Extensible, Transactional Architecture for IP Connectivity Management

Eduardo Grampín[1] , Javier Baliosian[2] , Joan Serrat[3]

Abstract

This paper presents an extensible, transaction oriented Element Manager System (EMS) architecture for IP connectivity management. The key aspects of the proposal are the extensibility, derived from the adoption of an object oriented, policy-based approach, and the transactional feature, which ensures both EMS repository data information and network configuration integrity. This means that an intended Connectivity Service is configured end to end in one atomic operation, as seen by the Network Management System (NMS), or not configured at all, if any of the steps of the ongoing operation fails. Management data and policies are stored as persistent objects in a common repository. This provides not only an inventory of the managed elements, but also a policy-driven application environment. The paper briefly describes the Information Model, the solution adopted, and presents a proof of concept implementation.

Keywords: IP management, policies, transactions

1. Introduction

Traditional connection-oriented telecommunications networks have adopted hierarchical, object oriented management models based on International Telecommunication Union (ITU-T) Telecommunications Management Network (TMN) [1] principles, which have been very useful for the telecommunications industry in the last decade. On the other hand, provisioning of IP networks has historically relied on control plane (i.e. routing and signalling protocols), with little significance of the management plane. The traditional Simple Network Management Protocol (SNMP) management model has proven to be enough in these best-effort networks. IP has been adopted as the technology of choice for service delivery among individual and corporate users, who demand service assurance with predictable levels of Quality of Service (QoS). The Multiprotocol Label Switching (MPLS) architecture [2] enables the application of Traffic Engineering techniques in IP networks, like explicit routing with Quality of Service (QoS) guarantees, to fulfil the requisites of IP applications. The adoption of an intelligent Control Plane help operators willing to deploy IP Connectivity Services over an MPLS infrastructure. Nevertheless, MPLS Operation, Administration and Maintenance (OAM) tools are scarce and have limited functionality. Therefore, there is a need for management models and applications to promote a widespread adoption of the MPLS model.

EMSs usually adapt proprietary management tools (e.g. SNMP-based) to the specifications of the Network Management Layer-Element Manager Layer (NML-EML) interface. These adaptations are usually not flexible and cannot easily accommodate changes on network devices. Therefore a requisite of modern EMSs would be their capability to be dynamically adapted to those changes (i.e. without the need to rewrite and recompile the adaptation code). Establishing a connection in the IP/MPLS layer requires the issue of several requests from the EMS to different routers under its domain. These requests must be in order and successfully executed; otherwise the system should be restored to the previous state, to avoid inconsistencies (and potential problems with network connectivity). Therefore, the set of operations intended to fulfil a connectivity request can be considered a transaction, and shall be treated as such.

With these two requirements in mind we propose to use policies and database technology to conceive an EMS solution that can be used in a variety of conditions. The adaptation mechanism is driven by policies, while ideas adopted from database technology are used to confer the transaction-oriented property to the whole process of connectivity establishment, modification or release.

After this section, the paper continues with section 2 and 3 which describe the EMS architecture. Section 4 presents a proof of concept implementation. Some concluding remarks are given at the end of the paper.

[1]Eduardo Grampín is with Universidad de la República, Montevideo, Uruguay. E-mail: grampin@tsc.upc.es
[2]Javier Baliosian is with Universitat Politècnica de Catalunya, Barcelona, Spain. E-mail: jbaliosian@tsc.upc.es
[3]Joan Serrat is with Universitat Politècnica de Catalunya, Barcelona, Spain. E-mail: serrat@tsc.upc.es

2. Proposed EMS Architecture

The functional architecture of the proposed policy-based EMS solution is depicted in Figure 1.

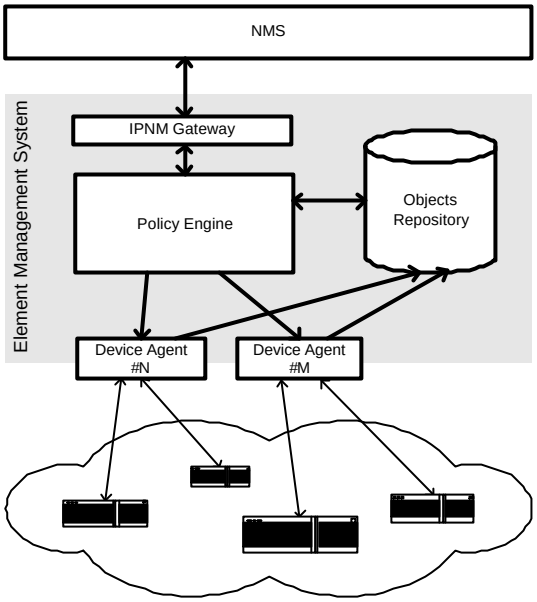

Figure 1 - Architecture of the NMS-EMS Interface

The Gateway component provides the adaptation for the upper layers in the TMN hierarchy, showing an standard interface to the NML and an internal, proprietary interface to the Policy Engine, enabling the extension of our data model (that is, to change the behaviour of the EMS in respect to the underlying network elements) without changing the view from the NML. The Repository is the heart of the system. It holds the persistent Managed Objects and Policy Objects, which are used by the Policy Engine to drive the management application. The Device Agents adapt the generic representation of the network to device-specific syntax, and provide functionality for communication with devices. The major components are described below.

2.1 IPNM Gateway

The Multi-Technology Network Management (MTNM) [3], is a technology-independent specification from the TeleManagement Forum (TMF) that enables multi-vendor provisioning on the Element Management Layer (EML). It is strongly connection-oriented, mainly targeted ATM and SDH networks. The adaptation of MTNM to the IP/MPLS Network Connectivity Model has proven to be quite cumbersome [4]. Another TMF initiative, the IP Network Management (IPNM) NML-EML interface specification [5], is much closer to the IP/MPLS model, maintaining the core MTNM concepts. The use of this interface allows the proposed EMS architecture to transparently integrate into the whole TMN/TMF framework of high-level management application.

Note that the EMS shows the above-mentioned interface to the upper layers, but is free to map this Information Model as needed towards its internal architecture. In this section we will briefly describe the IPNM model and operations. The internal object model is described in the Objects Repository section.

2.1.1 IPNM Information Model and Operations

The managed network is assumed to offer basic connectivity and to be MPLS enabled. In response to IP Connectivity Services requests from the upper layers of the management system, the EMS performs the configuration of LSPs with QoS constraints. The routing computation for such LSPs is done by the NMS using the appropriate Constraint Based Routing algorithms; the Control Plane can provide dynamic backup for these LSPs in case of failure (that is, the establishment of the connection through a different route). The IP EMS supports a subset of the IPNM “get” operations that respond to requests of information from the upper layers.

The Information Model is an extension of the IPNM interface. The fundamental add-on of the model is to extend the generic classes with MPLS-related information. The result of this extension is a model that describes the IP/MPLS network with enough detail to establish Connectivity Services. We consider a restricted model which main entity is the MPLS Connectivity Service (MPLS-CS), supported by SubNetwork Connections (SNCs).

Among the managed entities of the model shown by the EMS to the NMS it is worth mentioning the following:

- *MPLS Connectivity Service (MPLS-CS)* is the basic service over the MPLS network, meeting some QoS constraints resembling a connection-oriented circuit.
- *MPLS SubNetwork* is an abstraction provided by the EMS to upper layers in the management hierarchy. The Subnetwork is the unit of work of an EMS system. SNCs are provided within subnetworks.
- *MPLS SubNetwork Connection (SNC)* relates MPLS Connection Termination Points. A Network Connection provides a transparent end-to-end connection through or within a subnetwork, equivalent to a portion of one MPLS path.
- *MPLS Physical Termination Point* is an actual or potential endpoint of a topological (physical) link. Essentially, this is a representation of a physical port, belonging to a Label Switch Router (LSR).
- *MPLS Managed Element (MPLS-ME)* is an abstract class used to represent Label Switched Routers.

The SNC operations are:

- *createSNC* configures the SNC in the MPLS infrastructure.
- *activateSNC* activates the previously created SNC; this includes the assignment of traffic to the configured SNC.
- *createAndActivateSNC* executes the both previous operations together.
- *deactivateSNC* deactivates the target SNC but keeps configuration information.
- *deleteSNC* deletes the SNC configuration information from network elements.

2.2 Policy Engine

Many policy languages and frameworks can be used as support platforms for policy-based management. Several Internet Engineering Task Force (IETF) Working Groups (WGs) are involved in this field. Well known models like PCIM and extensions, and protocols like COPS are major achievements of these groups. There are some commercial implementations based on the IETF Three Tier Policy model, that cope with the problem of definition of Per Hop Behaviours on devices but fail to provide a solution for the basic problem of point-to-point connectivity provisioning with QoS constraints.

Among several options we selected Ponder[4] as the language to support the policy-based management system. The Ponder model includes a declarative language to specify policies of different types, and a deployment model [6] to make use of these policies. It also provides a policy compiler that transform the policy definitions into objects that are subsequently stored in a Lightweight Directory Access Protocol (LDAP) repository.

2.2.1 Policy Engine Implementation

The Policy Engine processes the IPNM operations (translated by the IPNM Gateway) and provides the management functionality using the persistent Managed Objects and Policy Objects. The internal architecture of the Policy Engine and its interactions with other components are illustrated in Figure 2.

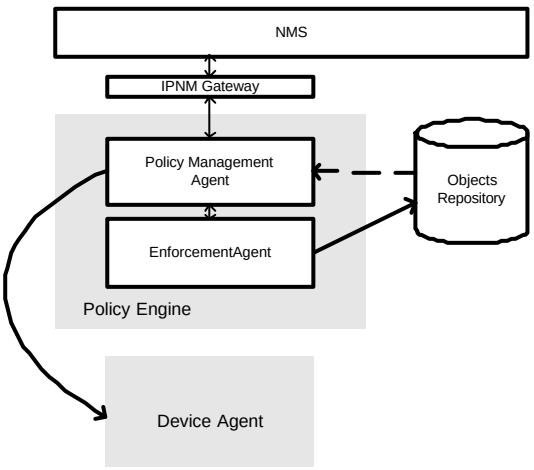

Figure 2 - Policy Engine Internal Architecture

[4] Besides this option others can be used and therefore the solution is not restricted to this language.

As we can infer from this figure, most of the work of the Policy Engine is performed on the network abstract representation stored in the *Objects Repository*. The *Device Agent* is requested to carry out the configuration of the network elements in an atomic operation.

2.2.2 Extensibility of the Management Functionality

Every action executed by the interface is policy-driven. This means that the system changes its behaviour dynamically in accordance with the attributes and methods of the management objects and system policies. Modifying or adding policies and managed objects is the way to reach the extension of management functionality.

The extensibility concept is a result of well-known strengths of object-oriented modelling. In this case, dynamic binding allows us to define operations for one object and then share the specification of the operation with other objects. These objects can further extend this operation to provide behaviours that are unique to those objects. At runtime, dynamic binding determines which of these operations are actually executed, depending on the class of the object requested to perform the operation.

The Policy Management Agent functionality is complemented by the Enforcement Agents. Given a particular policy and its targets, these agents can navigate the LDAP repository and instantiate actions contained in such targets. Then, specific functionality stored as a persistent object can be recalled by policies. This functionality may be completely unknown to the Policy Engine at compilation time; at runtime such functionality can be augmented.

2.3 EMS Information Model

The internal information model maps the IPNM model described above. Some relevant internal entities are the MPLS tunnels (*Tunnel* Class), which reside in a router (the *Router* Class) and are characterised by a traffic descriptor (*TrafficDescriptor* Class), built from their FEC (Forwarding Class Equivalence) and QoSAttributes. Every entity has its own set of operations that, as we will see, can be used as actions by applicable policies. A simplified view of the model is depicted in Figure 3. Note that the proposed policy-based model can be extended as new requirements appear. This is reflected by the *ToSpecialize* Class of the model.

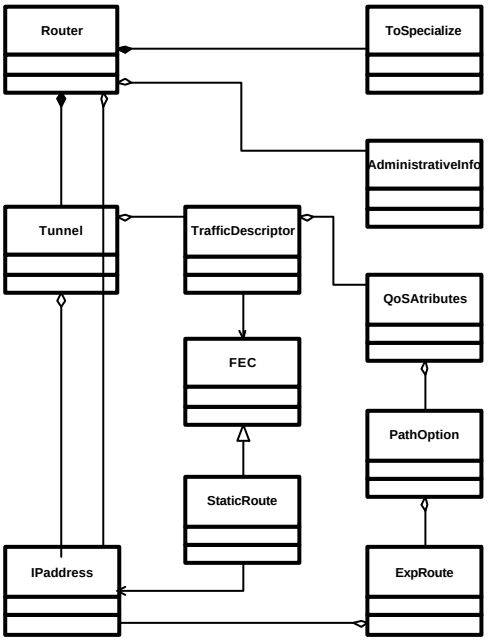

Figure 3 - Simplified EMS Information Model

2.3.1 LDAP Implementation of the Object Model

There are two kind of objects stored in the repository: the Policy Objects and the Managed Objects. There are many ways to replicate the model hierarchy into the repository tree. The managed objects can be stored in any

convenient way for the management application. (i.e. the persistent objects can be stored in "bulk" or de-aggregated into a given sub-tree, depending on the programmer's needs). A leaf of the LDAP tree could contain a complete Managed Object representation stored as a Java class, with its attributes and methods. A "fine grain" approach can also be used. Each element could be a node in the repository tree with recursive children that represent classes contained in the "main" class. For example, a SubNetwork Connection could be de-aggregated into MPLS-tunnels, which in turn could contain other attributes.

A main goal of our work is to recreate the usual *transaction* or *transactional operation* concept from databases in the configuration management context. A standard transactional operation performs a set of changes in an atomic way, and if the progression fails somehow, there is a rollback to the state previous to the transaction. This approach is used in our model, keeping consistency between network and inventory information, and avoiding useless operations on the elements. As we intended it, a transactional operation is not limited to one network element, but to the set of operations and elements needed to establish a SNC. This will be described in the Transactional Behaviour section.

2.4 Device Agent

The Device Agent (DA) is a mediator between the EMS and the managed elements. The DA takes care of the device-specific functionality. For example, a full SNMP configurable device could use a *GenericSnmp* DA, but a *MyRouter* brand proprietary router should use a *MyRouter* DA. The DAs shall provide the appropriate methods to access the device and change the configuration after translating the abstract representation to a device-specific syntax. In other words, the agent would understand the model and translate it into another device-specific model with its particular configuration methods[5].

As seen in Figure 1, the DAs are boundary entities. Some of their functionality could be built by third party developers. Figure 4 depicts the DAs' internal architecture. It can be seen that while the Translator Manager is an internal entity, the Communication and Discovery module can be thought of as an open interface, or Application Programming Interface (API).

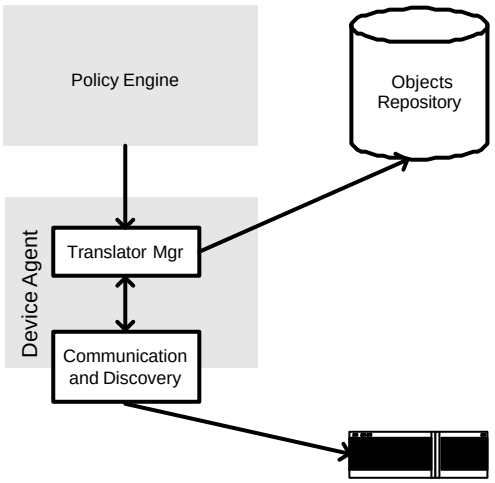

Figure 4 - Device Agent Architecture

Translator Mgr
This component is capable of understanding the OO abstract representation and translating it into device-specific syntax. This process may be triggered by the Policy Engine through a *Configure*() order. The translator accesses the Objects Repository and retrieves the relevant objects that need to be updated at the device level.
Communication and Discovery
This is a supporting component that provides communication methods with the network devices. It can be a basic SNMP API or a more sophisticated piece of software that supports CLI, HTTP, TFTP or other communication methods. The discovery function is responsible for maintaining information consistency. It performs this function by accessing the Translator module which in turn feeds the Repository with the updated information collected from the network devices. One candidate algorithm to perform the discovery task is the one proposed in [7].

[5] Note that this translation should be done back and forth.

3. Transactional Behaviour

One main goal of our proposal is to achieve repository consistency, in each element configuration and between those elements and the repository. This is analogous to the challenges faced by distributed database systems. Since the '80s this has being solved successfully with a Two-Phase Commit (2PC) protocol [8].

To achieve the transactional behaviour of our EMS architecture, the Policy Engine works as the *coordinator* in the 2PC protocol, and the Device Agents as the *cohorts*. The *coordinator* is responsible for trying the operations and commands the 2PC process.

In this paper we are addressing only the adaptation of the 2PC idea to the configuration of IP connectivity services, and although a complete adaptation to network configuration is a very complex task (subject of future work), the basic concepts are here.

LDAP server shall be enhanced with minimal transactional capabilities (as described in the Implementing Transactions in the Repository Section), and the Device Agents need the functionality to send notifications, as shown in Figure 5. An important change to the classic databases 2PC protocol is needed to cover the lack of transactional behaviour of Device Agents. The modified protocol is will be described further on.

3.1 EMS Two-Phase Commit Protocol Adaptation

In this section we describe our adaptation of the 2PC protocol, shown as a finite state diagram in Figure 5. The steps of the protocol are described bellow.

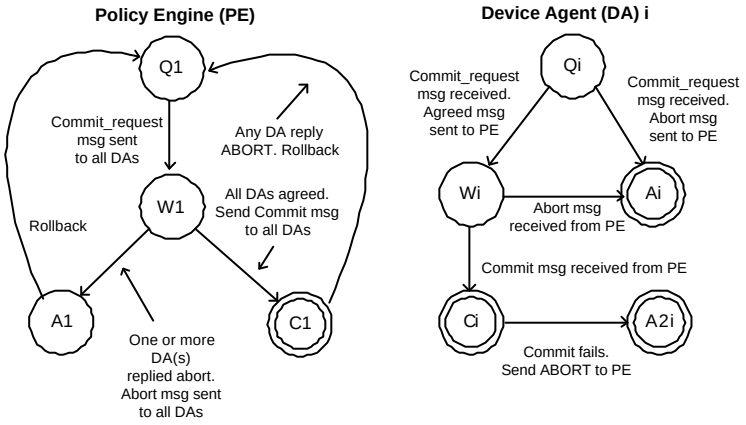

Figure 5 - Two-Phase Commit Finite State Diagram

At the Policy Engine (PE):

1. Send COMMIT-REQUEST message to each Device Agent (DA). The PE is now in the preparing transaction state.
2. Wait for responses from DAs. If any DA responds ABORT the transaction must be aborted, proceeding to step 5. If all DAs respond AGREED then the transaction may be committed, proceeding to step 3. If, after some time period any DAs do not respond, the PE can either transmit ABORT messages to all DAs or transmit COMMIT-REQUEST messages to the DAs that have not responded. In either case the PE will eventually go to state 3 or state 5.
3. Send COMMIT message to each of the DAs.
4. Wait for each DA to respond. If all DAs reply COMMITTED erase the original branch created in the LDAP as explained in the previous section. DONE, no more steps should be performed. If any DAs reply ABORT, erase the temporary sub-tree in the Repository, mark the original one as visible and start a new commit cycle with the original configuration to rollback to the previous stable network configuration[6]
5. Send the ABORT message to each DA. Recover the original configuration state in the Repository.

[6]We are supposing that the previous stable network configuration state will have no problems to be enforced. This hypothesis may prove not to be true, and should be considered for future research.

At Device Agents

1. When a COMMIT-REQUEST message is received for a t transaction, read the branch in the repository containing the abstract configuration representation of the elements under the responsibility of this DA. Translate it to *device dependent configuration*. Optionally, communication with devices could be tested as well. If those operations are successful, send AGREED to the PE, and perform step 2. Otherwise send ABORT. DONE.
2. If an ABORT message is received, then do nothing on the devices and erase the configuration information associated with the transaction read in step 1. DONE.
3. If a COMMIT message is received, then perform all the configuration operations obtained in the translation procedure in step 1. If it is already "committed", no further action is required.
4. If all the configuration operations in step 3 were successful, respond COMMITED to the PE. Otherwise respond ABORT. The configuration state of the managed elements could be inconsistent now, internally and with the Repository. This situation will be fixed later when the previous configuration state would be enforced by the Policy Engine.

3.2 Implementing Transactions in the Repository

An LDAP server is hierarchically organised, so it can be seen as a tree. To implement a transactional operation on the LDAP server, the following algorithm is proposed:

- First, the *minimal changed sub-tree* (the minimum sub-tree of the LDAP repository where changes will be performed) needs to be found.
- Once a transaction is started, a temporary copy of the minimal changed sub-tree is created. This is the analogue to the *do log* from relational databases; the original sub tree is marked as invisible to the Device Agents. Changes are written to this temporary sub tree.
- If the operation is successful, the original sub-tree is deleted from the repository without further operations.
- If the operation fails, the temporary sub-tree is erased and the original one is marked as visible (back to original state).

4. Proof of Concept

The implementation is programmed in Java using a standard SNMP API in the *Device Agent* component. The functionality is built from an object-oriented information model of the managed elements' configuration (depicted in Figure 3) and stored in an LDAP Server.

The actual implementation is based on Cisco routers, but it can be used to manage other vendors' equipment that supports the used SNMP MIBs. The application is able to configure MPLS LSPs with all of the functionality provided by the vendors' equipment using the described IPNM interface. A multi-vendor implementation would require specific *Device Agents*. It has been tested on a network composed by three Cisco 7204 core routers with MPLS and Traffic Engineering capabilities and several client routers and hosts. The management platform is a Linux PC. Though currently limited to Configuration Management, the implementation can be extended to support Fault and Performance Management, based on SNMP traps and MPLS-specific meters.

As stated before, the Policy Engine is based on the Policy Deployment Model. Regarding this model, the managed elements are the Targets of the policies and their methods are the Actions, such as *createTunnel*() and *deleteTunnel*(). The model can be extended specialising the *ToSpecialize* Class.

To illustrate the operation of the proposed EMS, let's review a *Create IP Connectivity Service* (ICS) scenario. The workflow of such request will depend on the objects instantiated, which in turn are defined by policies. When an action is defined in a policy, the correspondent method is instantiated in the relevant managed object. A couple of simplified obligation policies useful in this scenario are shown below (using Ponder syntax):

```
inst oblig /Private/Obligs/ics/set_tunnels {
      on createSNC(element_A, element_B);
      subject /Managers/GUI;
      do element_A.createTunnel(toElemet_B) ||
                  element_B.createTunnel(toElemet_A);
}
```

This policy determine the first step of an if-event-do-action cascade. The second step will be the one triggered by the createTunnel() event associated to the action with the same name. That event will match against the following policy:

```
inst oblig /Private/Obligs/ics/set_tunnel_elements {
      on createTunnel(Tunnel_ID, fromElement_A, toElement_B);
      subject /EMS/EnfAg;
      do element_A.Tunnel_ID.createTrafficDescriptor(FEC, QoSParams);
}
```

Following this method, all the needed steps for the ICS creation would be defined in the abstract model and finally, when a `connectionCreated()` event occurs, the entire new configuration will be committed on the elements.

Note that variations to the scenario can be created "on the fly" by modifying the applicable policies, without modification to the application's code.

5. Conclusions and Future Work

This paper presents an extensible, transaction oriented Element Manager System (EMS) architecture for IP connectivity management compliant with the IPNM interface. The core concepts of the proposal are the policy-driven application environment over persistent managed elements, and an approximation to a transactional behaviour regarding network inventory and configuration. The proposed model enables a specialisation of the Information Model at the EMS level, giving the applications the opportunity to manage vendor-independent IP Connectivity, using specific Device Agents where needed.

The proof-of-concept implementation gives us a good insight about the strength of the model. Further implementation using the policy-driven approach is being carried out simultaneously with a deeper specialisation of the model entities and in-depth study of the proposed transactional approach. Relevant future work shall include a comparison in terms of efficiency and flexibility against other proposals (i.e. Control Plane driven solutions).

6. Acknowledgements

Some of the ideas described in this paper are the result of the work undertaken in the context of the IST Project WINMAN, co-funded by the European Union. The authors would like to express their gratitude to the other members of the consortium.

References

[1] ITU-T, "M.3010, Principles of Telecommunications Management Network (TMN)," 1996.

[2] E. Rosen, A. Viswanathan, and R. Callon, *Multiprotocol Label Switching Architecture*, Jan. 2001. RFC 3031.

[3] TMF 608, "Multi-Technology Network Management Information Agreement NML-EML Interface." Version 2.0. TM Forum Approved , October 2001.

[4] IST 13305 - WINMAN, "WDM and IP Network MANagement." http: //www.winman.org.

[5] TMF 611, "IP Network Management Information Agreement NML-EML Interface," April 2002.

[6] IEEE/IFIP International Symposium on Integrated Network Management (IM'200)1, *A Policy Deployment Model for the Ponder Language*, (Seattle), May 2001.

[7] H. C. Lin, S. C. Lai, P. W. Chen, and H. L. Lai, "Automatic Topology Discovery of IP Networks," *IEICE Trans. Inf. and Syst.*, vol. E83-D, January 2000.

[8] E. Moss, *Nested Transactions: An Approach to Reliable Distributed Computing*, pp. 31–38. Cambridge, Massachusetts: The MIT Press, 1985.