Enhancements to Policy Distribution for Control Flow and Looping

Nigel Sheridan-Smith[1], Tim O'Neill[1], John Leaney[1], and Mark Hunter[2]

[1] Institute of Information and Communication Technologies,
University of Technology, Sydney
{nigelss, toneill, jrleaney}@eng.uts.edu.au
[2] Alcatel Australia
Mark.Hunter@alcatel.com.au

Abstract. Our previous work proposed a simple algorithm for the distribution and coordination of network management policies across a number of autonomous management nodes by partitioning an Abstract Syntax Tree into different branches and specifying coordination points based on data and control flow dependencies. We now extend this work to support more complex policies containing control flow logic and looping, which are part of the *PRONTO* policy language. Early simulation results demonstrate the potential performance and scalability characteristics of this framework.

1 Introduction

Policy-based Network Management (PBNM) systems require many additional capabilities in *carrier-class networks* - a service management orientation, and a capacity for *adaptive* and *dynamic* behaviour to respond to user and business needs. However, the dynamic behaviour required could cause scalability issues in large-scale networks when millions of policy decisions are required every minute for complex services. Policy-based management can potentially increase performance and scalability though the distribution of policies to multiple management nodes, but very little work has been done in this crucial area.

We are developing a policy-based service description language and management system called *PRONTO* [1] that manages dynamic and adaptive end-to-end services, whilst allowing for system evolution over time. Whilst some policies can be easily distributed according to different regions of the network, other policies will require coordination to ensure the correct sequencing of event, condition and action evaluation and information distribution. We now deal with more complex policies, involving control flow and looping. Sec. 2 briefly discusses the management system, and Sec. 3 and 4 discuss the existing work and the proposed extensions respectively. Sec. 5 presents the results of our discrete event simulator, and Sec. 6 examines related work.

2 The *PRONTO* Management System and Language

The *PRONTO* management system allows the specification of *dynamic* and *adaptive* services through a policy-based *service description language*. This

J. Schönwälder and J. Serrat (Eds.): DSOM 2005, LNCS 3775, pp. 269–280, 2005.

language allows services to be readily changed or replaced as market requirements dictate. Each service description defines the parameters, roles (e.g. devices and systems), resources, software components, and event-condition-action (ECA) policies involved in each service. Policies are written with regard to virtual model of the devices that has hierarchically structured model elements, or software components called *domain experts* that carry out different management tasks and strategies (at different levels of abstraction and according to feedback from the network). The domain experts are interchangeable, allowing Service Providers to change management functionality to suit their individual needs.

More complex policies are required in network management to have precise control over behaviour, particularly when different customers have conditional requirements, or when different behaviours need to be assigned to different devices in the network. *PRONTO* policies supports many additional actions that are not present in other policy languages, giving the policy writer greater flexibility in describing their service and network policies. An example service is shown in Fig. 1 for the management of a full mesh of MPLS LSPs. The LSP resources are allocated when the service is put into the **enable** state, and re-requested when any given Label Switch Path fails. The 'mpls' domain expert allocates the required resources using LDP signalling. Whilst this demonstrates the versatility of the language, implementing most complex behaviour in a policy-controlled domain expert will simplify the policy specification.

```
service /mplsService {
  params {
    required CEs: collection;
    n: int = CEs.size(); }
  components {
    mpls: [/mgmt/coreMplsDomainExpert]; }
  resources { LSPs[n*(n-1)]: LSPResource; }
  events { pathFail(path): LSPs[all].failure; }

  concurrent policies {
    on (enable) {
      // Setup full mesh of LSPs between CEs
      for (int i = 0; i <n; i++) {
        for (int j = i; j <n; j++) {
          LSPs[i*n+j].signalling = LDP;
          LSPs[i*n+j].source =
            mpls.selectPhysConn(CEs.get(i));
          LSPs[i*n+j].target =
            mpls.selectPhysConn(CEs.get(j));
          mpls.consume (LSPs[i*n+j]);
  } } }

    on (pathFail) {
      // Release the existing path
      mpls.release (path);
      // Initiate another LDP path
      mpls.consume (path);
} } }
```

Fig. 1. Management of VPN service

3 Policy Distribution and Coordination

Some complex dynamic services, though, will put incredible demands on centralised PBNM systems. For example, the admission control function on an pay-per-channel IPTV service could be swamped with hundreds or thousands of policy decision requests during commercial breaks as users change channels repeatedly. In these cases, it is desirable to distribute such decisions to increase the

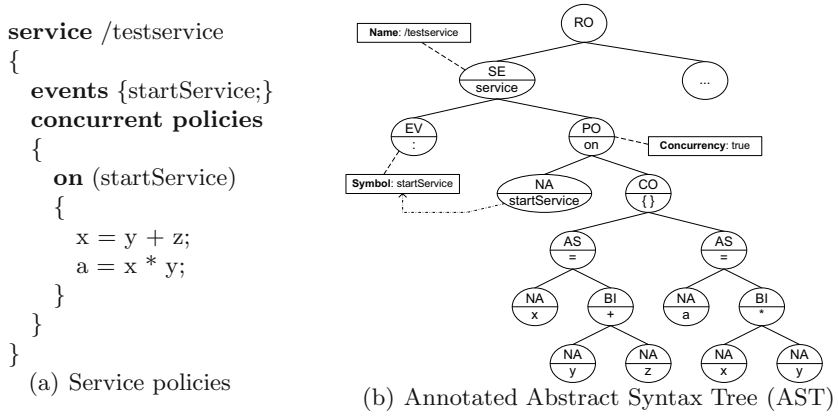


Fig. 2. Service to AST transformation

potential management load that can be handled. Our earlier work [2] described how the principles of *automatic parallelising compilers* and *Abstract Syntax Tree (AST) partitioning* could be used to distribute and coordinate policies to management nodes that are assigned the responsibility of different geographic partitions of the network. By adjusting the demarcation, the policy load on different management nodes can be adjusted. We give an overview of the basic algorithm here, but for simplicity, we only show the distribution and coordination of actions - events and conditions are treated similarly.

The **sequential** and **concurrent** keywords can be applied at different granularities to policies, to allow policy execution to be parallelised where required. However, *sequential execution semantics* are still obeyed for correct evaluation. Policies without inter-dependencies are simply distributed, as coordination is not required. Fig. 2(a) shows two policy actions, where there is a dependency between the expressions, enforcing sequential execution. We build an AST (similar to a normal compiler) as shown in Fig. 2(b): each node is marked with a type (two-letter abbreviation)[1] and the equivalent token text.

The different branches of the tree are partitioned by examining the name expression (NA) nodes that define different symbols (e.g. devices and their model elements, variable names). There are three major types of coordination messages: *data distributions* (DI) when data is exchanged, *sequence points* (SP) when sequential ordering is important, and *event notifications* (EN) to initiate distributed policy execution. We annotate the AST with additional DI and SP nodes, transforming it into a graph, so as to indicate where coordination is required between the AST partitions.

Using *depth-first traversal* to emulate sequential execution ordering of each policy action, we mark each AST node with directional arrows to indicate re-

[1] The nodes shown in this figure are as follows: Root (RO), Service (SE), Event (EV), Policy (PO), Compound Statement (CO), Assignment (AS), Name Expression (NA) and Binary Operator (BI).

gions of responsibility around different NA nodes. Descendants of NA nodes are marked with up arrows, and if all the children of any unmarked node have the same responsibility, they are marked with a down arrow. Any other unmarked nodes are assigned up arrows (or left arrows in the case of assignment operators). We then insert DI nodes between the partitions of the AST to indicate that some data needs to be exchanged between two or more management nodes, such as a calculated value or data that is retrieved from a device.

Whilst this simple process identifies some simple data dependencies, other types of dependencies go unnoticed. Wolfe [4] defines three types of dependencies: *flow dependence* when an assigned variable is later used; *anti-dependence* where a variable is used, and then reassigned; and *output dependence* when a variable is assigned twice at two different points in time. We perform a modified IN/OUT set analysis [4] to identify these dependencies; values and references are distinguished (INVAL and OUTVAL; INREF and OUTREF), and different parts of statements rather than whole statements are analysed to maximise parallelisation.

A post-order depth-first traversal is used to simulate the execution of policy expressions and statements in an sequential interpreter, and a *backpatch list* is maintained to keep track of prior IN/OUT markings, which are generated as each NA node (i.e. symbolic name) is encountered. The backpatch list stores pointers to the most recent OUTVAL for each symbol, and all INVAL/INREF markings since that NA node. Flow dependencies occur where an OUTVAL is followed by an INVAL for a symbol. A new DI node is inserted between these NA nodes to indicate this data dependency. Anti- and output dependencies occur when INVAL is followed by an OUTVAL, or OUTVAL is followed by another OUTVAL for any given non-temporary[2] symbol. New SP nodes are inserted between these NA nodes to indicate that sequencing of these actions is important, due to side-effects. Finally, setting references (i.e. OUTREF) results in one symbol *aliasing* another, and the backpatch list must maintain a pointer to the other symbol record to ensure that the history of IN/OUT markings for the correct symbol are used. Fig. 3 shows the semantic execution order and the generated IN/OUT markings and Fig. 4 shows the resulting dependency graph.

4 Extensions to the Distribution Algorithm

4.1 Control Flow Statements and Expressions

PRONTO supports statements and operators that require strict sequential evaluation order, such as **if-then-else**, **switch**, **break** and **continue** statements. It also supports exception handling, with a **throw** statement altering control flow. We continue to use sequence points, as these avoid the need for global synchronisation locking across all management nodes. Rather, sequencing only occurs between nodes that are directly inter-related.

[2] Temporary variables are implicitly duplicated, eliminating dependencies.

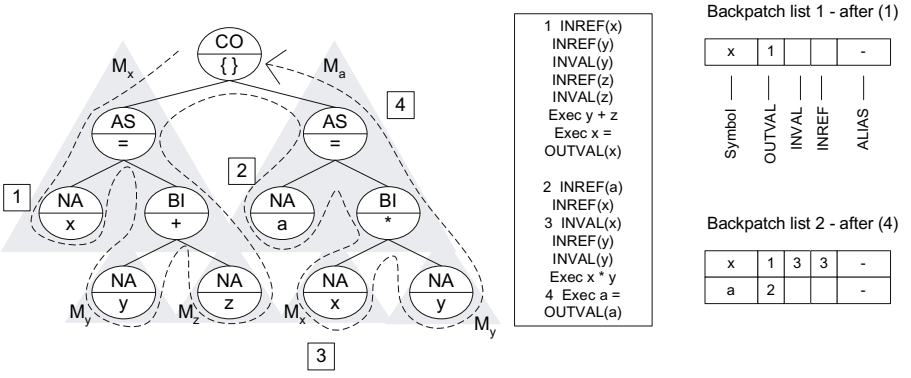


Fig. 3. Semantic execution order, IN/OUT markers and backpatching

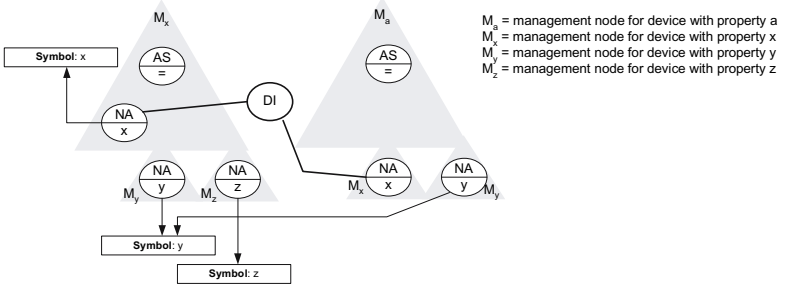
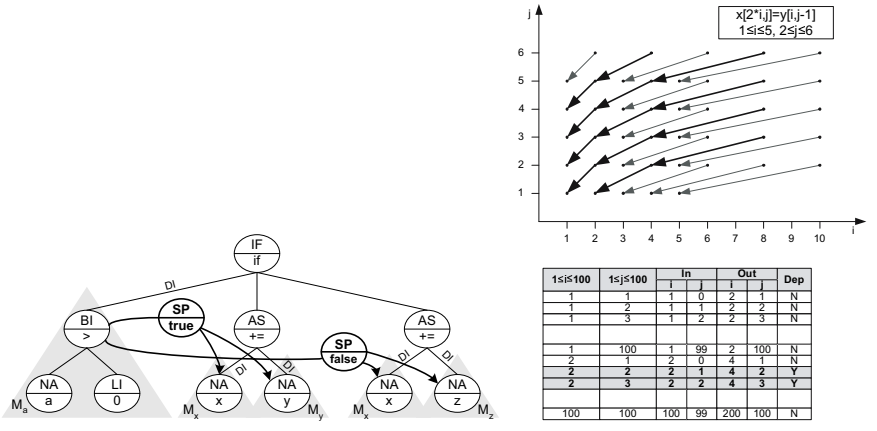


Fig. 4. Dependency graph

In the case of **if-then-else** or **switch** statements, a SP node is inserted to connect the conditional expression to any children of the statement or expression nodes that are conditionally executed, whenever a partition boundary is crossed (i.e. at an inserted DI node). All partitions hierarchically down the tree must be associated with an appropriate SP node. Each SP node is then marked with information describing its purpose. In Fig. 5(a), the **if-then-else** (IF) node has up to three child nodes - the evaluated boolean expression, and optional true and false outcome statements. Once the conditional expression is evaluated, other management nodes are notified of the outcome. Management node M_a must notify M_x and M_y when the condition evaluates to true, or notify M_x and M_z when the condition evaluates to false. In the case of **break**, **continue** and **throw** statements, SP nodes can also be inserted into the AST to ensure that any partitions containing statements following the conditional logic are not executed, based on the outcome of the conditional expression.

4.2 Loops with Counters

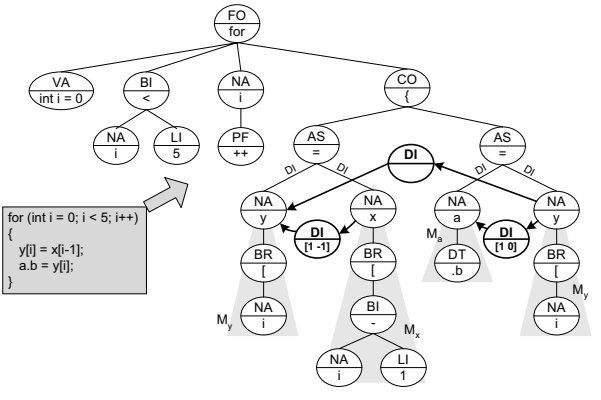
Loops can often be unravelled or sub-divided to maximise parallelisation [3], depending on the presence of any *loop-carried dependencies*. The *PRONTO* lan-



(a) **if** statement AST

1≤i≤100	1≤j≤100	In		Out		Dep
		i	j	i	j	
1	1	1	0	2	1	N
1	2	1	1	2	2	N
1	3	1	2	2	3	N
1	100	1	99	2	100	N
2	1	2	0	4	1	N
2	**2**	**2**	**1**	**4**	**2**	**Y**
2	**3**	**2**	**2**	**4**	**3**	**Y**
100	100	100	99	200	100	N

(b) Iteration spaces and tables



Loop Symbol List

Symbol	Count	Local?	INVAL	OUTVAL	Pre-dep	Post-dep	Deps
y[i]	N	N	Y	Y	N	N	i
x[i-1]	N	N	Y	Y	N	N	i
a.b	N	N	N	Y	N	N	
i	Y	Y	Y	Y	N	N	

(c) **for** loop AST

Fig. 5. Complex policies and SP/DI insertion

guage has four types of loops: **for**, **foreach**, **while** and **do..while**. Different loops have various levels of complexity, so we deal with the most common varieties and revert to sequential execution when it is too difficult to predict loop behaviour. The *counter symbols* which are modified in each iteration (with OUTVAL markings) are identified, as loop-carried dependencies are reliant on the counters and are modified during loop iterations. Dependency analysis then then used to determine a suitable execution ordering of the partitions within the loop.

Firstly, we build a table of symbols contained within the loop, identifying if they are local or global symbols, if they result in INVAL or OUTVAL markings, if they have pre- or post-loop dependencies, and if they depend on other symbols (e.g. the counters). An example is shown in Fig. 5(c). Symbols updated in every

iteration with no other dependencies are treated as counters[3]. We must then identify the expressions that lead to changes in the counters. If these expressions do not rely on any other symbols, then the counter behaviour can be predicted in advance, allowing localised counters at each management node.

The second stage is to identify relationships between the symbols dependent on the counters, and which loop iterations have dependencies on other iterations. To cover the majority of simple loops, we focus on array indices of the form $a_1 i + a_2 j + \ldots + a_m n + b$ when retrieving values set in previous loop iterations (i.e. INVAL operations). $i, j, \ldots, n$ are the counter variables and $a_1, \ldots, a_m, b$ are constants, and these values are stored in matrix $M = [a_1 \ \ldots \ a_m \ b]$. For example, the expression $x[2 * i - 1]$ with counters i and j has the values $a_1 = 2$, $a_2 = 0, b = -1$ and $m = 2$. Using the constructed symbol table and by examining the INVAL and OUTVAL markings on symbols (in relation to counter variable updates), we can identify flow dependencies between iterations of the loop. If the dependency is between distinct AST partitions, DI nodes are inserted into the graph between the NA nodes, indicating the direction of the flow dependency and the matrix M. The *antecedent* management node must notify the *dependent* management node of the calculated value when complete. For an example using matrices M, see Fig. 5(c).

In other cases where array indices do not fit a linear equation, or where the index equation is used on an OUTVAL operation, we can model an *iteration space* [4], simulating the loop and keeping a list of counter values that lead to dependencies. The list is then kept with the DI node added between NA nodes. The antecedent management node consults the list and notifies the dependent management node when a dependency exists - see Fig 5(b). The difficulty with this scheme is the potential table size, but non-shaded entries can be eliminated after dependencies are identified.

Special care is required when previous indexed variables are not set during the loop itself, according to sequential ordering. The expression $y[i] = y[i + 1]$ would access prior values for monotonically increasing i. We also have to ensure that prior values are retrieved before being overwritten. We can insert additional SP nodes into the AST, annotated with either the M matrix or an iteration table. Another difficulty is counters being modified in the middle of the loop - the first and last iterations are shortened, and execution of each partition must occur with the correct counter values.

4.3 Loops with Conditions

Loops based on conditional expressions require the expression to be evaluated before each iteration executes. An SP node is inserted to ensure that partitions contained within the loop are not evaluated prior to the conditional test, with notification being given on the first run and when the condition changes. This is similar to the control flow approach above. There are some difficulties though:

[3] We do not currently deal with complex counters with multiple updates per iteration or with dependencies.

how do we guarantee the loop is executed the correct number of times if there are side-effects between the conditional expression and the loop actions?

In these situations, we can enforce sequential ordering by using additional SP nodes. If the conditional expression or the loop actions are known to have side-effects that influence each other, or if it is unknown, then lock-step notifications are needed between the management nodes. This ensures that the condition is evaluated prior to other loop partitions and vice versa, so that execution occurs the correct number of times for both. Other more optimistic approaches could evaluate the conditional expression when only some of the loop actions have finished executing, or allow evaluation of the conditional expression whilst the other partitions are not lagging too far behind.

We anticipate that we might have access to additional *meta-data*, that would be beneficial in determining when side-effects might occur. Such meta-data, described as part of the management models or contained within descriptors for each vendor's devices, could indicate when property $dev001.x$ has an impact on property $dev001.y$, and would be used similarly to other meta-data for detecting and resolving conflicts. This is an area of future investigation.

4.4 Distributed Execution

Different management nodes take responsibility for different sets of devices in the network and the associated policies, and most policies are carried out autonomously. The EN, DI and SP messages must be exchanged at suitable times so that the behaviour is coordinated appropriately, depending on the policy complexity. EN messages must be sent to all participating management nodes whenever any given node recognises that a policy has been triggered, due to conditions or events that have been evaluated. The source of the EN message creates a unique execution identifier to ensure that each the policy executes exactly once, and so that later SP/DI messages apply to the same execution instance.

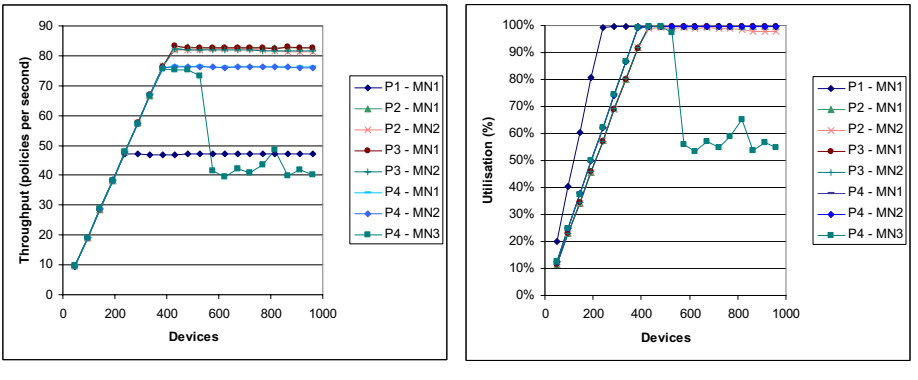
Once a management node receives a EN message, it can start executing any AST partitions that is has responsibility for, as long as they do not require an SP message (i.e. some other actions must occur first). If a DI message is required, the partition can commence execution, but the execution will halt until information that needs to be exchanged is received. Execution of the AST partition at each management node occurs independently, either sequentially or concurrently, depending on the current policy. However, localised analysis must be done, to ensure that concurrent statements are executed correctly, in the same manner as for policies that are distributed across multiple nodes. As each AST partition finishes executing, any outgoing SP and DI messages must be sent if they go to other management nodes. If the messages go to the same node, then other partitions can be triggered or finished locally. DI messages must contain calculated values which are to be exchanged between management nodes. Once each management node completes its own AST partitions, the policy finishes at each node.

5 Simulation

We are constructing a distributed policy simulator (based on OMNET++) to predict and examine the anticipated performance of this algorithm with different types of policies, with unique network and management architectures and different dynamic loads. The simulator models *execution units* representing the different partitions of the AST. Management nodes are informed of their responsibilities, and then retrieve policies and other data from a centralised relational database with ideal, localised caching at each management node. Devices generate events using an exponential distribution, resulting in policies being executed across one or more management nodes. We model the exchange of EN, DI and SP messages during execution. Each management node has a *coordinator* and *executor* with thread-pooling architectures.

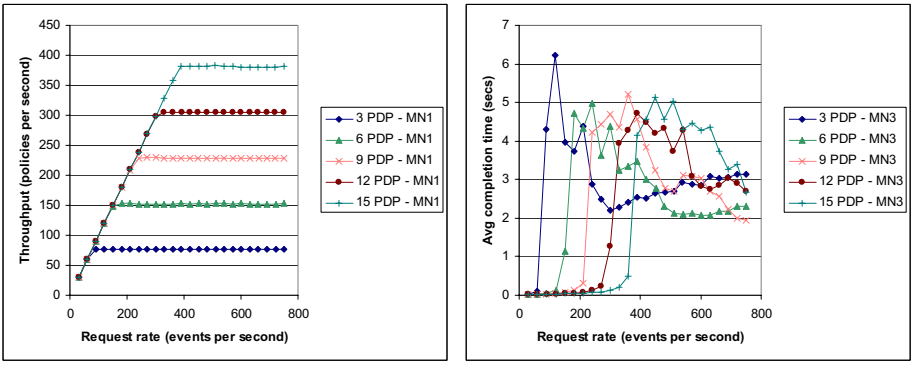
Although the simulation system is not complete, we have been able to get some early simulation results based on preliminary real-system measurements and estimated model characteristics. Our initial profile consists of: device response (uniform: $\mu = 3.431ms$, $\sigma = 799.8\mu s$), database response (uniform: $\mu = 1.445ms$, $\sigma = 1.6\mu s$), cache response (exp: $\mu = 500\mu s$), coordination delay (exp: $\mu = 500\mu s$) and execution delay (exp: $\mu = 5ms$). The simulation allows the calculation of the *utilisation*, *throughput* and *response time* (measured from the initial event timestamp) at each management node.

To evaluate performance, we simulate an increasing device load, with devices sending events every 5 seconds. As seen in Fig 6(a), four types of policies result in different load capacities. P1 involves setting a device property from a single node. P2 involves retrieving a device property on one node, followed by writing a property on a second node using a *sequence point*. P3 is identical to P2, except the value is exchanged between the nodes by using a *data distribution* (DI) message. P4 models retrieving two device properties on management nodes 1 and 2, followed by writing of a device property on management node 3.



(a) Throughput (b) Utilisation

Fig. 6. Performance of single management node set



(a) Throughput (b) Response time

Fig. 7. Scalability of management node sets

Increasing the number of management nodes increases event load capacity, but leads to a worsening of response time and capacity per node, due to coordination overheads. Furthermore, the final management node (MN3) under policy P4 is heavily overloaded by multiplexing competing events from MN1 and MN2, although selective discarding of queued messages improves the throughput and utilisation responses under excessive load conditions.

In Fig. 7(a) and Fig. 7(b), we examine scalability by simulating 540 devices with varying request rates on policy P4 (2 x reads + 1 x write over 3 management nodes), with equal device load on each management node. Here, the throughput clearly benefits from increasing the number of nodes. The response time is slightly worse with more nodes, but higher event loads are handled.

These preliminary results are promising, since they show that the distributed algorithm has the potential to scale easily, simply by adding management nodes and distributing the device load where necessary to alleviate bottlenecks. Additional nodes increases the overall throughput and load handling with a minimal impact on response time. We estimate that a single management node could handle approximately 47 policy events per second, or 240 devices with an event every 5 seconds. This compares well against the 20 events/s reported by Ponnappann et al. [5]. Eddie and Law [6] claim 400 events/s per node, although it is unclear whether caching is used and how the directory or database server is accessed.

The throughput appears to be highly sensitive to changes in the processing delays involved within each node. However, there are many factors that have an influence on performance - the complexity of the policies, the choice of language constructs used and the network and management architectures. Complex policies tend to increase the number of SP and DI coordination messages needed, and this has implications for response times. Ideally, the simulator will help in building a database of the metric cost of different policy constructs, or assist engineers in the design of dynamic and adaptive services that can meet oper-

ational load demands. Several improvements to the simulation are yet to be made, to more accurately model real conditions and more complex policies (as discussed in this paper) with different coordination strategies (e.g. *optimistic* vs. *pessimistic*).

6 Related Work

Policy distribution is occasionally mentioned in the literature but coordination has gone largely unnoticed.

The IETF policy architecture allows for some distribution through replicating PDPs and PEPs, but the responsibilities for policies must be manually configured. If two or more PDPs attempt to control the same device with overlapping configurations, the results would be unpredictable. Hamada et al. [7] attempted to increase scalability using hierarchical LDAP repositories and multicasted updates. Law and Saxena [6] also increased the scalability of the IETF architecture using transparent load-balancing agents. Corrente [8] showed that the PDP was a bottleneck in COPS-PR protocol handling. Strassner [9] introduces the concept of a *policy broker* for communication, but provides only limited suggestions for how conflict management should take place. Wies et al. [10] uses *management by delegation* to distribute policy management to extensible intelligent agents to increase scalability, but he does not characterise the global conflict resolution module proposed.

Howard et al. [11] provided separate event and policy statements, and these were deployed to separate management components by a Policy Distribution Service (PDS). The Policy Validation Service (PVS) generated events according to conditions that needed to be watched on the managed objects. However, they do not suggest how particular management components are selected. Similarly, Koch et al. [15] had separate event and policy languages, and policy objects were distributed to monitoring agents to poll managed objects and generate events when appropriate. However, a method of distributing policy actions was not discussed. Marriott and Sloman [12] distributed policy information and TCL scripts to obligation management agents using CORBA. Later work by Dulay et al. [13] allowed Ponder policies to be distributed to Policy Management Agents (PMAs) using Java RMI. However, both these two approaches require the agents to be identified by the *subject* in the policy, limiting the migration of policies between agents. Kohli and Lobo [14] created policy elements from PDL policies, which could be hosted between multiple policy servers to perform coordination. However, no clear method of doing this was specified.

7 Conclusion

We have demonstrated an algorithm for distributing policies amongst a number of management nodes based on geographical segregation, and extended this algorithm for flow control, loops and exception handling. Our early simulation results show that this approach offers great potential for scalability, with only

a marginal impact on response time when management nodes are correctly provisioned for the anticipated load. The algorithm and its simulator will be immensely helpful in the design of complex, adaptive and dynamic services based on policy-based management techniques. We would like to acknowledge the generous financial support of Alcatel Australia and the Australian Research Council through Linkage Grant LP0219784.

References

1. Sheridan-Smith, N., Leaney, J., O'Neill, T., and Hunter, M.: A Policy-Driven Autonomous System for Evolutive and Adaptive Management of Complex Services and Networks. Eng. Comp. Based Sys. (ECBS 2005)
2. Sheridan-Smith, N., O'Neill, T., Leaney, J., and Hunter, M.: Distribution and Coordination of Policies for Large-scale Service Management. LANOMS (2005)
3. Grune, D., Bal, H. E., and Jacobs, C. J. H.: Modern Compiler Design. John Wiley & Sons, West Sussex (2000)
4. Wolfe, M.: High Performance Compilers for Parallel Computing. 1st edn. Addison-Wesley, Redwood City CA (1996)
5. Ponnappan, A., Yang, L., Pillai, R., and Braun, P.: A Policy Based QoS Management System for the IntServ/DiffServ Based Internet. POLICY (2002)
6. Law, K. L. E., and Saxena, A.: Scalable Design of a Policy-Based Management System and its Performance. IEEE Comm. Mag. **41**(6) (2003) 72-79
7. Hamada, T., Czezowski, P., and Chujo, T.: Policy-based Management for Enterprise and Carrier IP Networking. Fujitsu Science and Technology **36**(2) (December 2000) 128-39
8. Corrente, A., De Bernardi, M., Rinaldi, R.: Policy Provisioning Performance Evaluation using COPS-PR in a policy based network. IM (2003)
9. Strassner, S.: Policy-Based Network Management: Solutions for the Next Generation. Morgan Kaufmann. ISBN 1-55860-859-1 (2003)
10. Wies, R., Mountzia, M.-A., and Steenekamp, P.: A practical approach towards a distributed and flexible realization of policies using intelligent agents. DSOM (1997)
11. Howard, S., Lutfiyya, H., Katchabaw, M., Bauer, M.: Supporting Dynamic Policy Change Using CORBA System Management Facilities. IM (1997) 527-38
12. Marriott, D., Sloman, M.: Implementation of a management agent for Interpreting obligation policy. DSOM (1996)
13. Dulay, N., Lupu, E., Sloman, M., Damianou, N.: A Policy Deployment Model for the Ponder Language. IM (2001) 529-43
14. Kohli, M., Lobo, J.: Policy Based Management of Telecommunication Networks. Policy Workshop (1999)
15. Koch, T., Krell, C., Kramer, B.: Policy Definition Language for Automated Management of Distributed Systems. Systems Management (1996)