Securing Distributed Applications Using a Policy-based Approach

Patrick McDaniel
AT&T Labs – Research
Florham Park, NJ 07932
pdmcdan@research.att.com

Atul Praskash
Department of EECS
University of Michigan
Ann Arbor, MI 48109-2122
aprakash@eecs.umich.edu

December 19, 2003

Abstract

Distributed applications are increasingly being used for communication, sharing data, and distributing data by users. However, incorporating security in them remains a significant challenge for both developers and users for several reasons. First, the security features required in an instance of an application may depend on the environment in which the application is operating, the type of data exchanged, and the capability of the end-points of communication. Second, the security mechanisms deployed could apply to both communication and application layers in the system, making it difficult to understand and manage overall system security. This paper presents a policy-based approach to meeting these needs. A security policy language framework, Ismene, is extended to allow security policy specification to be used by both the communication and the application layers. To illustrate the use of the framework, we specify security policy for a prototype distributed file mirroring application that operates in environments with different security requirements. We report on our experiences in using a policy-driven approach for securing such applications.

Keywords: security, policy, distributed systems, group communication

1 INTRODUCTION

Distributed applications are increasingly being used for communication, sharing data, and distributing data by users. Examples include conferencing applications, messaging applications, peer-to-peer file sharing systems, and data distribution applications. Many of these applications are currently used without any security, partly because incorporating the "right" level of security in them is difficult for both developers as well end-users of the system. remains significant challenge for both developers and users. Some problems in incorporating security include:

- *Heterogeneity in Security Requirements:* There may not be a right level of security that is appropriate for all instances of an application. For example, in a file distribution application, the level of security required is likely to depend on the nature of the file (e.g., public or confidential), the operating environment (e.g., private network or public network), the capabilities and policies of the users (e.g., cryptographic algorithms available to each user and authentication method), etc. Configuring the security incorrectly does have a potential cost: the system may become unavailable to some users, it may become more difficult to use or access, or it may perform poorly.

- *Incorporating Security Requirements:* Even after the right level of security can be determined for an application instance, incorporating it into an application can be non-trivial for developers.

- *Layering Problem:* It is usually not sufficient to apply security at one layer in the system. For example, using a protocol such as TLS or IPSec communication layer alone may be only one component of overall security. An application-layer security policy will also need to decide whether an authenticated user is permitted to send some application data over the available communication channels and the policy governing who is allowed to subscribe to the communication channels. Thus, the security mechanisms deployed will often apply to multiple layers in the system, making it difficult to understand and manage overall system security.

One possible solution to address the above problems is to develop different systems for each likely security environment. Development of TLS [**?**] and WTLS [**?**], two related, but different, secure communication protocols for wired and wireless domains can be considered to be an example of such an approach. If the environments are well known, few, and with wide applicability, such an approach may be feasible and can give the advantage of optimizing the solution for each environment. In most other cases, developing custom systems for each potential environment with different security requirements is likely to be too expensive, take too long, and of questionable value if the operating environment changes by the time the solution is developed.

A more promising approach to meeting the above requirements is to design policy-based applications where the policy can be configured to meet the different security requirements with minimal or no changes to the application. Policy has been used in different contexts as a vehicle for representing individual aspects of a system, such as authorization and access control [1, 2, 3, 4, 5], peer session security [6], quality of service guarantees [7], and network configuration [8, 9]. However, distributed applications pose a significant challenge for policy-based security systems because of the following issues

- *Provisioning and Reconciliation Problem:* Each member (computer or user) of the distributed application may have different cryptographic algorithms and multiple authentication methods available. *Reconciliation* among these choices may be required to achieve a common, agreed-upon set of security parameters for a session among the members.

- *Authorization Problem:* Domain-specific checks may be needed before operations are permitted.

- *Need for Flexibility:* It is difficult to anticipate the rules that different applications will need. The policy framework needs to be extensible to support different applications.

- *Layering Problem:* Distributed applications frequently have a layered architecture. Policy may need to be coordinated across layers.

Different policy languages address a subset of the above problems, but we are not aware of any integrated and general approaches to addressing all the above issues.

For example, several systems have examined ways of specifying authorization and access control [1, 3, 4, 5] but do not address the provisioning problem. These approaches govern access by mapping identities, credentials, and conditions onto a set of allowable actions. Depending on the system, the policy language can be specific to a particular application (e.g., a database system) or be more broadly applicable. The PolicyMaker [2] and KeyNote [10] trust management systems, for example, provide a very general solution to the authorization problem that is not domain-specific, including allowing the establishment of chains of conditional delegation defined in authenticated policy assertions, and allows application-specific customization of policy rules. However, they do not address the

session provisioning problem and rely on the availability of a public-key infrastructure to authenticate policies and authenticate principals.

Some recent systems have examined the problem of provisioning security mechanisms in a multi-party application, given that different members may have different preferences regarding how to provision a group. In the two-party case, the emerging Security Policy System (SPS) [6] defines a framework for the specification and reconciliation of security policies for the IPSec protocol suite [11]. Reconciliation is largely limited to the intersection of data structures. In the multi-party case, the DCCM system [12] provides a negotiation protocol for provisioning. DCCM defines the session policy from the intersection of policy proposals presented by each potential member of a secure group.

None of the approaches above meet all the above requirements for handling security in distributed applications. In some applications, provisioning aspects and authorization aspects may be related. The authorization rules may be depend on the provisioned security mechanisms (e.g., allow a user to receive the group key provided a strong cryptographic algorithm is being used for authentication and communication). In other applications, the application may wish to add a policy for logging and auditing important events at the communication layer. Considerable flexibility is required in the policy infrastructure to handle these situations.

This paper presents a policy-based approach to meeting these needs. A security policy language, Ismene [13, 14], is used to specify both provisioning and authorization requirements of distributed applications. We show that the policy framework needs to be extensible in application-specific ways to allow security policy specification at both the communication and the application layers.

To validate the approach, we present a detailed case study where we used Ismene to specify security policy for a prototype distributed file system mirroring application, AMirD, that operates in environments with different security requirements. Policies appropriate for the AMirD mirroring service in four diverse environments are specified, analyzed, and benchmarked. These environments capture the diversity in threat and trust models encountered by many information services. In this investigation, we endeavor not only to demonstrate the power of a policy-based behavior, but also illuminate the construction and use of diverse policies in a candidate distributed application.

Ismene allows a group policy to be reconciled with multiple local policies of the members to achieve a session policy in an efficient way. We do not discuss the theoretical foundations of Ismene and the reconciliation algorithms in this paper. For interested readers, they can be found in [13, 14].

The remainder of this paper is organized as follows. The following section presents a brief overview of the Ismene policy language and algorithms. Section 4 describes the design of AMirD. Section 5 considers the construction of policies appropriate for AMirD in a number of operating environments. Section 6 presents a performance analysis of AMirD under these policies. We conclude in section 7.

2 ISMENE POLICY LANGUAGE

Ismene is a session-oriented security policy language and currently targeted for secure group communication applications. Evaluation of Ismene policy results in a a *session policy* that defines the security-relevant properties, parameters, and facilities used to support a session. Thus, a policy states the identifies and abilities of session participants (i.e., *authorization*), and the mechanisms used to achieve security objectives (i.e., *provisioning*). This broad definition extends much of existing policy; dependencies between authorization, access control, data protection, key management, and other facets of a communication can be represented within a unifying policy. Moreover, requirements frequently differ from session to session, depending on the nature of the session and the environment in which it is conducted. Hence, the conditional requirements of all parties are defined by policy.

Figure 1 presents a scenario in which a distributed session is established between two or more entities in an instance

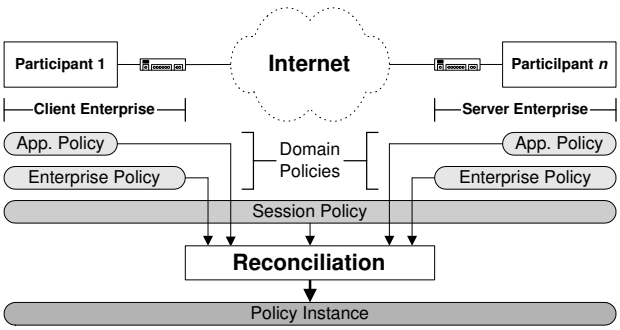

Figure 1: Policy construction - A session-specific policy instance for two or more participants is created by an initiator. Each participant submits a set of domain policies identifying the requirements relevant to the session. The initiator constructs the policy instance compliant with each domain and the session policy through reconciliation.

of a distributed application. Each participant in the session submits a set of relevant domain policies to the *initiator*. The initiator may be a participant or external entity (e.g., policy decision point [15]). A special entity that normally creates the session,*policy issuer*, states a *session policy* that describes the required set of security mechanisms and their configuration (provisioning) and the set of rules used to govern actions in the system (authorization). *Domain* policies state conditional requirements and restrictions placed on the session by each of the interested parties.

The initiator uses a *reconciliation* algorithm to create a *policy instance* compliant with the session and each domain policy. A policy is compliant if all stated requirements and restrictions are realized in the resulting instance. If an instance is found, it is used to govern the provisioning and authorization of the subsequent session. If an instance cannot be found, then the participants must revise the domain policies or abort the session or some domain policies excluded from reconciliation. The reconciliation algorithm in Ismene allows domain policies to be prioritized so that a restrictions placed by a lower-priority policy does not end up excluding higher-priority policies [14]. A policy instance concretely defines session provisioning (e.g., cryptographic algorithms, key lengths, security guarantees such as confidentiality, integrity, source authentication, as well as any rekeying policies). The policy instance also includes rules for authentication of members and authorizing actions by members.

A session policy in Ismene is authoritative; any policy instance must be compliant with the session policy. The session policy may give choices so that reconciliation can be achieved when members have different sets of capabilities. During reconciliation, domain policies are consulted only where flexibility is expressly granted by the issuer. Hence, the session policy acts as a template for operation, and domain policies are used to further refine the template toward a concrete instance. Conversely, domain policies represent the set of requirements that are deemed mandatory and relevant by a potential member. The member will not participate in an active session if the policy instance used is not compliant with its domain policy.

If no session policy is specified, a default session policy that places no constraints on session security is used. In that case, participant domain policies are reconciled to derive the instance. If a session policy is specified, but no domain policies are specified,the policy instance is derived from the session policy.

2.1 Policy Specification and Determination

Ismene session and domain policies are defined through totally ordered sets of *clauses*.

A given clause specifies either provisioning requirements or authorization requirements and consists of a tuple:

$$[tag] : [conditionals] :: [consequences];$$

```
% Provisioning Policy
provision: :: config(kerberos()), trans;
trans: inLan() :: config(ssh());
trans: :: config(ssl());

% Authorization Policy
login: credential(&tick,tgt=$tgt) :: accept;
```

Figure 2: Session Policy - the session policy provides a template for session configuration. The policy is reconciled with the relevant domain policies to arrive at a policy instance enforced at run-time.

```
% Provisioning Policy
provision: :: config(kerberos()),
              pick( config(ssl()), config(ssh()) );

% Authorization Policy
login: :: accept;
```

Figure 3: Domain Policy - a domain policy specifies a set of requirements and constraints to be placed on the session. Each session participant may supply zero or more domain policies for reconciliation.

The basic idea in evaluating a clause for a given tag is to enforce the consequences if the conditionals are true. Empty conditionals imply a value of `true`. A tag simply denotes a provisioning requirement, or an action for which an authorization decision needs to be made.

Each clause defines a rule for either the the session provisioning or the authorization policy (but not both). Provisioning clauses are evaluated when reconciling a session policy and given domain policies. The result of an evaluation is a set of mechanisms and their parameters to configure the mechanisms for implementing the services in a distributed session.

Mechanisms are simply software modules that implement some session-relevant service. For example, a Kerberos [16] mechanism could be specified in the provisioning policy and configured to contact a local KDC when authenticating users.

The use of clauses for determining both provisioning and authorization aspects in a session is best illustrated via a simple example shown in Figure 2. We discuss the various aspects of policy evaluation below.

2.2 Evaluation of provisioning clauses

Consider the clauses in Figure 2. The first three clauses specify the provisioning policy. `provision` is a distinguished tag in the language that indicates the starting point of provisioning policy evaluation. The evaluation of the provisioning policy occurs as follows:

1. First evaluate the clauses with the tag `provision`;
2. To evaluate the clauses with a given tag, determine the first clause with the given tag for which the conditional is true. Assert (i.e., apply) the consequences for that clause and ignore other clauses for the tag. The consequences can contain provisioning directives or subsequent tags (no recursion is allowed). Provisioning directives must be enforced in the session. Any tags is then evaluated using the same algorithm and its consequences enforced.

If for the clauses corresponding to a tag being evaluated, no conditional is true, then the policy evaluation fails and the session's security policy cannot be determined. A policy should thus normally contain a default rule for each tag.

In the first clause in Figure 2, because the `provision` clause does not contain conditionals, the consequences are always applied. The configuration consequence (`config(kerberos())`) states that the Kerberos mechanism must be used for authentication.

The `trans` tag in the first provisioning clause indicates that the transport protocol sub-policy specified in the two `trans` clauses must also be evaluated. The first `trans` clause specifies context; the `inLan()` predicate tests whether the session is between two hosts within the same LAN environment. If the predicate evaluates to *true* (within a given environment), then the `ssh` protocol is used and configured. If *false*, then evaluation drops to the next `trans` clause, and the `ssl` protocol is selected.

2.3 Evaluation of authorization clauses

An authorization policy states the conditions under which access to protected actions is given. Actions are defined by the tags of the authorization clauses (e.g., the `login` action in the Figures). The set of actions appropriate for a given session are dictated by the application domain an the enforcement architecture. Applications are expected to govern all actions, and hence the session, by consulting this authorization policy at run-time.

The `accept` consequence is the only legal consequence in an authorization clause. Ismene represents a closed world in which denial is assumed. Therefore, an action is allowed only where explicitly granted by successful evaluation of an associated authorization clause. If the `accept` consequence is not reached when evaluating clauses for a specified action because all the conditionals are false, the action is not permitted.

The `login` clause defined in the example session policy states that users are given access only if they acquire and present the proper Kerberos ticket (which proves their authenticity). The `credential` conditional is implemented by the Ismene infrastructure and tests whether the entity attempting to perform a protected action has provided the appropriate rights or identity proving credential. Similar to Keynote credential evaluation [10], this conditional matches credential fields against values specified in policy. The `$tgt` symbol indicates attribute replacement. Attributes are Ismene specific variables assigned by the application, and are replaced during clause evaluation. In this case, the `$tgt` attribute is replaced with the identity of the local Kerberos server (identified by the application), and access given where the ticket was issued from that server.

2.4 Domain Policies and Policy Reconciliation

Figure 3 shows one domain policy for the same application. This domain policy is for an entity that wishes to participate in this distributed application (in this case, remote login). The domain policy provisions the Kerberos mechanism as previously described. The entity, however, does not care whether ssl or ssh is configured for the login session. In Ismene, this choice is specified via a `pick` statement. The semantics of `pick` state that exactly one of the configurations must be selected to implement the session. Hence, an issuer uses pick statements to identify choices for acceptable behavior. Much of the power of Ismene stems from this feature.

The domain policy also includes a clause for authorization. The `login` clause defined in the domain policy does not specify any conditional; it thus simply defers governance of the `login` action to the session policy. The reconciliation algorithm constructs logical conjunction of all policies to form the authorization policy in the policy instance. Hence, in this example, the authorization policy will just contain the conditions identified in the session policy and be enforced as described earlier.

The result of evaluation of a policy is a set of configuration and pick statements. Reconciliation attempts to find some instance that is consistent with all the policies. In our above example, the reconciled policy would include `keberos()` and either `ssl()` or `ssh()`, depending on whether `inLan()` evaluated to true or not. For brevity we omit further details of the reconciliation algorithm.

2.5 Mechanism Parameters

Parameter(s) can be optionally passed in to a mechanism to further configure it. For example, consider the following clause.

```
% Provisioning Policy
provision: :: config(kerberos(KDC="kerberos.engin.umich.edu")),
              pick( config(ssl()), config(ssh()) );
```

The above clause passes a parameter to the Kerberos mechanism for initializing it. The value of a parameter is dictated by the the session policy (unless the session policy gives an explicit choice by using a `pick` statement). For a

domain policy to be reconciled, it must either omit the parameters or provide values that are consistent with the session policy. For safety reasons, if the session policy does not specify a parameter value, a default value is assumed.

3 USE OF THE POLICY LANGUAGE AND ITS EXTENSIBILITY

The only keywords in an Ismene specification are `provision`, `pick`, and `config`. Everything else is opaque to the language and the semantics are interpreted and enforced by either our group communication libraries or the application. In a distributed application, Ismene library and APIs are provided to help decide the set of mechanisms to be configured and for making authorization decisions.

The current set of applications that use Ismene also use the Antigone secure group communication toolkit [17, 18]. Antigone consists of several types of security mechanisms, a subset of which is given below:

- *Authentication mechanisms:* Several authentication mechanisms are provided in the Antigone toolkit, including SSL, Kerberos, and a null authentication scheme for testing.
- *Group key distribution mechanisms:* Keys can be distributed in a group using either a centralized scheme, a key-encrypting key scheme, or a scalable scheme based on logical key hierarchies.
- *Data handling mechanism:* The data handling mechanism can be configured to provide confidentiality, sender authenticity, and data integrity. Each of these can be further parameterized to specify the crypto to be used, including key lengths.

In order to support application-specific security requirements, an application may need to go beyond the mechanisms provided by the Antigone group communicatoin layers. An application may need to do the following in order to implement application-level security policies:

- Defining application-specific mechanisms and paramters
- Defining application-specific predicates
- Defining application-specific actions and related attributes

Below, we discuss each of these aspects. Consider an application that needs to include a new mechanism `mymech` with a parameter called myconfig. This mechanism may be used in a policy clause as:

```
Mypol: :: config(mymech(param=20));
```

The Ismene system provides an API to applications to register new mechanisms, along with any configuration parameters. The Ismene library also provides a *Policy Engine* that is used to evaluate input policies and to check if an action should be authorized.

To define a new mechanism, the programmer simply inherits the new mechanism from an existing class `AMechanism`[1] A C procedure is also defined to initialize the mechanism. This procedure is registered with the Ismene Policy Engine via a call:

```
registerMechanism(‘‘mymech’’, void *createMyMechMechanism);
```

The policy engine invokes the specified procedure if the policy instance dictates the `mymech` mechanism needs to be configured.

The initialization procedure for a mechanism is implemented as follows:

[1]The initial letter `A` in `AMechanism` and other names in the system denotes Antigone, since Ismene was developed as part of the Antigone project though, in principle, it can also be used independently of Antigone.

```
void *createMyMechMechanism(void *data)
{
  return (new AMyMech(*(AMechEngine *) data));
}
```

The `data` pbject contains a reference to an Ismene object that contains all attribute-value pairs that were configured in the policy instance, including `param` and its value of 20. The mechanism can then configure itself, given the configuration parameters.

New predicates can also be added by an application. The Policy Engine provides an interface for registering new predicates. We omit the details here since the APIs are similar to those used for registering mechanisms.

Note that a Policy Engine exists at each communication end-point in a distributed application. The underlying Antigone protocols provide protocols for distribution of policy instance after members are authenticated as part of the protocol to join the group. The join protocol and the policy distribution protocol needs to be implemented carefully to avoid security flaws. We omit the details of those protocols here since it is beyond the scope of this paper.

Ismene's policy engine uses an event-based architecture. Events include timer expirations, message sends and receives, as well as events generated by the mechanisms. Each mechanism receives all the events (software bus architecture). Upon seeing an event, it can choose to ignore it, do some processing on it (such as logging), and/or generate another event. With careful definition of events, ignoring events can be done very efficiently since events have an integer type ID and a mechanism uses a bit-mask to filter out events to be ignored.

When a new event is generated, the data in the new event may be a function of the data in the event being processed (e.g., a cryptographic operation applied on the buffer associated with the processed event or adding a header to an earlier event). A library is provided for efficient copying of buffers and sharing of data between buffers.

Note that the event bus delivers an event to every mechanism within a node in the distributed application (events are always local). Events received on the bus are processed in accordance with each module's purpose and configuration. Event delivery is modeled as being simultaneous. The event bus guarantees that *a)* events are delivered in FIFO order and, *b)* an event will be delivered to all mechanisms and the application interface before any other event is broadcast. The event bus provides no guarantees on the ordering of mechanisms to which the event is delivered. We considered an alternative option of providing APIs for associating events with only a subset of mechanisms, defining ordering within mechanisms, etc., but the complexity did not appear to be worthwhile as long as the number of mechanisms instantiated in any given system is small, as we have experienced in practice. Furthermore, where ordering semantics are desired, we can achieve that be defining new event types. For example, suppose that an event of type A is desired to be processed by a mechanism X, followed by Y. We can achieve that semantics in our model by the following setup:

1. Event of type A is only processed by X and ignored by Y.
2. Mechanism X processes event of type A and generates an event of type A' (possibly reusing some of the data from event A).
3. Mechanism Y processes event of type A'. Event of type A' is ignored by X.

The number of event types can go up in our model where ordering is important, but we have not found this to be a significant issue. One major advantage is that it is easy to add logging and auditing services in the infrastructure by simply defining a new mechanism that logs specified events.

Finally, an application can invoke the Policy Engine at any time to determine if an action should be permitted. The action is simply identified by its string name in the Ismene specification. The application is responsible for setting the values of any attributes that are required to evaluate the predicates to determine if the action should be permitted..

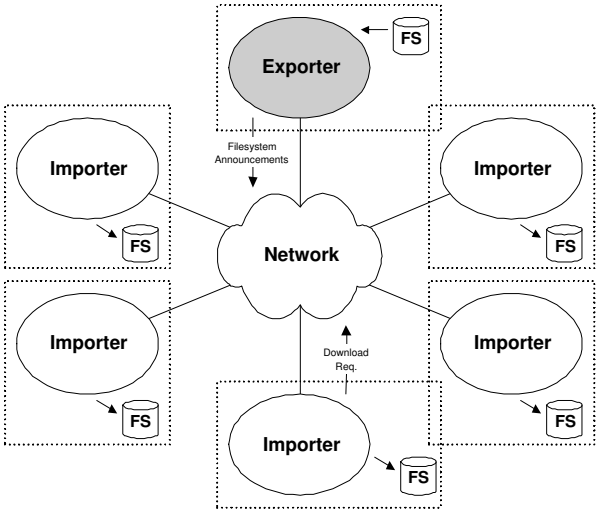

Figure 4: The Control Group - filesystem announcements are periodically broadcast by *exporters* to the control group. *Importers* noting missing or stale content request updates via download requests.

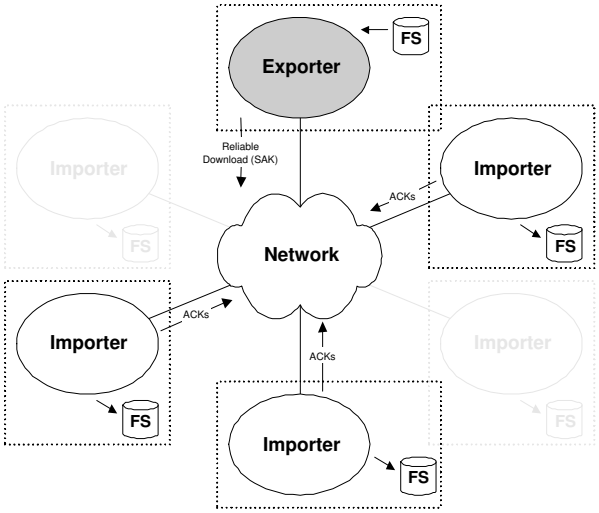

Figure 5: The Download Group - the per-file download group reliably broadcasts content to authorized importers (which are a subset of the control group) using a run-time developed policy.

4 EXAMPLE SYSTEM: AMirD DISTRIBUTED FILE MIRRORING

This section details the design and operation of the AMirD filesystem mirroring service. Built upon the group services available in the Antigone secure communication framework [17, 18], an AMirD exporter periodically distributes the contents of an AMirD file-system to an arbitrary collection of importer agents. In our implementation, IP multicast communication is used to amortize the costs of reliable distribution and support scalable replication.

While multicast-based replication affords efficiency, it complicates security. One possible solution to security at the communication level would be to encrypt the broadcasts after establishing a common key among all the importers. That may be adequate in some mirroring services, but we introduce an additional application-specific requirement that a subset of files must only be mirrored to the specified set of participants. How such policies are crafted in Ismene and mapped to management of multicast groupst at the communication layer is considered in depth in the following sections.

AMirD operates as follows. The exporter agent replicates (exports) a local directory tree (AMirD filesystem) to a set of importer agents. In general, an AMirD agent in our implementation can simultaneously import and export any number of filesystems. For ease of exposition, in this paper, we restrict ourselves to the environment in which there is only one exporter agent and multiple importer agents, and only one AMirD filesystem (identified by the path to the top directory) is being exported.

A control group is created by the exporter and used to communicate filesystem state and content requests (see Figure 4). The exporter creates a policy instance for the control group by reconciling the session policy and the domain policies of the exporter and the importers. The resulting policy instance is used to initialize the AMirD software services and to establish the group via enforcement of provisioning and authorization policy.

Exporters describe the contents of filesystems through periodic *export announcements* in the control group. Export announcements contain path information, modification times, and permissions of each directory entry. File entries are augmented with a collision resistant hash of the file content (e.g., MD5 [19]). Importers compare the announcements against the local filesystem state. Directories entries that are inconsistent with an announcement are updated (or cre-

ated) with the identified permissions and modification dates. Files and directories not identified in the announcement are removed. Stale file contents are detected via content hashes. Files whose content is consistent (as determined by the content hash) are updated with the permissions and modification dates as needed.

Resulting from the detection of missing or stale files, importers indicate the need for update by broadcasting download requests on the control group. Note that in a naive approach, *implosion* can result from many simultaneous requests; each importer is likely to detect and request update for an out of date or missing file at roughly the same time. The sudden burst of requests following an announcement can overwhelm the exporter, cause congestion, and ultimately delay update. Similar to techniques found in SRM [20], we mitigate implosion by randomly delaying requests (for 100-1000 milliseconds). Duplicate requests received prior to the expiration of the delay timer are suppressed.

Non-redundant download requests are queued for processing by the exporter. The exporter initiates a download group for each requested file (in the order that the associated requests are queued) by reconciling the appropriate session and domain policies. Hence, the policy used to protect file distribution can be derived from the file itself, the expected importers, or any other measurable aspect of the environment. The exporter broadcasts a download group announcement at the time at which the group is initialized. This announcement contains file and network addressing information for the spawned download group. The number of simultaneous download groups supported by an exporter is explicitly stated in the exporter agent configuration.

Depicted in Figure 5, the download group reliably distributes the file to the admitted importers. The current download protocol ensures content reliability through a AMirD-specific one-to-many selective acknowledgment protocol [21]. The use of other reliability techniques [22, 23, 20, 24] with different performance characteristics and delivery semantics may be integrated into AMirD as future needs dictate. The use of probabilistic reliability protocols (e.g., FEC [25] and SRM [20]) is currently under investigation [18]. Once the transfer is completed, the group is disbanded.

5 SECURITY POLICY DESIGN FOR AMirD SYSTEM

This section considers how policy-based software can be used to address complex and dynamic security requirements. We capture the diversity in threat and trust models encountered by general-purpose information services through example environments. The construction and enforcement of policies appropriate for these environments are studied. In investigating policy specification and enforcement, we endeavor not only to demonstrate the power of policy-based behavior, but also illustrate the flexibility of Ismene.

The Antigone group services define mechanisms for authentication, membership, key management, data handling, and failure detection and recovery [17, 26]. These mechanisms address particular aspects of group security. The selection of implementations, protocols, and configurations through mechanism provisioning defines the underlying communication service, and indirectly the threat model under which the application will operate. Similarly, authorization policy articulates a particular trust model for the application.

We begin an investigation of policy in AMirD in the next section by introducing policies for a range of diverse environments. Section 5.2 further explores the use of policy-based management in Ismene by considering in depth the construction and enforcement of these policies.

5.1 Deployment Environments

The following text introduces four hypothetical environments in which AMirD is deployed. Requirements and policies appropriate for these environments are described. Note that the policies represent singular views of environmental

requirements. Policy construction is a subjective enterprise. Depending on the nature and constraints of a given environment, any number of other policies will be appropriate. Such is the promise of policy defined behavior; alternate interpretations leading to other application requirements can be addressed through policy. The policies described in this section are presented in their entirety in Appendix A.

Local LAN

The Local LANs characterized by this environment exist within a single administrative domain. Depicted in Figure 6, an *enterprise internal network* supports users and services within a small geographic area (a single building). The network itself is protected with standard security infrastructure (i.e., firewalls, intrusion detection, etc.). The users within this community largely trust the local services and each other. However, as determined by the local filesystem access control, access to file content should be predicated on the UNIX read permission (i.e., by UID).

Note that, for this environment, the existence and characteristics of exported files may not be a concern. Hence, the control group can implement a low cost policy; content announcements need not be protected. Similarly, requiring users to be authorized before being allowed to participate in the control group is not necessary, and all control group messages are broadcast in cleartext.

The distribution of files within the LAN should be protected, but users participating in the distribution are trusted to not interfere with the distribution protocol. Thus, the download group can avoid the costs associated with integrity and authenticity guarantees. Because the network is relatively isolated, weak (and thus efficient) data security is acceptable. However, the level of secrecy should be commensurate with the sensitivity and value of the exported files. Users must be authenticated by the download group via UNIX UID (and allowed access where the UID has read permission).

Mobile Users

Mbile users place a number of unique security and performance requirements on AMirD. These members are assumed to exist in untrusted environments with often sporadic and limited connectivity to their *home enterprise*. Depicted in Figure 7, the AMirD service is implemented by a (hardened) gateway machine at the border of the home enterprise. Participants are expected to connect to the home enterprise for short periods during which filesystem synchronization occurs.

The security of the control group is driven by the need for secrecy and authenticity. Where characteristics of the filesystem are sensitive (such as the existence and names of technical documents), the control group must be confidential. However, if the existence of file content exposes little (such as email cache), confidentiality is not a concern. In all cases, the authenticity and integrity of control group communication must be preserved. The typically limited computational power of mobile devices combined with unreliability of the transport networks requires low cost mechanisms be employed.

Similar to the control group, the requirements of download groups are primarily driven by data sensitivity and resource constraints. Thus, content secrecy should be predicated on the sensitivity of the data transmitted. Due to resource limitations, low cost and robust protocols should be used for distribution. For example, it is likely that many download groups in this environment may contain only two participants (mobile users are likely to perform synchronization at their own schedule). Thus, the distribution protocol can be tuned for smaller groups.

Mobile users are trusted members of the home enterprise, and as such may be trusted with control group content. However, download groups must only allow access to members with local read permission. Note that exporters must of often map the credentials used for group authentication (e.g., certificates) to local filesystem access identities (e.g., UNIX id). Entities who's credential maps to a UID with read permission to the file are accepted into the download group.

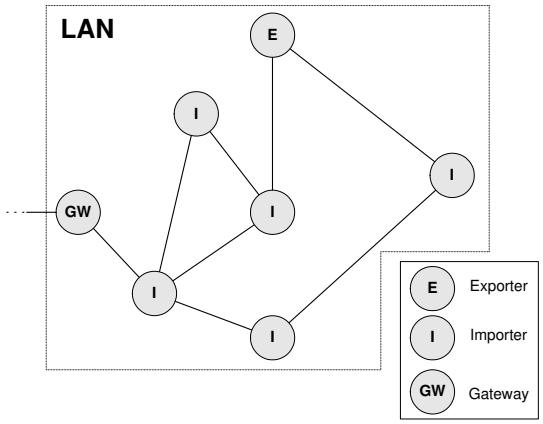

Figure 6: Local Lan - members and services in this environment are mutually trusted. Access to filesystem content is predicated on local filesystem access rights.

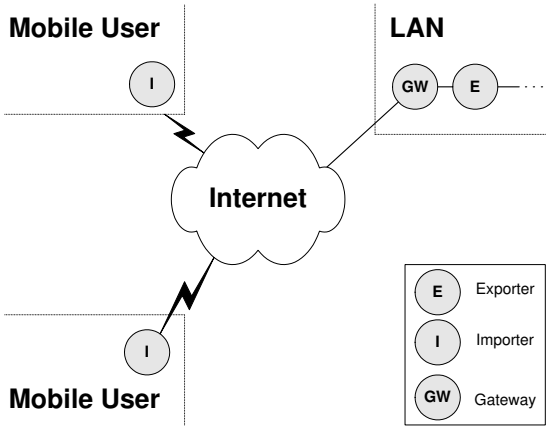

Figure 7: Mobile users - this environment use AMirD to synchronize mobile devices with filesystems in their home environment.

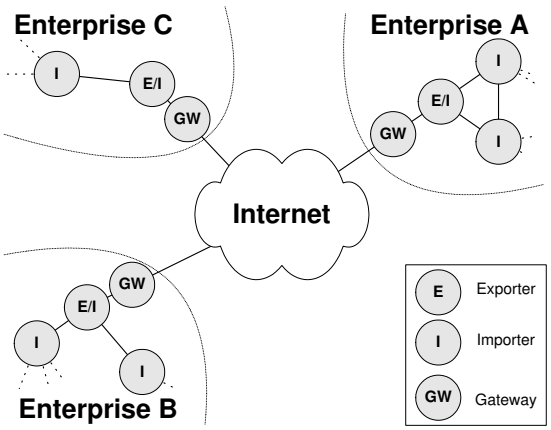

Figure 8: Coalition - The enterprises comprising the coalition have conditional and fluid trust. The policy under which the control and download content is distributed is a reflection of this trust.

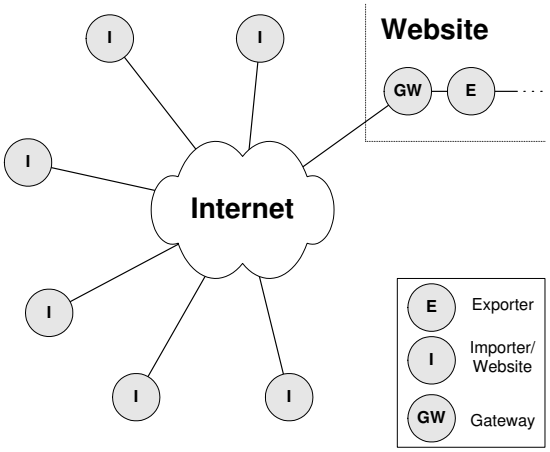

Figure 9: Website - web content is synchronized to a large body of largely untrusted mirror sites. The authenticity of the content is of paramount importance.

Coalition Networks

A coalition network allows independent enterprises to share information in a secure and controlled manner [27]. Depicted in Figure 8, the example coalition contains separate networks communicating over the Internet. Each enterprise shares information with the other enterprises via a single AMirD gateway host. The gateway host imports the filesystems of the other enterprises in the coalition group, and exports a local filesystem. Each enterprise maintains an enterprise internal session within which the external enterprise filesystems are exported to local hosts.

The coalition session communicates over a potentially hostile network. Moreover, there is limited and fluid trust between the coalition partners. For these reasons, the group must be able to make progress in environments in which external adversaries or partners attempt to disrupt the group. Hence, the coalition session should employ secrecy, integrity, and authenticity guarantees. Strong cryptographic algorithms should be used to protect the potentially sensitive announcements and content. The coalition must be able to eject misbehaving members, which is often implemented

through an appropriate group key maintenance policy.

The coalition session should allow only negotiated coalition behavior. For example, it may be necessary to restrict access to filesystems to only those coalition partners to which the files are important and necessary. Moreover, it is unlikely that all enterprises will use the same mechanism to evaluate access. Thus, a "lingua franca" must be agreed at the point at which the coalition group is initiated. The reconciliation algorithm determines, based on the enterprise local policies, which (authentication and provisioning) mechanisms are appropriate for the session.

Enterprise local groups operate entirely within the local scope. As such, the policies may be constructed in a manner similar to that identified in Local LAN. However, it is likely that allowing uniform access to (external) imported filesystems is insufficient. Thus, it may be necessary to further restrict access by partitioning imported filesystems into smaller filesystems exported to the local enterprise.

Site Mirroring

Website replication is increasingly being used to reduce client latency and network load [28]. Described in Figure 9, an AMirD *authoritative web server* distributes web content to a number of *mirror sites* over the Internet. Updates are announced at a configured schedule. Hence, replication is performed automatically during periods of low usage (e.g., maintenance windows). However, emergency updates are initiated without prior announcement as needed.

Because the content announcements of a mirrored website are unlikely to be sensitive, secrecy of control group communication is unlikely to be a chief concern. However, the authenticity, integrity, and freshness are necessary to ensure the correct and timely updates. Similarly, download groups associated with public web-sites simply require authenticity and integrity. Private or restricted web-sites (such as those provided by password or certificate protected content) may have very strict requirements for secrecy, authenticity, and integrity.

While the remote mirrors are not likely to be under the administrative control of the home website enterprise, it is likely that they can require a uniform set of services be supported by all mirrors. Thus, the provisioning of a session can be statically defined in a session policy provided by the home website, and accepted as mandate by the mirror sites.

It is important that web content be authentic. Thus, the access control policy should state that only the authoritative web server is allowed to export, and that the some guarantee that the content originated from the web server be provided (i.e., source authentication). Similarly, where desirable, one must ensure authorized mirror sites are allowed to import the web filesystems.

5.2 Illustrating Policy

The following text considers the flexibility with which the requirements defined in the preceding section are met through Ismene Policy. We briefly review the contents of the policy. The policy summarized in Table 1 and presented in Appendix A is used to define policy for each environment. Throughout this section, line numbers (#) refer to the policy presented in Appendix A.

As is true of any Ismene specification, evaluation of the policy begins with the `provision` tag. This clause states that the `antigone` and `application` sub-policies must be evaluated (5).

```
provision: :: antigone, monitor, application;
antigone : grouptype(locallan)   :: lanprov;
antigone : grouptype(mobileuser) :: mobileprov;
antigone : grouptype(coalition)  :: coalprov;
antigone : grouptype(website)    :: webprov;
% Error on non-matched grouptype predicate
```

The application implements the predicate `grouptype` and registers it with the Ismene's Policy Engine. The `grouptype` predicate is used to check the environment in which AMirD is currently running, and guides evaluation toward an envi-

Policy	Control Group		Download Group	
	mechanism	**configuration**	**mechanism**	**configuration**
Local LAN				
Authentication	nullauth		nullauth	
Membership	Antigone	no membership	Antigone	no membership
Key Management	KEK	static key	KEK	static key
Data Handling	Antigone	cleartext	Antigone	confidentiality
Failure Detection	*none*		*none*	
Mobile Users				
Authentication	OpenSSL		OpenSSL	
Membership	Antigone	no membership	Antigone	no membership
Key Management	AGKM	AES or Blowfish	AGKM	AES or Blowfish
Data Handling	Antigone	integ/(conf)	Antigone	integ/(conf)
Failure Detection	*none*		*none*	
Coalition Networks				
Authentication	OpenSSL		OpenSSL	
Membership	Antigone	membership	Antigone	no membership
Key Management	LKH	leave/eject/fail sens	KEK or AGKM	AES or Blowfish
Data Handling	Antigone	sauth/integ/conf	Antigone	sauth/integ/conf
Failure Detection	Chained FP		*none*	
Site Mirroring				
Authentication	OpenSSL		OpenSSL	
Membership	Antigone	no membership	Antigone	no membership
Key Management	AGKM	Blowfish	AGKM	Blowfish
Data Handling	Antigone	sauth/integ	Antigone	sauth/integ
Failure Detection	*none*		*none*	

Table 1: Provisioning Policy Summary - policies appropriate for the four AMirD environments.

ronment specific sub-policy (lines 6-10). The last line (10) (beginning with a '%' symbol[2]) denotes a comment stating that reconciliation will fail if no previous grouptype predicate evaluates to true. Further evaluation of sub-policies (see below) is directed by the `isControlGroup` and `isDownloadGroup` predicates identifying the type of group being initiated.

Triggered by the `lanprov` consequence, AMirD enforces a basic policy implementing a service similar to those found in contemporary group communication frameworks (lines 13-19). The evaluation of LAN policy results in the control group being largely unprotected; members are not authenticated (and thus trusted to provide their real identity), and all group text is sent in the clear. These features are implemented through the simple mechanisms for null authentication, key management (KEK [29]), data handling, and membership management. No authorization policy is enforced over the control group; anyone on the local network is free to join. This egalitarian policy is represented through clauses of the form (lines 21-32):

```
join : grouptype(locallan), isControlGroup() :: accept;
```

These clauses state that a member is permitted to perform any action within the group (including join). Note that an exporter is free to further restrict access as needed through reconciliation with domain policies containing additional authorization conditionals (requirements).

The `hasReadAccess` conditional regulates access to the download group (line 22). Semantically, the predicate determines whether the requesting entity has permission to access the download file. The user identity ($id) download ($file), and filesystem ($fsys) are *application attributes* asserted by AMirD at run-time. The predicate implementation maps the user identities to UNIX UID and GID. The predicate returns true if either the UID or GID has read access to the file or filesystem. The `inJoinPhase` predicate implemented by AMirD represents a further refinement download

[2] All Ismene comments begin with the percent symbol and are implicitly terminated with an end-of-line character.

group access. Members joining a download group in which a transfer is in progress cannot successfully complete the download protocol. For this reason, no members are permitted to join the group after the transfer is begun.

Mobile users require strong security (lines 35-59). Agents are initially authenticated via certificate over an OpenSSL [30] channel. Sensitive files and content (as classified by the application through the `isSensitiveFile` and `hasSensitiveFilesystem` conditionals) are kept confidential. How this confidentiality is maintained is driven by domain policy; the `pick` statement in the clause (line 38)

```
mkey : :: config(agkmkey(kychlen=64,rekeyperiod=60,hash=sha1)),
          pick(config(agkmkey(crypt=aes)),config(agkmkey(crypt=blowfish)));
```

states that all application traffic should be encrypted with either AES or Blowfish. Domain policies will inform the reconciliation algorithms on which of these options to implement to select. Where no guidance is provided, the algorithm arbitrarily selects the first value.

Access to the control and download groups is partially predicated on the assertion of X.509 certificates [31]. The `credential` test in the clause (line 48-50),

```
member_auth : grouptype(mobileuser), inlist($id, $ssl_acl),
              credential(&ca,issuer_CN=Antigone_SSL_CA),
              credential(&cert,subject_CN=$id,issuer_CN=$ca.subject_CN) :: accept;
```

states that the agent must provide a certificate issued by the known issuer `Antigone_SSL_CA`. Note that the enforcement infrastructure must perform validation internally; a certificate is should deemed valid only if a certification chain rooted by the identified CA can be found.[3] The statically defined ACL attribute (line 47),

```
ssl_acl := <member1:member2:member3:member4>; % ACL of Acceptable Members
```

further regulates group access. The ACL attribute (as used by the Ismene `inlist` conditional) enumerates the entities that have permission to access the group. Hence, one can view the credential test as enforcing authentication policy (establishing identity), and the ACL enforcing access control.

Access to other actions is predicated on the assertion of the cryptographic keys. The policy assumes two types of keys are established and maintained by the provisioned mechanisms. Established during authentication, *pairkeys* are ephemeral shared secret keys known only to the initiator and a joining agent. *Session* keys are negotiated between the agents by the key management mechanism and replaced as directed by policy. Knowledge of these keys is used as a form of authorization. For example, the clauses (lines 51 and 57)

```
join : grouptype(mobileuser), credential(&ky,name=$id) :: accept;
send : grouptype(mobileuser), credential(&ky,name=$gid) :: accept;
```

describe the keys required to join and send data to the group. The run-time asserted attribute `$id` is used to identify the user. The pairkey specific `name` attribute identifies the user or initiator. Hence, the credential condition is used to whether the pairkey associated with the joining member was used to request the join. Similarly, the `$gid` attribute identifies the current session key, and the credential test determines whether the session key was used to generate the message sent to the group. Hence, only entities in possession of the pair or session can join or send to the group.

The coalition environment consists of a collection of independent enterprises. Note that the services available to each enterprise may be very different. Hence, AMirD must allow groups to converge on an acceptable and interoperable policy. This is achieved for one aspect of the policy in the clause (line 72),

```
ckey : :: pick(config(agkmkey(crypt=aes)),config(kekkey(crypt=aes)));
```

[3] Revocation, and certificate management in general, present a number of difficult challenges [32]. For simplicity, we deem certificate acquisition, validation, and revocation to be outside the scope of this work.

which states that either AGKM [13] or a KEK mechanism may be used for key management. The decision of mechanism will be determined by the domain policies supplied by the coalition partners. Moreover, access to the group will be predicated on this determination; members whose domain policies are satisfied by the resulting decision will be free to participate. However, any member who requires a mechanism that is not selected will not participate.

Agents must assert a certificate credential issued by an authority indicating the right to participate in the control group. The clause (line 80),

```
member_auth : grouptype(coalition), inlist($id, $ssl_acl),
              credential(&ca,issuer_CN=Antigone_SSL_CA),
              credential(&cert,subject_CN=$id,issuer_CN=$ca.subject_CN) :: accept;
```

illustrates the use of *credential binding*. The first credential conditional binds all matching authority certificates to the name `ca`. The second clause states that the member should be allowed access only if the supplied certificate was issued by one of these authorities (bound in the second clause). Hence, through binding, we can delegate access for the control group; any entity issued a certificate by Antigone_SSL_CA is permitted to join.

The policy appropriate for a particular situation are not always simple. For example, the clauses (lines 86-91),

```
eject : grouptype(coalition), config(amember(ejecttype=pairkey)),
        IsServer($id) :: accept;
eject : grouptype(coalition), config(amember(ejecttype=pairkey)),
        Credential(&ky,name=$id), inlist($id, $eject_acl) :: accept;
eject : grouptype(coalition), config(amember(ejecttype=cert)),
        credential(&ca,issuer_CN=Antigone_Ejection_CA),
        credential(&cert,subject_CN=$id,issuer_CN=$ca.subject_CN) :: accept;
```

demonstrate how several access control methods can be used in conjunction. The first clause indicates that the server (initiator) should always be allowed to eject, and the latter two state that any member listed in the ejection ACL or who can produce a ejection authority certificate should be allowed to perform ejection. The `config` conditional states that all access to eject should be predicated on the ejection service being enabled in the membership mechanism.

The website environment implements a canonical distribution policy. Members are allowed into the group if they can provide an rights-proving certificate, and access to group functions is entirely predicated on this initial authentication. Hence, members are free to receive all content or none. Because the authorized agents are not trusted, the authenticity of the content is of paramount importance. Hence, the clause (lines 127-128),

```
content_auth : grouptype(website), credential(&ca,issuer_CN=$authorid),
               credential(&cert,subject_CN=$id,issuer_CN=$ca.subject_CN) :: accept;
```

states that the content must be received from the authenticated source. As directed by provisioning, the authenticity of the data is inferred from the stream signature (`satype` configuration parameter). However, it is not realistic to assume that any entity will ultimately be the authenticating body (CA) for all web sites. Ismene cannot *a priori* identify the content authority; the application must identify an authenticating body based on the content. An application signifies this judgment by identifying an authority through `$authorid` attribute. Hence, a member will only be able to validate web site content after obtaining the appropriate authority's certificate. Similarly, the clause (line 105),

```
wath : ::config(sslauth(interval=10,retries=2,crypt=blowfish,cafile=$author));
```

indicates that the application will assert the desirable certificate filename at run-time. Application attributes are replaced in provisioning statements prior to reconciliation. In this case, the CA filename will be asserted by the application. Hence, they are subject to the same evaluation processes as other policy defined configuration statements.

Application policies are relevant to (and enforced by) all environments. The reserved mechanism designator `applic` is interpreted by Ismene as application policy. All parameters defined in the instance for the `applic` mechanism are stored in the attribute set. Applications obtain the relevant policy by querying the attribute set at run-time. To illustrate, the group policy defines the clauses (lines 133-136),

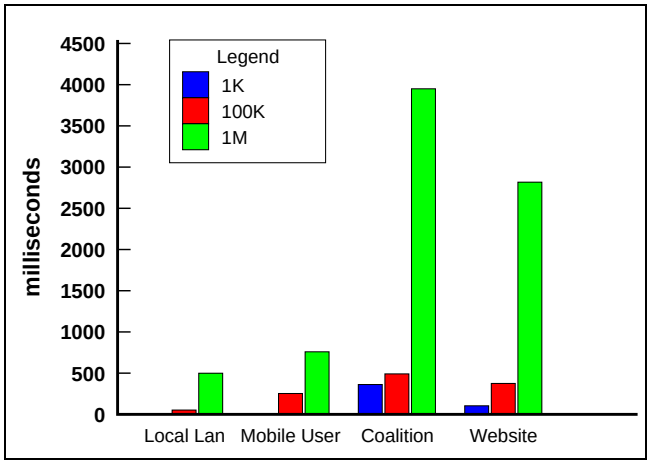

Figure 10: Reliable Broadcast Transfer - file transfer times in the example AMirD deployment environments.

Member	**Policy**			
	Local LAN	Mobile User	Coalition	Website
Member 1	68	151	155	149
Member 2	68	136	155	148
Member 3	68	152	156	164
Member 4	69	151	155	152

Table 2: AMirD Performance - time (seconds) to mirror two filesystems in the example AMirD deployment environments.

```
application : :: config(applic(followsymlinks=true)), apsauth;
apsauth : grouptype(locallan) ::
          config(applic(maxexportsubgroups=10, maximportsubgroups=10));
apsauth : :: config(applic(maxexportsubgroups=5, maximportsubgroups=5));
```

which indicate that symbolically linked files and directories should be exported. The latter two clauses place a maximum on the number export or import groups in which a agent may simultaneous participate. We consider how this policy affects performance in the following section.

6 PERFORMANCE CONSIDERATIONS AND DISCUSSION

To get experience with the framework, we implemented AMirD and all the policies given in the previous sections. Some of the research questions that we wanted to answer are:

1. Can applications reliably use Ismene to implement application-specific policies by defining their own actions, predicates, and mechanisms?

2. Is Ismene adequate to support security requirements for multiple environments?

3. Are there significant performance tradeoffs in configuring security policies?

The answer to the first two questions were positive. In AMirD code, we were able to define new predicates, actions, mechanisms without changing the underlying group communication services. One action that was added was `content_auth` for authenticating the source of file. Examples of predicates that were added include `grouptype`,

`isDownloadGroup`, `hasReadAccess` and `hasSensitiveFileSystem`. One application-level mechanism was added, called `applic` (lines 126-127) that started the exporter and the importer agents.

We now briefly address the last question. We profile the performance of AMirD under the environmental policies in two series of experiments, all carried out on a clustter of Pentium III Netfinity machines on a 100 Mbps network. We chose to conduct the experiments in the same network environment for all four policy scenarios, so that we could isolate the impact of choosing a security policy.

The first set of experiments estimate the cost of broadcast distribution and are summarized in Figure 10. These experiments measure the average time of 10 file transfers of 1 kilobyte, 100 kilobyte, and 1 megabyte files. The block size for all experiments is set to 1 kilobyte.

For all file sizes, the local LAN and mobile user policies exhibited significantly shorter transfer times than the other, relatively costly policies. The confidentiality guarantee enforced by the LAN requires data encryption. However, the cost of encryption only marginally delays transmission. In addition to confidentiality, the mobile user policy specifies integrity, which leads to slightly slower transfers.

In the test implementation, AMirD used a windowed acknowledgment distribution protocol (similar to TCP [33]), with a 1k block size and 100 packet window size. Thus, the transmission of small (1k) and medium (100k) files is completed through a single send/acknowledge exchange. Hence, in all policies, the transmission of such files is completed in less than 500 milliseconds. The difference between transfer rates between the LAN and mobile environments and the coalition and website is due to overheads associated with source authentication (as implemented by stream signatures). The coalition policy implemented a 250 millisecond data forwarding timer (used to amortize the cost of stream signatures) that delayed the packet transmission at both the exporter (data) and importers (acknowledgments). This timer lead to the longer transfers observed by the collation partners. The shorter 50 millisecond data forwarding timer used by mobile users yielded faster transfers.

Our second series of experiments attempts to characterize the long term effects of policy on AMirD performance. Table 2 describes the total time required by an AMirD agent to synchronize two filesystems to four receiver hosts. The first filesystem contains ten 1 megabyte files, and the second contains one hundred 1 kilobyte files. Note that each transfer is delayed by a 3 second join and 3 second shutdown protocol. The setup protocol is required to must allow ample time for members to join. The shutdown protocol is required to gracefully disband the group.

Synchronization times are partially a reflection of the file transfer rates and application policies. Simple, efficient policies (LAN and Mobile users) allow the file transfers to proceed rapidly. As defined by the application policy, the Local LAN supports more simultaneous download groups (10) than the other policies (5). The limited amount of processing required by the simple policy leads to greater opportunities to transmit data. As shown in Figure 2, the number of simultaneous transmissions dominates filesystem synchronization performance.

In practice, such experiments can be used to do tradeoff analysis between security policy and performance. Note that not all changes to a policy specification need to be security-specific. For example, number of simultaneous download groups impacts performance but not security.

7 CONCLUSIONS

Application behavior should reflect the often complex and dynamic requirements of the run-time environment. Security, in particular, is subject to changing requirements. Policy one way to address these requirements. However, a prerequisite to the proper use of policy is an understanding of their design and enforcement. In this paper, we have considered the construction, semantics and performance of non-trivial Ismene security policies. Ismene is a language and associated infrastructure for flexible security policy determination. We have explored how general-purpose secu-

rity languages such as Ismene can be used to specify and implement security policies for distributed applicationsin vastly different environments.

One advantage of a policy-based framework is that performance of an application can be more easily analyzed for different policies to determine tradeoffs between performance and security. To demonstrate the tradeoffs, we evaluated the performance of AMirD system using different security policies.

Acknowledgements

We thank our colleagues Alok Manchanda and Jim Irrer for contributions to the AMirD system and comments on earlier drafts of the paper. This work is supported in part by the Defense Advanced Research Projects Agency (DARPA) and Air Force Research Laboratory, Air Force Materiel Command, USAF, under agreement number F30602-00-2-0508. The U.S. Government is authorizeed to reproduce and distribute reprints for Governmental purposes notwithstanding any copyright annotation thereon. The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of the Defense Advanced Research Projects Agency (DARPA), the Air Force Research Laboratory, or the U.S. Government.

References

[1] T. Woo and S. Lam. Authorization in Distributed Systems; A New Approach. *Journal of Computer Security*, 2(2-3):107–136, 1993.

[2] M. Blaze, J. Feigenbaum, and Jack Lacy. Decentralized Trust Management. In *Proceedings of the 1996 IEEE Symposium on Security and Privacy*, pages 164–173, November 1996. Los Alamitos.

[3] L. Cholvy and F. Cuppens. Analyzing Consistancy of Security Policies. In *1997 IEEE Symposium on Security and Privacy*, pages 103–112. IEEE, May 1997. Oakland, CA.

[4] T. Woo and S. Lam. Designing a Distributed Authorization Service. In *Proceedings of INFOCOM '98*, San Francisco, March 1998. IEEE.

[5] T. Ryutov and C. Neuman. Representation and Evaluation of Security Policies for Distributed System Services. In *Proceedings of DARPA Information Survivability Conference and Exposition*, pages 172–183, Hilton Head, South Carolina, January 2000. DARPA.

[6] J. Zao, L. Sanchez, M. Condell, C. Lynn, M. Fredette, P. Helinek, P. Krishnan, A. Jackson, D. Mankins, M. Shepard, and S. Kent. Domain Based Internet Security Policy Management. In *Proceedings of DARPA Information Survuvability Conference and Exposition*, pages 41–53, Hilton Head, South Carolina, January 2000. DARPA.

[7] David C. Blight and Takeo Hamada. Policy-Based Networking Architecture for QoS Interworking in IP Management. In *Proceedings of Integrated network management VI, Distributed Management for the Networked Millennium*, pages 811–826. IEEE, 1999.

[8] S. Bellovin. Distributed Firewalls. *;login:*, pages 39–47, 1999.

[9] Yair Bartal, Alain J. Mayer, Kobbi Nissim, and Avishai Wool. Firmato: A novel firewall management toolkit. In *IEEE Symposium on Security and Privacy*, pages 17–31, 1999.

[10] M. Blaze, J. Feignbaum, J. Ioannidis, and A. Keromytis. The KeyNote Trust Management System - Version 2. *Internet Engineering Task Force*, September 1999. RFC 2704.

[11] S. Kent and R. Atkinson. Security Architecture for the Internet Protocol. *Internet Engineering Task Force*, November 1998. RFC 2401.

[12] P. Dinsmore, D. Balenson, M. Heyman, P. Kruus, C Scace, and A. Sherman. Policy-Based Security Management for Large Dynamic Groups: A Overview of the DCCM Project. In *Proceedings of DARPA Information Survivability Conference and Exposition (DISCEX '00)*, pages 64–73. DARPA, January 2000. Hilton Head, S.C.

[13] P. McDaniel. *Policy Management in Secure Group Communication*. PhD thesis, Univeristy of Michigan, Ann Arbor, MI, August 2001.

[14] P. McDaniel and A. Prakash. Methods and Limitations of Security Policy Reconciliation. In *2002 IEEE Symposium on Security and Privacy*, pages 73–87. IEEE, MAY 2002. Oakland, California.

[15] D. Durham, J. Boyle, R. Cohen, S. Herzog, R. Rajan, and A. Sastry. RFC 2748, The COPS (Common Open Policy Service) Protocol. *Internet Engineering Task Force*, January 2000.

[16] B. C. Neuman and T. Ts'o. Kerberos: An Authentication Service for Computer Networks. *IEEE Communications*, 32(9):33–38, September 1994.

[17] P. McDaniel, A. Prakash, and P. Honeyman. Antigone: A Flexible Framework for Secure Group Communication. In *Proceedings of the 8th USENIX Security Symposium*, pages 99–114, August 1999.

[18] P. McDaniel, A. Prakash, J. Irrer, S. Mittal, and T. Thuang. Flexibly Constructing Secure Groups in Antigone 2.0. In *Proceedings of DARPA Information Survivability Conference and Exposition II*, pages 55–67. IEEE, June 2001.

[19] R. Rivest. The MD5 Message Digest Algorithm. *Internet Engineering Task Force*, April 1992. RFC 1321.

[20] S. Floyd, V. Jacobson, C. Liu, S. McCanne, and L. Zhang. A Reliable Multicast Framework for Light-weight Sessions and Application Level Framing. *IEEE/ACM Transactions on Networking*, pages 784–803, December 1997.

[21] K. Fall and S. Floyd. Simulation-Based Comparisons of Tahoe, Reno, and SACK TCP. *ACM Computer Communication Review*, 26(3):5–21, July 1996.

[22] K. Birman. The Process Group Approach to Reliable Distributed Computing. *Communications of the ACM*, 36(12):37–53, December 1993.

[23] R. Van Renesse, K. Birman, and S. Maffeis. Horus: A Flexible Group Communication System. *Communications of the ACM*, 39(4):76–83, April 1996.

[24] O. Rodeh, K. Birman, M. Hayden, Z. Xiao, and D. Dolev. Ensemble Security. Technical Report TR98-1703, Cornell University, September 1998.

[25] L. Rizzo. Effective Erasure Codes for Reliable Computer Communication Protocols. *ACM Computer Communications Review*, 27(2):24–36, April 1997.

[26] P. McDaniel and A. Prakash. An Architecture for Security Policy Enforcement. Technical Report TD-5C6JFV, AT&T Labs - Research, Florham Park N.J., July 2002.

[27] G. Patz, M. Condell, R. Krishnan, and L. Sanchez. Multidimensional Security Policy Management for Dynamic Coalitions. In *Proceedings of Network and Distributed Systems Security 2001*. Internet Society, February 2001. San Diego, CA, (*to appear*).

[28] Kenneth Birman. Replication and Fault-Tolerance in the ISIS System. In *Proceedings of 10th ACM Symposium on Operating Systems Principles*, pages 79–86. ACM, 1985.

[29] H. Harney and C. Muckenhirn. Group Key Management Protocol (GKMP) Specification. *Internet Engineering Task Force*, July 1997. RFC 2093.

[30] The OpenSSL Group. OpenSLL, May 2000. `http://http://www.openssl.org/`.

[31] R. Housley, W. Ford, W. Polk, and D. Solo. Internet X.509 Public Key Infrastructure Certificate and CRL Profile. *Internet Engineering Task Force*, January 1999. RFC 1949.

[32] P. McDaniel and A. Rubin. A Response to 'Can We Eliminate Certificate Revocation Lists?'. In *Proceedings of Financial Cryptography 2000*. International Financial Cryptography Association (IFCA), February 2000. Anguilla, British West Indies.

[33] Postel J. Transmission Control Protocol - DARPA Internet Protocol Program Specification. *Internet Engineering Task Force*, September 1981. RFC 793.

Appendix A - AMirD Environment Policy

A complete description of the Ismene language, mechanisms, configurations can be found at the Antigone/Ismene project website at `http://antigone.eecs.umich.edu/`.

```
% Description : This is a AMirD group policy for the Chapter 6 scenarion.
%               policy processing algorithms.

% Antigone Group-based Policy Derivation
provision: :: antigone, application;
antigone : grouptype(locallan)   :: lanprov;
antigone : grouptype(mobileuser) :: mobileprov;
antigone : grouptype(coalition)  :: coalprov;
antigone : grouptype(website)    :: webprov;
% Error on non-matched grouptype predicate

% Scenario 1 - Local Lan
lanprov : :: lath, lkey, lmem, ldat;
lath : isDownloadGroup() :: config(nullauth(interval=10,retries=2,crypt=des));
lath : :: config(nullauth(interval=10,retries=2));
lkey : :: config(kekkey(rekeyperiod=65536,hash=md5,crypt=des));
lmem : :: config(amember(memdist=none,retries=3,interval=2));
ldat : isDownloadGroup() :: config(adathndlr(conf=true));
ldat : :: config(adathndlr(none=true));

join : grouptype(locallan), isControlGroup() :: accept;
join : isDownloadGroup(), inJoinPhase(), hasReadAccess($id,$file) :: accept;
member_auth : grouptype(locallan) :: accept;
leave : grouptype(locallan) :: accept;
shutdown : grouptype(locallan) :: accept;
eject : grouptype(locallan) :: accept;
key_dist : grouptype(locallan) :: accept;
rekey : grouptype(locallan) :: accept;
send : grouptype(locallan) :: accept;
group_mon : grouptype(locallan) :: accept;
member_mon : grouptype(locallan) :: accept;

% Scenario 2 - Mobile User
mobileprov : :: math, mkey, mmem, mdat;
math : :: config(sslauth(interval=10,retries=2,crypt=aes,cafile=ssl_ca));
mmem : :: config(amember(memdist=none,retries=5,interval=5));
mkey : :: config(agkmkey(kychlen=64,rekeyperiod=60,hash=sha1)),
          pick(config(agkmkey(crypt=aes)),config(agkmkey(crypt=blowfish)));
mdat : isControlGroup(), hasSensitiveFilesystem() ::
       config(adathndlr(integ=true,conf=true)), halg;
mdat : isDownloadGroup(), isSensitiveFile($file) ::
       config(adathndlr(integ=true,conf=true)), halg;
mdat : :: config(adathndlr(integ=true,conf=false)), halg;
halg : :: pick(config(adathndlr(hash=md5)),config(adathndlr(hash=sha1)));

ssl_acl := <member1:member2:member3:member4>; % ACL of Acceptable Members
member_auth : grouptype(mobileuser), inlist($id, $ssl_acl),
              credential(&ca,issuer_CN=Antigone_SSL_CA),
              credential(&cert,subject_CN=$id,issuer_CN=$ca.subject_CN) :: accept;
join : grouptype(mobileuser), credential(&ky,name=$id) :: accept;
leave : grouptype(mobileuser), credential(&ky,name=$id) :: accept;
shutdown : grouptype(mobileuser), credential(&ky,name=$gid) :: accept;
key_dist : grouptype(mobileuser), credential(&ky,name=$id) :: accept;
rekey : grouptype(mobileuser),credential(&ky,name=$gid) :: accept;
send : grouptype(mobileuser), credential(&ky,name=$gid) :: accept;
content_auth : credential(&ca,issuer_CN=Antigone_Content_CA),
               credential(&cert,subject_CN=$id,issuer_CN=$ca.subject_CN) :: accept;

% Scenario 3 - Coalition
coalprov : isControlGroup() :: cath, cmem, ckey, cdat, cfdr;
coalprov : :: cath, cmem, ckey, cdat;
cath : :: config(sslauth(interval=10,retries=2,crypt=aes,cafile=ssl_ca));
```

```
cmem : :: config(amember(retries=5,interval=5)), config(amember(ejectenabled=true)), memd;
memd : isControlGroup() :: config(amember(memdist=conf, intlen=60)),
       config(amember(ejecttype=pairkey,joinsens=true,leavesens=true,
                      ejectsens=true,failsens=true));
memd : :: config(amember(memdist=none,ejecttype=cert));
ckey : isControlGroup() :: config(agkmkey(kychlen=64,rekeyperiod=60,hash=sha1));
ckey : :: pick(config(agkmkey(crypt=aes)),config(kekkey(crypt=aes)));
cdat : isControlGroup() :: config(adathndlr(integ=true,conf=true)),
       config(adathndlr(sauth=true,satype=signpkt));
cdat : isDownloadGroup() :: config(adathndlr(integ=true,conf=true)),
       config(adathndlr(sauth=true,satype=online,frmsize=32,datfwd=250));
cfdr: :: config(afpchain(hash=sha1,maxdrophb=5,hbperiod=5,chainlen=20));

eject_acl := <member1,member4>; % Acceptable Pair Ejection Members
member_auth : grouptype(coalition), inlist($id, $ssl_acl),
              credential(&ca,issuer_CN=Antigone_SSL_CA),
              credential(&cert,subject_CN=$id,issuer_CN=$ca.subject_CN) :: accept;
join : grouptype(coalition), credential(&ky,name=$id) :: accept;
leave : grouptype(coalition), credential(&ky,name=$id) :: accept;
eject : grouptype(coalition), IsServer($id) :: accept;
eject : grouptype(coalition), config(amember(ejecttype=pairkey)),
        Credential(&ky,name=$id), inlist($id, $eject_acl) :: accept;
eject : grouptype(coalition), config(amember(ejecttype=cert)),
        credential(&ca,issuer_CN=Antigone_Ejection_CA),
        credential(&cert,subject_CN=$id,issuer_CN=$ca.subject_CN) :: accept;
shutdown : grouptype(coalition), credential(&ky,name=$gid) :: accept;
key_dist : grouptype(coalition), config(kekkey()), credential(&ky,name=kek) :: accept;
key_dist : grouptype(coalition), credential(&ky,name=$id) :: accept;
rekey : grouptype(coalition), config(kekkey()), credential(&ky,name=kek) :: accept;
rekey : grouptype(coalition),credential(&ky,name=$gid) :: accept;
send : grouptype(coalition), credential(&ky,name=$gid) :: accept;
content_auth : credential(&ca,issuer_CN=Antigone_Content_CA),
               credential(&cert,subject_CN=$id,issuer_CN=$ca.subject_CN) :: accept;

% Scenario 4 - Website Mirroring
webprov : :: wath, wmem, wkey, wdat;
wath : :: config(sslauth(interval=10,retries=2,crypt=blowfish,cafile=$author));
wmem : :: config(amember(memdist=none,ejectenabled=false));
wkey : isControlGroup() :: config(agkmkey(kychlen=64,rekeyperiod=60,hash=sha1));
wkey : :: config(agkmkey(crypt=blowfish,hash=sha1));
wdat : isControlGroup() ::
       config(adathndlr(integ=true,sauth=true,conf=false,satype=signpkt,hash=sha1));
wdat : isSensitiveSite($author) ::
       config(adathndlr(integ=true,sauth=true,conf=true,hash=sha1)), sapl;
wdat : :: config(adathndlr(integ=true,sauth=true,conf=false,hash=sha1)), sapl;
sapl : :: config(adathndlr(satype=online,frmsize=15,datfwd=50));

member_auth : grouptype(website), credential(&ca,issuer_CN=$authorid),
              credential(&cert,subject_CN=$id,issuer_CN=$ca.subject_CN) :: accept;
join : grouptype(website), credential(&ky,name=$id) :: accept;
leave : grouptype(website), credential(&ky,name=$id) :: accept;
shutdown : grouptype(website), credential(&ky,name=$gid) :: accept;
key_dist : grouptype(website), config(kekkey()), credential(&ky,name=kek) :: accept;
key_dist : grouptype(website), credential(&ky,name=$id) :: accept;
rekey : grouptype(website), config(kekkey()), credential(&ky,name=kek) :: accept;
rekey : grouptype(website),credential(&ky,name=$gid) :: accept;
send : grouptype(website), credential(&ky,name=$gid) :: accept;
content_auth : grouptype(website), credential(&ca,issuer_CN=$authorid),
               credential(&cert,subject_CN=$id,issuer_CN=$ca.subject_CN) :: accept;

% AMirD Application Policies
application : :: config(applic(followsymlinks=true)), apsauth;
apsauth : grouptype(locallan) ::
          config(applic(maxexportsubgroups=10, maximportsubgroups=10));
apsauth : :: config(applic(maxexportsubgroups=5, maximportsubgroups=5));
```

Information about the Authors

Patrick McDaniel is a member of the technical staff at AT&T Research in Florham Park, New Jersey. He received his Ph.D. in Computer Science at the University of Michigan, Ann Arbor in 2001. His research efforts have focused on distributed systems security, scalable public key infrastructures, and component architectures. As the architect and developer of the DARPA funded Antigone project, Patrick has investigated languages and architectures for security policy determination and enforcement in multiparty communication. Patrick's interests lie in experimental computer science focusing on systems evaluation, design, and implementation. Patrick is an active member of the IRTF working group developing standards for secure multicast.

Atul Prakash is a Professor in the Department of Electrical Engineering and Computer Science at the University of Michigan. Prof. Prakash received his Ph.D. from UC Berkeley in 1989 and subsequently joined University of Michigan as a faculty member. His research interests include security and groupware systems. He has actively worked in identifying and addressing security needs of emerging applications in various domains such as computer-supported cooperative work, mobile computing, and distributed systems. His research work has formed the basis of several systems, including Antigone for secure group communication and the SPARC and UARC collaboratories for scientific collaboration. Professor Prakash is a member of IEEE, ACM, and Usenix.