTime() equals the specified scalar, the time is struck.

One could imagine several types of such *Clock*s; examples are a common 'system time' clock, a 'number of currently logged in users' clock, or even more exotic clocks like a 'currently sold units' clock. It is possible to mix different types of clocks in the same model; the only requirement is that all instances of clocks can compute a single scalar when called at a certain point in (world-)time.

The IndefiniteTime class implements an indefinite time as introduced in Section III-A. The IndefiniteTime.earliest() and IndefiniteTime.latest() are the implementations of the corresponding functions defined in Section III-A. They traverse all intervals that start (respectively end) in the associated event for finding the earliest (or latest) possible time the IndefiniteTime can occur.

C. Modeling Timing Specifications

Figure 5 shows the basic entities of the Timing Specification model, with the *Interval* class described in the previous section being the central element. There exist three concrete types of intervals which carry different semantics: The lifetime of a resource, a simple sequence of two events, and the lifetime of a more complex timing condition following the timing patterns depicted in Figure 1.

Every Resource that is managed or that is used for management by the system, such as policies or users, is modeled as an interval. When the interval starts, a resource (instance) 'is born' and it 'dies' when its interval ends. This lifecycle affects resource *instances*

and not their meta-definition. For example, when an interval I starts or ends, it may cause a policy A (the resource) to become active or inactive; this can translate into 'an instance of policy A is born' (or 'has died' respectively). The metadata *describing* the policy is not affected (as long as the metadata are not managed by such a time model themselves, of course). Accordingly, the expiration of a user means that the user left the managed system and not that he has 'died' literally.

An EventSequence implements a simple interval between exactly two events mapping directly to one arrow of the timing patterns in Figure 1.

TimingConditions are complex temporal relationships between two intervals. Each timing condition is associated with a *TimingPattern* that specifies the basic structure of the condition following the patterns of Figure 1. Depending on the associated pattern, the TimingCondition manages one to three instances of EventSequence objects (corresponding to the arrows).

All TimingConditions have a well-defined lifetime and are therefore modeled as intervals themselves, i.e., they are associated with a start and an end event. The concrete start and end event of a timing condition depends on the applied timing pattern. When the arrows in Figure 1 are interpreted as messages that are sent between the intervals, the first sending event and the last receiving event denote the boundaries of the interval. For example, the boundaries of a timing constraint $r = i_1$ before i_2 between two intervals are defined as:

$$(endEvent(i_1), startEvent(i_2))$$

while the boundaries for a $r = i_1$ cobegin i_2 constraint would be:

$$(startEvent(i_1), startEvent(i_2))$$

Because TimingConstraints are modeled as intervals, recursive relationships allow the construction of higher order timing constraints.

IV. Implementation of the Timing Service

This section describes the architecture of a Timing Service (TS) that is capable of executing timing specifications. Section IV-A describes the overall architecture of the TS and Section IV-B outlines some possibilities how such a service can be integrated with existing management systems. Section IV-C describes our current implementation of the Timing Service for the Java platform.

A. Architecture

The architecture of the Timing Service is depicted in Figure 6. All subsystems of the service are built on top of the Runtime layer. The runtime layer provides an entry point to start, initialize, run, and shutdown the service. On startup, the runtime subsystem accesses a Repository that stores the timing specification and creates an initial runtime-representation of this specification. During the service is in use, the runtime layer monitors internal changes of all objects and propagates them into the repository so that they become persistent.

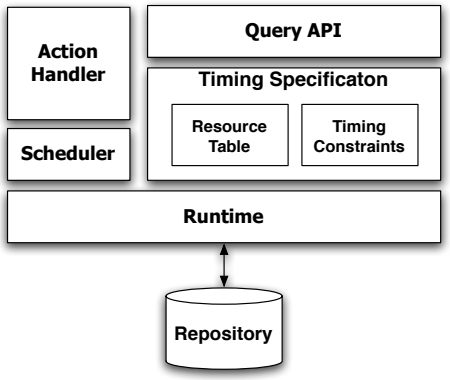

Fig. 6. Architecture of the Timing Service

The runtime-representations of the timing specification elements are located in the Timing Specification component. This component consists of a Resource Table subsystem, that manages all resources together with their events, as well as the Timing Constraints subsystem that contains all created timing constraints, namely the sequences, constraints, and applied timing patterns. Both subsystems allow creating, retrieving, updating, and deleting elements and assure the correctness of the overall model. A Query API provides access to the timing specification from outside the TS. Such a query API may be provided by the repository directly (e.g., SQL when a relational database is used, or RDQL when the data is stored in an RDF framework); other options are to implement the API as remote objects of the runtime timing specification objects (e.g., Java RMI objects) or by developing a dedicated Timing Service query protocol (TSQP).

The Timing Service includes a Scheduler that watches the event times and schedules occurring and upcoming events. It also provides a mechanism to plug-in different types of clocks that translate the world-time scalar into some clock-dependent scalar value. The scheduler offers a notification mechanism when any clock strikes a certain time. Whenever a time is struck, the scheduler searches for the affected event objects and notifies them. This notification will then cause the Action Handler component to trigger actions for adapting its environment such as an external PBM system.

Action handlers allow the Timing Service to actively communicate with its environment. For example, an action handler is notified that a certain interval has

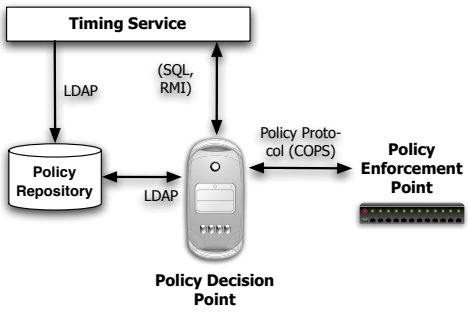

Fig. 7. Integration of the Timing Service with a Policy-Based Management System

become active (e.g., a policy should be activated). The handler then may connect to a Policy Decision Point and inform it about the changes. Through introducing an active component (the action handlers) to the Timing Service, not only *outsourcing* of timing information (i.e., a passive, queryable information-base) but also *provisioning* scenarios (active delivery of timing information) are supported.

B. Integration with Role-based Management Systems

Figure 7 illustrates how the TS can be integrated into a role-based management system. Following the reference model described in RFC 3198, a Policy-Based management system comprises the following modules:

- **A Policy Repository.** "A specific data store that holds policy rules, their conditions and actions, and related policy data. A database or directory would be an example of such a store." [15]
- **A Policy Decision Point (PDP).** Often called Policy Server. The PDP makes decisions for itself or for other network elements that request such a decision.
- **A Policy Enforcement Point (PEP).** A PEP is any logical entity that enforces a policy by applying the policy rules.

The integration of the Timing Service can be realized in several ways, which differ in how an existing management system is affected. In the simplest case, the Timing Service can be integrated without affecting the management system at all. In this scenario, the Timing Service would be equipped with action handlers that access the Policy Repository and directly change the time conditions of the affected policies.

While integrating the Timing Service by directly editing policies in the Policy Repository is feasible it is still a questionable approach because the Timing Service overwrites manual changes of the policies' time conditions. A better integration could be achieved when the action handlers can access the Policy Decision Points by some defined protocol or API. Then, action handlers could trigger events in the PDP which cause the PDP to recalculate its decision.

The tightest integration requires an adaption of the Policy Decision Point to make it aware of the Timing Service. In such cases, the PDP actively queries the TS and access the specification. While some current implementations may already offer mechanisms to plug in such extensions as the Timing Service (e.g., programmable filters that can query the timing component), other PDPs would need a change in source code.

C. Current Implementation

The current implementation of the TS is written in Java 1.4. Several interfaces and their concrete implementations realize the model described in Section III.

Each model class implements the observer pattern by providing a listener mechanism that can be used to get notified whenever the internal state of the object changes. In addition to the listener mechanism, the Event and Interval classes offer the possibility to add so-called VetoableChangeListeners. In contrast to common listeners, which are notified *after* the internal state has changed, VetoableChangeListeners are notified *before* such a change occurs. By throwing an exception, each of these vetoable change listeners has the possibility to prevent a value to be changed. This mechanism is used, for example, by intervals, which register themselves as VetoableChangeListeners at their start and end events to prevent inconsistent changes such as changing the time of the end event to a time before the start event time.

The Scheduler component is implemented as a thread that monitors the current system time and manages the clock implementations. Events register their associated times with the scheduler. Whenever a time is triggered, the scheduler informs the events. This notification may then cause action handlers that are associated with the event to execute. A scalar time is triggered when its associated clock switches to the scalar value. For the other time objects — dependent and indefinite times —, the scheduler keeps track of their state and triggers them as needed to assure the correct execution of the timing specification.

The repository of the timing specification is implemented on top of the Jena RDF framework [3] that is loaded by the Runtime class (the main class that is instantiated when the Timing Service is started). Since the Runtime class registers itself with every object in the specification, it can update the repository whenever changes occur; equally, it propagates changes in the repository to the specification objects.

In the current implementation, only relatively simple action handlers are implemented, mainly for testing purposes. These action handlers only perform local

actions, such as making output on the terminal or opening and closing windows (as representatives for resources such as applications) when certain intervals start and end. Even though the timing component has not yet been tested with a running Policy-Based management system, the timing mechanism works and is independent from the concrete implementation of the action handlers. Therefore, realizing one of the integration scenarios described in Section IV-B is mainly a matter of implementing the appropriate action handlers.

V. Concluding Remarks

This paper presents a time model that can be used for specifying timing constraints between resources. Specifications can be executed on a Timing Service for which we propose a model together with an architecture. Such a service can be integrated with existing PBM systems. The Timing Service watches the model and assures its validity. During execution the Timing Service monitors the environment for triggering events. Such events usually denote the start and/or the end of certain time intervals. Action handlers translate changes in the internal state of the model into changes of the state of a Policy Repository or even the Policy Decision Point directly.

An interesting application area for this architecture is the management of policy changes concerning authentication, authorization, and accounting of distributed resources in peer-to-peer networks, for example, 'granting access to a resource during a session' or 'publishing the document while the project is running'.

Since in peer-to-peer networks no central authority exists, and users are more responsible for their resources — compared to users in centrally managed networks — it is difficult to offer a comprehensive administration service in such environments. Therefore, users become local administrators of their own resources. Such a setup requires powerful but yet easy to understand management tools. Policy-Based management together with high-level Timing Services are promising tools to realize such distributed administration.

References

[1] E. Bertino, P.A. Bonatti, and E. Ferrari. *TRBAC: A temporal role-based access control model*. ACM Transactions on Information and System Security, 4(3), pages 191?233, 2001.

[2] D. F. Ferraiolo, D. R. Kuhn, and R. Chandramouli. *Role-Based Access Control*. Artech House Publishers, 2003.

[3] *Jena ? A Semantic Web Framework for Java*. available from http://jena.sourceforge.net/.

[4] S. Jha and M. Hassan. *Java implementation of Policy-Based bandwidth management* International Journal of Network Management, Vol. 13, pages 249?258, 2003.

[5] A. Kern, M. Kuhlmann, R. Kuropka, and A. Ruthert. *A meta model for authorisations in application security systems and their integration into RBAC administration*. In Proceedings of the 9th ACM symposium on Access control models and technologies, ACM Press, pages 87?96, 2004.

[6] P. King, P. Schmitz, and S. Thompson. *Behavioral Reactivity and Real Time Programming in XML: Functional Programming meets SMIL animation* ACM Symposium on Document Engineering, pages 57-66, 2004.

[7] K.L.E. Law and A. Saxena. *Scalable design of a Policy-Based management system and its performance*. IEEE Communications Magazine, Vol. 41, no. 6, pages 72-79, 2003.

[8] D. Lewis, K. Feeney, and V. Wade. *Policy Based Management for Internet Communities*. In Proceedings of IEEE 5th International Workshop on Policies for Distributed Systems and Networks (Policy 2004), IBM Thomas J Watson Research Center, New York, USA, pages 23-34, 2004.

[9] B. Moore, E. Ellesson, J. Strassner, and A. Westerinen. *Policy Core Information Model*. IETF, RFC 3060, 2001. available from http://www.ietf.org/rfc/rfc3060.txt?number=3060.

[10] B. Moore, Editor. *Policy Core Information Model Extensions. (PCIM)*. IETF, RFC 3460, 2003. available from http://www.ietf.org/rfc/rfc3460.txt?number=3460.

[11] R. Prakash and M. Singhal *Dependency sequences and hierarchical clocks: efficient alternatives to vector clocks for mobile computing systems* Wireless Networks archive, Vol. 3, no. 5, pages: 349 - 360, 1997

[12] T. Pyron. *Special Edition Using Microsoft Office Project 2003*. Que, Feb. 2004

[13] R. S. Sandhu, E. J. Coyne, H. L. Feinstein, and C. E. Youman. *Role-based access control models*. IEEE Computer, 29(2), pages 38?47, 1996.

[14] T. Wahl and K. Rothermel. *Representing time in multimedia systems*. In IEEE 1st. International Conference on Multimedia Computing and Systems, pages 538-543, 1994.

[15] A. Westerinen, J. Schnizlein, J. Strassner, et al. *Terminology for Policy-Based Management*. IETF, RFC 3198, 2001. available from http://www.ietf.org/rfc/rfc3198.txt?number=3198.

[16] M. Wright. *Using Policies for Effective Network Management*. International Journal of Network Management, Vol. 9, pages 118-125, 1999

[17] S. Wright, P. Lapiotis, and R. Chadha. *Policy-Based Networking*. IEEE Network, Vol. 16, no. 2, pages 8-9, 2002.