Policy Interoperability and Network Autonomics

Shane Magrath[1], Robin Braun[1], and Fernando Cuervo[2]

[1] University of Technology, Sydney,
Broadway, Australia
shane.magrath@uts.edu.au
[2] Research and Innovation, Alcatel Canada
600 March Rd, Ontario Canada K2K 2E6,
fernando.cuervo@alcatel.com

Abstract. Autonomic behaviours in network operations will alleviate much of the labour intensive and error prone interventions of today's complex networks. The Service Provider must be able to manage the infrastructure and services at an abstract level, focusing on what the desired behaviour should be rather than how it might be specifically achieved. Policy-Based Network Management (PBNM) appears as one of the leading mechanisms to describe desired behaviours and abstract the programmability of an autonomic network infrastructure to the Service Provider. For massive-scale and complex networks, the current understanding of the Higher Level to Lower Level (HL→LL) refinement process commonly used in PBNM today is not completely effective. One problem encountered is the need to provide a bind mechanism between Higher Level and Lower Level policy specifications such that cross-layer policy requests in the policy continuum can be made by lower policy layers in a dynamic policy refinement cycle (LL→HL→LL). In this paper, we illustrate the problem with a policy-based simple admission control (SAC) application. We then show that policy specifications with a join operator ($\bowtie$) simplify the SAC specification. We also investigate the performance considerations of this enhancement in Internet size applications. Our future goal is to provide a policy inference engine that can support complex specifications appropriate for PBNM systems that support autonomic behaviours in large networks, made of Network elements with realistic memory and processing constraints.

1 Introduction

There can be no question that infrastructure and services are harder to manage now than they were perhaps five years ago. In that time, technology has improved, customer expectations have risen, services have become more complex, the weight of legacy infrastructure heavier and the collision between traditional Telecommunications applications (*voice*) and Enterprise technology (TCP/IP) have threatened the sustainability of conventional market models for Carriers and Service Providers.

Telecommunication systems management is *complex*. Conventional management architectures and standards have proven inadequate when faced with new

M. Smirnov (Ed.): WAC 2004, LNCS 3457, pp. 25–43, 2005.

sets of complicating requirements - pervasive operational security, differentiated service level agreements and a plethora of Next Generation Services. One interesting reason is that these legacy management architectures maintain the semantic locus in the *individual* elements that comprise the domain of managed objects. Data is collected from *each* device, and stored centrally. Information about service status is attempted to be re-constructed from a collating of the properties of the *individuals* from the network. This is analogous to considering a person's health as a function of "simply" probing the state of each of the body's cells and collating the findings - an ultimately ineffective approach since it can not describe the state of the higher physiological functions that *emerege* when aggregates of cells interwork to give rise to a new system functions.

In this context, policy-based management architectures have been considered as viable and necessary part of these new management frameworks ([1, 2, 3, 4]). Service Providers need to be freed to manage their systems at a higher level of abstraction than the mere technology configuration. They need to consider and specify the *business* requirements of the *applications* that comprise the *services* being operated. With a proper understanding of the *roles* that comprise the service operations, the Service Provider can formulate HL policies appropriate for a PBNM system that link *business* requirements with *technology* configuration - a level of interoperability that has previously not been achieved.

However, the HL→LL refinement process is not always sufficient for the needs of complex networks and services. Typically, the managed objects within the domain will encounter situations not covered by their present configuration. These devices need a means to determine appropriate behaviour when faced by these conditions. By referring the request to a policy server that is authoritative and capable of interpreting the request against the HL policies, an appropriate LL policy can be identified and deployed to the device. This LL→HL→LL cycle can be problematic. It implies that the policy server is able to bi-directionally refine HL and LL policies in real-time. This further implies a mechanism whereby the policy server can provide a binding between the LL and HL policies. Unfortunately, most of the current policy specification approaches and languages do not innately have this ability.

In this paper, we consider the value of a *join* operator in a policy specification. The join operator allows a linkage to be achieved between HL and LL policy information. This can greatly simplify policy specification for the LL→HL→LL requirement. Furthermore, the join operation is required to perform at the level appropriate for these real-time systems where servicing massive-scale applications involve transaction rates that are measured in thousands of events per second. For autonomic system architectures involving a centralised PBNM system these issues need to be considered.

Our paper proceeds as follows. Section Two considers the current state of the research in this area. Section Three presents a scenario involving a simple access control application (SAC) and examines the issues presented by this application. Section Four presents a contribution towards alleviating the problem identified by

SAC and a consideration of the issues raised by the proposed solution. We finish with a summary of our findings and an outline of future work to be undertaken.

2 Previous Work

We consider the literature from the perspectives of:

- policy semantics, specifications and languages
- policy refinement

2.1 Policy Semantics

A bit of space will be taken to outline the semantics of policy seen in the literature since a characterisation of these systems is needed here.

Within the literature, the term "policy" generally means an *administrative rule* - that is, a *declarative* statement of requirement. More specifically though, the semantics of policy vary slightly across the literature. The IETF [5] defines policy as follows:

> **Policy**
>
> "Policy" can be defined from two perspectives:
>
> - A definite goal, course or method of action to guide and determine present and future decisions. "Policies" are implemented or executed within a particular context (such as policies defined within a business unit).
> - Policies as a set of rules to administer, manage, and control access to network resources [RFC3060].
>
> Note that these two views are not contradictory since individual rules may be defined in support of business goals.

As can be seen from the IETF's definition, policy involves notions of "context" and abstraction. More helpfully, Verma [6, 7, 8, 9] has drawn the distinction between high-level policies and low-level policies. High-level policies are used to express "business-level" rules. Low-level policies are used to express "technology-level" rules.

Again, the IETF [5] also develops the notion that policies have varying levels of abstraction:

> **Policy Abstraction**
>
> Policy can be represented at different levels, ranging from business goals to device-specific configuration parameters. Translation between different levels of "abstraction" may require information other than policy, such as network and host parameter configuration and capabilities. Various documents and implementations may specify explicit levels of abstraction. However, these do not necessarily correspond to distinct processing entities or the complete set of levels in all environments. (See also "configuration" and "policy translation".)

Abstraction is an important concern in autonomic systems engineering because it allows us to describe and be concerned only for the important functions of the required abstracted autonomic behaviour.

The most significant body of research contributing to PBNM has been undertaken at Imperial College under Morris Sloman. A summative work in [1] describes their definition of policy to include "*types*". They classify policy as either being typed as:

- Authorization - related to the permissions that a "*domain subject*" can perform,
- Delegation - related to the ability of a domain subject to delegate its privileges,
- Obligation - related to the actions a domain subject must perform in response to conditions and events,
- Refrain - related to the actions a domain subject must refrain from performing on *"targets"* in the domain.
- Composite - a grouping of the more basic types described above for administrative reasons.

Closely associated with policy is the concept of "*roles*". Roles are generally used as a container for policies. That is, a role can contain a collection of policies. Moreover, roles express the rights, duties and obligations of a position or function. The role concepts have been developed by Sloman et al in [1, 10, 2].

Sloman et. al. also add to the concepts of policy and PBNM by incorporating the concept of "*Domains*" into the semantics ([11, 12, 13]). Domains are a collection of managed objects that are under one administrative control and are related by "subject". That is, those objects that are part of the same policy application space may be collected together in a domain. Policies may operate on the entire set of domain objects or a sub-set defined by some selection criteria. For example, a Service Provider may place all DIFFSERV edge routers in the same domain. A policy may then be authored such that the *scope* of the policy is limited to a sub-domain of those routers - for example, all DIFFSERV edge routers located in Victoria (a subset of Australian routers) should authenticate Operations Support staff who wish to log on to the router via the Victorian RADIUS server.

A common thread to policy definition is the importance placed on the concept of "*events*". Events are a signal from the management environment that a possible state-change to the Domain has occurred. Events are used to trigger policy evaluation ([14]). In contrast to most other approaches, the IETF COPS formulation has no precise operationally explicit syntax for event management. However, there is an implicit concept of events in the QOS policy applications where packet arrival events are used to trigger the appropriate evaluation and marking of the packets as specified by the QOS PIB.

Several contributions have been made to the specification of policy *languages*. Two significant contributions are:

Ponder The formal specification from Imperial College ([15, 14]). Ponder is a declarative, object-oriented language that supports events, constraints, roles, templates and other useful language features.

$\mathcal{PDL}$ Policy Description Language ($\mathcal{PDL}$) is from Bell Labs ([1, 16, 17]). $\mathcal{PDL}$ is a declarative event-condition-action language originally developed for specifying network management policies.

Ponder. A whimsical and hopefully self-explanatory illustration is provided:

```
inst oblig /Policies/HomeLandSecurityPolicies {
  on       Event(TerroristAction, Hostage) ;
  subject  /Government/MI5 ;
  target   t = /Agents/Agent007 ;
  do       t.CaptureTerrorists(TerroristAction)->
           t.RescueTheGirl(TerroristAction, Hostage) ->
           t.SaveTheWorld(TerroristAction) ;
  when t.isNotInBed() ;
}
```

$\mathcal{PDL}$. Policy Definition Language ($\mathcal{PDL}$) ([16, 17]), like Ponder, is an event-condition-action (ECA) declarative language though it does not have all the features that Ponder has. $\mathcal{PDL}$ is reviewed by Sloman in [1]. $\mathcal{PDL}$ was originally developed for the specification of network management policies.

$\mathcal{PDL}$ policies consist of two types of expressions:

- policy *rule* propositions of the form :
  `<event> causes <action> if <condition>`
- policy *defined event* propositions of the form :
  `<event> triggers pde(`$M_1 = T_1, ..., M_k = T_k$`)`
  `if <condition>`

The policy rule is the conventional ECA specification of a rule. The policy defined event read "if the *event* occurs and the *condition* is satisfied then the policy defined *event* is triggered". $\mathcal{PDL}$ supports a basic event calculus for causal specification purposes. $\mathcal{PDL}$ does not support the notion of "roles", nor does it have a concept of "domains". These are serious weaknesses to have in a *generalised* PBNM system.

Despite the simplicity of the language, $\mathcal{PDL}$ has shown itself capable in managing a range of network management tasks involving telecommunications switch products ([1]).

2.2 Policy Refinement and Inter-operability

Policy refinement is concerned with the process of mapping a set of HL policies to a set of LL policies. Bandara ([14]) considers refinement as having three requirements: Correctness, Consistency and Minimality. Verma identifies the correctness requirements for successful refinement by describing the process as a

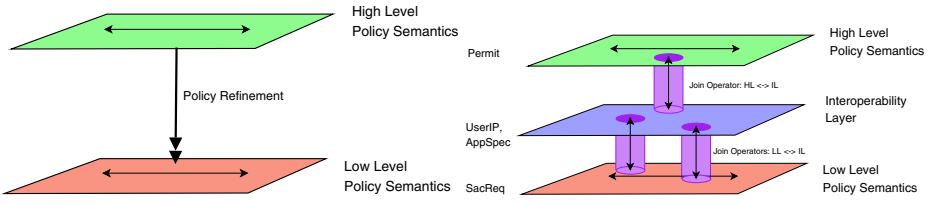

Fig. 1. (a) Policy Refinement (b) Policy Interoperability with joins via Intermediate Layer

consideration of translation, bounds, relation, consistency, dominance and feasibility checking ([6]). Kanada considers refinement more critically in [18, 19] in terms of the significant problems of optimally refining HL policies to LL policies involving policy division and fusion.

In contrast to refinement, we introduce *inter-operability* (see Figure 1). Refinement is concerned with the one directional mapping: HL→LL. Inter-opera bility is the bi-directional mapping: HL ↔ LL. The refinement process occurs *before* policies are operationally deployed. Inter-operability occurs *as part of* the operational deployment. The function of inter-operability mapping is to allow LL policies at run-time to dynamically refer to their HL parents as the need arises. We will see that interoperability requires the presence of an interoperability layer (IL) that mediates between HL and LL representations.

In the most general case where there exists a progression of abstraction layers (the "policy continuum" [20]), the HL ↔ LL mediated by an IL remains useful as a fundamental pattern. By cascading the inter-operability model, a more general abstracted policy continuum can be achieved - "one man's HL is another man's LL" so to speak. It is quite common for autonomic systems to contain quite a sizable stack of abstraction layers to provide the final functional service, so the ability for inter-operability to be cascaded in order to maintain the interchange between layers is reassuring. In contrast, policy refinement presents some problems to autonomics because each downward refinement from one HL to the next LL further distances the final operations from the true functional intent of the uppermost policy management layer. Without the ability for the LL to interact with the upper layers, policy refinement only provides a partial mechanism for autonomic operations.

3 Theory

We begin our consideration of the policy refinement and inter-operability problem with a problem scenario. We imagine a *large* network (carrier-scale) of users who enjoy the benefits of *individually* tailored service levels differentiating the quality of service their *applications* receive (see Figure 2). Moreover, the network provides a rich set of features such as broadband mobility, ubiquitous service access, and continuous context sensitive display to multi-modal terminals.

In this scenario, it is the *access* network that needs to perform the necessary access controls and QoS management to enforce the service level requirements.

Moreover, in this environment it does not make sense necessarily to *pre-provision* the edge devices with complete and specific LL policy configuration: there are a lot of users and they are mobile. Pre-provisioning policy to the edge consumes resources, with consequences if poorly deployed:

- large LL policy tables occupy more memory and therefore slow search times;
- mobility means that not all the required LL policies are in the tables. Moreover, there may exist LL policies in the tables that won't be used in practice;
- the policy server is committed to maintaining the state of deployed LL policies with no benefit if the policies are inappropriately deployed.

By deferring the deployment of LL policies until it becomes clear of the location of the user and their *contextual* requirements, a better match between policies usefully deployed and the resources consumed is made. This late binding of policies can benefit the system by mediating the effects of poor policy deployment, as argued in [21].

If we restrict our consideration to *simple access control* with deferred policy provisioning, then we require each edge device to ask the policy server what to do when it detects a new session flow (a "context request") that involves a previously unknown user, or a previously unencountered combination of source/destination and application specifications for the new session.

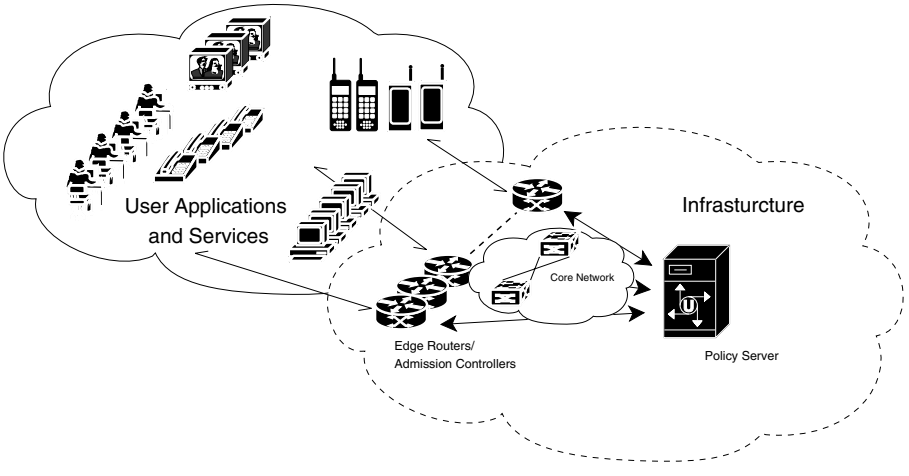

Fig. 2. Simple Access Control in a Network

In this case, the edge device informs the policy server by sending an *event* containing the tuple:

```
{Sac_Request SrcAddr SrcPort DestAddr DestPort Protocol SessionID}
```

This tuple contains the minimum information specifying a new session flow in the network. It is the LL specification of information appropriate for the technology plane of systems management.

However, the Administrator is happily oblivious to these specifics. He relates to the PBNM system through the specification of HL policies appropriate to the requirements of the business. In the SAC case, the Sys Admin has previously made known to the system the following tuples of information:

```
{Permit Shane Web}
{Permit Shane email}
{Permit Shane SSH}
{Permit Leanne Web}
{Permit Leanne email}
```

where each tuple is of the format

```
{Permit UserName ApplicationName}
```

Here, the user SHANE is permitted to use web, email and SSH applications. Implicitly, any application not in the list is denied access to the network. Similarly, user LEANNE is permitted to use only web and email. Here the requirements are analogous to a firewall. However, *every* edge device in the domain performs the firewall enforcement function.

If this were the only system information available, there is no possibility for the PBNM system to effect the meaning of the HL policies at the LL operational level. Additional information, from ad hoc sources, that serves to bind the HL and LL representations is required. These relations form an *Interoperability Layer* (IL) and facilitate the cross-domain mapping between LL and HL representations.

We need two IL relations:

- A User Name $\Longleftrightarrow$ IP Address binding, sourced from RADIUS, or DHCP :
  ```
  {UserIP Shane 138.25.41.126}
  {UserIP Leanne 142.53.16.7}
  ```
- An Application Name $\Longleftrightarrow$ IP Specification binding, made known by the application provider :
  ```
  {AppSpec Web 80 TCP}
  {AppSpec email 25 TCP}
  {AppSpec SSH 22 TCP}
  ```

where each tuple is of the format[3]

```
{AppSpec Name Port Protocol}
```

It is now conceivable that the policy server can determine what to do in response to SAC_REQUEST events. For instance, if the server were to receive:

[3] In reality, the specification of applications in terms of protocol and port numbers are more complicated than this since more than one tuple may be required and the tuples may be dynamic. However, this is *simple* access control after all.

Algorithm 1. Policy Rule for SAC (expressed in the Jive! language)

```
policy SAC_Policy {
    condition {
      event Sac_Request : (srcAddr * * destPort protocol SessionID}
            UserIP      : (userName srcAddr}
            AppSpec     : (appName destPort protocol}
            Permit      : (userName appName}
    }
    action {
      main {
        Send(Sac_Response, SrcRouter, SessionID, Permit);
      }
      default{
        Send(Sac_Response, SrcRouter, SessionID, Deny);
      }
    }
}
```

{Sac_Request 138.25.41.126 1078 204.32.45.61 80 TCP}

it should reply with a POSITIVE authorisation.
However, to do so involves the server in several join operations:

- It has to perform a LL→HL resolution of the source address (138.25.41.126 →SHANE);
- It has to perform a LL→HL resolution of the application (80/TCP →WEB)
- It needs to determine if the request is permitted (SHANE + WEB →PERMIT)
- It finally needs to resolve the HL policy requirement to a LL deployable specification:

{Sac_Response SrcRouter SessionID Permit}

This simple application can be specified with the policy rule in Algorithm 1., using the Jive! language we have developed for experimenting with these systems.

The *condition* clause of this policy involves a syntax similar to conventional rule-based production systems. Join operations are identified by a "¡join_name¿" syntax on the right hand side of the ":" operator. An isolated "*" signifies a "don't care" conditional match for that particular field/attribute. The policy also includes a *default* action clause that allows for the efficient handling of requests that *fail* to cause the condition clause to evaluate to TRUE.

The most interesting feature of the policy rule is the *condition* clause. The condition clause defines the pattern matching relationship between the HL and LL data as well as the more general constraints of that relationship. By taking the set of set of data and looking for *all* combinations that can satisfy the condition clause pattern constraints, a set of activations that can be executed is achieved. Moreover, in this example, the use of the join operations fulfill the needs of the interoperability requirements.

This simple example provides support that the join operator is an elegant and effective means for dynamic policy refinement and interoperability *between* HL business rules and LL technology configuration. This policy has achieved the valuable goal of enabling the system to inter-operate between HL and LL levels of abstraction and simultaneously maintain the very nice separation of concerns relating to the specification of HL and LL information.

However, most of the current examples of policy languages and specifications do not support this operator. This is not necessarily an oversight. PBNM specifications originally developed to service a specific need: the management of QoS services, and DIFFSERV in particular. The requirements here were to provide a PBNM framework that can operate in real-time by providing LL policy configuration in a form appropriate for the domain of managed devices, namely *tables* of a *limited* range of *typed* information that devices such as routers and switches could interpret efficiently. In this context, keeping policy free of join semantics is conducive to the QoS management problem. It does however, make things difficult for dynamic HL↔LL interoperability.

To be considered is the question of *efficiently* evaluating the join in real-time. Despite the apparent simplicity of the SAC application, as the *number* of tuples grow, the number of candidate matches that need to be considered by the rule also grows exponentially. This presents a tension between requiring the ability to support HL policy specification that is fully interoperable with the LL specifications, and the certain need for maintaining system throughput performance at very high transaction rates and low round-trip latency times.

We proceed to consider these issues.

4 Considerations

4.1 Action Clause

One observation that can be made is that a similar effect of the join can be procedurally achieved as part of the *action* clause of the policy. That is, if we restrict the condition clause to just the SAC_REQUEST event specification, then the remaining information can be determined as a series of functional lookups to a directory, or a location service, etc as part of the action clause, an approach sometimes seen.

There are a few issues with this:

1. Functional lookups require the existence of the functions to perform them. These functions are either to be made available as libraries as part of the policy language, or the Sys Admin would need to develop them.
2. The question exists whether the sort/search/select operations required as part of the lookup leads to best performance of the policy server.
3. If every interesting event that is raised triggers a positive evaluation in the condition clause, only to be later discarded by further constraints in the action clause, where was the benefit? Moreover, by raising an abortive activation, other activations in the set that contend for service risk delayed resourcing.

4. Good design would suggest that the requirements are best served by maximising the necessary constraints on the condition clause leaving the action clause to be as specific and productive as possible. This approach is consistent with the very event-condition-action character of the system.

4.2 A More Formal Consideration

We will make some observations about the policy computational complexity by formalising the description of SAC.

We first define the set AS as the *activation set*. It is the set of all instances of *current* rule activations that have yet to be serviced.

We define the *system state* as a series of *relations* on the data known to the system. For example, in the SAC example, data about users is made known through the relation schema:

$$UserIP = \{UserName, IPAddr\} \tag{1}$$

Similarly,

$$\begin{aligned} Apps &= \{AppName, Port, Protocol\} \\ Permits &= \{UserName, AppName\} \\ SacReq &= \{SrcAddr, SrcPort, DestAddr, DestPort, Protocol\} \end{aligned} \tag{2}$$

We are particularly interested in the effects of the $SacReq$ event on the activation set:

$$AS = AS \cup Rules(SacReq) \tag{3}$$

where

$$\begin{aligned} Rules(SacReq) = R_1(SacReq) \cup R_2(SacReq) \\ \cup ... \cup R_n(SacReq) \end{aligned} \tag{4}$$

and R_i is rule i in the system.

This expresses the idea that a *single* event may be responsible for *multiple* activations as more than a single rule may be satisfied by the event. This representation is particularly relevant when the underlying inferencing engine does not support "default action" semantics. See Appendix One for a short discussion.

For the next section, it helps to know that the join operator can be defined as:

$$A_a \bowtie_b B = \sigma_{a=b}(A \times B) \tag{5}$$

As an example:
if

$$A = \{a, b\}$$
$$B = \{1, 2\} \tag{6}$$

then

$$A \times B = \{(a, 1), (a, 2), (b, 1), (b, 2)\}$$
$$\therefore \sigma_{a=b}(A \times B) = \emptyset$$
$$\therefore A_a \bowtie_b B = \emptyset \tag{7}$$

We now consider the rule from Algorithm 1.:

$$AS = AS \cup SAC\_Policy(SacReq) \tag{8}$$

Procedurally, we express the rule predicate as a process of relational refinements using relational algebra:

$$\begin{aligned} J1 &= UserIP_{IpAddr} \bowtie_{SrcAddr} SacReq \\ S1 &= \sigma_{SacReq.SrcAddr}(J1) \\ J2 &= Apps \underset{protocol}{\overset{port}{}} \bowtie \underset{protocol}{\overset{port}{}} SacReq \\ S2 &= \sigma_{\substack{SacReq.Port\ AND \\ SacReq.Protocol}}(J2) \\ J3 &= Permits_{UserName} \bowtie_{UserName} S1 \\ J4 &= J3_{AppName} \bowtie_{AppName} S2 \\ AS &= AS \cup J4 \end{aligned} \tag{9}$$

Now, if $|J4| = 0$ then the condition clause is *not* satisfied and the *default* action is added to the activation set. Otherwise, each tuple within $J4$ is added to the activation set for *executor* scheduling. When will $|J4| = 0$? When the SacReq event presents a context (that is, set of field values) that:

1. can *not* be mapped into the Intermediate Layer (IL) - (the "unknown application" or "unknown user" case)
2. *can* be mapped to the IL *but* can *not* be mapped from the IL to the HL - (the "no permission" case) (see figure 1).

If either of these two conditions hold, then $|J4| = 0$.

In terms of computational complexity, since the join operator can be defined as:

$$A_a \bowtie_b B = \sigma_{a=b}(A \times B) \tag{10}$$

So, if $|A| = N$ and $|B| = M$ then

$$\begin{aligned} |\ A_a \bowtie_b B\ | &= |\ \sigma_{a=b}(A \times B)\ | \\ &\leq M \times N \end{aligned} \tag{11}$$

That is to say, the join operator is performed by :

1. taking the Cartesian product of the two relations (forming a new relation whose cardinality is $M \times N$),
2. selecting tuples from this intermediate relation that satisfies the join condition.

In this way, we can see that joining two *large* relations, there can be a substantial computational cost proportional to the product of their cardinalities. Within the SAC application, we should reasonably expect:

$$| \, Permits \, | \gg | \, UserIP \, | \gg | \, Apps \, | \gg 1$$

Given this, one might expect $J3$ to be the most expensive operation in the procedure.

A few observations can be made:

- the algebraic procedure above for the SAC_Policy is *not* unique. That is, there are other equivalent derivations that are mathematically identical in the final result. However, they are *not* cost identical. This means in practice we would wish to *optimise* the method for computationally determining the SAC_Policy result. The basis for this optimisation follows from the following observation.
- there is much to be gained to being as *selective* as possible *early* in the pattern matching process. By reducing the cardinalities of the intermediate relations as early as possible, the join operations become more efficient in both memory and time requirements;

4.3 Experiment

Design. We built a prototype PBNM system ("Step") that supports Join semantics for the purpose of testing, amongst other things, the performance qualities of the system against increasing domain size. As stated earlier, it is a requirement of all Service Provider PBNM systems to adequately perform for domain sizes consistent with those found in massive-scale Service Provider networks.

With reference to Figure 3, the experimental system consisted of three main components:

- GENEVA: a configurable Elvin Event Generator for producing the SAC_REQUEST events (developed by us),
- Elvin Server: the Elvin content-based routing server ([22] [4]),
- STEP: The policy enforcement point that we developed.

The system was established on the university's research computing cluster (Orion). Each of the machines in the system is composed of a 3.0GHz Pentium 4 with 800MHz FSB and 2GB 400MHz DDR-RAM and runs Red Hat Linux 8.0.

[4] Refer HTTP://elvin.dstc.com/

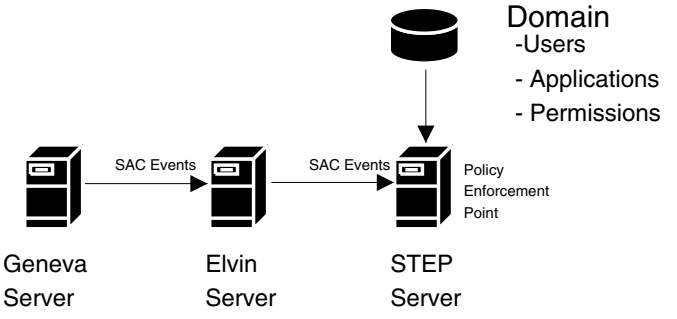

Fig. 3. Experiment Configuration

For each of the runs, the GENEVA application was configured to produced 20,000 SAC events at the rate of 40 events per second distributed negative exponentially to emulate the stochastics of large group behaviour. The field contents of each SAC event is randomised in such a way to range over the entire domain dimensionality. The STEP sub-system receives and enqueues the SAC_REQUEST events from the Elvin server. At the heart of STEP is an externally sourced rule inferencing engine (Jess: Java Expert System Shell[5]) that implements the Rete algorithm [23]. This algorithm performs the computation of Join operations and is the typical algorithm found in most commercial and academic inferencing engines. We encoded the SAC application into STEP so that it would dequeue the SAC events and take appropriate action (permit or deny) according to the policy. We instrumented STEP in order to measure the performance of the inferencing sub-system under increasing domain sizes. This data was captured for each experimental run that we performed.

The reported results are for a series of runs consisting of the number of known applications held constant at 10, and a fixed Permit cardinality of 40% of UserIP x Apps. The free variable is the number of Users the system knows about. The runs consists of User populations of 10, 100, 1K, 10K, 100K, 1M users.

Results and Discussion. Figure 4 reports the throughput characteristics of the system that was determined from the experimental runs described above. The results are not encouraging for the massive-scale applications envisaged by a Service Provider. For even moderately sized domains, the performance of the system deteriorates significantly reaching a minimum of around 20 events per second throughput. Service Providers need to maintain system throughput of the order of thousands of events per second for massive domains. We are several orders of magnitude below the requirement.

As indicated in Section 4.2, this result is not surprising because of the multiplicative effects observed in the Cartesian products comprising J4 (via J3 and J1). The

[5] Refer to HTTP://herzberg.ca.sandia.gov/jess/

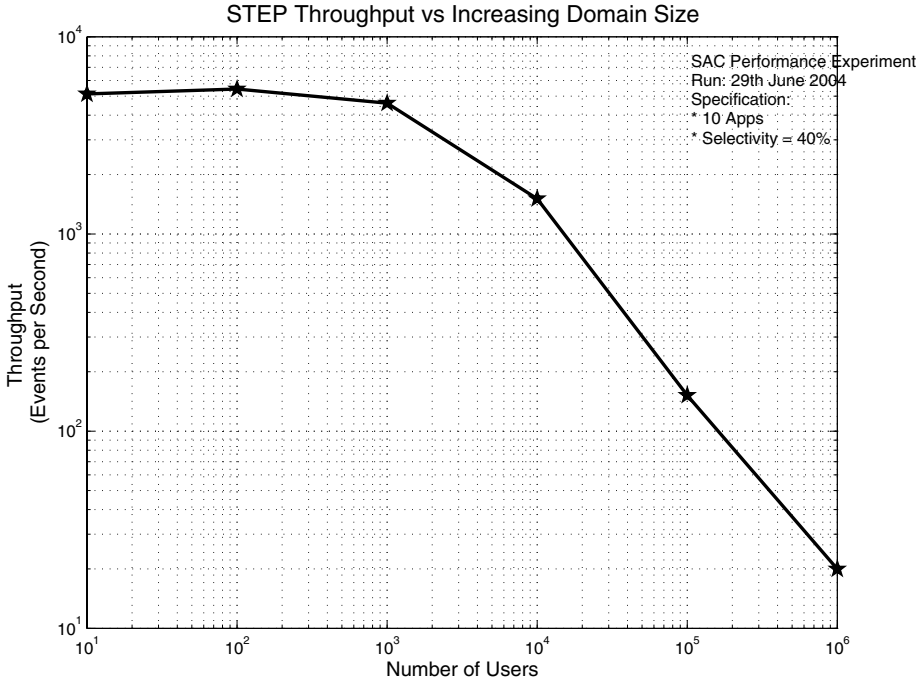

Fig. 4. Performance Results for STEP

Rete algorithm, as a finite-differencing algorithm, maintains increasingly large data structures commensurate with increases in domain size. Whilst it is not surprising that performance diminishes with domain size, the question then becomes how do we maintain performance *and* the use of the Join semantics which is so useful for autonomic and policy-based network management. This is our future work. The logical next step is to try other algorithms besides Rete, such as TREAT and Matchbox ([24, 25]). This is useful, necessary work however any purely centralised architecture will ultimately be defeated by sheer size. Ultimate improvements will largely be made through increased processing speed (Moore's Law) and improvements in compiler optimisation techniques.

An obvious alternative is a more distributed approach such as may be achieved by a multi-agent system, however our feeling is that the impact of inter-agent communications may defeat the advantages of the distribution. An interesting alternative, and certainly more consistent with physiological and biological autonomic systems, are the *swarm* algorithms for performing task allocation and resource distribution ([26, 27]). Their main advantage is their *lack* of inter-agent communication, and robust ability to *adapt* to changing environments. However, the engineering of such systems is still far from mature so this forms another line of development in our research.

5 Conclusion

We have established that LL→HL→LL policy interoperability provides significant Administration benefits to complex network management. The concept of an interoperability layer that mediates LL and HL layers of abstraction is seen to be an important component for the autonomic management of systems since it allows the bidirectional policy interaction between the layers during system run time without Administrator intervention. This is in contrast to policy refinement approaches that seek to "compile" policies from a HL representation into a LL specification prior to operational deployment.

We have also established the need for a high performance policy inferencing engine that can service the needs of massive-scale real-time applications found in large Service Provider networks. The use of standard algorithms for inferencing may not be the best choice for the specific needs of Service Providers and the type of real-time policy applications they may wish to run.

Our future work consists of developing a set of benchmark real-time policy applications that are relevant to Service Providers in general. Using these benchmarks we expect to develop and test different inferencing algorithms and determine which may best fit the operational requirements of these massive-scale applications.

Acknowledgment

The authors also wish to thank Alcatel for their support and interest in this research.

References

1. Sloman, M., Lupu, E.: Security and Management Policy Specification. IEEE Network **2** (2002) 10–19
2. Lupu, E., Milosevic, Z., Sloman, M.: Use of Roles and Policies for Specifying and Managing a Virtual Enterprise. In: Research Issues on Data Engineering: Information Technology for Virtual Enterprises, 1999. RIDE-VE '99. Proceedings., Ninth International Workshop on. (1999) 72–79
3. Strassner, J.: Policy-Based Network Management - Solutions for the Next Generation. Morgan-Kaufmann (2003)
4. TMF: The NGOSS Technology Neutral Architecture Specification V3.0. Technical Report TMF053, TMF (2003)
5. Westerinen, A., Schnizlein, J., Strassner, J., Scherling, M., Quinn, B., Herzog, S., Huynh, A., Carlson, M., Perry, J., Waldbusser, S.: Terminology for Policy-Based Management. Informational RFC 3198, IETF, Network Working Group (2001)
6. Verma, D.: Simplifying Network Administration Using Policy-Based Management. IEEE Network **16** (2002) 20–26
7. Rajan, R., Verma, D., Kamat, S., Felstaine, E., Herzog, S.: A Policy Framework for Integrated and Differentiated Services in the Internet. IEEE Network **13** (1999) 36–41

8. Verma, D.C.: Policy-Based Networking: Architecture and Algorithms. New Riders (2001)
9. Verma, D., Beigi, M., Jennings, R.: Policy Based SLA Management in Enterprise Networks. Lecture Notes in Computer Science **1995** (2001) 137–??
10. Lupu, E., Sloman, M.: A Policy Based Role Object Model. In: Enterprise Distributed Object Computing Workshop [1997]. EDOC '97. Proceedings. First International. (1997) 36–47
11. Damianou, N., Dulay, N., Lupu, E., Sloman, M., Tonouchi, T.: Tools for Domain-Based Policy Management of Distributed Systems. In: Network Operations and Management Symposium, 2002. NOMS 2002. 2002 IEEE/IFIP. (2002) 203–217
12. Robinson, D., Sloman, M.: Domains: A New Approach to Distributed System Management. In: Distributed Computing Systems in the 1990s, 1988. Proceedings., Workshop on the Future Trends of. (1988) 154–163
13. Sloman, M., Magee, J., Twidle, K., Kramer, J.: An Architecture for Managing Distributed Systems. In: Distributed Computing Systems, 1993., Proceedings of the Fourth Workshop on Future Trends of. (1993) 40–46
14. Bandara, A.K., Lupu, E., Russo, A.: Using Event Calculus to Formalise Policy Specification and Analysis. In: IEEE Workshop on Policies for Distributed Systems and Networks (Policy 2003). (2003)
15. Damianou, N.: A Policy Framework for Management of Distributed Systems. PhD thesis, Department of Computing, Imperial College, London (2002)
16. Bhatia, R., Lobo, J., Kohli, M.: Policy Evaluation for Network Management. In: INFOCOM 2000. Nineteenth Annual Joint Conference of the IEEE Computer and Communications Societies. Proceedings. IEEE. Volume 3. (2000) 1107–1116 vol.3
17. Kohli, M., Lobo, J.: Realizing Network Control Policies Using Distributed Action Plans. Journal of Network and Systems Management **11** (2003)
18. Kanada, Y.: Policy Division and Fusion: Examples and a Method-or, Multiple Classifiers Considered Harmful. In: Integrated Network Management Proceedings, 2001 IEEE/IFIP International Symposium on. (2001) 545–560
19. Kanada, Y., O'Keefe, B.J.: Rule-Based Building-Block Architectures for Policy-Based Networking. Journal of Network and Systems Management **11** (2003)
20. Strassner, J.: Autonomic networking - theory and practice (tutorial seven). In: 2004 IEEE/IFIP Network Operations and Management Symposium. (2004)
21. Cuervo, F., Sim, M.: Policy Control Model: a Key Factor for the Success of Policy in Telecom Applications. In: IEEE 5th International Workshop on Policies for Distributed Systems and Networks (POLICY 2004). (2004)
22. Seagall, B., Arnold, D.: Elvin has left the building: A publish/subscribe notification service with quenching. In: Australian Unix and Open Systems Group. (1997)
23. Forgy, C.L.: Rete: A Fast Algorithm for the Many Pattern/Many Object Pattern Match Problem. Artificial Intelligence **19** (1982) 17–37
24. Miranker, D.P.: TREAT: A Better Match Algorithm for AI Production Systems. In: National Conference on Artificial Intelligence. Volume 1. (1987) 42–47
25. Perlin, M.: Incremental binding-space match: The linearized matchbox algorithm. In: Proceedings of the IEEE Conference on Tools for AI. (1991)
26. various: Design Principles for the Immune Systems and Other Distributed Autonomous Systems. Santa Fe Institute Studies in the Sciences of Complexity. Oxford University Press (2001)
27. Bonabeau, E., Dorigo, M., Theraulaz, G.: Swarm Intelligence: From Natural to Artificial Systems. Santa Fe Institute Studies in the Sciences of Complexity. Oxford University Press (1999)

Appendix

Most PBNM languages and the supporting inferencing engines do not provide default action semantics. That is, if the policy has no "default" clause that is invoked when the condition clause fails to evaluate to TRUE in response to an event. This may be appropriate for some applications, but many Telecoms policy applications involve what we term as a "Definite Response" policy pattern. That is, when presented with a particular question via an event, the policy server *must* present an answer ("permit/deny", "yes/no", "gold/silver/bronze" etc).

For such environments requiring to implement the SAC application, the policy described in 1. requires two rules to perform correctly:

```
policy R1 {
    condition {
        event Sac_Request: (srcAddr * * destPort protocol}
              UserIP     : (userName srcAddr}
              AppSpec    : (appName destPort protocol}
              Permit     : (userName appName}
    }
    action {
        Send(Sac_Response, SrcRouter, SessionID, Permit);
    }
}
policy R2 {
    condition {
        event Sac_Request: (srcAddr * * destPort protocol}
          not (UserIP    : (userName srcAddr}
                AppSpec  : (appName destPort protocol}
                Permit   : (userName appName}
               )
    }
    action {
        Send(Sac_Response, SrcRouter, SessionID, Deny);
    }
}
```

It is important that for *any* SacReq event presented to the policy server, only one of the two rules is activated and admitted into the Activation Set. An interesting question is how might one prove the correctness of the two rules under all conditions? We note the following pre and post conditions hold:

Pre-Condition: $|AS| = 0$ and $|SaqReq| = 1$
Post-Condition: $|AS| = 1$

Let

$$J = \sigma_{\substack{UserName \\ AppName}} (\sigma_{\substack{DestPort \\ Protocol}} (\sigma_{SrcAddr}(SacReq \times UserIP) \times AppSpec) \times Permit) \tag{12}$$

Then the operation of the two rules together may be described as:

$$AS = R1 \cup R2 \tag{13}$$

where

$$\begin{aligned} R1 &= J \\ R2 &= \sigma(SacReq \times \bar{J}) \end{aligned} \tag{14}$$

Therefore,

$$\begin{aligned} |AS| &= |J \cup \sigma(SacReq \times \bar{J})| \\ &\leq |J| + |\sigma(SacReq \times \bar{J})| \end{aligned} \tag{15}$$

But the following observation holds:

- if $|J| = 1$ then $|\sigma(SacReq \times \bar{J})| = 0$, and
- if $|J| = 0$ then $|\sigma(SacReq \times \bar{J})| = 1$

Therefore $|AS| = 1$ under all conditions and the conditional operation of the two rules is shown to be correct.