Managing Security in Dynamic Networks*

Alexander V. Konstantinou	Sandeep Bhatt
Yechiam Yemini	S. Rajagopalan
Computer Science Dept.	*Telcordia Technologies*
Columbia University	*(formerly Bellcore)*
New York, NY	*Morristown, NJ*

Abstract

This paper describes our initial steps towards self-configuring mechanisms for automating high-level security and service policies in dynamic networks. We build on the NESTOR system developed at Columbia University for instrumenting and monitoring constraints on network elements and services such as DHCP, DNS zones, host-based access controls, firewalls, and VLAN switches.

Current paradigms for configuration management require that changes be propagated either manually or via low-level scripts suited to static networks. Our longer-term goal is to provide fully automated techniques which work for dynamic networks in which changes are frequent and often unanticipated. Automated approaches, such as ours, are the only viable solution for global and dynamic networks and services. In this paper, we focus on one specific scenario to illustrate our ideas: providing transparent and secure access to selected services from a mobile laptop. The challenge is that reconfiguration must satisfy the security policies of two independent corporate networks.

1 Introduction

As the technologies to deploy new internet services have progressed rapidly, the tools to manage them have lagged behind. This is especially true of security management. Sophisticated services are often vulnerable to unanticipated and sophisticated attacks. With inadequate management tools, the tendency is to deploy new services in a restricted manner that limits their usefulness.

Existing tools for static security are inadequate to meet the current demands of user mobility and diversity. These tools require frequent, expensive, and error-prone reconfigurations that may make legitimate access cumbersome and time consuming. This happens because the primary first-generation technique for reconfiguration — low-level scripting — cannot handle rapid change easily. There are no tools to verify the correctness or composability of scripts, properties critical for security. As a result, unpredictable security gaps can appear during changeovers. In a dynamic network with frequent upgrades, this uncertainty becomes untenable and leads to over or under-management of resources. The task of systems administration is increasingly human-intensive and administrators often must make decisions with little or no basis to justify their choice. The need for automation of configuration management is immediate. History also tells us that any security mechanisms that obstruct the legitimate use of services becomes unpopular and will eventually be bypassed. Balancing

*Presented at the 19th USENIX Systems Administration Conference (Lisa'99), Seattle, WA, USA, November 1999.

the demand of users for new services with the security vulnerabilities that the new services cause is an important and challenging problem.

The long-term goal of our project is a management platform that automates both the management of security and service availability. This paper describes our approach and initial steps towards this goal. We demonstrate our solution with a simple scenario in which a user moves from one network to another while expecting transparent access to services. This example illustrates the complexities of reconfiguring networks which are independently administered to provide transparent access. It should be noted that while we describe our solution for one specific example in this paper, our approach is more general and geared towards the larger problems of service and security management. The scenario in this paper is a vehicle to illustrate our ideas; indeed, we do not address every conceivable aspect of the scenario.

We describe how we use the NESTOR configuration management system prototyped at Columbia University to manage security requirements. We also discuss our plans for further work towards self-configuring network systems that maintain high-level security and service policies dynamically.

1.1 The Scenario

Jane Consultant, who is employed by Corporation A, is visiting a client in Corporation B. During her meeting, Jane realizes that she needs to access files in her home directory which have not been copied onto her laptop. When plugged into her home network in A, Jane simply clicks an icon on her desktop to access her files. What are her choices while plugged into B's network? She can establish a slow but potentially insecure modem connection to Corporation A (over a wireless connection for example, or perhaps the phone call is routed over the Internet). Alternatively, she can plug her laptop into an ethernet port within Corporation B; assuming she gets connected at all, it will likely be a window-less connection to B because Corporation A may not open X services in its network to hosts outside. Neither method offers access comparable to what Jane would get within her home network.

What makes the problem more challenging is that the two networks are separately administered, with independent security policies. For example, Corporation A might filter certain services when the user is plugged into a remote network. Corporation B might require that guest machines not be able to send or receive traffic directly from any machine within B's network, and that guest machines may only access remote VPN nodes. If there is a way to provide access without violating either company's policy, we would like all necessary reconfigurations to be automatic and not require manual intervention. Of course, if there is no way to provide access without violating one security policy or another, Jane cannot be provided this service. The difference in our approach is that we have a language designed to express these concerns explicitly at a high level as policies and mechanisms to support the semantics of these policies by appropriately reconfiguring the network.

A number of configuration changes are necessary to provide Jane access when the security policies allow it. In our example, some of the changes involve the Dynamic Host Configuration Protocol (DHCP)[3] server, switches and firewalls in B, and firewalls, file servers and encryption protocols within A, and decryption in Jane's laptop.

Today, most of these configuration changes must be done manually by systems administrators. There is no standard method for performing dynamic reconfiguration of services or network elements in response to changes in the network. Script-based solutions are usually highly dependent on network topology and service/element configuration mechanisms which differ across vendors and even between different versions of the same product. Scripts are frequently highly customized and therefore non-transparent and non-universal. Moreover, a single change in the network can require changes in multiple scripts, reducing reliability and further worsening the maintenance overhead.

Errors in scripts can result in inconsistent network configuration states, and manual recovery made difficult by lack of logging. Moreover, each script must carefully enforce exclusive access to the respective configuration repositories. Another drawback with the lack of centralized meta-information description and repository means that each script must rediscover information, for example network topology, that is not directly instrumented. Our solution is the first step towards standardizing and automating these configuration changes. First, using NESTOR we build a universal platform to treat network elements in a vendor-independent way. Second, by automating implementation of high-level policy we allow the SA to implement custom policies (which is what she wants) without having to write low-level scripts (which she does not want). The platform is open so that SAs can change existing models or add new ones as necessary to support new policies. Auxiliary services such as logging can also be dealt with at the policy level. The most important aspect of our approach is to ensure that the network remains (at a low level of granularity) in a consistent state; we achieve this with transaction-like semantics for operations on configuration states.

1.2 Our Approach

We used the NESTOR system [16] developed at Columbia University to build an experimental prototype for our experiments. NESTOR is a configuration management system that automates most of the network configuration tasks. Constraints between configuration parameters model interdependence between different subsystems; NESTOR provides support for network-level instrumentation and monitoring of constraints to provide predictable and error-free operations.

In our work we use constraints to also capture security policy. We focus specifically on enabling transparent and secure access in the scenario described in the previous section. This paper describes briefly the architecture of the NESTOR system, how we used NESTOR to build our prototype, how constraints are modeled and managed to provide secure transparent access in a fully automated manner. Future reports will detail progress on high-level security and service policy specifications, compilation to NESTOR constraints, and experiences with larger systems.

2 NESTOR Architecture

The NESTOR project underway at Columbia University provides a management platform for common network elements. The novelty of the NESTOR manager lies in the fact that it not only automates SA actions but also maintains consistent network state. Consistency is guaranteed by enforcing constraints between various network elements to ensure that change propagation is done correctly. A two-phase commit protocol allows all network changes to be viewed as distributed transactions with rollback features. Our next prototype will handle cases where some network elements do not allow undoing changes. For a general discussion of NESTOR and the details of its architecture please see [16].

Figure 1 shows an overview of the NESTOR system. Systems administrators manage network operations through a unified object-relationship model where managed network elements and services are represented as objects interconnected via a network of relationships. Operationally, the object repository is accessed through the NESTOR Directory Interface (NDI). The interface provides methods for initiating transactions and manipulating repository objects (lookup, create, update, delete). For example, in order to change the name of the host, a systems administrator will look up the matching *host* object in the repository, and modify its *name* attribute. All such updates are performed in the context of a transaction.

Constraints associated with the affected objects are evaluated in order to assure consistency. A constraint is composed of two parts: a declarative predicate and an imperative action. The predicate

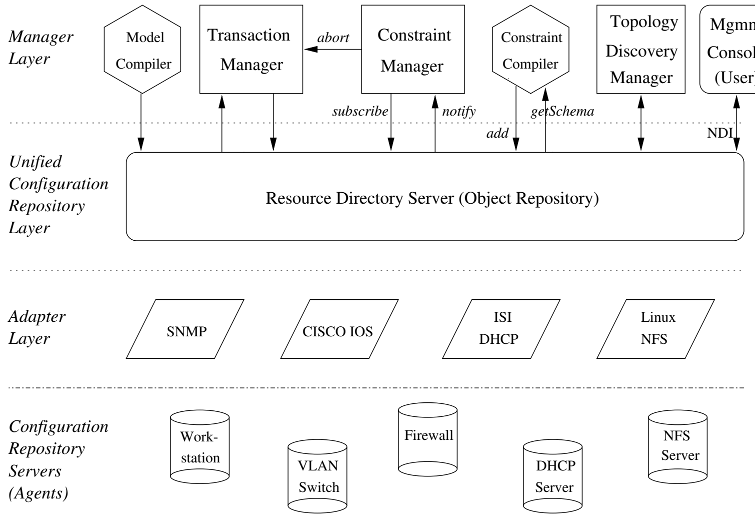

Figure 1: NESTOR Architecture

is evaluated by the constraint manager whenever a change is attempted that may affect its value. When a predicate is violated, the corresponding action associated with the predicate is executed. If the action cannot be taken (perhaps because the element being controlled is not responding) the error will propagate so that the system registers the fact that the constraint has been violated and takes appropriate actions to satisfy the high-level policy. Actions have priority levels to manage the relative order in which multiple actions must be executed.

Constraints are first-class objects and are distinguished from other modeled elements. In the host renaming example above, a constraint will state that all host names must be unique. The predicate associated with the constraint will check whether the new name will maintain this uniqueness property. If not, the action will reject the change and may indicate the error on the SA console. If the name change is accepted, the system launches a sub-transaction that may in turn cause other changes as necessary which may launch sub-transactions themselves. In order for the transaction to be committed, the system must reach a consistent state where all constraints are satisfied and all sub-transactions are complete. Otherwise the transaction is aborted and the changes discarded. Unbounded transactions are caught by time-outs and aborted.

2.1 Deploying NESTOR

To deploy NESTOR the systems administrator first models the key resources in the network as classes, in the sense of object-orientedness. A model class consists of data members and objects that describe the state and interfaces of the network element. In addition, the model has constraints which define the relationship with other network elements. A detailed example is available in Section 4 in Figure 4. It is important that the model include all the information necessary to implement the stated policies. Examples include network elements (switches, routers, workstation, servers), network services (HTTP, NFS, NIS, YP, Windows Domains), and optionally policy objects such as a security manager. Common elements models currently available in NESTOR include: Linux (RedHat SysV style configuration) workstation which instruments /etc/* configuration information, SNMP MIB II (to instrument WinNT workstations) and Generic Cisco IOS adapter. Service models in NESTOR include Apache HTTPd, Linux NFSd, ISIC DHCPd, LDAP adapter, Java JNDI and, through the JNDI adapters, YP/NIS access. For the purpose of this work, models of firewalls and other security-related objects were also created. Constraints may be defined at the time of modeling, or added incrementally to the system. Although some constraints may refer to attributes on a single object, most will navigate the object relationships to establish constraints over multiple objects. Examples of modeling and constraint languages are given in the next section.

The next step is to populate the NESTOR repository with model instances. This is done either manually, or in combination with the network topology discovery manager. The topology manager is a process executing on a NESTOR server (or, possibly, some other network node) which scans an IP network for active nodes, and attempts to discover their type (host, router, etc), and services (HTTP, NFS, X11, etc). This discovery process is successful to the extent that the topology manager can only infer information based on SNMP MIB values, port scans, and possibly remote shell commands. For example, a web server may be discovered, but it may not be possible to instrument its configuration unless an appropriate agent (SNMP HTTPd MIB or NESTOR) is installed.

2.2 Transactions in NESTOR

Repository transactions are overseen by the transaction manager using a two-phase commit protocol to maintain the transaction properties of atomicity, consistency, isolation, and durability (ACID). There are always at least two participants in every transaction: the initiating entity (e.g. a systems administrator, or the topology discovery manager), and the constraint manager. The constraint manager is responsible for enforcing the constraints stored in the repository. When the transaction initiator requests a commit, the constraint manager verifies that the new state is consistent, that is, all constraints are satisfied. If a violation is detected, the manager will invoke the action associated with the violated constraint. In the case of multiple constraint violations, the order of execution is based on the action priority level, with same level actions executed in arbitrary order. The constraint manager commits the transaction only if all constraints can be satisfied. If a cycle is detected in action execution is detected, the transaction is aborted.

2.3 Resource Discovery

The NESTOR Resource Directory Server is responsible for maintaining the network model and constraint object repository. Repository objects implement one or more model interfaces (e.g. *Host*, *HttpServer*). Constraint objects implement the constraints between various objects and thereby encapsulate the mechanism for propagating changes to the underlying resource. For example, an object implementing the *Host* model interface may use SNMP to propagate changes to the name attribute back to the host. Changes are only propagated when a transaction is moved to the commit phase, and are applied in the same order in which the transactions commit (as opposed to the order in which they are initiated).

In order to simplify the task of implementing model interfaces, NESTOR provides adapters for several management protocols and services. For example, the SNMP adapter enables system modelers to map the aforementioned *Host* name attribute to an SNMP MIB object. In addition to SNMP, adapters are provided for the LDAP and NIS protocols, as well as particular implementations of the HTTP, DHCP, and NFS protocols.

Sub-transactions will be ordered so that an aborted transaction does not expose the system. The cost of transactions and the practical implications of system lock-up during configuration changes will be addressed in future reports.

2.4 Prototype Implementation

The NESTOR system prototype we used is implemented in the Java language and runtime system. The NESTOR Directory Interface was defined as a Java interface using the Remote Method Invocation (RMI) interface protocol for its underlying communication. The Jini[12] distributed transaction and leasing mechanisms were used in implementing the transaction manager and performing garbage collection at the object repository. Java object serialization was used for persistent repository operations.

3 The Experimental Testbed

This section describes in more detail the reconfiguration necessary for the scenario of Section 1.1, and presents the specific network employed in our experiment.

Recall that in our scenario Jane, whose home is corporate network A, is now connected to company B's network and wants Web/E-mail/Telnet access to files in her company A. To simplify the exposition, we make a few simple assumptions about the two networks. These assumptions are not necessary in practice; the approach is more general that the example we choose to illustrate the capabilities of our management platform. In particular, let us suppose that Company B uses a switched network which supports Virtual LANs (VLAN) and that company A's firewall supports Virtual Private Networks (VPNs) in order to provide remote access to its users over the Internet. Our actual implmentation uses Linux for firewalls and hosts and Cisco switches for VLAN support. Further details of the equipment used are provided later in this section.

3.1 Requirements for the Scenario

In order to achieve transparent access to the services that Jane wants from her laptop the following configuration changes are necessary:

1. An available Ethernet port will need to be located and Jane's laptop physically connected (in the premises of company B).

2. The laptop will need to be configured for the local network environment, including parameters such as IP address, netmask, DNS servers, default gateways, SOCKS servers, etc. Ideally, this will be achieved automatically using DHCP.

3. In order to maintain Company B's security policy, the laptop will either have to be connected to a special "guest" network and the switch port must be configured for a "guest" virtual LAN, or all the internal services must be guaranteed to require authentication. The last option is exceedingly difficult to implement in typical networks today and are, in fact, the main motivation for using firewalls to protect corporate networks.

4. Depending on the configuration setting of Company B's firewall, the laptop's address may have to be explicitly allowed to initiate outgoing connections. Company B's security policy may require disabling the laptop's access to any external sites other than Company A since it holds an IP address in Company B's domain. This can be achieved by limiting external connections to the VPN protocol.

5. Once the laptop can reach the Internet, it will need to establish a Virtual Private Network (VPN) connection with Company A's firewall whose policy may be to grant remote hosts

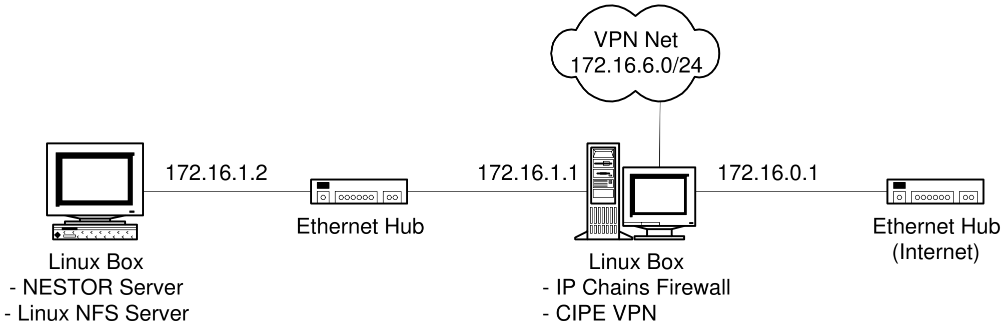

Figure 2: Company A Network

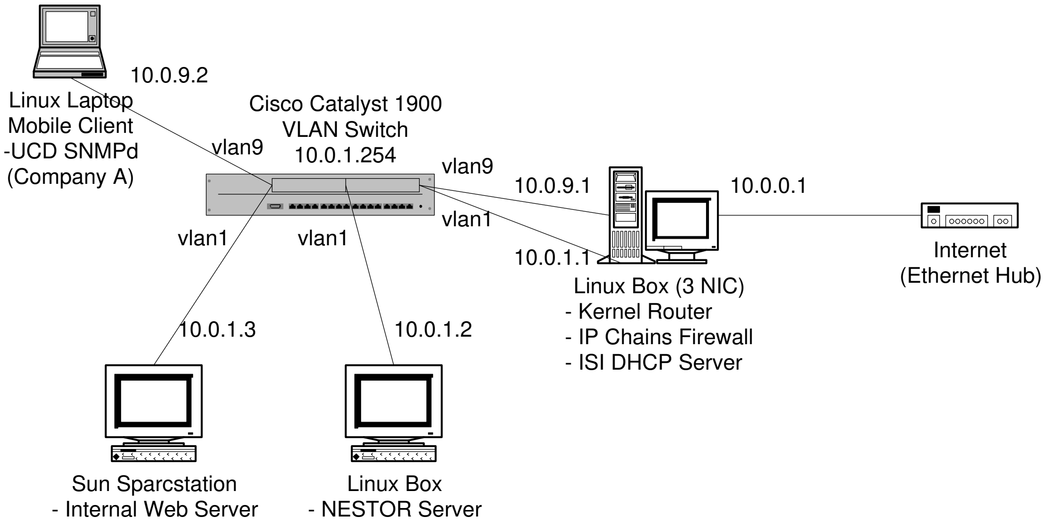

Figure 3: Company B Network

limited access to internal resources. Such policies will need to be enforced by all internal services in Company A's network.

3.2 Network Topology

The two networks are shown in Figures 2 and 3. For simplicity, Company A's network consists of the internal subnet 172.16.1.0/24 (net-1) and the VPN subnet 172.16.6.0/24 (net-6). VPN clients are allocated addresses from the second network (net-6). The internal Linux NFS server is configured dynamically by the company-A NESTOR server to restrict mobile user access to their home directory. A Linux workstation is used to provide routing, firewalling, and VPN services for the network of company A. The route table is statically set with paths to the internal network, the VPN network, and company B's network. Firewall rules are added to deny all incoming traffic access to the internal network of company A (using Linux Kernel IP Chains). Finally, the CIPE[14] Linux software is configured to enable the remote establishment of a VPN tunnel.

The network of company B is slightly more involved. Instead of an ethernet hub, the internal network is a switched network. Layer-2 switching is provided by a Cisco Catalyst 1900 with support for Virtual LANs (VLAN). The VLAN switch provides for the physical separation between trusted and untrusted IP nodes mandated by the policy of company B. Vlan port assignments will be handled dynamically by the NESTOR server. Company B also has a trusted subnet 10.0.1.0/24 (net-1) and untrusted subnet 10.0.9.0/24 (net-9). Virtual LANs with IDs 1 and 9 physically separate traffic to net-1 and net-9 respectively. A Linux workstation provides routing, firewall and DHCP services to the internal network. The router has three interfaces, one connected to vlan-1/net-1, another connected to vlan-9/net-9 and the last connected to the external network. Static routes are configured to route traffic from the internal networks (net-1 and net-9) to the external network. Firewall rules prevent any incoming traffic from the external network to the internal network with the exception of established TCP connections to net-9 (guest network). Furthermore, a firewall rule restricts outgoing connections from net-9 exclusively to the VPN port of external network destinations. Obviously, no routing is configured between net-9 and net-1. The DHCP server on the host is configured to listen to the two net-1 and net-9 interfaces for DHCP requests. All unknown hosts are allocated IP addresses from net-9, while a list of trusted hosts (based on their unique client ID which may be their ethernet address) are set to be allocated addresses from net-1. It is assumed that the list of trusted client identifiers is supplied to the NESTOR server. These trusted identifiers will be dynamically configured into the DHCP server by NESTOR.

The networks of the two companies were connected through a common ethernet hub, with static routes configured between the gateways. In this experiment, the NESTOR servers in each company operate independently of each other. The details on how dynamic configuration of the aforementioned networks occurs are discussed in the next section.

3.3 Constraints and Automatic Reconfiguration

We now outline how Jane gets transparent access without violating the security policy of either network. The policy of company B not to allow guest machines to gain access to the internal (trusted) network can be translated into the following topology-dependent constraints on device and service configuration. (The next section describes how these constraints can be expresssed formally in the NESTOR configuration language).

- Switch ports to which trusted hosts hosts are connected must belong to the internal (net-1) VLAN. Switch ports to which guest (or unknown) hosts are connected must be configured for the guest VLAN (net-9). In cases where internal and guest hosts are connected to the same switch port, the port should be set to the guest VLAN (an alternative would be to disable the port). Violation of this predicate is handled by reconfiguring the VLAN membership of the offending port.

- A firewall rule should prohibit any traffic (including the guest network) from entering the internal network (except for established TCP connections).

- The DHCP server should allocate internal IP addresses only to trusted hosts.

Additionally, company B policy further restricts guest Internet access by limiting guest connections to remote VPN servers. The goal of this policy is to prevent guests from attacking or misusing other networks while in ownership of a company B IP address. This policy is translated into a simple configuration constraint limiting all outgoing traffic from the guest network to well-known ports of VPN services.

The security policy of company A states that remote (VPN) users should receive limited services. In this example, remote users are restricted to accessing only their home directory. We map this policy (there are other ways) to file server configuration by initially denying all directory mounts by VPN hosts. When users sign in with the VPN server to obtain an IP address, permission is added for the particular host to temporarily mount the user's directory. The constraint on file server configuration states that VPN hosts should only be allowed to mount the directory whose user is logged on to the VPN server with that address. This constraint spans the configuration of the VPN server as well as the file server.

In the overall scenario, these constraints (predicates and actions) achieve dynamic reconfiguration in the following manner :

1. User Jane cannot find an available Ethernet port for her laptop, so she "borrows" the network connection of an existing host (thereby obtaining physical access to the internal network).

2. Jane's laptop requests configuration information from the DHCP server,

3. The DHCP server returns a lease on a guest network IP address since the MAC address of Jane's laptop is not included in the DHCP daemon internal network list (at this point the laptop's IP configuration is inconsistent with the link-layer network to which it is connected).

4. By polling the Ethernet switch, the company B NESTOR server discovers the new laptop host. The constraint manager evaluates the constraints which might be violated by the connection of a new host to the switch. The security constraint will be found to be violated, since an unknown (therefore untrusted) host is connected to the internal network. The action component of the constraint is executed, resulting in the reconfiguration of the affected switch port VLAN. Note, that if Jane had connected to an unused switch port, which by default is assigned to the guest network, this constraint would not have been violated. Future prototypes will include mechanisms other than polling.

5. Jane's laptop is now connected to the guest network and will attempt to establish a VPN connection with its home server (at company A) (any other access request, such as web access, is filtered by the firewall of company B).

6. Once the laptop has authenticated with company A's VPN server it is assigned a virtual IP address.

7. The NESTOR server of company A detects this new lease (by polling for the configuration state of the VPN server). The constraint on the file server configuration will be violated since the file server will not be configured to allow home directory mounts from that IP address. This violation will be handled by adding the IP address of the VPN host to the access list of the user's home directory on the file server.

8. After completing her work, Jane disconnects from the VPN server. Again, the company A NESTOR server detects this change, and re-evaluates the affected constraints, resulting in the removal of the file access permission for the previously allocated VPN IP address.

9. After Jane disconnects her laptop from the company B network, and graciously reconnects the host whose connection she had "borrowed", the company B NESTOR server will detect this event and the ensuing constraint violation, leading to the reassignment of the affected switch port back to the internal VLAN.

```
interface companyB::EthernetVlanSwitch {

    // relationshipset declares a many-to-many relation between instances
    // of this class, and instances of the EthernetVlanSwitchPort class.
    // The definition includes the role names at both ends of the relation.

  relationshipset consistsOfPorts, EthernetVlanSwitchPort, partOf "a
    many-to-one relation between a switch port to its enclosing switch";

    // An IpInterface is a NESTOR java class maintaining configuration
    // information for IP network interface (IP addresses, netmasks,
    // protocols, etc).

  attribute IpInterface ipInterface;
}

interface companyB::EthernetVlanSwitchPort : CompanyB::Node {
  attribute boolean isEnabled "true if the port is enabled, false otherwise";

  readonly attribute int portNumber "the number of the port in the enclosing
                                     switch";
  attribute int vlanId "the integer Virtual LAN id of this port";

  relationshipset forwardsNodes, EthernetNode, forwardedBy "a many-to-many
    relationship between instances of this class and EthernetNode instances";

  // relationship declares an any-to-one relation between instances of this
  // class, and instances of the EthernetVlanSwitch class.  By examining
  // the matching role declaration in the definition of EthernetVlanSwitch,
  // the model compiler determines that this is a one-to-many relation.

  relationship partOf, EthernetVlanSwitch, consistsOfPorts "a one-to-many
      relation between the switch and its ports";
}

interface companyB::SecurityManager {

  boolean isTrusted(int vlanId) "returns true if the VLAN with the given id is
                           considered a trusted network, false otherwise";

  boolean isTrusted(EthernetNode node) "returns true if the EthernetNode is
               considered trusted, false if it is untrusted (or unknown)";

  relationshipset manages, Node, securityManager "a one-to-many relation
          between a security manager and the network nodes it is managing";

}
```

Figure 4: Network Model Example

```
package companyB;

// [ This code is automatically generated by the model-to-Java compiler.
//   Comments have been manually inserted for clarity. ]

public interface EthernetVlanSwitchPort extends Node {

  public boolean getIsEnabled() throws RemoteException;
  /** returns true if the switch port is enabled, false otherwise. */

  public void setIsEnabled(boolean value) throws RemoteException;
  /** sets the configuration state of the switch to enabled/disabled */

/** model relationships are mapped into Java classes which handle the
    consistent management of relation membership.  The declared "relationship
    partOf, EthernetVlanSwitch, consistsOfPorts" is compiled into this Java
    method returning a reference to an object implementing the
    OneToManyRelation interface.  The EthernetVlanSwitch interface definition
    will conversely include a getConsistsOfPorts() method which will return the
    same object, but viewed as implementing nestor.ocl.ManyToOneRelation.  The
    mapping of model relations to Java is an area which will be further
    explored in the future. */

  public nestor.ocl.OneToManyRelation getPartOf() throws RemoteException;
}
```

Figure 5: Model to Java compiler output example

```
/** The following is an arbitrarily named Java package where
    implementations of the Java model interface definitions will be
    defined for particular managed elements. */

package companyB.impl;

/** The following class instruments the companyB.EthernetVlanSwitch
    interface for the Cisco Catalyst 1900 switch.   */

public class CiscoCatalyst1900
  implements companyB.EthernetVlanSwitch,
    nestor.adaptor.snmp.SnmpTableListener {

      /** The class constructor is invoked by the topology discovery
          manager, or explicitly by the systems administrator through a
          custom Java program, or in the future through the NESTOR GUI.
          The implementation further includes code which rediscovers the
          SNMP adaptor after the class has been deserialized (in order
          to support installation in the NESTOR repository).  */

  public CiscoCatalyst1900(IpAddress address,
                           SnmpAuthenticationObject snmpAuth,
                           CiscoIosAuthenticationObject iosAuth) {

    nestor.adaptor.snmp.SnmpAdaptor adaptor =
       nestor.repository.Repository.getAdaptor("SNMP");
    adaptor.addSnmpTableListener(target, tableOids, handback, this);
  }

  public boolean getIsEnabled() throws RemoteException { ... }

  public void setIsEnabled(boolean value) throws RemoteException { ... }

  public OneToManyRelation getPartOf() { ... }
}
```

Figure 6: Java interface example

4 Network Model and Configuration Constraints

The first step in using NESTOR for our experiment is modeling the network. This section gives some examples of model definitions for network B. Models are expressed in the MODEL language[13] which is an extension of the CORBA[7] IDL with support for relationships, and other features useful for event correlation. Figure 4 shows a subset of the model definitions for company B.

Consider the `EthernetVlanSwitchPort` interface definition. This interface models the configuration of a port in an Ethernet switch supporting VLANs. The MODEL definition states that the interface is part of the `companyB` package and inherits from the `Node` interface. Three attributes are declared to model: the state of the switch port (enabled/disabled), the port number (a read-only value), and the integer ID of the Virtual Lan to which the port is assigned. The relationship definitions declare a many-to-one relation mapping the port to its enclosing switch, and a one-to-many relation associating the port with the Ethernet (layer 2) nodes which are actively connected to the port.

The `EthernetVlanSwitchPort` interface represents a device-independent configuration model for an Ethernet switch port supporting Virtual LANs (VLAN). In order to instantiate such an object in the NESTOR repository, an implementation of that interface must be provided with support for the configuration protocols of the actual device being modeled. The next step will therefore be to compile the MODEL interface definitions into a target implementation language. The current NESTOR prototype is built in the Java language and the model compiler converts the extended IDL interface definitions to a set of Java interfaces. As part of the compilation, attribute definitions are converted to a pair of get/set methods (one for read-only attributes) following a simple design pattern. Relationships are compiled into references to collections implementing the OCL[8] (Object Constraint Language) collection semantics. OCL is a language for expressing declarative constraints (side-effect free) and was originally created to state the semantics of the Unified Modeling Language (UML). Unfortunately, due to Java's lack of a parameterized type facility, the translation loses the relationship type information, forcing users to use class casts. An alternative approach of automatically generating new classes for different relation types will be evaluated in the future.

The Ethernet switch supporting VLANs used in this experiment was a CISCO Catalyst 1900 with enterprise edition firmware. The Catalyst supports several SNMP MIBs and may also be configured using a menu system as well as from the command-line. The Bridge SNMP MIB `dot1dTpPortTable` table was used to instrument the `consistsOfPorts` attribute of the Catalyst `EthernetVlanSwitch` implementation. The implementation class registers with the NESTOR SNMP adaptor to receive notification of updates to the table. When a new port is detected, a new instance of the `CiscoCatalyst1900Port` class is constructed. The port `forwardsNodes` relationship is instrumented through the Bridge MIB `dot1dTpFdbTable`. See [2] for details of the components of these tables. The VLAN ID attribute is instrumented using the Cisco IOS adaptor parameterized by the command sequence appropriate for obtaining the VLAN id of this port. The `CiscoCatalyst1900Port` class, implementing the `companyB.EthernetVlanSwitchPort` interface, was defined with a single constructor parameterized by the IP address of the managed switch, the switch port number, and an SNMP and IOS authentication object. The authentication objects encapsulate protocol-specific security access information such as passwords and certificates.

4.1 Programming Constraints

Based on this model of the network, constraints are defined to maintain the security policies of each domain. To take an example, consider the constraint caused by company B's policy that untrusted hosts should not have access to the internal network. This policy is translated into several constraints

```
EthernetVlanSwitchPort->allInstances
  ->select(port : EthernetVlanSwitchPort | port.isEnabled)
  ->forAll(port : EthernetVlanSwitchPort |
     if (port.securityManager.isTrusted(port.vlanId))
       ( (port.forwardsNodes->size > 0)
        and
         (port.forwardsNodes->forAll(node : EthernetNode |
                                    port.securityManager.isTrusted(node))))
     else
       ( (port.forwardsNodes->size = 0)
        or
         (port.forwardsNodes->exists(node : EthernetNode |
                                   (!port.securityManager.isTrusted(node)))))
```

Figure 7: A Declarative Constraint: Trusted ports should only forward frames of trusted nodes

on the configuration of network devices. For example, a constraint on the switch states that trusted ports (i.e., those configured for a trusted VLAN) must only be connected to trusted hosts. This constraint, expressed in the OCL language, is shown in figure 7. In this example, the constraint generates the set of all instances of objects implementing the `EthernetVlanSwitchPort` interface. The `select` operator constructs a new set containing only the ports whose `isEnabled` attribute is true. The `forAll` operator makes an assertion which must hold for all elements of the selected set of ports. The assertion states that if the port's VLAN is trusted, all the Ethernet nodes which are connected must be also be trusted, otherwise, at least one of them must be untrusted. The checks for size handle the case where no hosts are connected to the port in which case the constraint states that it should be in the untrusted state.

Self-management is achieved by associating a policy script with each constraint. For example, violation of the above constraint, that trusted ports should only forward frames for trusted nodes, may be handled by switching the VLAN id of the port to one which is untrusted. Policy scripts are expressed in an imperative language, which is Java in the current prototype. The policy script is invoked with two parameters, the constraint evaluation stack, and a reference to the transaction object.

```
public class TrustedPortTrustedHostHandler
  implements nestor.repository.ConstraintHandler {

  // (simple constructor) ...

  /** Handle violation of the constraint that active trusted ports should
      only forward frames of trusted nodes by changing the VLAN id of
      violating ports to the public VLAN id */

public void constraintHandler(Object[] stack, Transaction trans) {
  // Stack: < port, node >
```

```
  if (stack.size != 2) throw
    new InternalError("Unexpected stack size=" + stack.size + ": " + stack);

  EthernetVlanSwitchPort port = (EthernetVlanSwitchPort) stack[1];

  // Obtain the public VLAN id from the security manager of the port.
  port.vlanId = port.securityManager.getPublicVlan();
}
```

The constraint compiler parses the OCL syntax and lists the events which may trigger a violation of the constraint. In the previous example the switch constraint may be violated after the following repository events: create/remove `EthernetVlanSwitchPort` instance, update in the `isEnabled`, `vlanId`, `securityManager`, and `forwardsNodes` attributes of an `EthernetVlanSwitchPort` instance, or changes to the `SecurityManager.isTrusted` map.

Constraints are first class objects in the repository, which implement the `nestor.repository.Constraint` interface. When a constraint is written to the repository, the Constraint Manager is notified of the constraint predicate and requests to be listener to the list of relevant events which when triggered may result in a change of state of the constraint.

```
package nestor.repository;

public interface Constraint extends java.io.Serializable {
  public void checkConstraint(Repository repos, Transaction trans)
              throws ConstraintException;
  public void checkConstraint(RepositoryEvent[] events, Repository repos,
                        Transaction trans) throws ConstraintException;
  public RepositoryEvent[] getConstraintEvents();
}
```

Constraints are evaluated at the end of a configuration transaction. Transactions may be initiated by a manager, using the repository API, or may be initiated by an adaptor to propagate direct changes to device configuration. For example, the previous constraint example may be violated by a systems administrator if in the course of a transaction a host which was previously untrusted was marked as trusted. The constraint may also be violated when a new (unknown) host is connected to a switch port and this fact is propagated to the repository by the switch adaptor. If an external state change transaction is rejected for any reason resulting in the violation of a constraint, then this fact is propagated back to the affected objects through the relationship network which may find an alternative method of constraint satisfaction (such as an alternative setting of a firewall). If an object cannot find any way of satisfying its constraints, it raises an error message on the SA console.

4.2 Populating the Repository

The NESTOR repository may be populated manually or using a graphical user interface that can generate objects given the model and the appropriate parameter values. The repository is accessed through a Java Remote Method Invocation API. The API supports methods for adding and removing objects, locating objects based on the class and attributes, and initiating transactions. To add an object to the repository a systems administrator initiates a transaction and then adds the object within the transaction. The object must implement one or more model interfaces and support the serializable interface (i.e. may be stored as a byte string for transport over the network). Storage in the repository is provided on a lease basis which must be renewed by some entity such as a lease renewal manager, or the object itself. If a lease expires an object may be killed or archived if possible. If there are constraints that may be affected, an error message is raised on the console. This obviates the need for vigorous garbage collection. The current NESTOR prototype utilizes the Jini[12] distributed leasing, event, and transaction APIs.

The repository can also be populated with the help of a utility for topology discovery. The utility executes on a host, and periodically pings each network address to establish a map of active nodes[1]. Currently, our topology manager accepts classless IP network and netmask combinations. Once a node is detected as being active, the utility attempts to extract information using the SNMP protocol, and tests for service availability by attempting to connecting to different services (such as Telnet, HTTP, NFS, FTP, etc). In its current incarnation, the topology manager is mostly focused on discovering workstations (such as Linux and Windows NT boxes) and supplying information about their interface configurations, route tables, and active services.

Returning to our scenario, the administrator constructs a new instance of the `CiscoCatalyst1900Switch` object using the IP address assigned to the management interface of the VLAN switch and the appropriate authentication information for administering the switch which are the SNMP community and IOS passwords. NESTOR repository objects implement an initialization and control interface (analogous to Java applets) so that their execution can be controlled by the NESTOR server. An object may query the repository for services such as adapters using an instance of the `RepositoryContext` interface. For example, the switch object will use an SNMP and IOS adaptor instances. New adaptors may also be used provided they implement the `NestorAdaptor` interface). After obtaining the necessary adaptor references, the object will subscribe for notification of changes in the relevant SNMP objects, and IOS results.

When the administrator commits the transaction to create the new switch object, the transaction manager will verify that the addition did not violate any constraints. The constraint, shown in figure 7, may be violated when new instances of objects implementing the `EthernetVlanSwitchPort` interfaces are created. Assume the initial switch state does not violate the aforementioned constraint. When user Jane connects her laptop computer to network B, and in particular to a switch port, the switch SNMP bridge MIB table `dot1dTpFdbTable` will add the laptop's MAC address. At the time of the next poll by the NESTOR SNMP adapter, the change will be detected resulting in notification of the subscribing `CiscoCatalyst1900Switch` object. The switch object will look up for an instance of `EthernetNode` with the same MAC address, creating a new instance if one is not found. The `EthernetNode` is then added to the switch's `forwardsNodes` relation. At the point where all propagated changes have been reflected in the model, the switch object will commit the changes. At this point the constraint manager will again verify the set of constraints which may have been affected by the transaction. In this example, since Jane connected her laptop to a switch port previously assigned to the internal network, the constraint on switch port VLAN state will be violated, and the policy script will be executed as outlined earlier in the paper.

5 Future Extensions and Applications

As mentioned in the introduction, the work described in this paper is the first step towards our long-term goal of building a management platform for security and service availability. Several interesting theoretical and implementational issues have been identified and examined in this project. Foremost amongst them is the need for formal tools to express network security and service policy. In the current work, we have assumed that the two networks in question have security policies that have

[1]The security warnings that these may generate will have to be handled.

to be obeyed when reconfiguring the network. Upcoming reports will deal with the questions of how general network security policy can be stated formally and implemented automatically by integrating it with NESTOR with appropriate translation mechanisms. This requires settling questions of an appropriate language for security policy and checking consitency of policies.

Another dimension of our approach that we did not address in this paper is scalability. For the network management platform to be viable in wide-area networks, management discipline must be lightweight and modular. Furthermore the response time of the management software to proposed changes in the network must be fast enough to keep pace with the rate of change. Our current architecture uses centralized NESTOR servers but in a large network this is likely to be infeasible. To ensure fast response times, some of the management functionality must be localized. Local decision-making capabilities have to be balanced against the more important goal of automatic network consistency. Since verifying (and consequently, propagating) each and every change in a dynamic network with a central global policy server is not likely to be successful, our approach is to distribute the task of maintaining global network properties such as consistency and security policy using constraints between network elements. We plan to investigate a distributed architecture for our management platform wherein every network element such as a switch or laptop has some NESTOR functionality within it so that network elements can directly communicate with each other using a NESTOR-like interface which implements constraints between these elements. For example, a switch may be enabled to reconfigure within a predefined set of allowed configuration (as defined by the global security policy) in response to local changes. Our future work on scalability will address the problem of partitioning policy and constraints in a network between its various elements.

In the short term, the current design of NESTOR will be extended with a focus on the design of the security, distribution, replication and caching protocols, as well as optimization issues in the current prototype implementation. NESTOR will be applied to verifying the configuration of the Columbia University department of Computer Science. Deployment on a large, live network such as Columbia CS will help identify areas for performance optimization. The NESTOR source code and sample models will also be made freely available for downloading[2].

6 Related Work

The most closely related management architecture to NESTOR is the ICON system[5] which used active database style Event-Condition-Action (ECA) rules to state restrictions on objects instrumented by SNMP MIB values. The NESTOR system also incorporates services such as multi-protocol access to heterogeneous resource information, configuration transactions, declarative constraint, and constraint propagation through policy scripts. The Dolphin project[9] developed a declarative language for modeling network configuration and operation for fault analysis where the emphasis was on deducing the cause of failures that have occurred by tracing the propagation of operational rules in the model. Constraint-based management has been pursued previously in [10] and [11] where constraints are employed for the diagnosis of network faults. In the area of configuration management automation, the GeNUAdmin system[6] is an off-line tool for extracting network configuration information into a centralized database, performing updates on that database which are checked for consistency, and pushing the changes back to their respective configuration files. Simple consistency checks are performed to assure that added values are valid and that key values are unique. The RPI service dependency tool[4] detects service dependencies and generates up to date server listings. The goal of the system is to prevent unforeseen service interruptions caused by hidden service dependencies. Ganymede[1] is an extensible and customizable directory management framework

[2] See `http://www.cs.columbia.edu/dcc/nestor`

applied to the central management of user and host data, which is distributed in different databases. Ganymede supports transactions on the central repository objects, but does not provide a constraint mechanism beyond a few built in security, and deletion propagation checks. NESTOR can support these functionalities given an appropriate set of constraints on the unified configuration model.

7 Conclusions

Providing guest users secure network services requires automatic, dynamic, and safe configuration of multiple devices and services. Even though the operational and security constraints span multiple devices, services, and profiles, NESTOR provides a unified model of network configuration which significantly simplifies specification and management of network constraints. By separating the model definitions from the instrumentation layer implemented by NESTOR, we can implement high-level security policies automatically and effectively by compiling them into NESTOR constraints on network elements and services. Constraint resolution is achieved through the automatic execution of policy scripts whose actions are subject to the constraints defined. This, combined with the fact that all configuration changes are logged, promises to make use of automated reconfiguration a practical reality.

Acknowledgments

Part of this work was done when Alexander Konstantinou was a summer intern at Telcordia. The NESTOR project at Columbia University is sponsored by DARPA Contract DABT63-96-C-0088. The project on self-management of security properties at Telcordia Technologies is sponsored by DARPA Contract F30602-99-C-0182.

References

[1] J. Abbey and M. Mulvaney, "Ganymede: An extensible and customizable directory management framework," in *12th USENIX System Administration Conference (LISA '98)*, 1998. http://www.arlut.utexas.edu/gash2/.

[2] E. Decker, P. Langille, A. Rijsinghani, and K. McCloghrie, "Definitions of managed objects for bridges," Tech. Rep. RFC 1493, IETF, July 1993.

[3] R. Droms, "Dynamic host configuration protocol," Tech. Rep. RFC 2131, IETF, Mar. 1997.

[4] J. Finke, "Automation of site configuration management," in *11th USENIX System Administration Conference (LISA '97)*, 1997.

[5] S. K. Goli, J. Haritsa, and N. Roussopoulos, "ICON: A system for implementing constraints in object-based networks," in *IFIP/IEEE Integrated Network Management, IV*, 1995.

[6] M. Harlander, "Central system administration in a heterogeneous unix environment," in *8th USENIX System Administration Conference (LISA VIII)*, 1994.

[7] Object Management Group, "The Common Object Request Broker: Architecture and specification," Tech. Rep. Revision 2.3, OMG, June 1999.

[8] Object Management Group (OMG), "Object constraint language specification," tech. rep., OMG, 1 1997.

[9] A. Pell, K. Eshgi, J. J. Moreau, and S. Towers, "Managing in a distributed world," in *IFIP/IEEE Integrated Network Management, IV*, 1995.

[10] M. Sabin, R. D. Russel, , and E. C. Freuder, "Generating diagnostic tools for network fault management," in *Fifth IFIP/IEEE International Symposium on Integrated Network Management (IM'97)*, 1997.

[11] M. Sabin, A. Bakman, E. C. Freuder, and R. D. Russel, "Constraint-based approach to fault management for groupware services," in *International Symposium on Integrated Network Management*, 1999.

[12] Sun Microsystems, "Jini architecture specification," Tech. Rep. version 1.0, Sun Microsystems, Jan. 1999.

[13] System Management Arts (SMARTS), "Event modeling with the MODEL language: A tutorial introduction," tech. rep., SMARTS, 1996. http://www.smarts.com/.

[14] O. Titz, "Cipe - crypto ip encapsulation," tech. rep., inka, 1999. http://sites.inka.de/%7eW1011/devel/cipe.html.

[15] Y. Yemini, A. V. Konstantinou, and D. Florissi, "NESTOR: An architecture for self-management and organization." To appear in IEEE JSAC special issue on network management.

[16] Y. Yemini, A. V. Konstantinou, and D. Florissi, "Nestor: An architecture for self-management and organization," tech. rep., Dept. of Computer Science, Columbia University, Sept. 1999. http://www.cs.columbia.edu/dcc/nestor/.

AUTHOR BIOGRAPHIES

SANDEEP BHATT (bhatt@research.telcordia.com) is Senior Research Scientist at Telcordia Technologies. His current research involves self-configuring network management systems. His previous work has included network monitoring and fault-management systems, high-performance distributed simulations of communication networks and physical N-body systems, theory of graph embeddings, with applications to VLSI circuit layout and parallel computing. He received his S.B., S.M., and Ph.D. degrees from the Massachusetts Institute of Technology in 1978, 1980, and 1984 respectively. He was previously Associate Professor of Computer Science at Yale University, and in 1990 Visiting Associate Professor of Computer Science at California Institute of Technology.

ALEXANDER KONSTANTINOU (akonstan@cs.columbia.edu) is a doctoral candidate in the Computer Science Department at Columbia University in the city of New York. He was awarded the M.S. degree in computer science from Renselaer Polytechnic Institute (RPI) in Troy, NY (1996), and a B.A. in history and computer science from Macalester College in St. Paul, MN (1994). His research interests include computer networks, network management, and distributed systems. Currently, he is leading the NESTOR project at Columbia University under the supervision of Professor Yechiam Yemini. He has previously been employed as systems administrator managing the Unix laboratory in the Macalester College Mathematics and Computer Science department.

SIVARAMAKRISHNAN RAJAGOPALAN is Research Scientist at Telcordia. His Ph.D. thesis at Boston University devised secure ways of accelerating block ciphers. His research interests include cryptography, security of web applications, and traffic and fraud analysis. He holds patents on provably secure video-rate encryption, and efficient, data-compression. These are used by phone companies to compress logging data on telephony networks. Together with colleagues in 1995 he found security flaws in firewalls which could be exploited by Java programs. This attack was widely reported and has influenced security administrators strongly on the dangers that downloaded executable content pose for protected networks. He has published in the fields of Cryptography, Data Compression, Internet Security, and Complexity Theory.

YECHIAM YEMINI (yemini@cs.columbia.edu) is Professor of Computer Science at Columbia University. His research interests include computer networks, network management, high-speed networks, and protocols. He has authored over 150 publications and lectured extensively in these areas. Research at his Distributed Computing and Communications (DCC) laboratory resulted in widely exported network software technologies applied by hundreds of sites and commercialized by several companies; see http://www.cs.columbia.edu/dcc. He was a co-founder of Comverse Technology Inc., the lead vendor of voice mail systems for telecom networks.- see http://www.comverse.com/, and of System Management Arts, Inc., a DCC lab spin off producing software products to automate event correlation and fault diagnosis in networked systems; see http://www.smarts.com.