A Policy Management Framework for Content-based Publish/Subscribe Middleware

Alex Wun and Hans-Arno Jacobsen

University of Toronto
Toronto, Canada
{wun,jacobsen}@eecg.utoronto.ca

Abstract. Content-based Publish/Subscribe (CPS) is a powerful paradigm providing loosely-coupled, event-driven messaging services. Although the general CPS model is well-known, many features remain implementation specific because of different application requirements. Many of these requirements can be captured in policies that separate service semantics from system mechanisms, but no such policy framework currently exists in the CPS context. In this paper, we propose a novel policy model and framework for CPS systems that benefits from the scalability and expressiveness of existing CPS matching algorithms. In particular, we provide a reference implementation and several evaluation scenarios that demonstrate how our approach easily and dynamically enables features such as notification semantics, meta-events, security zoning, and CPS firewalls.

Key Words: Publish/Subscribe, Policy, Security, Configurability

1 Introduction

To date, many publish/subscribe (pub/sub) systems have been developed to provide loosely-coupled, event-driven messaging services [1–6]. In particular, the Content-based Publish/Subscribe (CPS) paradigm is designed to support flexible and dynamic enterprise applications by routing on message content rather than destination identities or explicit network routes. Although the general CPS model is well understood, many CPS feature details still remain non-standardized for the good reason that different application scenarios have different requirements. While some CPS features can be addressed with system reconfigurability [7, 8], others are more suitably expressed in policies that separate application requirements from infrastructure mechanisms [9, 10]. For example, advanced features such as notification semantics, meta-events, security zoning, and CPS firewalls are appropriate for being realized as policies. These kinds of novel CPS features depend on being able to dynamically change system behaviour and are achievable through the flexibility of policies. However, no such policy framework currently exists in the CPS context. To address this problem, we present a content-based policy framework that is scalable, expressive, and extensible. Our policy framework supports a novel approach that applies policies based on the results of content-based matching. We find that this approach

enables many unique CPS capabilities that would otherwise be difficult or costly to achieve. In particular, we present a novel *post-matching* policy model capable of achieving scalable and expressive CPS policies. We also present a reference implementation of our policy framework using the PADRES [1] CPS middleware platform and a number of evaluation scenarios to highlight several unique and novel CPS features that become possible with our approach.

We first overview related work in Sec. 2 before presenting the concepts for our policy framework in Sec. 3. Our implementation is presented in Sec. 4 and several scenarios used to evaluate our approach are presented in Sec. 5. Finally, we conclude and discuss future work in Sec. 6.

2 Related Work

While there has been little research to date on policies in the CPS context, we are aware of the following related work. Opyrchal *et al.* [11] address issues of publication privacy in the context of pervasive environments using a centralized policy engine. Our work is different from theirs in many respects since they focus specifically on providing access control on publications. In addition to being distributed, our policy framework does not specifically target access control policies but also general feature and service policies such as notification semantics. Belokosztolszki *et al.* [12] incorporate Role-Based Access Control (RBAC) into the Hermes pub/sub system [3]. They address issues of policy management, broker trust, and access control optimization. Our work represents a different approach to pub/sub policies that targets issues orthogonal to RBAC in unstructured rather than structured overlays. Sturman *et al.* [13] propose a pub/sub architecture capable of message transformations. Our focus is not on the transformations themselves, but a framework that can support specifying policies on when and how to perform transformations among other features. In general, we are introducing a policy model that has significant expressiveness benefits complementing existing work.

Reconfigurable pub/sub systems allow the customization of middleware to suit the needs of different applications. Cugola and Picco [7] address issues of overlay and routing configurability by implementing a modular system architecture customizable at deployment time. Sivaharan *et al.* [8] present a component-based framework that allows pub/sub systems to easily cope with the diversity of mobile and heterogeneous network environments. Both are flexible systems that can be reconfigured with different pub/sub semantics as necessary. Our work is complementary because it addresses a different problem of separating system policies from mechanism, allowing applications to specify how a configured and running system should provide its services based on message content. Indeed, a benefit of our approach is that the main framework can be implemented in well-componentized, interceptor-based, or aspect-oriented system architectures without too much difficulty.

[1] http://padres.msrg.utoronto.ca (extended version of paper also available)

In the domain of traditional network environments, there is already a significant amount of work on policies addressing various issues from Quality of Service (QoS) to network management and security [14–16]. Stone *et al.* [17] present a survey of existing network policy languages and also propose their own Path-based Policy Language. Their approach explicitly declares the nodes in a network path to which policies are applicable. This approach is clearly not suitable in the CPS domain since it fundamentally conflicts with the paradigm of decoupling clients from message routing details. Agrawal *et al.* [18] present a policy-based system for autonomic management of computing resources. However, their work is again applicable in a different domain. The WS-Policy framework [19] focuses on providing an extensible syntax to express policies between Web service end-points. However, not only is our focus on developing an actual policy mechanism rather than a syntax for expressing policies, the distributed CPS domain also has many concerns not addressed by end-point interactions such as routing. Existing policy frameworks for traditional network environments generally do not migrate easily into the CPS domain.

3 Content-based Policy Framework

In this section, we introduce the main concepts of our content-based policy framework and discuss the implications of our approach with respect to policy composition and application in a distributed CPS system.

3.1 The Post-Matching Policy Model

Since content-based matching algorithms are an integral part of CPS systems, the natural intuition is to protect these systems by enforcing policies before messages reach the matching algorithm. Although our policy framework supports enforcing policies before matching, such an approach does not easily achieve content-based expressiveness without duplicating the functionality of matching algorithms and incurring additional overhead. CPS systems generally provide highly scalable and expressive message filtering capabilities already [1, 2, 20]. By leveraging the high-performance matching algorithms that already exist, it is possible to build a policy framework that achieves the same scalability and expressiveness as the host CPS system itself. The basic concept behind our policy framework is summarized in Alg. 1 using an event-condition-action policy model [21].

when content-based match occurs
  if *additional policy condition(s) satisfied* **then**
    perform
      Action_1;
      ...;
      Action_n;

Algorithm 1: Post-matching policy model

In this model, a *content-based match event* serves as the trigger for policy *application*, which involves evaluating policy conditions and executing policy actions. Hence, we refer to this semantic as the *post-matching policy model*. While the model itself is deceptively simple, it enables a powerful policy framework since any application context that surfaces as message content is also reflected in the policy framework. Note that since we only depend on the notion of a content-based match event, this model is applicable to any CPS system that performs matching at the message granularity[2]. The remainder of this paper focuses on the post-matching policy model even though we also support enforcing policies before matching in our framework.

3.2 Policy Framework Approach

More formally, our approach associates each filter F (advertisement or subscription) with an optional *policy statement* T[3], which contains one or more *policy rules*. Policy rules specify the conditions to evaluate and actions to execute when the policy is applied. When a message M is processed by a CPS broker, the matching algorithm computes a set $\Phi = \{(F_1, T_1), (F_2, T_2) \dots (F_n, T_n)\}$ of matching filters F_i and their associated policy statements T_i containing policy rules applicable to M. Applying the policies $T_1 \dots T_n$ against M involve evaluating the conditions and executing the actions specified in the policy rules of each policy statement. The result of applying the policies could include the rejection of M for routing, transformations on the format or content of M, or the triggering of other actions such as broker state maintenance and debugging. Essentially, our policy framework extends the CPS paradigm by giving applications the ability to specify policies intercepting content-based match events. In Sec. 5, we present example scenarios to highlight the benefits of the post-matching model and this approach.

It is important to note that the computation of Φ does not require any additional processing beyond what is already performed by the existing matching algorithm. If M is a publication, then Φ contains matching advertisements and subscriptions as computed by the matching algorithm. If M is a subscription, then Φ contains matching advertisements. For example, suppose a client issues two advertisements $A_1 = [(x < 100), (y < 50)]$ and $A_2 = [(x > 75), (y < 100)]$ to its local broker. Policies T_{a1} and T_{a2} are associated with advertisements A_1 and A_2, respectively. When the broker receives a subscription $S_1 = [(x > 25), (y < 75)]$ that intersects with both A_1 and A_2, the application of both policies T_{a1} and T_{a2} against S_1 is triggered. In contrast, a subscription $S_2 = [(x < 100), (y > 75)]$ would only trigger application of policy T_{a2} because the subscription only intersects with advertisement A_2. Suppose there is a further policy T_{s1} associated with S_1. Then a publication $P_1 = [(x, 90), (y, 30)]$ would trigger application of all three policies T_{a1}, T_{a2}, and T_{s1} against P_1. In contrast, a publication

[2] This excludes matching algorithms that compress message sets into bit vectors, for instance, but includes topic-based approaches.

[3] From here on, we will use the terms "policy statement" and "policy" synonymously.

$P_2 = [(x, 30), (y, 30)]$ only triggers application of policies T_{a1} and T_{s1}. In this way, the content-based expressiveness of the hosting CPS system is reflected in the policy framework.

Although we have discussed our approach in the context of advertisement-based semantics, these concepts are equally applicable in the context of subscription-based semantics.

3.3 Implications for Policy Composition

Deploying an application in a CPS system involves the decomposition of application contexts into messages. In this process, the application developer thinks in terms of event schemas[4] and event spaces[5]. Consequently, it is natural for an application developer to compose an overall policy by designing policies around the event schemas and event spaces that make up the application. By associating policies with filters, we implicitly achieve policy composition [22] with content-based expressiveness. For example, consider a supply-chain scenario where inventory report publications $P_i = [(\textit{class,report}), (d_1, x_{i1}) \ldots, (d_n, x_{in})]$ with many data attributes are issued regularly. A management application subscribing to reports may consider d_1 to be a critical attribute. As such, if the value x_{i1} of that attribute is above a certain threshold, then the client would like to know the identity of the previous overlay hop of the message for tracking purposes. On the other hand, if x_{i1} is below a certain value, then the remaining attributes are uninteresting so the client would like the broker to remove them before delivering the notification. To achieve this, the management application can issue two subscriptions $S_1 = [(\textit{class=report}), (d_1 > X_{\text{high}})]$ and $S_2 = [(\textit{class=report}), (d_1 < X_{\text{low}})]$ with policies T_1 and T_2 associated with each, respectively. Policy $T_1 =$ *AppendPrevHop()* specifies a single action that appends the attribute (*PrevHop ID*) to the notification while policy $T_2 =$ *RemoveAttributes*$(d_2, \ldots, d_n)$ specifies a single action that removes the given list of attributes from the notification. With these policies in place, notifications delivered to the management application may now have an extra *PrevHop* attribute, missing $d_2 \ldots d_n$ attributes, or both depending on the value of the d_1 attribute. Furthermore, the management application is able to specify this notification policy without affecting other clients subscribing to the same events since the policies are only associated with subscriptions belonging to the management application clients. In this example, policies T_1 and T_2 have been composed together to specify a notification semantic for inventory reports by leveraging the content-based filtering capabilities of subscriptions, which already exist as a fundamental concept in CPS systems. No additional policy-specific composition language or content-based processing is needed in our approach. In contrast, a generic policy framework layered on top of the CPS system would need to explicitly process the contents of publications to achieve the same result.

[4] Advertisements or message type definitions.
[5] The set of all possible messages matching a filter.

3.4 Interception Points in CPS Overlays

The CPS policy concepts we have presented so far are equally applicable in both centralized and distributed CPS systems [1, 2, 4, 20]. In particular, we have addressed *when* policy application occurs – either before or immediately after a content-based match event. For distributed CPS systems however, it is equally important to address *where* in the overlay policy application occurs. For instance, are policies only applied at edge brokers? Or are they applied at every overlay hop? Since there are valid scenarios for either case, our approach lets applications specify where policy application occurs based on *interception points*. The three important interception points are ingress, egress, and routing, which correspond to the brokers at which a message enters, leaves, and routes through an overlay. Fig. 1 illustrates the concept of interception points. Note that for a single isolated overlay, ingress and egress points correspond to the brokers at which injection and notification occurs between brokers and clients. However, in a federated CPS system, ingress and egress points correspond to the brokers at the edges of sub-overlays.

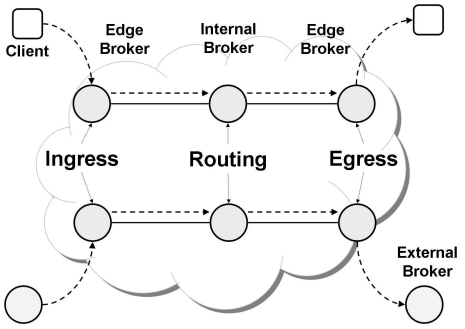

Fig. 1. Policy interception points

4 Policy Framework Implementation

In this section, we present the implementation of our policy framework, which builds upon the model and approach described earlier. In particular, we describe the mechanisms for creating, distributing, and enforcing content-based policies in a distributed CPS system. Our framework is built on top of PADRES [5], an existing rule-based CPS middleware platform implemented by our research group in Java.

4.1 API and Language

Only minor changes to the API are needed to support our policy framework. The `subscribe(msg)`, `advertise(msg)`, and `publish(msg)` methods previously used by clients have simply been extended to accept an optional policy statement argument, resulting in `subscribe(msg, policy)`, `advertise(msg, policy)`, and `publish(msg, policy)` as the new API. For advertisements and subscriptions, `setPolicy(msgID, policy)` also allows for specifying policies after the message has already been issued. There is a factory class that can create commonly used policy statement objects directly, but it is also possible to build a policy statement from either XML specifications or a more compact language shown

in Fig. 2. However, our focus in this paper is on developing the policy framework mechanisms rather than providing a specific syntax for writing policies. This language represents the construction of a single self-contained policy statement. Each policy statement contains one or more policy rules enclosed by the `On(...)` keyword and two mandatory parameters that define policy rule types.

```
1   PolicyStatement {
2     On (
3       [Forward | Insert],
4       [Advertisement | Subscription
5        | Publication | Unsubscription
6        | Unadvertisement]) {
7
8       @matching: [Before | After]?
9       @broker: [Ingress | Egress | Routing]*
10      @attach: [Never | Always | KeepExisting
11                       | IfYield]?
12      @yield_attach: [False | True]?
13
14      If <conditions ...> Then <actions ...>
15      Elseif <conditions ...> Then <actions ...>
16      ...
17
18      OnException {
19        If <conditions ...> Then <actions ...>
20        Elseif ...
21      } } }
```

Fig. 2. Policy language

The parameter choices on line **3** specify whether the rule is applicable to messages being forwarded or to messages being inserted into broker routing tables. The parameter choices on lines **4** to **6** specify which type of message the policy rule is applicable to. Lines **8** to **12** show optional qualifiers that further define when and where the policy is applicable. When the rule is applied, the conditions specified on line **14** are evaluated and the actions are executed if the conditions all return true. Subsequent condition clauses are only evaluated if the preceding condition clause fails. Line **18** encloses conditions to evaluate and actions to attempt if an exception occurs when applying the rule. In the following sections, we discuss how this is used to specify policies and control how they are applied.

4.2 Creation and Distribution of Policies

Using the new API, both clients and brokers can create policies either when CPS messages are first issued or by associating policies with filters (advertisements and subscriptions) at any time afterwards. For instance, `advertise(msg, policy)` attaches a policy to the advertisement when it is issued. The attached policy is routed along with the advertisement and stored by brokers, who associate the policy with the advertisement. Similarly, `subscribe(msg, policy)` attaches a policy to the subscription that is routed through the overlay and stored by brokers. The policies stored by brokers can also be set using `setPolicy(msgID, policy)`, which updates the policy associated with either an advertisement or subscription. In general, a policy that routes with a message in the overlay is said to be *attached*, while a policy that is stored by a broker and linked to a filter is said to be *associated*. Policies can be attached to any CPS message type but can only be associated with either advertisements or subscriptions.

Table 1 summarizes the available methods for specifying policies applicable to each message type. For example, publication policies (i.e., policies applied to publications) can either be specified by policies associated with advertisements and subscriptions using the `On(Publication)` qualifier or attached to the

publication itself, while unsubscription policies can only either be specified by policies associated with subscriptions using the `On(Unsubscription)` qualifier or attached to the unsubscription itself. Similarly, advertisement policies can only be created and attached to the advertisements they are to be applied to. However, subscription policies can either be attached directly to the subscription or associated with advertisements as `On(Subscription)` policy rules. In the latter case, the `@attach` and `@yield_attach` qualifiers can additionally be used to allow subscriptions to inherit the policy from the advertisement. That is, the subscription policy associated with the advertisement can be attached to the subscription rather than applied normally. These additional qualifiers allow greater control over the specification of default policy attachments.

Table 1. Specification methods for policies.

Policy \ Message	Adv.	Sub.	Pub.	Unadv.	Unsub.
Adv.	Attached	×	×	×	×
Sub.	On(*)	Attached	×	×	×
Pub.	On(*)	On(*)	Attached	×	×
Unadv.	On(*)	×	×	Attached	×
Unsub.	×	On(*)	×	×	Attached

4.3 Enforcing Applicable Policies

Brokers are solely responsible for interpreting and enforcing the policies applicable to messages they receive. When a policy is enforced by evaluating conditions or executing actions, we say that the policy is being *applied* to a message. In general, if a broker receives a message M with a policy T_M attached to it and M matches a set of filters $\{F_1, \cdots F_n\}$ associated with a set of policies $\{T_1, \cdots, T_n\}$, then the set $\{T_M, T_1, \cdots, T_n\}$ contains all policies potentially applicable to M. However, the applicability of a policy rule to any given message depends on a combination of the policy rule type and the policy rule qualifiers[6]. For instance, a publication matching an advertisement-associated policy that contains only subscription rules will not have any of those rules applied to it. Two qualifiers are currently supported to further specify *when* and *where* a policy rule is applicable.

The `@match` qualifier specifies whether the rule is applied before or after the message goes through content-based matching. Rule application before matching is supported since some policies may require checking conditions or executing actions before accepting the message for matching. Policies for fast message forwarding that bypass matching altogether or content-independent authorizations are more appropriate for application before matching, for example. However, such policies do not benefit from the advantages of scalability and expressiveness that are possible with rules applied after matching. More specifically, a powerful implication of evaluating rules after matching is that the rules are selectively applied based on message content. We focus on exploring the benefits of post-matching policy rules with our evaluation scenarios in Sec. 5.

[6] From here on, we will use the terms "policy rule" and "rule" synonymously.

The `@broker` qualifier specifies the broker overlay contexts where the rule is applicable and can be any combination of *ingress*, *egress*, and *routing* as shown in Fig. 1 and discussed in Sec. 3. Ingress rules are evaluated for messages entering the CPS system, egress rules are evaluated for messages leaving the CPS system, and routing rules are evaluated for messages at internal brokers.

Together, the `@match` and `@broker` qualifiers give applications significant flexibility in specifying when and where policy rules are applicable.

4.4 Framework Extensibility with Modular Rule Elements

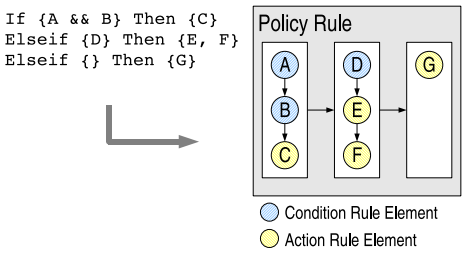

Fig. 3. Policy rule structure

The level of functionality achievable in our framework depends on the conditions and actions supported inside policy rules. In our framework, all conditions and actions are implemented as *rule elements* chained together inside policy rules. Every policy rule contains one or more rule element chains. Applying a policy rule essentially involves traversing its rule element chains, evaluating and executing the corresponding conditions and actions as appropriate. Fig. 3 shows an example of how conditions and actions are represented as a policy rule. In this example, the rule elements A, B, C are accessed in order first. Should the conditions corresponding to either rule elements A or B fail for instance, then the next chain consisting of rule elements D, E, F is accessed. Recall that a single policy statement may also contain multiple policy rules, one for each type of message at each interception point. If an exception occurs while traversing the rule elements, compensation policy conditions and actions as specified in the `OnException` clause shown in Fig. 2 are accessed. Further exceptions during compensation actions are no longer handled by the policy framework itself and instead, a meta-event (as presented in Sec. 5) that describes the exception is generated.

Although we have already implemented a number of rule elements presented in Sec. 5 that cover a wide range of CPS functionality, our framework is designed to be easily extensible with new rule elements in response to emerging application requirements.

5 Evaluation Scenarios

In this section, we evaluate our policy framework by applying it to a number of different scenarios, demonstrating the expressiveness and flexibility achieved using the language presented in Sec. 4. Several of these scenarios represent novel CPS features that become easy to specify and implement using our policy framework. Where appropriate, we also present experimental data resulting from the

implementation of these scenarios. We do not include any experimental data for scenarios that are purely functional and instead present only the associated policies. Since we focus on the post-matching model, all policy statements shown are implicitly qualified with `@matching: After` to avoid repetition in presentation. All experiments presented in this section were run using separate Intel Dual Xeon 3.xGHz processor, 2GB memory systems for each broker or client. We divide our evaluation into two broad scenario categories: CPS semantics and security.

5.1 Specifying CPS Semantic Policies

Since there has been no standardization of CPS implementations, there are still many subtle operational semantics that are open to interpretation by implementers. As such, it is useful to have a flexible CPS system that allows customization of operational semantics according to the needs of applications. The following examples highlight how we can dynamically tune system semantics using policies.

Notification Semantics Although the semantic of delivering notifications only to interested subscribers is well-established [1–4, 6], the actual content delivered in notifications typically remains an implementation decision. However, different applications may want notifications delivered to them in different forms. Suppose there is a stream of publications of the form $P_i = [(class,event), (a_1, v_{i1}), \ldots , (a_n, v_{in})]$. A subscriber issuing a subscription $S = [(class{=}event), (a_1 > x_1), \ldots , (a_k > x_k)]$ can optionally associate the policy in Fig. 4 with S. This policy specifies that just before notifications are delivered to the subscriber (at `Egress` brokers), they are "trimmed" to match attributes in the subscription. In this example, attributes $a_{k+1} \ldots a_n$ would be removed from all P_i since they do not appear in the subscription. The `TrimAttributes()` action automatically selects attributes for removal based on the subscription, but other possible notification semantic actions include `RemoveAttributes(attributeList)` and `KeepOnlyAttributes(attributeList)`, which allow a subscriber to remove or keep a specified list of attributes, respectively. Although we expect some performance improvement from removing unnecessary attributes, it is not immediately obvious exactly how much improvement can be achieved because of other factors such as message header overhead.

```
PolicyStatement {
  On(Forward,Publication) {
      @broker: Egress
      If {} Then {TrimAttributes()}
} }
```

Fig. 4. Notification policy

Fig. 5 shows that the effects of trimming attributes on network traffic are still very significant in our system despite message header overheads. We used three different streams of publications consisting of 10, 20, and 30 attribute publications. A subscription was associated with policies for removing from 0 to all attributes. The solid lines for each stream show the network usage of receiving full publications and the dashed lines show the network usage of delivering the

same publications if notification policies are applied. The values shown are averages over 100 publications. Clearly, even removing a small number of unwanted attributes could mean substantial overall network performance improvements when delivering to large numbers of subscribers. Since the infrastructure cannot always predict application workloads, our framework allows applications to help optimize performance by specifying exactly which attributes are relevant and should be delivered.

In addition to improving network performance, notification policies have functional benefits as well. Transformation from one syntax to another is also easily expressed using the same policy by using the appropriate action such as `ToXML()`. The actions can of course be stacked as well to compose more complicated notification policies such as

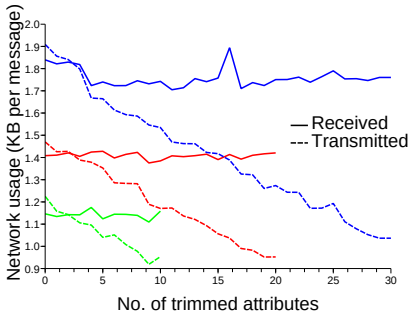

Fig. 5. Trimming notifications

```
Then {TrimAttributes(), ToXML()}
```

that both trims the notification and then converts it to XML syntax. For example, such transformation policies can be used to create proxy brokers between different CPS infrastructures that require different message formats by using the `@broker: Routing` qualifier and specifying appropriate conditions and actions such as

```
If {AuthenticateReceiver(Domain1)} Then {ToFormat1()}
Elseif {AuthenticateReceiver(Domain2)} Then {ToFormat2()}
```

The important point is that different subscribers may specify different notification policies, thereby receiving different versions of the same event. Note also that no condition has been specified in the policy we show here, but it is easy to imagine how notification semantics can be combined with conditions such as authentication to achieve access control.

Distributed Tracing Although keeping the infrastructure transparent to clients is an important CPS feature, applications sometimes need information about the infrastructure for monitoring or debugging purposes. Policies are well-suited for specifying this type of message content augmentation on an as-needed basis for applications. For example, consider the policy in Fig. 6.

This policy specifies that at every broker hop, publications are augmented with information about the broker, the load state of the broker, and the total time spent processing the publication. When attached to publications, this policy is applied on a per-publication basis and does not affect other publications that do not have the policy attached. Consequently, this is most useful if tracing is only needed occasionally. Alternatively, the policy can be automatically

attached to publications by adding the `@attach: Always` qualifier and associating the policy with an appropriate advertisement, which would give tracing information to all subscribers by default. Associating the policy with subscriptions would instead allow the augmentation to occur on a per-subscriber basis. The `@broker` qualifier and `If{}` conditions can of course also be changed to restrict augmentation to certain brokers. The unique combination of content-based expressiveness, policy language, and policy framework gives applications great flexibility in choosing a suitable tracing semantic.

```
PolicyStatement {
  On(Forward,Publication) {
    @broker: Ingress,Egress,Routing

    If {}
    Then {AugmentBrokerHostInfo(),
          AugmentBrokerLoadIndex(),
          AugmentProcessingTime()}
} }
```

Fig. 6. Tracing policy

The actions shown here place augmented data into a binary payload that is part of the publication, but similar alternative actions can instead extend the publication by placing augmented data into reserved CPS attributes. The second method would allow subscribers to further specify notification semantics on tracing attributes even when advertisement-associated or publication-attached tracing policies are used.

Meta-Events and Triggers Sometimes events in the CPS system itself can be of interest to clients and brokers. As such, our policy framework enables generating publications based on system events such as matches occurring under certain conditions. For example, consider the policy in Fig. 7. This policy

```
PolicyStatement {
  On(Insert,Subscription) {
    @broker: Ingress

    If {MessageSizeIndex() > 0.8
       && BrokerLoadIndex() > 0.75}
    Then {UninsertMessage(),
      Publish("[class,DropMessage]
              ,[cause,'Broker load']
              ,[message,$Message]")}
  }

  On(Forward,Subscription) {
    @broker: Ingress

    If {MessageSizeIndex() > 0.8
      && BrokerLoadIndex() > 0.75}
    Then {BlockMessage()}
} }
```

Fig. 7. Meta-event policy

specifies that if a large subscription is injected at a time when the broker is sufficiently loaded, then the subscription is not stored in routing tables (uninserted using the `UninsertMessage()` action) and also prevented from propagating any further (blocked using the `BlockMessage()` action). Furthermore, a publication is internally generated by the policy framework regarding this event using the `Publish()` action. The variable `$Message` inserts the offending subscription as a string into the generated publication content. Internally generated publications are processed by the same broker that generated it and treated as a normal publication for matching and routing purposes. Effectively, this policy specifies a simple load resilience scheme where interested subscribers are notified of dropped subscriptions. The dropped subscription event may be relevant to applications for recovery purposes or to system management services for resource provisioning.

Note that this policy does not necessarily affect all subscriptions since the application can choose which subscriptions are potentially dropped by associating

this policy with the appropriate advertisements. For instance, associating this policy with the advertisement $A = [(class = CustomerOrder), (priority < 5)]$ specifies that only subscriptions to low priority customer orders will be dropped and all other subscriptions will be unaffected by the policy. This kind of policy is not possible in normal pre-matching policies or generic policy framework layers without duplicating content-based functionality.

Flooding Semantics In terms of routing efficiency, there are some situations in which flooding subscriptions may be preferable to flooding advertisements. This can be the case if a particular application consists of many publishers and only a few subscribers interested in content from all publishers or if publishers are highly mobile while subscribers are mostly stationary. Our CPS system is based on advertisement flooding by default, but preference for subscription flooding can be specified using the policy in Fig. 8.

```
PolicyStatement {
  On(Forward,Advertisement) {
      @broker: Ingress,Routing

      If {} Then {BlockMessage()}
    }
  On(Forward,Subscription) {
      @broker: Ingress,Routing

      If {} Then {FloodMessage()}
} }
```

Fig. 8. Flooding policy

When attached to advertisements, this policy prevents the advertisement from propagating beyond a single broker hop using `BlockMessage()`. Furthermore, any subscription that matches an advertisement associated with this policy will be tagged for flooding to all neighbours. A broker can control which event schemas are flooded by internally generating an appropriate advertisement and associating this policy with it. For example, a group of brokers can agree to flood infrastructure management subscriptions by internally storing the advertisement $A = [(class = BrokerManagement), \cdots]$ in each of their own routing tables associated with the above policy. Notice that the enforcement of subscription flooding is left up to the discretion of brokers and does not occur at brokers that do not similarly store this policy.

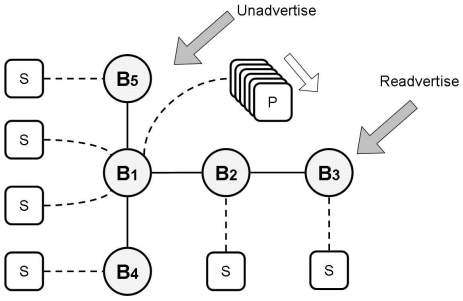

Fig. 9. Highly mobile publishers

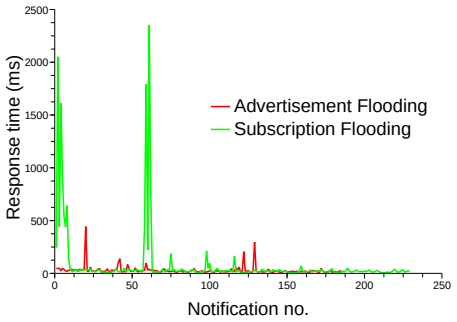

Fig. 10. Negligible policy overhead

We set up the scenario shown in Fig. 9 where subscribers are situated at different brokers and remain stationary while publishers move from broker to broker frequently in between issuing publications. This scenario reflects characteristics found in applications where mobile clients need to continuously send location and status updates to home servers, for instance. Fig. 11 shows that under

the normal advertisement flooding scheme, advertisement, unadvertisement, and subscription messages are continuously routed throughout the network as the application runs. Subscriptions are routed as a result of the unadvertisements/re-advertisement process, which triggers removal and re-propagation of subscriptions. However, if a subscription flooding policy is used, then no additional advertisement or subscription messages need to be routed while the application runs since subscribers remain stationary. Fig. 12 shows that significant network traffic is saved by using subscription flooding. Furthermore, the subscription flooding policy only incurs overhead when the advertisements are initially issued and subscriptions are flooded. Subsequent notification response times are unaffected as Fig. 10 shows.

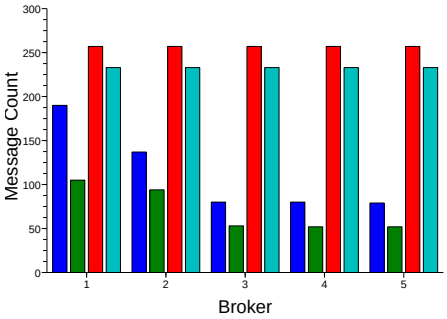

Fig. 11. Normal advertisement flooding

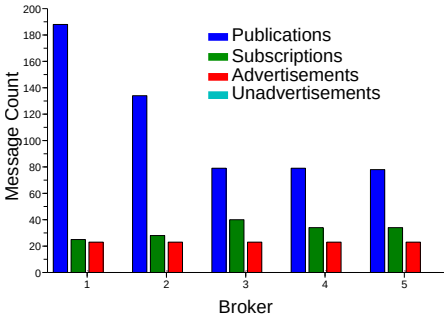

Fig. 12. Subscription flooding policy

Of course, there are reverse scenarios (such as subscriber mobility) that favour advertisement flooding instead. However, our purpose is only to show that different application scenarios can benefit significantly from different flooding semantics. With our policy framework, both semantics can be active simultaneously and specified on a per event schema basis.

5.2 Specifying Security Policies

Although security mechanisms are typically orthogonal to the policy framework, security behaviours can still be specified at the CPS level. We implemented a simple security mechanism for use with our policy framework in which authentication and encryption is based on *Trust Group* membership. Trust groups are conceptually similar to secure multicast groups [23]. Each trust group is associated with a shared group secret K_g so that members of the same group are able to perform authentication and encryption within the group. To establish K_g, there must be an out-of-band bootstrapping process to either set up K_g directly or set up public/private keys on the appropriate clients and brokers so that K_g can be exchanged securely. We support both bootstrapping methods since the first has the advantage of simplicity and low overhead while the second method is more flexible.

Authenticated Event Scope Although advertisements are normally flooded in our CPS system, trust group authentication can be used to limit the visibility

of events in the overlay on a per schema basis by issuing advertisements attached with the policy in Fig. 13. This policy specifies that the advertisement must only be sent to brokers belonging to either the `TrustGroup1` or `TrustGroup2` trust groups. If the receiver of the advertisement is successfully authenticated, the advertisement is sent normally and no additional special actions are performed. However, if authentication fails for both groups, the delivery of the advertisement is blocked by the `BlockMessage()` action. Alternatively, the condition

```
If {AuthenticateReceiver(TrustGroup1)
  && AuthenticateReceiver(TrustGroup2)}
```

can be used to specify that only brokers belonging to both trust groups will receive the advertisement. Although authentication is currently based on trust group membership, the same policies can be used to express authentication based on other mechanisms such as public key identities or Role-Based Access Control [12] since the actual authentication process uses out-of-band mechanisms.

```
PolicyStatement {
  On(Forward,Advertisement) {
    @broker: Ingress,Routing

    If {AuthenticateReceiver(
       TrustGroup1)} Then {}
    Elseif {AuthenticateReceiver(
       TrustGroup2)} Then {}
    Elseif {} Then {BlockMessage()}
} }
```

Fig. 13. Authentication

Fig. 14 shows publication processing time when a sender-authentication policy is in place between two brokers (the policy is associated with a subscription). Each step in the plot represents 0, 1, 3, and 5 different trust group authentications required by the policy. For the "Authorization" line, the receiving broker is able to authenticate the sending broker for all five trust groups. Since authentication results are not cached, the authentication protocol must run for every publication, resulting in worst case performance that is proportional to the number of trust groups specified in the policy. For the "Denial" line, the sending broker belongs to no trust groups so that authentication fails on the first attempt regardless of how many trust groups are specified in the policy. However, by caching authentication results, we can avoid running the authentication protocol for every message at the expense of lower responsiveness to trust group membership changes. The "Cached" line shows that since cached entries do not expire simultaneously, performance remains acceptable even when several groups are specified in the policy. Therefore, incurred overhead is due to the authentication process itself rather than processing and management performed by the policy framework. Note that we set the expiry time to a low value here in order to observe the effects of authentication cache expiry. Since our focus is on the policy framework and not the authentication mechanism itself, we implemented a protocol similar to CHAP [24] for the purposes of this evaluation. Without the post-matching model, the policy framework would have to duplicate content-based functionality to achieve expressive, fine-grained authentication policies based on content.

Security Zones Suppose a broker network is divided into restricted, controlled, and uncontrolled security zones as shown in Fig. 15. This setup is not uncommon

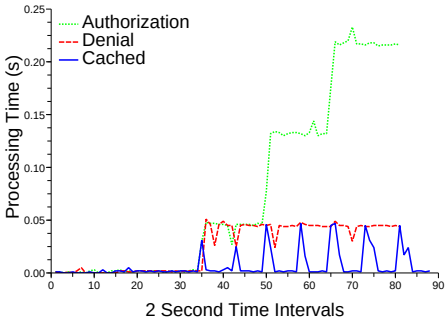

Fig. 14. Authentication processing time

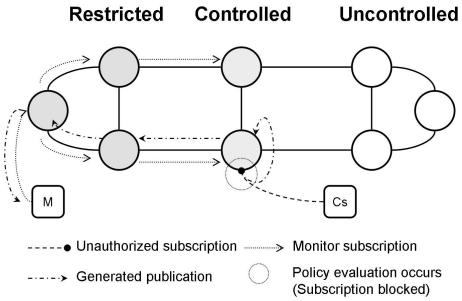

Fig. 15. Monitoring subscription attempts

in organizations separating their intranet (restricted) systems from the Internet (uncontrolled) using a demilitarized zone (DMZ, controlled). To enforce privacy, all attributes may be visible within the restricted zone but some attributes must not appear in the controlled zone. No events from the application should be visible at all in the uncontrolled zone. Furthermore, only authorized clients may subscribe from either zone. These application requirements can be expressed by attaching the policy in Fig. 16 to an advertisement issued from within the restricted zone.

```
PolicyStatement {
  On(Forward,Advertisement) {
    @broker: Ingress,Routing

    If {AuthenticateReceiver(Restricted)}
      Then {}
    Elseif {AuthenticateReceiver(Controlled)}
      Then {RemoveAttributes(a, ... ,n)}
    Elseif {} Then {BlockMessage()}
  }
  On(Forward,Subscription) {
    @broker: Ingress

    If {AuthenticateSender(AuthorizedSubscribers)}
      Then {}
    Elseif {}
      Then {Publish("[class,UnauthorizedSubscribe],
                    [message,$Message]")}
  }
```

```
  On(Forward,Publication) {
    @broker: Routing (Ingress)
    @attach: Always  // Routing only

    If {AuthenticateReceiver(Restricted)} Then {}
    Elseif {AuthenticateReceiver(Controlled)}
    Then {RemoveAttributes(a, ... ,n)}
    Elseif {} Then {BlockMessage()} } }
```

Fig. 16. Security Zones policy

This policy combines the use of authentication, message transformations, and meta-events to enforce privacy across different security zones. Fig. 15 illustrates the resulting meta-event message flow for an event schema using this policy.

Content-based Firewall In CPS systems, subscriptions are analogous to firewall "allow" rules on publications while advertisements are analogous to "allow" rules on subscriptions. In this respect, the existing filtering capabilities of CPS systems already provide some firewall functionality. However, consider a stable application in which advertisements have been established and no longer need to change. Subscriptions originating from an "internal" overlay are sent to a neighbouring "external" overlay and attract publications. In order to temporarily

block certain publications from entering the internal overlay, the subscriptions used by the application must change. For instance, this may be necessary as a reaction to detecting fraudulent publications that suddenly need filtering. Not only would such a change affect subscriptions throughout both overlays, the resulting subscriptions could potentially become a cumbersome mix of filters for attracting wanted publications and filters for fine-grained blocking of unwanted publications. Depending on the subscription language, this could be very difficult or even impossible to express in a single subscription. Similarly, preventing certain subscriptions from exiting the internal overlay would require changing the advertisements that originated from the external overlay. The same issue of expressing "allow" and "deny" filters in a single advertisement exists.

To block publications from entering the internal overlay, we can issue subscriptions from an internal firewall broker B_{if} to an external firewall broker B_{ef} as shown in Fig. 17 with the policy in Fig. **??** attached. This policy blocks forwarding of all publications *strictly* matching the subscription as determined by the `StrictMatch()` condition. A publication strictly matches a subscription if the publication contains exactly the same attributes as the subscription, while a subscription strictly matches an advertisement if their filters are the same. For example, a subscription $S = [(class = C), (a < 10)]$ is strictly matched by the publication $P_1 = [(class, C), (a, 9)]$ but not $P_2 = [(class, C), (a, 9), (b, 5)]$ even though P_2 normally matches S. Strict matching conditions can be used to achieve content-based firewall rules with more precision if needed but are not required in situations where the normal matching semantic is sufficient. The subscription and its associated policy is analogous to a single content-based firewall rule on publications. Note that authenticated event scoping is used to restrict firewall subscriptions and advertisements to the firewall brokers. Similarly, subscriptions are blocked by issuing an advertisement from B_{ef} to B_{if} and attaching the same policy using `On(Subscription)` instead of `On(Publication)`. In Figs. 18 and

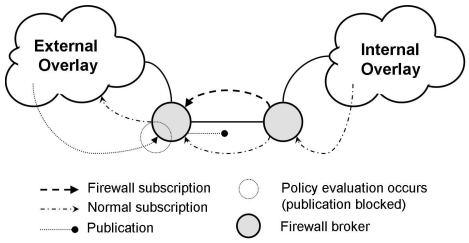

Fig. 17. Content-based firewall setup

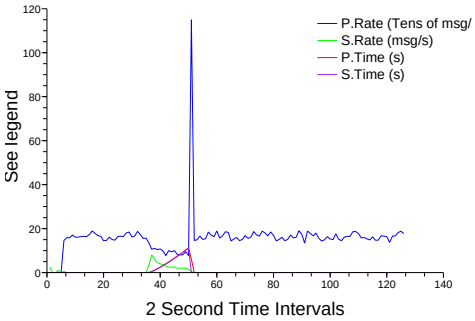

Fig. 18. Individual firewall subscriptions

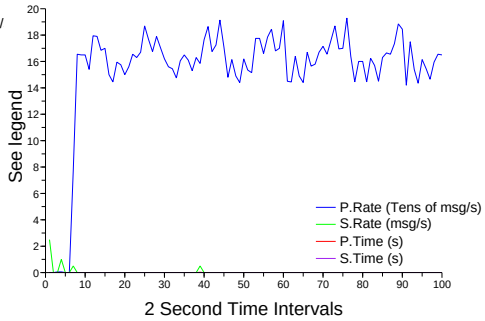

Fig. 19. Merged firewall subscription

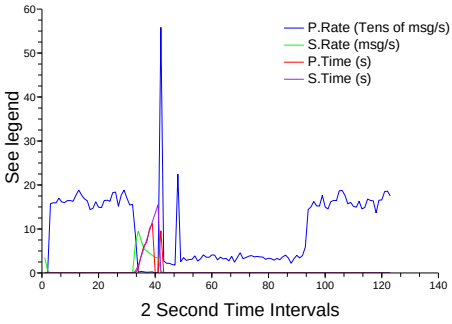

Fig. 20. Individual firewall subscriptions

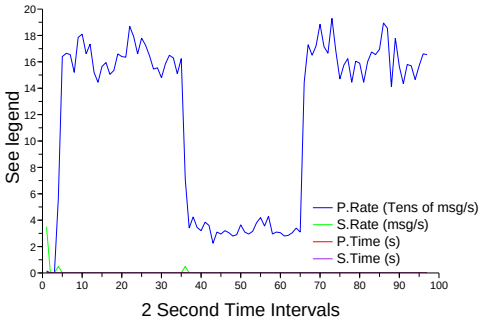

Fig. 21. Merged firewall subscription

20, we issue 100 separate firewall subscriptions to the internal firewall broker that block roughly 80% of the incoming publications overall. The publication rate `P.Rate` remains steady at the external broker (Fig. 18) but is much lower at the internal broker (Fig. 20) when the firewall policies are in effect. The time to process both publications and subscriptions (`P.Time` and `S.Time`, respectively) increases when the 100 firewall subscriptions are received.

```
PolicyStatement {
  On(Forward,Publication) {
    @broker: Routing

    If {StrictMatch()}
      Then {BlockMessage()}
} }
```

Fig. 22. Firewall policy

When firewall subscriptions are first issued and processed with their policies, broker processing times spike briefly before returning to normal sub-millisecond values. Subsequent removal of the same 100 firewall subscriptions via unsubscription is significantly faster, incurring no noticeable overhead. In Figs. 19 and 21, we issue a single subscription merged from the 100 separate firewall subscriptions that block the same amount of traffic. As there is only a single subscription and policy rule to process, Fig. 21 shows that there is no noticeable disruption to broker processing when the policy takes effect and is later removed. The original subscription issued by the application did not need to change in either case. This technique allows us to dynamically specify content-based firewall rules that are totally independent of the filters specified by existing applications. In these experiments, the firewall subscriptions were issued to the internal broker by a normal CPS client, but the authentication policies described earlier can be used to place access control policies on who is able to issue firewall filters. Note that a reverse scenario where publications are blocked from leaving the internal overlay and subscriptions are blocked from entering is also possible.

6 Conclusion and Future Work

In this paper, we have presented a content-based policy framework for distributed CPS systems that supports a novel *post-matching* policy model. Evaluations of our reference implementation show that this model is capable of achieving scalable and expressive policies in distributed CPS systems with little overhead. In

particular, we showed that our policy framework enables new features related to both CPS semantics and security such as notification semantics, meta-events, security zoning, and CPS firewalls. By leveraging the capabilities of existing CPS matching algorithms, our policy model allows these features to be specified easily and dynamically. Since our model is based on generic CPS matching concepts, our approach is appropriate across different CPS systems using either advertisement or subscription based semantics.

Although we have addressed many concepts in our policy framework implementation, some future work still remains. In particular, we have not discussed self-management features such as conflict resolution in any detail. Although many conflict resolution strategies are possible [25, 21], none are universally applicable across all conflict situations. At the moment, we use our own meta-notification feature to inform the application about policy conflicts and exceptions when they are detected. However, certain conflicts may be resolvable automatically by the system. We have started work in this area by identifying conflict situations amongst authorization and message transformation policies in the CPS context. Also, the policies we presented in this paper are based mostly on authorization and message transformation. There are still other types of policies that need to be explored, such as generic obligation actions [10] involving logging, persisting messages to a database, and other similar actions. The meta-notification feature implemented using our policy framework is work in this direction.

Acknowledgements

This research was funded in part by OCE, NSERC, CA and Sun. We would also like to thank the anonymous reviewers and the members of the Middleware Systems Research Group for their valuable feedback regarding this work.

References

1. Banavar, G., Chandra, T., Mukherjee, B., Nagarajarao, J., Strom, R.E., Sturman, D.C.: An Efficient Multicast Protocol for Content-based Publish-Subscribe Systems. In: ICDCS. (1999)
2. Carzaniga, A., Rosenblum, D., Wolf, A.: Design and Evaluation of a Wide-Area Event Notification Service. ACM Transactions on Computer Systems **19(3)** (2001) 332–383
3. Pietzuch, P.R., Bacon, J.M.: Hermes: A Distributed Event-Based Middleware Architecture. In: DEBS. (2002)
4. Fiege, L., Mezini, M., Mühl, G., Buchmann, A.P.: Engineering Event-Based Systems with Scopes. In: ECOOP. (2002)
5. Fidler, E., Jacobsen, H.A., Li, G., Mankovski, S.: The PADRES Distributed Publish/Subscribe System. In: Feature Interactions in Telecommunications and Software Systems. (2005)
6. Aekaterinidis, I., Triantafillou, P.: PastryStrings: A Comprehensive Content-Based Publish/Subscribe DHT Network. In: ICDCS. (2006)

7. Cugola, G., Picco, G.P.: REDS: A Reconfigurable Dispatching System. In: International Workshop on Software Engineering and Middleware. (2006)
8. Sivaharan, T., Blair, G.S., Coulson, G.: GREEN: A Configurable and Reconfigurable Publish-Subscribe Middleware for Pervasive Computing. OTM Conferences **1** (2005) 732–749
9. Calo, S., Lobo, J.: A Basis for Comparing Characteristics of Policy Systems. In: POLICY, Washington, DC, USA, IEEE Computer Society (2006) 183–194
10. Sloman, M.: Policy driven management for distributed systems. Journal of Network and Systems Management **2** (1994) 333–360
11. Opyrchal, L., Prakash, A., Agrawal, A.: Supporting Privacy Policies in a Publish-Subscribe Substrate for Pervasive Environments. JOURNAL OF NETWORKS **2** (2007) 17–26
12. Belokosztolszki, A., Eyers, D.M., Pietzuch, P., Bacon, J., Moody, K.: Role-Based Access Control for Publish/Subscribe Middleware Architectures. In: Distributed Event Based Systems. (2003)
13. Sturman, D., Banavar, G., Strom, R.: Reflection in the Gryphon Message Brokering System. In: Reflection Workshop at OOPSLA. (1998)
14. Strassner, J., Schleimer, S.: Policy Framework Definition Language. In: http://www3.ietf.org/proceedings/98dec/I-D/draft-ietf-policy-framework-pfdl-00.txt. (1998)
15. Brownlee, N.: SRL: A Language for Describing Traffic Flows and Specifying Actions for Flow Groups. In: http://www.rfc-archive.org/getrfc.php?rfc=2723. (1999)
16. Blunk, L., Damas, J., Parent, F., Robachevsky, A.: Routing Policy Specification Language next generation (RPSLng). In: http://www.ietf.org/rfc/rfc4012.txt. (2005)
17. Stone, G.N., Lundy, B., Xie, G.G.: Network Policy Languages: A Survey and a New Approach. IEEE Networks **January/February** (2001) 10–21
18. Agrawal, R., Srikant, R., Thomas, D.: Privacy Preserving OLAP. In: SIGMOD. (2005)
19. WS-Policy: http://www.w3.org/Submission/WS-Policy/
20. Li, G., Jacobsen, H.A.: Composite subscriptions in content-based publish/subscribe systems. In: Middleware. (2005)
21. Aib, I., Agoulmine, N., Fonseca, M.S., Pujolle, G.: Analysis of policy management models and specification languages. Network control and engineering for Qos, security and mobility II **2** (2003) 26–50
22. Dulay, N., Lupu, E., Sloman, M., Damianou, N.: A policy deployment model for the Ponder language. In: IEEE/IFIP International Symposium on Integrated Network Management. (2001)
23. Rafaeli, S., Hutchison, D.: A Survey of Key Management for Secure Group Communication. ACM Computing Surveys **35(3)** (2003) 309–329
24. CHAP: http://www.networksorcery.com/enp/rfc/rfc1994.txt
25. Dunlop, N., Indulska, J., Raymond, K.: Methods for Conflict Resolution in Policy-Based Management Systems. EDOC **00** (2003) 98