POLICY MANAGEMENT IN SECURE GROUP COMMUNICATION

by

Patrick Drew McDaniel

A dissertation submitted in partial fulfillment
of the requirements for the degree of
Doctor of Philosophy
(Computer Science and Engineering)
in the University of Michigan
2001

Doctoral Committee:

Professor Atul Prakash, Chair
Adjunct Professor Peter Honeyman
Research Staff Member Trent Jaeger, IBM
Assistant Professor Sugih Jamin
Associate Professor Paul Resnick
Principal Researcher Aviel Rubin, AT&T

For Megan

ACKNOWLEDGEMENTS

I would like to thank first and foremost Atul Prakash, Peter Honeyman, Avi Rubin, and Sugih Jamin who have taught me much of what I know about computer science. They have had a profound affect on my life and any future success I may have is due in large part to their efforts.

Atul Prakash has been my advisor since arriving at the University of Michigan. He has been a key figure in all aspects of my academic life, and drove me to be successful. Antigone and this thesis exist because of Atul's constant questioning of my assumptions and solutions.

Peter Honeyman introduced me to security. His unfounded belief that I could someday become a useful graduate student during my first years at Michigan allowed me to explore interesting research. Simply put, I would not have pursued a career in security research if it were not for Peter's advice and encouragement.

Avi Rubin has been a friend and mentor over the last few years. The time I spent at AT&T research was among the most value of my academic career. Avi has helped me to understand the process of completing and presenting research.

Many of the tools I use to perform research I learned from Sugih Jamin. Sugih demonstrated infinite patience while working with me early in my academic career. In particular, his many editorial comments taught me the basics of presentation.

I would also like to thank Trent Jaeger, Paul Resnick, Kevin Compton and Brian Noble for their help in understanding the execution of research. Their technical and philosophical assistance in formulating and completing the thesis has been invaluable. I would like to particularly thank Brian for his help in navigating the academic world.

Hugh Harney has been a friend, collaborator, and sounding board for ideas. Our phone calls during the final days of the thesis allowed me to maintain some sense of perspective. I value our friendship immensely.

I would like to thank my family for their seemingly inexhaustible well of faith and support. I wish to thank my mother for giving me the strength to persevere, and my father with the drive to be successful. I would like to thank Robert and Nancy who have always been there for me. Thanks to Sean and Matthew for remembering who I am. I would like to thank Dean and Donna for inviting me into their family and always treating me as one of their own. Thanks to Alissa, Liam, Thomas, Shannon, and Ella for helping me to understand what is important.

I would like to thank the National Security Agency, National Aeronautics and Space Administration, and the Defense Advanced Research Projects Agency for their financial support. In particular, I would like to thank Doug Maughan for his support and direction of the Antigone project.

Finally, I would like to thank Megan. She has served at different times as my personal cheerleader, critic, editor, psychologist, and caretaker. She has listened to my rants, put up with my moods, encouraged me during my failures, and celebrated my successes. More than any other single person, Megan has made the completion of this degree possible.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

CHAPTER 1

INTRODUCTION

Policy is a temporary creed liable to be changed, but while it holds good it has got to be pursued with apostolic zeal.

Mahatma Gandhi (1869 - 1948), [Gan22]

Advances in the use of digital technologies to carry out our personal and professional lives have changed the fundamental nature of communication. Where once sending electronic mail to your mother was considered an aberration, it has become a matter of course for many. However, these emerging technologies have a cost; we have sacrificed much of the privacy and security afforded by traditional forms of communication in exchange for the ease of online life.

Applications increasingly combat the limitations of existing Internet security and privacy by providing basic security services. However, the types of security provided are often the result of decisions made by either software architects or system administrators. As a result, one must accept the security and trust model embodied by the application.

Group communication, in particular, has suffered from the lack of security. Long used as a foundation upon which distributed systems are built, groups provide a single communication context under which participating entities can efficiently share information. The security requirements of groups are more complex than found in traditional peer communication; groups embody associations that are larger and more fluid than their pair-wise counterparts. Therefore, the mismatch between changing user needs and statically defined security is felt more forcefully and frequently in groups.

Security policies bridge the gap between static implementations and user requirements. A security policy is a statement of a communication participant's security desires, abilities, and requirements. In systems supporting policy expression and enforcement, each communication is subject to the reconciliation of the policies stated by all interested parities. The way in which communication security is served (i.e., the definition of the guarantees

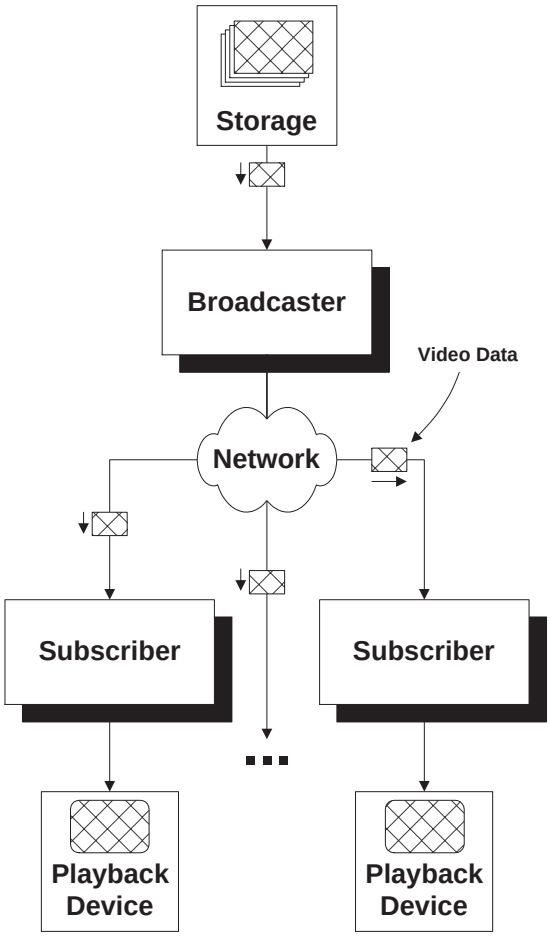


Figure 1.1: Scenario 1 - A pay-per-view service broadcasts a movie to a group of subscribing clients.

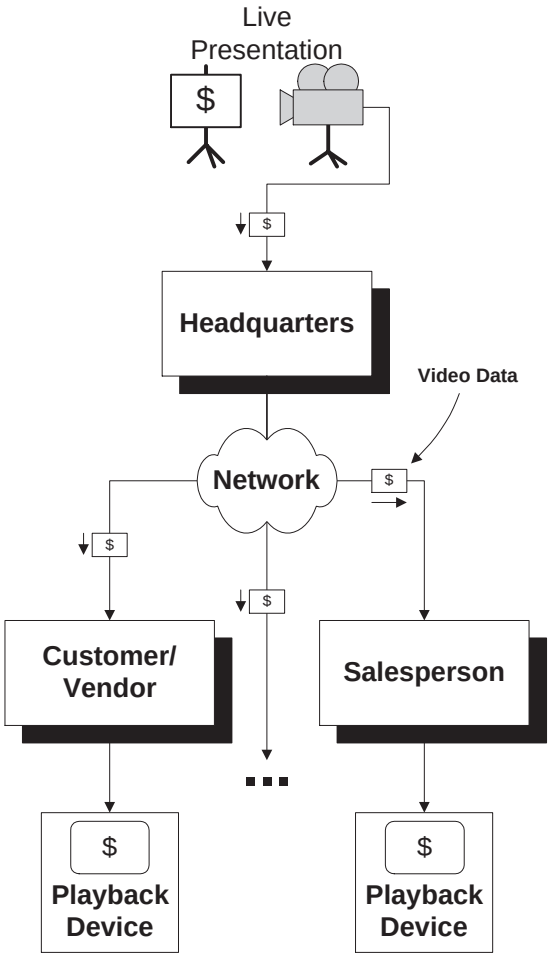


Figure 1.2: Scenario 2 - The headquarters company X broadcasts a series of presentations.

provided and the means by which they are achieved) is a direct result of the reconciliation process. Communication can occur where the requirements of all parties can be met through a single definition. Where not, the parties can choose to revise their policies or abort the session.

Standard security services for group communication have yet to emerge. This is due in large part to the security and user communities' inability to articulate a single model addressing the needs of many useful environments. However, policy may provide sufficient flexibility to make a secure general-purpose group communication framework feasible [HCM01]. Thus, through the precise definition and enforcement of policy, each session can implement a security model appropriate for its needs.

The following scenarios illustrate two distinct environments in which a group application

can be deployed. These environments, a pay-per-view broadcast and a sales-conference, represent very different kinds of sessions that are built upon the same video-conferencing application.

The pay-per-view service defined in scenario one and depicted in Figure 1.1 broadcasts movies to client subscribers. The video-stream is generated from persistent storage and transmitted at an established schedule. Clients playback received video content on local devices. Several secure group communication frameworks are designed specifically to address the requirements of this kind of pay-per-view application [BF99, Bri99].

In a pay-per-view environment, the ability to receive video data is predicated solely on client subscription. The broadcaster restricts access to the content based on evidence of payment. Clients arriving at a session present payment evidence (e.g., an electronic receipt), and receive the keying material necessary to view video content. All video content is directly or indirectly decoded from this keying material.

The pay-per-view session forms a loosely connected group. The server need not have knowledge of the subscribers viewing content over the course of the session. However, the server must be able to ensure that only paying subscribers can view video content. For the purposes of this example, there is no need to protect the group from past or future members; each client who has paid for the broadcast is free to view any part of the session. However, the server may wish to reduce the vulnerability of the session content to cryptanalysis by periodically changing the keys.

Client admission (i.e., payment, key distribution) is performed by a service external to the video source. Little, if any, interaction occurs between the server and clients. Clients assume their identity is shielded from other viewers (e.g., anonymity). In meeting this expectation, it is important that due diligence be exercised in protecting membership information from the clients and non-participants.

The video content itself must be protected. The broadcaster wishes to prevent eavesdropping by enforcing data confidentiality. In contrast, the client desires only the contents be delivered at a rate that makes viewing acceptable (i.e., playback frame-rate).

Figure 1.2 depicts a second scenario supporting a video sales-conference. A video source at the headquarters of company X uses the session to broadcast a series of presentations describing a new product line to its sales-force, suppliers, and customers. At the end of an initial product line presentation, the customers and vendors are required to leave the

session. Following their departure, the video source convenes a presentation informing the sales-force of the product line pricing structure.

Initially, the group should consist only of employees, suppliers, and customers. Thus, each participant must be authenticated prior to being admitted to the group. Similar to subscription, this requires that participants establish a means by which they can authenticate themselves before joining the session. As the expected participants are likely to have previously established a relationship with X, it would be efficient to use existing forms of authentication. Assume that X has issued digital certificates for all of its employees and passwords to all customers and suppliers. In this case, the admittance service authenticates potential session participants via certificate (e.g., using SSL authentication [Gro00]) or password.

The content and tone of each presentation is a direct result of the participants present. Hence, the video source requires knowledge of the membership throughout the session. Failure to provide membership information may lead to the (possibly undetected) disclosure of important information by X. For these reasons, company X additionally wishes to ensure that its customers and suppliers are not able to continue receiving content during the pricing presentation. In this case, the keys used to protect the video stream must be made unavailable to customers and suppliers at all points following the conclusion of the initial presentation.

As in the pay-per-view environment, the video feed is required to be confidential. However, additional protection may be needed. X may wish to prevent malicious or compromised participants from altering the video stream. This requires that any modification to the data be detectable (and ultimately discarded), and that the source of a message be accurately identified.

These scenarios highlight two of many session environments appropriate for groups. While the sessions use the same conferencing application to distribute video data, the session requirements dictate very different kinds of security. The description of each scenario defines a unique trust and threat model. A policy is the embodiment of these models. Architectures supporting policy construction and subsequent enforcement address the requirements of these and many other session environments in a flexible and secure way.

As it exists today, new environments require customized software or the acceptance of existing application security. The acceptance of existing application security can lead to un-

addressed requirements, and ultimately insecure operation. For example, an infrastructure designed for scenario one does not address all the requirements of scenario two. Similarly, applications can incur performance penalties where unnecessary security infrastructure is provided. A policy management infrastructure addresses these mismatches through the explicit specification and subsequent enforcement of application policies meeting session requirements.

While recent advances have increased the quality and availability of standard security services for peer communication (i.e., IPSec [KA98], SSL [Gro00]), similar advances in group communication have not been forthcoming. Policy-based group communication systems present us with a unique opportunity: using previous successes and failures as a guide, we can accelerate the adaptation of group communication as a building block of distributed systems. The flexible security afforded by policy management allows new approaches to be constructed and deployed rapidly. These frameworks require that policy specifications, rather than the applications or infrastructure, be modified to take advantage of new security services.

The goals of this thesis are summarized in the following statement:

> An essential aspect of group security is the definition of infrastructures in which the abilities and requirements of communication participants can be efficiently reconciled and enforced.

Hence, this thesis investigates the algorithms, construction, and performance of architectures supporting policy *determination* and *enforcement* in secure group communication. The investigation of policy determination considers the form, meaning, and processes used to represent and derive group and member security requirements. The investigation of policy enforcement considers the means by which the requirements are addressed.

Policy management is not a panacea. The fundamental costs associated with systems supporting policy often take the form of infrastructure complexity and system performance. Hence, a central goal of this thesis is quantification of the costs of policy management.

1.1 Secure Group Policy

Policy is used to address changing user and environmental needs. Through policy, software services state and reconcile the (sometimes conflicting) needs of all communication partic-

ipants in real time. Each session occurs within the context of a shared policy defining the acceptable behavior and requirements of its participants. Thus, rather than relying solely on the system designers or network administration to define service behavior, the interests of all parties are considered at the point at which communication occurs. While this thesis focuses on the definition and enforcement of policies appropriate for secure groups, the identified techniques are likely to be applicable to other contexts.

A group is defined as a collection of collaborating entities communicating over a real or virtual broadcast medium[1]. Early work in group communication focused on the study or reliable communication [KT91, Bir93, RBM96]. Often, groups are organized around a single *session leader*[2] and potentially many *group members*. The session leader directs all group behavior by coordinating the addition or ejection of members, the ordering of messages (where delivery semantics are implemented), and recovery from lost messages or failed members. Group members are clients acting at the behest of the session leader. More flexible models have been investigated as group communication has matured. Recent work has investigated groups containing multiple or backup session leaders [DM96] and hierarchies of subgroups [Mit97]. In the extreme, an egalitarian (or peer) group consists of a collection of equal partners [SSDW98].

The advent of the Internet heightened concerns over the lack of security available to groups. The first protocols implementing secure groups applied well-understood techniques previously used in peer communication. This led to useful, but limited, security [HM97b, HM97a, Bal96, Gon96]. Other works focused on the development of techniques for secure reliable group communication [Rei94], or on techniques addressing security requirements unique to multiparty communication [PSTC00, WGL98].

In this thesis, a group security policy is defined as the statement of the entirety of security-relevant parameters and facilities used to implement the group. This best fits the viewpoint of policy as defining how security directs group behavior, which entities are allowed to participate, and what mechanisms will be used to achieve mission-critical goals. Note that this definition is not restricted to electronically distributed statements; facets of

[1] Typically, group broadcast is implemented by a transport layer protocol such as IP multicast [Dee89]. However, where unavailable, groups can be implemented over a logical broadcast channel built on point-to-point protocols [Fra99, BCG+00, CRZ00, JGJ+00, CMB00].

[2] No single name has been accepted for the group authority. Largely defined by their responsibilities within a given system, other labels include; controller, arbiter, and sequencer. Throughout, this entity will be referred to as the *session leader*.

policy are often implicitly stated through the design and configuration of software.

This definition differs from previous work in security policy. Authentication [TJM+99, RN00] and trust management systems [BFL96, BFIK99b, CFL+98] view security policies as statements of acceptable authentications and access control. Policy is specified and evaluated within a well-defined and often rigorously evaluated framework. However, provisioning and enforcement is largely outside the scope of these systems. Conversely, in *policy based networking* [SWM+99], a policy defines generalized rules for the configuration of network resources. Used for network management, these systems define how hardware and software present in a network are configured, and in the presence of changing environments, reconfigured.

Recent work in secure group communication embraces a more flexible definition of group policy. Policies in contemporary secure group communication systems [HM97b, HM97a, DBH+00, MPH99] are chiefly designed to dictate the provisioning of security mechanisms (e.g., parameters and algorithms for session keying). Although these systems provide increased flexibly in defining group behavior, they often dictate the trust embodied by the group. Moreover, it is not immediately clear that the enforcement infrastructure adequately addresses all session security requirements.

The approaches described in this thesis are evaluated through the construction and evaluation of the Antigone group communication framework. The core components of Antigone include the Ismene Policy Determination Engine and Antigone Policy Enforcement Architecture. Ismene defines the representation and processes used to derive security policy. The Antigone enforcement architecture implements a secure group communication service meeting the requirements stated in the derived policies.

Group policies in Antigone are created by *policy issuers*. The policy issuer is the logical owner of the group and is trusted to faithfully identify session requirements. Once specified, the policy is stored in a well-known and available *policy repository*. The repository commonly treats policies as opaque data. For example, LDAP [YHK95] may be used for this purpose.

A policy can be stated abstractly or conditionally. An abstract policy states some conceptual aspect of security that is to be achieved or implemented. For example, while a confidentiality policy is abstract, a DES encryption policy is not. A conditional policy indicates, based on the operating environment, which policies should be enforced. An exam-

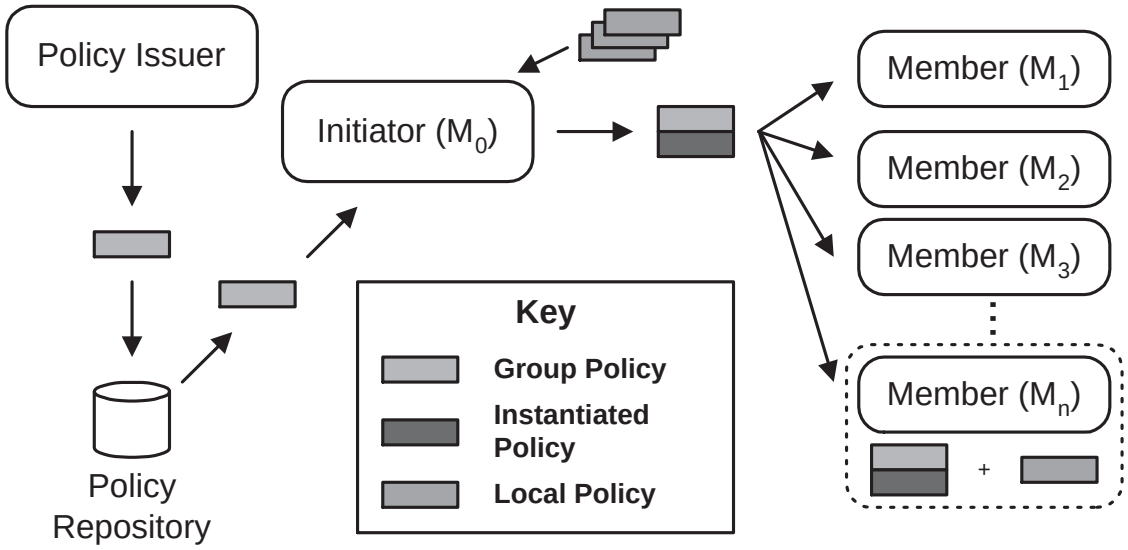


Figure 1.3: Antigone Session Policy - A session defining group policy is created by a policy issuer and stored in the policy repository. The initiator reconciles the local policies of the expected participants prior to session initialization. The resulting policy instantiation is transmitted to group members prior to their acceptance into the group.

ple conditional authentication policy states that employees of company X may participate during normal working hours, but not at other times.

A policy may be discretionary. Statements of discretionary policy allow the initiator some flexibility in defining the group. An example discretionary policy states that either Triple-DES [Nat99] or DESX [KR96] be used for confidentiality. The semantics of such a statement indicate that one of the two algorithms must be used, but not both or neither.

Each participant states its set of local requirements on a future session through a *local policy*. These policies explicitly enumerate the conditions upon which the member's participation is predicated. Similar to group policies, these statements may express provisioning, authentication, and access control requirements.

Figure 1.3 describes the use of policy over the course of one session of an Antigone group M. M is a centralized group containing a session leader M_0 and a set of members M_i (where $1 \leq i \leq n$ and n is the number of members of group). Note that depending on the user and system needs, other attributions of trust and group organization will be appropriate. Antigone is not restricted to groups of the type defined in this example; peer, hierarchy, or multiple session leader groups can be constructed from a collection of

appropriate mechanisms.

Based on a session announcement or invitation, M_0 begins session initialization by retrieving the group policy and the local policies of the expected participants. The group and local policies are reconciled to arrive at a *policy instantiation*. A policy instantiation contains unambiguous configuration directives and statements of authentication and access control.

M_0 initializes the session as directed by the instantiation. At some later point, a group member M_1 attempts to join the group. However, before joining, M_1 must first acquire the policy instantiation. This leads to a problem; if the policy must be protected (e.g., confidential), how does M_1 perform authentication before knowing the means by which authentication is required to be performed? In this case, it is necessary to establish a secure channel between M_0 and M_1 prior to the knowledge of the policy [HCM01]. This issue is addressed through the establishment of a publicly available authentication policy (see Chapter 5). M_1 obtains the instantiation following the authentication process.

Note that it may not be possible to derive an instantiation that meets the requirements of all members. Therefore, M_1 tests the *compliance* of the received instantiation with its local policy via a compliance algorithm. A compliant instantiation is consistent with all statements in the local policy. Failure of the instantiation to comply with the local policy can result in the modification of the local policy or the abstention of the participant from the session.

The enforcement of policy occurs in two phases. Initially the provisioning policy defines the mechanisms and configuration used to implement the group. The mechanisms and configurations used to support group communication are explicitly stated in the instantiation. The statements are used by M_1 to initialize all local interfaces and software. The second phase of policy enforcement occurs over the course of the session. The fine-grained authentication and access control policies defined in the instantiation restrict access to specific group actions. Based on supporting credentials and contextual information, these policies are used to assess an entity's right to perform actions within the group.

Any number of events occur over the course of a session. Those represented by messages require each receiver to evaluate sender authentication. For example, consider the member M_1 receiving a rekey message. M_1 must determine that some authority authorized to modify the session sent the message. The member maps the credentials and conditions associated

with the rekey message onto a set of rights. If the right to assert a new session key is granted, then the new session key is accepted.

1.2 Thesis Structure

The remainder of this thesis is structured as follows. Chapter 2 develops a definition of secure group policy and presents a taxonomy of group security requirements. Chapter 3 considers the technologies supporting security policy and group communication. Chapter 4 defines Ismene and analytically evaluates the algorithms used for determination. Chapter 5 describes the design and use of the Antigone policy enforcement architecture. Chapter 6 considers the expressiveness of Ismene via an investigation of the security requirements of the AMirD content distribution service in diverse environments. Chapter 7 evaluates of the costs of policy determination and enforcement within Antigone. Chapter 8 recapitulates the goals of this thesis and identifies avenues for future work.

This thesis investigates architectures and algorithms used for policy determination and enforcement in secure group communication. The main contributions are:

Identification of the policy space relevant to secure groups:

> This thesis presents a map of group security policies derived from an evaluation of group frameworks, applications and environments. Group security requirements can be categorized into policies defining *provisioning*, *authentication*, and *access control*. *Provisioning* policies define how the group addresses session needs by the appropriate configuration of security mechanisms. An *authentication* policy states the identities allowed to participate in the group. This policy not only identifies the parties that may participate, but also the conditions under which they may participate. An *access control* policy defines in the capacity by which authorized parties can participate. This largely defines the trust embodied by the group by mapping authorized participants or credentials to a set of relevant actions. A comprehensive survey of techniques used to define and enforce these policies is presented in Chapter 3.

A language and algorithms for multiparty policy determination

Policy determination in multiparty communication is more complicated than those found in peer environments. This is due in large part to the number of entities involved, the complexity of services used, and the multitude of entity relationships that must be supported. Converging on singular definitions that meet the requirements of all interested parties is non-trivial. The Ismene language and associated algorithms allow conditional and flexible statements to be used to derive a universally accepted policy. Moreover, unlike many previous policy approaches, policy encompasses the totality of the security context. The tractable nature of all algorithms used in the critical path of session creation and maintenance is determined via formal analysis. Confidence in the efficiency of these algorithms is further established experimentally.

The use of policy to define flexible group services

The means by which enforcement is performed ultimately determines the effectiveness of any policy approach. The Antigone framework presented in this thesis provides flexible interfaces for the enforcement of group security policy. Antigone constructs group services from software *mechanisms* that implement the basic services required by groups. The composition and configuration of mechanisms is directed at run-time by policy. Policy is enforced through the response to relevant *events* observed by the composed mechanisms. The design and limitations of event and component-based systems construction is evaluated, and policy implementing abstractions described.

The investigation of the requirements and costs of policy determination and enforcement

While many mechanisms designed to address specific security requirements in groups have recently emerged, little is known about their requirements and performance in integrated application environments. This thesis considers the policy requirements of group systems in several diverse environments through the systematic definition and implementation of the AMirD content distribution services. A number of policy alternatives are considered and trade-offs identified. A further exploration assesses the cost of enforcement. Experiments evaluating

the throughput and latency characteristics of Antigone have shown that policy is not in conflict with groups requiring high-performance networking.

CHAPTER 2

SECURE GROUP POLICY

The promise of policy is its ability to conform to changing environmental and user requirements. However, existing group security policy is limited in scope and flexibility; much of the group security is defined *a priori* in its implementation. This chapter considers the design space of group security policy. The processes used to develop and ultimately enforce policy are explored. A subset of the policy dimensions relevant to secure group communication is considered. The structure and operation of the *Ismene Policy Language* used by Antigone (see Chapter 4) is based on the results in this chapter.

The remainder of this chapter is organized as follows. The next section defines a group security policy through the decomposition of its expository requirements. Section 2.2 identifies the entities, algorithms, and life-cycle of a group security policy. Section 2.3 considers the general requirements of policy determination and enforcement. Section 2.4 presents a subset of the group policy design space derived from an investigation of contemporary applications, frameworks, and environments. Section 2.5 summarizes the chapter and revisits the goals of this thesis.

2.1 Secure Group Policy Definition

Security policy has been used in many contexts to define the means by which desired security properties are guaranteed. Often implicit, a definition of policy is derived from the goals of the system being designed. For example, trust management systems [BFL96, BFIK99b, CFL+98] are designed to evaluate authentication and access control using policies defining trust and delegation. Conversely, in the Route Policy Specification Language (RPSL) [ABG+98], policy is defined (in part) as sets of packet filtering rules.

Often, policy is defined as the statement of the entirety of security relevant parameters of a group. More specifically, this thesis adopts the policy definition found in the GSAKMP

specification, where:

> (a policy defines) ... the group security relevant behaviors, access control parameters, and security mechanisms. [HCH+00]

This definition best fits the viewpoint of policy as defining how security defines group behavior, which entities allowed to participate, and what mechanisms will be used to achieve mission critical goals.

In meeting this definition, any policy specification must explicitly define the following aspects of group security [MHC+00]. Note that these aspects may be stated using simple or implicit rules, but no aspect may be left unspecified.

1. **Identification** - The group must have some means by which it can be unambiguously identified. Failure to correctly identify the group policies, messages, and participants can lead to incorrect and insecure operation.

2. **Authentication** - A group policy must be able to identify the entities allowed to perform action. Thus, each operation within the group must be performed in the presence of some identifying context (e.g., credentials). An authentication policy states the required context.

3. **Access Control** - A mapping of contextual information to allowable action must be specified. An access control policy defines this mapping. For example, a simple access control policy states that any application level message encrypted under the session key should be accepted. This policy could be extended to accept only rekeying messages signed with session leader private key.

4. **Mechanisms** - Each policy must identify the security services and parameters used to implement the group. The configuration of these services determines the performance and robustness of the resulting solution, and greatly influences the quality of the security provided to the group. Each service can have further parameters and policies unique to its implementation. An example mechanism policy identifies the algorithms and protocols used to derive session keys (e.g., a key management mechanism; see Chapter 5).

5. **Verification** - Each policy must present evidence of its validity. The means by which the origin, integrity, and freshness of the policy is asserted (e.g., via digital signatures) must be known prior to its acquisition.

Policy is stated at varying levels of abstraction. Explicit policy systems define all aspects of the group in concrete terms (e.g., GKMP data structure [HM97b, HM97a]). Conversely, abstract policy systems provide interfaces for the formulation of abstract policies (e.g., as found in the Policy Working Group framework [SWM+99]). An abstract policy makes a statement defining a requirement of group operation, but does not specify how that requirement is to be achieved. For example, a policy can state that a *strong confidentiality* service be used for the transmission of application messages. How the group achieves strong confidentiality is subject to the interpretation by the policy infrastructure. Often, statements in the policy itself direct the mapping of abstract policies to concrete configuration.

2.1.1 Local Policy

A local policy defines the capabilities and requirements of each potential group member. This policy identifies, at a minimum, the services and credentials available to the local member. The member assesses of whether they will be able to participate in the group using local policy. A member incapable of implementing the group policy cannot participate in the group.

A local policy also states the local requirements that must be met by the group. These requirements can be viewed as a set of minimal standards placed on the group by the member. Participants determine- whether the group policy meets these requirements through a *compliance* test. The means by which the compliance test is performed is a direct result of the policy representation and semantic. Participants encountering a group policy that does not comply with the local policy can either refrain from participation, revise their local policy, or attempt to effect a change in the group policy through negotiation.

2.1.2 Policy Instantiation

Group and local policies need not be dictatorial; each may state the conditions under which a set of requirements are relevant or specify a range of acceptable behaviors. However, enforcement requires that a group operate under an unambiguous specification. Hence,

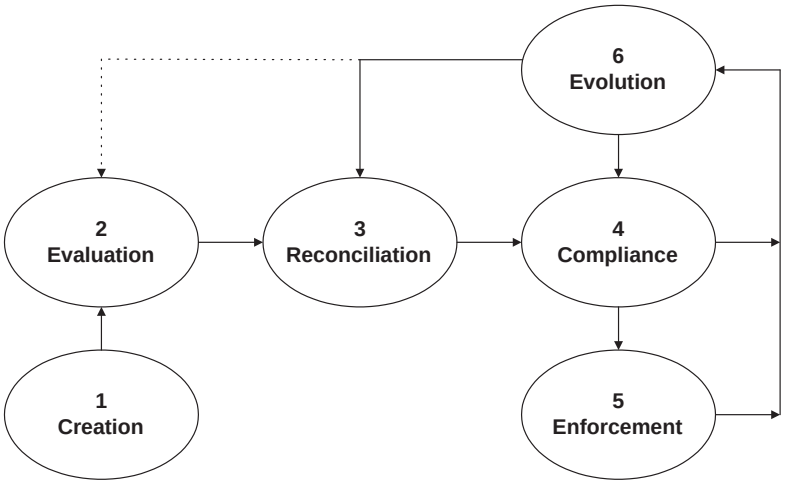


Figure 2.1: Policy Life-cycle - issuers specify policy during creation. The policy is subsequently interpreted towards an unambiguous instantiation identifying the group provisioning, authentication, and access control enforced by the group.

group participants must agree on a uniform policy (see policy requirements in Section 2.3). The means by which agreement is achieved is central to the design of any policy management infrastructure.

A *policy instantiation* is a fully specified group policy in which all configurations and statements of authentication and access control are explicit. The instantiation is the result of the interpretation of a group policy with respect to the evaluation of run-time conditions, local policies, and where supported, negotiation (i.e., *reconciliation*, see below). Policy instantiations can evolve as group requirements change. Evolution can occur in response to changes of membership, environmental conditions, or content sensitivity. The policy instantiation itself must specify the conditions under which evolution occurs.

2.2 The Life-cycle of a Group Policy

Group behavior is defined by interdependent and ongoing processes of policy determination and enforcement. Policy *determination* is the process used to derive session defining-specifications (i.e., policy instantiations). Group participants implement the semantic of established policies through *policy enforcement*.

Figure 2.1 and the following text present a view of the process used to construct and enforce a group policy; the ordering illustrates one possible policy life-cycle. Depending on environmental and security requirements, other orderings are possible and appropri-

ate. Where less flexibility is required, some processes can be omitted entirely (with the obvious exception of policy creation and enforcement). These abstract policy processes are [MHC+00]:

Creation - One or more informed authorities specify the group policy during creation. Policies can be abstract or specific to an implementation. A specific policy could state that all group messages be encrypted using the 3DES-CBC [Nat99] algorithm. Conversely, the (abstract) *strong confidentiality* policy could be mapped to the 3DES-CBC algorithm. While both policies could result in the same behavior, an abstract policy allows the group to select an alternative enforcement mechanism when 3DES-CBC is unavailable or inappropriate.

Evaluation - When a session is to be created, the group policy is retrieved and evaluated. Evaluation arrives at the set of acceptable configurations and statements of authentication and access control rules that *can be* used to implement the group. However, this may not represent a fully instantiated policy; rules may state ranges of acceptable behaviors.

Reconciliation - Further resolution of policy is necessary when evaluation does not result in a fully-specified policy instantiation (as is the case where ranges of acceptable behaviors exist). Reconciliation resolves these statements via consultation with local polices or through negotiation. Reconciliation attempts to find an instantiation that is consistent with the evaluated group policy and all local policies. However, it is not always possible to find an instantiation meeting the requirements of all policies. Therefore, the reconciliation process must implement a resolution discipline. The chosen discipline will directly determine which members can participate. Chapter 4 considers several reconciliation resolution disciplines.

Negotiation attempts to reach an acceptable instantiation via explicit participant co-ordination. For example, a negotiation protocol allows the group members to propose (and possibly counter-propose) new policies meeting group and local requirements. Once agreement of group policy has been reached, the session may proceed. If no agreement can be reached, those members with unaddressed requirements must choose to accept the group policy or refrain from participation. This thesis does not address policy negotiation protocols [DBH+00].

Compliance - Each member should test the compliance of a received instantiation with respect to its local policy. If the instantiation is not compliant, the member may request policy evolution, revise its local policy, or abstain from participation in the group. Note that compliance is an ongoing process; new instantiations resulting from policy evolution may fail to meet local policy requirements.

Enforcement - Enforcement occurs at each group member. This includes monitoring for relevant events and executing existing policies. For example, as directed by a rekeying policy, a group may distribute a new session key following each group member departure. Other policies, such as the 3DES-CBC policy described above, require only that mechanisms be correctly configured at the beginning of the session.

Evolution - Policy evolution occurs when some event requires modification of an existing policy. For example, the arrival of a member that lacks the ability to implement an existing policy may result in policy evolution. This process can lead to further evaluation, reconciliation, and compliance testing.

2.3 Requirements of Policy Management

The direct application of policy approaches used in two-party communication is unlikely to meet the needs of groups. This is due in large part to fundamental differences between peer and group policies; group policy conveys information about an association greater and more abstract than its pair-wise counterpart. The following text identifies and illustrates universal principles resulting from an analysis of group and peer communication policies [HCM01].

Principle 1: Enforcement of group policy must be consistent across a group

While it may evolve over the course of a session, the group requires a singular policy definition. Failure of members to operate under the same security context can lead to vulnerable or incompatible solutions. Similarly, policy frequently requires trust among the group members[1]. Each member trusts that all participants have been admitted, and that

[1]Several works have investigated support for groups whose members do not trust each other [Rei94]. However, the performance of the resulting solutions has been poor. The current mechanisms of Antigone implement only groups in which the members are largely trusted not to expose group information (e.g., session keys, policy, etc.). Other mechanisms designed to address other threats can be integrated into the framework.

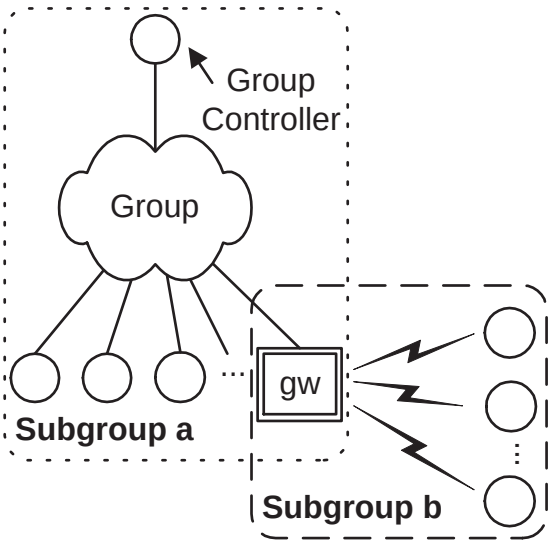


Figure 2.2: Mechanism equivalence - Policies implemented by members of Subgroup a must be equivalent to those implemented by Subgroup b. Failure to implement equivalent policies may result in undetected vulnerabilities.

they enforce the policy specification correctly. If a consistent view of policy cannot be established, members have no way to infer this trust.

Two facets of policy consistency are *mechanism equivalence* and *synchronization*. Two mechanisms are *equivalent* if a) they implement the same service (e.g., data confidentiality), and b) the security of the mechanisms is not qualitatively different. For example, Figure 2.2 describes a group implementing a confidentiality policy. Subgroup a (in the figure) implements confidentiality using a strong data encryption algorithm. Furthermore, a cryptographic gateway gw co-exists in both Subgroup a and a second Subgroup b. Subgroup b contains mobile devices with limited computing resources. gw translates all communication between the strong algorithm implemented by a to a weaker algorithm implemented by the mobile devices in b.

Clearly, an adversary attempting to uncover group content will mount an attack against data transmitted under the weaker algorithm. Thus, for this group, the confidentiality is only as strong as provided by the weaker algorithm. Because the algorithms are not equivalent, the security of the group as a whole is weakened. Worse, members of Subgroup a may be unaware of the use of the weak algorithm.

Session keying in two party communication is well understood [AN96]. Peer end-points

exchange a new key via an agreement protocol. Because both participants assert acceptance, subsequent use of the key is unambiguous. The issues, design, and vulnerabilities of two party key exchanges have been thoroughly researched and are well understood [LABW92].

Session rekeying in group communication is inherently more difficult. As defined by the group threat model, rekeying is triggered by security relevant events. Rekeying is often initiated, for example, when a session key lifetime is reached, following member arrival and departure, and to complete recovery from the compromise of a group member [MHDP00]. However, knowledge of these events is not often universally available.

Rekeying of the group is required to be *synchronized*. An arbitrary number of endpoints must reach agreement not only on the new secret key, but synchronize its subsequent use. For example, consider a group that has recently distributed a new session key. A member receiving a message encrypted under an old session key is faced with a dilemma; in the absence of synchronized delivery, the message may represent *a*) delayed delivery of a correct message encrypted under the old session key, or *b*) a message generated by an adversary who has gained access to a deprecated key.

Synchronization requires all received policies be fresh, authentic, and unmodified. The means by which policy freshness is assessed must conform to some *a priori* meta-policy. For example, group members could verify that a policy revision number increases monotonically. The group member would never accept a policy update with an unexpected revision number. Group members must be able to verify that the policy has not been modified during dissemination (e.g., integrity of received policies is preserved). This requires that each member be able to verify that a policy originated from an authoritative source. Of course, any policy must contain some evidence of its authenticity (i.e., policy verification). For example, a keyed message authentication code (HMAC) [KBC97] or digital signature [DH76] can be used to assert the authenticity of a policy.

Rekeying, and the synchronization of policy in general, are instances of *distributed consensus*. Agreement on the new session key or policy is reached through group communication protocols. However, in the general case, distributed consensus algorithms are complex and expensive [FLP85, Mul93]. Many existing group systems attempt to avoid these costs by relaxing synchronization requirements.

One way to relax key synchronization requirements is to allow several session keys to be simultaneously valid. For example, suspending transmission of data during rekeying in

a video-conference is highly undesirable [AAC+99]. So, a group may elect to continue to use (accept packets encrypted under an old session key) previous keys until consensus on the new key has been reached, if ever. Attacks that delay the consensus process can be mitigated by limiting the time a session key is used (and by direct corollary the minimal freshness of received messages). This approach and other relaxations to the general case of this *key transition* problem are considered in [MPH99].

Policy is also required to be synchronized. The policy a group member enforces must be identical at all members of the group. If not, then members may diverge from the session specification arbitrarily, introducing any number of vulnerabilities.

Principle 2: Only authorized entities can affect the group security context

Groups require an authorization model that is more complex than those commonly found in two-party communication policy (see Section 2.4.5). This fact is a direct reflection of the many actions that can alter the group security context. Because each of these actions can affect the security of *all* group members, they must be associated with the set of entities that are authorized to perform them. For example, a group allowing an adversary to perform security-relevant actions would be, among others, vulnerable in:

- *Policy creation* - The unauthorized entity can modify policy in arbitrary ways. Thus, the group may be manipulated into operating in an insecure way.
- *Key dissemination* - An unauthorized group controller can distribute bad, compromised, or old keying material.
- *Initiate rekey* - An unauthorized entity can perform a denial of service attack by forcing the group to rekey continually. The group would expend considerable resources performing key management functions.
- *Group destruction* - If an unauthorized group destruction command is accepted, the group will disband prematurely. Clearly, this represents a grave denial of service vulnerability.

Groups commonly designate one or more entities to act as authorities within the group. To illustrate, an entity wishing to join the group communicates with an entity authorized to admit members. A member is allowed into the group only after the admittance authority

verifies that the member possesses the appropriate credentials. However, unless otherwise specified by policy, the admitted member should not have the authority to admit other members. The group resulting from the admittance of the member represents a new security context; a new group member is trusted with the group key.

Principle 3: Group content must be protected

Data security mechanisms provide the means by which content confidentiality, authenticity, and integrity can be protected. These mechanisms implement protection by transforming content using cryptographic algorithms and session keys. Thus, the security of a group is predicated on the security of the processes restricting access to session keys.

As stated indirectly by principle 2, access to session keys must be restricted to entities with authority to receive them (i.e., through an authentication and access control policy). For example, consider a group policy stating that a member must prove possession of company X credentials (in the form of a public key certificate) before being admitted to the group. Thus, the associated authentication and access control policy directly or indirectly states: the entity must possess the private key of a certificate, the certificate must state that the organization of the entity is the desired company, and the certificate must be issued from the company's certificate authority.

An admittance authority enforces the access control policy (on a signed join request containing the certificate) by verifying the certificate organization and issuer fields, validating the signature, and checking that the certificate has not been revoked (e.g., through an appropriate certificate revocation service [Ken93, Koc98, NN98, MJ00]). If this process is successful, the member receives and subsequently uses the session key to communicate with the group.

Because the group policy is enforced correctly, and the underlying cryptographic algorithms are secure[2], group content protection is ensured. However, if any of these authentication, access control, or data security policies is incorrectly enforced, then the security of the group as a whole may be lost. This demonstrates the fragility of security; incorrect implementation of any one function can invalidate guarantees provided by others.

[2]There is significant debate on the correct design of secure group data transforms. For the purposes of this discussion, we assume that all mechanisms are fundamentally secure; the cryptographic algorithms and data transforms are sound.

Principle 4: Groups must be able to recover from security-relevant failures

It is necessary for groups to recover to a secure operating state when a subset of its membership is found to be untrustworthy. Thus, a policy must state the way in which compromise is detected and, if available, the mechanisms used for recovery. There are myriad ways that a group may recover from member compromise. Early systems disbanded immediately following compromise [HM97b, HM97a]. More recent, group systems employ sophisticated rekeying approaches to recover from member compromise [WHA98, WGL98, MPH99]. In these systems, compromised members are ejected by way of their exclusion from the subsequent rekey process.

A group may also require recovery from member failures. The effect of network partitions [DM96], process crashes [MP00], and other failures on the group security context is an open area of research. Mechanisms used for failure detection and recovery have unique security requirements. For example, the heartbeat-based chained failure detection mechanism requires heartbeats be authentic [MP00]. Otherwise, in the absence of authentic failure detection, an adversary may be able to mask the failure of group members through forged heartbeats.

2.4 Policy Design Space

A group policy defines the ways in which a session's security requirements and performance constraints are addressed by member applications. This section outlines policies commonly implemented by secure group communication systems. While not exhaustive, the enumeration of policies is representative of the services available in contemporary group frameworks and applications.

A *mechanism policy* defines the services used to achieve the group's mission-critical goals. These policies state what and how information is to be shared and the means by which these goals implemented. This section points out some of the important dimensions along which mechanism policies vary. Mechanism policy can be decomposed into: *session rekeying policies*, *data security policies*, *membership policies*, and *process failure policies*. A session rekeying policy defines how and under what circumstances group security contexts (e.g., session keys) are refreshed. A data security policy defines the security guarantees provided to application messages. Membership policies dictate the availability and guarantees

associated with the distribution of membership information. A failure policy defines the type of failures handled by the system.

An *authentication and access control policy* defines not only the entities that can participate in a group, but also the actions they may perform. Moreover, this policy states the prerequisites for participation; members can perform actions only after providing the appropriate credentials, and/or when a set of environmental conditions are met. Section 2.4.5 discusses the model and requirements of policies defining authentication and access control. The following sections consider these dimensions in further detail.

2.4.1 Session Rekeying Policy

A popular approach used to implement secure groups among trusted members is the distribution and subsequent maintenance of shared session keys. An important issue in the use of these keys is determining when a session must be rekeyed, i.e., the old session key is discarded and a new session key established. The session rekeying policy states the desired properties of the rekeying process. These properties indicate the lifetime and acceptable exposure of the session keys to past and future members of the group, and represent threats from which the group is required to be resistant. Four important rekeying properties are:

- *session key independence* - Knowledge of a session key does not provide any meaningful information about past or future session keys.
- *membership forward secrecy* - A member joining the group cannot obtain meaningful information about past group communication. This requires that knowledge of a session key does not give any meaningful information about past session keys (called *perfect forward secrecy*, PFS[3]), and session keys are replaced after each member joins the group.
- *membership backward secrecy* - A member leaving the group cannot obtain meaningful information about future group communication. This requires that knowledge of a session key does not provide any meaningful information about future session keys (called *perfect backward secrecy*, PBS), and that the session be rekeyed when any member leaves, fails or is ejected from the group.

[3]There are conflicting definitions of perfect forward (and backward) secrecy within the security literature. This thesis accepts Canetti et al.'s definition, where PFS (PBS) is defined as the property that compromise of a session key does not provide any meaningful information about past (future) content [CGI+99].

- *limited lifetime* - This property states that a session key has a maximum lifetime measured in time, bytes transmitted, or other globally measurable metric. Thus, a session key with a limited lifetime is required to be discarded (and the session rekeyed) when its lifetime is reached. Limited lifetime rekeying is typically used to combat cryptanalysis (i.e., limiting the amount of content transmitted under a session key reduces the amount of data available for cryptanalysis).

Session key independence is a prerequisite for both membership forward and backward secrecy. If the process through which a key is derived is not independent of other session keys, then a member may surreptitiously obtain past and future session keys.

Rekeying properties may be combined to define the desired group security. For example, a group may wish to enforce a policy with both membership forward secrecy and limited lifetime. Thus, current group content would be protected from future members, and each session key would have some maximum lifetime. The combination of these two properties identifies a particular threat model for the group (where the group is resilient to past members and the exposure of session keys to cryptanalysis is limited by the session key lifetime). Thus, as user environments evolve, their associated threat models may be addressed through composition of basic rekeying properties.

Session rekeying and group membership are clearly related. Applications often need protection from members not in the current *view*[4]. Therefore, as determined by the group threat model, changes in membership require the session to be rekeyed. If rekeying is not performed after each change in membership, the view does reflect a secure group, but indicates only the set of members that are actively participating in the session. Past members may retain the session key and send and receive the group content. Future members may record and later decrypt current and past content. Thus, applications that need protection from past or future members require rekeying after each relevant membership event.

A group security policy is *sensitive* to an event if the group changes the security context in response to the observation of the event. Typically, the security context is changed by distributing a new session key (rekeying). A group security policy is often sensitive to *group membership events*. Group membership events include:

[4]A group *view* is the set of identities associated with members of the group during a period where no changes in membership occur. When the membership changes (i.e., a member joins, leaves, fails, or is ejected), a new view is created. This is a similar concept to Birmans's group view [Bir93].

`JOIN` - triggered when a member is accepted into the group

`LEAVE` - triggered when a member leaves the group.

`FAILURE` - triggered when it is determined (or assumed) that a member has failed

`EJECT` - triggered when a previously admitted member is purged from the group.

With respect to a session key, policy sensitivity directly defines the group threat model. For example, consider a group model that is sensitive only to `EJECT` events. Assuming rekeying provides independence, because the session is always rekeyed after an ejection, no ejected member can access current session keys. However, the session is not protected from members that have left voluntarily, are assumed to have failed, or join in the future. This policy defines the *ejection secrecy* rekeying property.

Sensitivity mechanisms can be used to build a large number of session rekeying policies. The following text defines and illustrates four rekeying policies representative of those found in existing systems.

Time-sensitive Rekeying Policy

Groups implementing a time-sensitive policy periodically rekey based on a maximum session key lifetime measured by wall-clock time. Thus, the group limits the exposure of the session to cryptanalysis by using the key only for a limited period. Other kinds of limited lifetime rekeying operate essentially in the same way, save the means by which the lifetime is calculated. For example, a system supporting limited lifetimes based on bytes transmitted rekeys when a threshold of data has been transmitted under a particular session key. The GKMP [HM97b, HM97a] protocol implements a time-sensitive rekeying policy.

By periodically rekeying, the group is partially protected from an adversary who wishes to block the delivery of new session keys. An adversary who blocks rekeying messages may intend the group continue to use an old session key. If a new key is not successfully established after the current session key expires, members can suspend operation until a fresh key is obtained. The majority of existing secure group communication systems provide some form of limited lifetime rekeying.

The MARKS group key management system [Bri99] illustrates the use of time-sensitive rekeying. In this service, paying members receive session keys valid for a predetermined

subscription interval. Because knowledge of a session key is predicated only on member subscriptions, there is no need to support sensitivity to membership events. Members may join or leave the group without loss of security; they have the right to all content for which they have paid.

Typically, systems implementing limited lifetime rekeying (only) use *Key Encrypting Keys* (KEK) [HM97b] to reduce the costs of rekeying. KEKs do not provide session key independence. Because the KEK provides access to all session keys and content, the group is not protected from past or future members. Note that systems that use KEKs cannot forcibly eject members without additional infrastructure.

Another promising approach is to use KEKs only where no relevant membership events have occurred since the last rekey. In this way, a group is able to achieve the performance of KEKs when no loss of security results, and the strength afforded by other rekeying approaches elsewhere. Key hierarchies [WHA98, WGL98] use a similar approach to reduce the costs associated with group rekeying (where the root of the key hierarchy acts as a KEK).

Leave-Sensitive Rekeying Policy

Groups implementing a leave-sensitive policy rekey after `LEAVE`, `FAILURE`, and `EJECT` events. Leave-sensitive groups are resistant to the malicious past members (i.e., any member who has left the group does not have access to current or future content). For example, a business conferencing system that supports negotiations between a company's representatives and a supplier may benefit from leave-sensitive rekeying. Once the supplier leaves, a leave-sensitive rekey policy would prevent subsequent discussions from being available to the supplier, even if the supplier is able to intercept all the messages. The Iolus [Mit97] implements a form of leave-sensitive rekeying.

Groups that implement leave-sensitive policies must consider liveness requirements. Unless each group member periodically asserts its presence in the group, process failures cannot be detected. Hence, the group members may incorrectly believe they are communicating with failed member. The cost of these assertions may be high in large or highly dynamic groups.

Join-sensitive Rekeying Policy

Groups implementing a join-sensitive policy rekey only after `JOIN` events. Join-sensitive groups are resistant to the malicious future members (i.e., any member joining the group is unable to access past content). In large or highly dynamic groups, the cost of rekeying after each join can be prohibitive. For example, the number of receivers in a network radio broadcast is often large and little control over member arrival and departure can be asserted. However, the threat models associated with join sensitivity are not commonly found in applications such as broadcasting. Several techniques are commonly used to mitigate the costs of implementing join sensitivity (e.g., batched joins, minimum session key lifetimes [SKJH00]). In practice, a join-sensitive rekeying policy is likely to be used in conjunction with a time-sensitive or leave-sensitive policy.

Membership-sensitive Rekeying Policy

Groups implementing a membership-sensitive policy rekey after every membership event. Membership-sensitive groups are resistant to the malicious past and future members (i.e., joining members will not have access to past content, and that past members will not have access to current or future content). Thus, this policy is the combination of leave-sensitive and join-sensitive rekeying. Membership-sensitive policies achieve both membership backward and forward secrecy. Because each membership event triggers rekeying, the group view defines exactly those members who have access to current content. Membership-sensitive policies are often among the most expensive to implement. Thus, these policies are typically avoided unless strictly needed by an application. The RAMPART [Rei94] system provides a type of membership-sensitive service.

Other Rekeying Policies

Application events can influence rekeying. For example, in a business conferencing application, a policy may state that rekeying occur only when a member with the role `Supplier` leaves. In providing policies that integrate application semantics with rekeying, the group can achieve exactly the desired behavior at a minimal cost. Similarly, it may be important for the group to be more sensitive at certain times, but less at others. Similarly, groups may wish sensitivity to be a function of group size or resource availability. In this way, a group can adapt to the capabilities of the available infrastructure.

2.4.2 Data Security Policy

A data security policy states the security guarantees applied to application messages. The most common types of data security are: *integrity*, *confidentiality*, *group authenticity*, and *sender authenticity* [CGI+99]. These policies are essential for protecting application content, and define in large part the quality of security afforded by the group.

Integrity guarantees that any modification of a message is detectable by receivers. Standard reliable communication mechanisms (point to point TCP [J81], reliable group communication [FJL+97]) do not provide any integrity guarantees. Adversaries can trivially alter sequence numbers, checksums, and other components of these protocols to manipulate message content. The use of keyed message authentication codes (HMAC) [KBC97] is an inexpensive way to achieve message integrity.

Confidentiality guarantees that no member outside the group may gain access to session content. Although typically implemented through encryption under the session key, other techniques may be used to limit content exposure. For example, confidentiality may be achieved via steganography, or via encryption of only critical portions of messages.

Group authenticity guarantees that a received message was transmitted by some member of the group, and is typically a byproduct of other data security policies. In many cases, proof of knowledge of the session key (as achieved through most confidentiality and integrity guarantees) is sufficient to establish group authenticity.

Sender authenticity (also known as source authentication) guarantees that the sender of a message can be uniquely identified. Achieving sender authenticity is expensive using known techniques (e.g., off-line signatures [EGM96] and stream signatures [GR97]). Thus, for high throughput groups, sender authenticity is often avoided.

One may consider a number of other useful data security policies. For example, some systems may require non-reputability, where the originator of non-reputable data cannot later deny its source. Another policy is anonymity, in which the sender of a message specifically cannot be identified.

Closely related to data security policies, a *cipher-suite* policy is one or more cryptographic algorithms used to enforce specified policies. As encryption algorithms have varying availability and characteristics, a cipher suite policy specifies acceptable algorithms, parameters, and modes. A cipher suite policy is relevant not only to data security, but to any mechanism using cryptographic techniques.

Note that a single policy need not apply to every message. In many applications, individual messages have unique data security requirements, depending on the nature of the message and the assumed threat model. Thus, it is useful to provide facilities for the specification of data security policies at the granularity of a single message.

2.4.3 Membership Policy

Distribution of group membership is an important requirement for a large class of applications. For example, many reliable group communication systems need accurate membership information for correct operation. Conversely, as seen in typical multicast applications, members need not be aware of group membership at all. Here, providing other services such as reliability and fault-tolerance is often left to the application. Because each relevant change in membership requires the distribution of new group views, guaranteeing the correctness and availability of membership views can be costly.

A membership policy states the availability and accuracy requirements of view distribution. Views need be only as accurate as required by an application. Thus, it is useful to provide a range of membership guarantees with associated costs. Several useful membership policies include:

- *best-effort membership* - membership data is delivered as available and convenient. No guarantees about the accuracy or timeliness of this information are provided. However, it is expected that due-diligence is expended in providing accurate views.

- *positive membership* - guarantees that within specific time bounds, no member who has left, failed, or been ejected from the group is listed in the view.

- *negative membership* - guarantees that within specific time bounds, every member who has joined the group is listed in the view.

- *perfect membership* - guarantees that within specific time bounds, no member who has departed and every member who has joined is listed in the view. That is, both positive and negative membership information is provided.

Confidentiality of group membership is a requirement of some applications. However, concealing membership from members and non-members is difficult in current networks. This is primarily due to the ability of adversaries to monitor messages on the network.

These messages expose the source and destination of packets (in the case of unicasts) and at the multicast tree (in the case of IP multicasts). In mounting this *traffic analysis attack*, an adversary may deduce a close approximation of group membership [RR98].

2.4.4 Process Failure Policy

A process failure policy states the set of failures to be detected, the security required by the failure detection process, and the means of recovery. The defining characteristic of a failure detection mechanism is its fault model. The fault model defines the types of behavior exhibited by a faulty process that the mechanism detects. Typical crash models include fail-stop, message omissions, or timing errors [Mul93][5]. In *fail-stop* failures, a failed member immediately and permanently stops transmitting messages to the group. A message omission occurs when a process does not generate or transmit an expected message. Timing failures arbitrarily delay the generation or transmission of group messages. In the Byzantine failure model, a faulty process may exhibit any behavior whatsoever. Many variants of these models exist. For example, network failures can cause a group to be become partitioned, and no communication between partitions is possible.

Often, the failure detection process itself is required to be secure. In securing failure detection, the group is protected from the adversaries masking of process failures. However, protecting the group from an adversary who attempts to generate false failures may be more difficult. Failures may be forced by blocking all communication between the group participants. This *denial of service attack* is difficult to address in software alone.

2.4.5 Authentication and Access Control Policy

An authentication and access control policy states the prerequisites and conditions under which participants may perform group action. These actions, or *rights*, provide a road-map for the group operation. The definition of participant requirements, conditions, the enumeration of rights, and the mapping of identities and conditions to rights are the core components of an authentication and access control policy.

An *authentication policy* states the means by which members can establish their right

[5] Another widely accepted taxonomy defines failures within *time* and *value* domains [Pow92]. Time domain failures occur when an event is observed outside its expected (time) window. Value domain failures occur when an unexpected data value is observed.

to perform an action. Members are often authenticated at or before joining a group [NS78] using public key certificates (e.g., PGP [Zim94]), or through the use of centralized authentication servers (e.g., Kerberos [NT94]). In other applications, such as pay-per-view broadcasts, group members can establish rights through credentials obtained from application specific subscriptions [BF99]. In many cases, the true identity of the member need not be known (e.g., anonymous groups [SCP98, RR98]). The credentials provided during initial authentication typically serve as authorizing data throughout the session. However, where specific group rights (e.g., member ejection) cannot be inferred from these credentials, other authentication infrastructure is necessary.

An *access control policy* dictates the conditions under which an action can be performed. Historically, access control in groups has been course-grained; once admitted, the group member is able to perform any action within the group. This model is in direct conflict with the requirements of secure groups (see Principle 3 in Section 2.3). Hence, fine-grained access control is required by many groups to meet this goal. Environmental conditions often dictate when a member is allowed to perform an action [RN00].

For example, consider a group supporting space shuttle monitoring instruments at NASA, Kennedy Space Center. The KSC technical staff is free to use the group to calibrate instruments outside of launch windows. However, during launch windows, modification of instrument configuration is prohibited. Hence, a group access control policy supporting this environment must state that write access (group send) is not allowed during launch windows. The determination of a launch window is contextual; policy must state the conditions under which the send action may be performed[6].

Access control policies are defined by models gleaned from the organization of the participants or underlying communication infrastructure. For example, as presented in Table 2.1, the draft Secure Multicast Research Group (SMUG) policy framework assigns rights based on the assumed *roles* [MHDP00]. In implementing this *role base access control model* [SCFY96, SBCY97], the authentication and access control policy must state a) what authentication information and environmental conditions are required to assume a role and, b) what rights are associated with the assumed role. Other models and rights assignments are appropriate for groups with differing requirements. Hence, it is highly desirable to allow

[6]This scenario represents an issue observed at the KSC launch control facility. The failure to impose a context sensitive access control policy has limited the ability of technical staff to control launch monitoring equipment [Bee97].

Role	**Description**
group owner (GO)	group initiator and policy issuer
group key authority (GK)	controller of keying actions within the group
group membership authority (GM)	controller of membership actions within the group
member (M)	admitted member of the group

Action	**Description**	**Rights**			
		GO	GK	GM	M
key creation	create a session key/rekeying material		√		
key dissemination	distribute keying material		√		
rekey action initiation	initiate a group rekey		√		
key access	gain access to the session key		√		√
policy creation	create/assert a group policy	√			
policy modification	modify the group policy	√			
grant rights	delegate group rights	√			
authorize member	authorize/state member authenticity			√	
admit member	admit a member to the group			√	
eject member	remove a member from the group			√	
audit group	monitor group access information	√	√		

Table 2.1: SMUG Authentication and Access Control Policy - group members assume roles as defined by authorizing credentials. Access to group action is governed by the roles assumed by members.

policy, rather than the communication or security infrastructure, to dictate a proper model.

2.5 Goals

This thesis adopts the definitions and requirements of a general-purpose policy management infrastructure presented in this chapter. Based on these requirements, the goals of this thesis are:

- *Flexible Representation* - a group policy should encompass the entirety of the session security context. Hence, the policy representation (language) should allow the specification of both provisioning and authentication and access control. Moreover, the language must support both conditional and discretionary policies.
- *Multiparty Determination* - policy must be efficiently derived from the desires and abilities of all communication participants. This requires that a policy instantiation should be the result of the reconciliation of all participant requirements. Moreover, members must be able to determine the compliance of an instantiation with local policies.
- *Flexible Policy Enforcement* - the way in which security requirements are addressed is determined by the available technology and performance requirements. Therefore, any enforcement architecture should allow the integration of a wide range of security services.
- *Efficient Enforcement* - the usefulness of any policy management infrastructure is largely determined by its efficiency. Hence, overheads associated with policy enforcement must not significantly hamper application performance.
- *Transport Agnostic* - for numerous economic and technological reasons, multicast is not yet (and may never be) globally available [AAC+99]. Hence, it is important to provide alternative transport channels where standard services [Dee89] are not available.

This thesis does not address the following aspects of policy management:

- *Multiparty Negotiation Protocols* - negotiation protocols are an area of active investigation [KT93, MC94, WYL+99]. These works investigate techniques by which the participant can perform policy determination through coordinated communication.

- *Participatory Group Communication* - recent works have investigated self governed groups of peer participants [SSDW98, CC89, FN93, BD96, BW98, AST00] (e.g., egalitarian groups that do not contain a single guiding authority). While Antigone does not implement participatory group infrastructure, the architecture is not restricted to centralized groups.

- *Policy Storage and Retrieval* - Antigone assumes a service supporting storage and retrieval of group and local policies. Previous policy management services suggest numerous possible repository solutions (e.g., LDAP [YHK95]). It is expected that each environment use local information services for this purpose.

The remainder of this thesis considers the means by which these goals are met in a flexible and efficient way in Antigone.

CHAPTER 3

RELATED WORK

A number of efforts have touched on the issues of group policy management identified in the preceding chapter. Although these works have not fully addressed flexible determination and enforcement, they do provide insight into the problems and pitfalls of policy management infrastructures. This chapter considers services and technologies used to represent, construct, and ultimately enforce group security.

This chapter is organized as follows. The next section considers the design of existing policy management infrastructures that address various aspects of policy representation and determination. Section 3.2 reviews approaches for group formation and maintenance. Section 3.3 investigates services used enforce group security policies. Section 3.4 considers the design of component based middleware. Section 3.5 discusses technologies supporting broadcast communication.

3.1 Policy Management

Policy is used in many different contexts as a vehicle for representing authentication and access control [WL93, BFL96, CC97, WL98, RN00], peer session security [ZSC+00], quality of service guarantees [BH99], and network configuration [WSS+00]. These approaches define a policy language or schema appropriate for their target problem domain. Policy in these systems is largely *static*. Hence, the ability to alter behavior in the presence of changing conditions or requirements is limited.

Recent group communication systems support the notion of a group defining security policy that identifies the services and configuration of the group. The Group Secure Association Key Management Protocol (GSAKMP) [HCH+00, HCM01] defines an architecture and protocol used to implement secure multicast groups. Policy is implemented in GSAKMP through the distribution of session-specific *policy tokens* distributed to each group member.

Similar in spirit to the security associations (SA) of IPSec [KA98] used for peer communication, the policy token defines the security requirements and access control for a lightweight multicast session. Policy tokens define the entirety of the security provided to the group, and participant requirements are addressed during token construction (prior to the creation of the group). Hence, the token is the direct result of out-of-band specification, and cannot react to changing conditions or fluid group membership.

General-purpose policy reconciliation is largely unaddressed by existing policy based communication frameworks. In the two-party case, the emerging Security Policy System (SPS) [ZSC+00, SC98] defines a framework for the specification and reconciliation of local security policies for the IPSec protocol suite [KA98]. SPS supports the flexible definition and distribution of policies used to define IPSec (peer communication) security associations (SA). In SPS, policy databases warehouse and distribute specifications to policy clients and servers. Policy servers coordinate (with clients) the interpretation, negotiation, and enforcement of SA policies. The scope of policy in SPS is limited. To simplify, SPS policies state acceptable connections and identify the use of cryptographic message transforms (i.e. connection access control and message security). The IPSec SA policy approach has been extended to multicast groups in a Group Domain of Interpretation [BHHW01]. The GDOI specification adopts a policy definition modeled from the GSAKMP policy token.

Reconciliation need not be centralized. The DCCM system [DBH+00] reconciles local provisioning policies though negotiation. In the first phase of the DCCM negotiation protocol, an initiator sends a policy proposal to each potential member. Clients assess the proposal with respect to a local policy and transmit counter proposals. The initiator subsequently declares a final policy that potential members can accept or reject, but not modify. Policy proposals define an acceptable configuration (which, for particular aspects of a policy, can contain wildcard "don't care" configurations). An advantage of this protocol is that the local policy need not be revealed to the initiator. However, the authors of DCCM view authentication and access control as orthogonal to the definition of the group policy. Hence, no services are provided for authentication and access control policy negotiation.

Reconciliation has also been used as a means of coordinating the definition of independent sessions occurring within dynamic environments. Patz *et al.* describe a policy management system targeted to dynamic coalitions in [PCKS01]. The Multidimensional Security Management and Enforcement (MSME) system describes a model and processes

used to reconcile member Policy Level Agreements (PLA) towards a unified Resolved Policy Level Agreement (RPLA). All peer communication between entities within the coalition is governed by session definition policies identified in the RPLA.

Each PLA (and RPLA) in MSME is constructed from a top down policy model. In this model, security services describe the end system security goals (e.g., confidentiality, integrity, etc.), security mechanisms describe the means by which these goals are achieved (e.g., AES, MD5 based HMACs), and security implementations describe specific instances of the security mechanisms (e.g., IPSec). The policy for a particular communication is identified by mapping services to mechanisms, and ultimately to implementations. The acceptable mappings are identified in the member PLA.

PLAs consist of policy *rules*. Each rule describes a condition/action pair, where each action and condition consists of the tuple $[consumer, producer, service]$, where the consumer is an entity receiving the service, the producer provides a service, and the service describes some policy goal. For example, consider the following MSME policy rule;

$$R_1 : (bob, umdb, confidentiality) \rightarrow (bob, umdb, integrity)$$
$$R_2 : (x, web, transit) \rightarrow (x, CA_1, authenticity)$$

Logically, R_1 states that any communication from *bob* to *umdb* that is confidential must also have integrity guarantees. Rule R_2 states that anyone communicating with the *web* must receive authenticity guarantees from the CA_1 certificate authority.

To simplify, the PLAs of the members of the coalition are reconciled through intersection; all policy rules that are not in conflict and can be resolved to an implementation are added to the RPLA. After resolution completes, the result is distributed to the coalition members. A second model in which each member performs reconciliation is suggested, but a number of technical issues remain.

Policy Architecture

Traditional network management infrastructures are increasingly strained by the size and complexity of contemporary networks. The overhead of solutions that require network administrators manage each device independently is quickly becoming infeasible. One way to address this problem is to use policy to manage the network.

(a) *if* `network is congested`
then `drop all packets from subnet 1 received at border routers`

(b) *if* `congestion monitor counter > 5`
then `delete route sn1 /dev/eth0`

Figure 3.1: IETF Policy Working Group Representation - abstract (a) and translated policy (b) rules in an example schema.

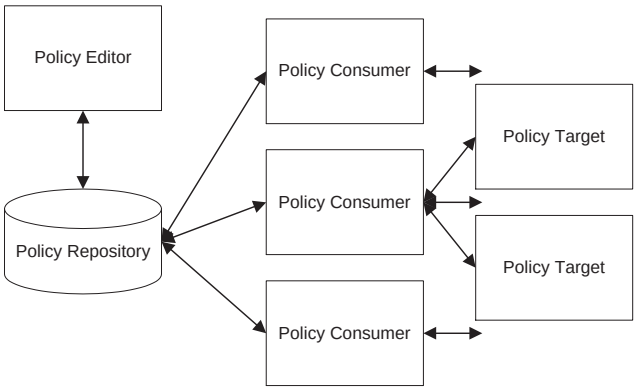


Figure 3.2: IETF Policy Framework working group architecture - architecture supporting the creation, distribution and enforcement of network management policies.

The IETF Policy Framework working group (PWG) is charged with developing an architecture and set of specifications for the management of network devices through abstract policies. The group's draft framework specification [SWM+99] identifies the entities and functions of policy implementing components for IP network devices.

Policy is represented in the PWG framework through sets of *policy rules* [DBC+00]. A policy rule is a conditional statement identifying a set of abstract configurations to be applied in those situations where the conditions are true. For example, policy rule (a) in Figure 3.1 states that border routers drop packets received from subnet 1 when network congestion is detected. This rule differs from traditional network management in that the rule is stated abstractly; the mechanism used to detect network congestion, how border routers are identified, and the means by which router interfaces are configured during times of congestion are outside the scope of the policy. The policy infrastructure must interpret and enforce these aspects of policy.

The reference architecture supporting the use of PWG policies is described in Figure 3.2. The architecture is defined over four types of policy components; *policy editors*, *policy repositories*, *policy consumers*, and *policy targets*. A policy editor provides interfaces for

adding, deleting, and updating sets of policy rules. The policy editor is responsible for detecting (and potentially resolving) instances of global conflicts. A global conflict occurs when two rules imply conflicting actions in the absence of any operational context. For example, rules stating that subnet A should get both best effort and guaranteed service for the same traffic constitutes a global conflict. The policy editor delivers rule updates to the policy repository.

The policy repository is a persistent store for policy rules. Policy consumers obtain relevant rules in the process of evaluating policy. In general, the policy repository does not act on or interpret policy rules. For example, LDAP [YHK95] is suggested as a reasonable repository solution.

A policy consumer is responsible for acquiring, evaluating, and translating policy rules into device-dependent action[1]. Given some operational environment, the consumer acquires relevant rules from the policy repository and evaluates their contents. The policy consumer is responsible for detecting local conflicts. A local conflict occurs when two rules conflict only in the presence of contextual information. For example, a rule stating that a device should be enabled only during a maintenance window may conflict with a second rule that states that the device should be disabled when a *denial of service* attack is in progress. Resolution of both global and local conflicts is an area of active research.

Rule evaluation is performed following the acquisition of relevant information through active probing of the network and target device. The policy is translated into device action (enforcement) when the conditions of the rules are satisfied. Generally, this requires the re-provisioning of the device itself. For example, the rule "drop all packets from the subnet 1 received at the border routers" is translated into individual commands that alters the internal tables of border interfaces of the relevant routers.

A policy target is an entity that alters its behavior in accordance with policy rules. Policy targets can have varying levels of support for policy. It is assumed in the above example that the border routers have no native understanding of the policy rules. However, in other environments, the targets might understand the policies themselves. Returning to the example described in Figure 3.1, a policy consumer might translate the abstract rule (a) into a device specific rule (b).

[1]It is suggested that some translation duties may be carried out by the policy target itself. In general, this does not significantly affect the operation of the architecture.

Note that the PWG architecture can be applied to other problem domains. Any policy infrastructure must support services implementing policy creation (editors) and policy enforcement points (consumers and targets). Antigone loosely adopts the PWG architecture, where policy issuers acts as editors (through the *apcc* compiler; See Chapter 4), and participants assume the roles of both policy consumers and targets.

Authentication frameworks regulate access to resources in distributed systems. Wong and Lam [WL98] introduce a generalized authentication service. derived from theory and design heuristics [WL93]. The resulting Generalized Access Control List (GACL) language and associated architecture is used to express and evaluate authentication requests in distributed environments. Central to the GACL architecture is a set of authentication servers to which evaluation of access control is delegated. The ability of an authentication server to authorize clients for an end service is established through a contract protocol. Authentication certificates are later obtained by clients (if the client is to be allowed access) and presented to end services (when services are used).

Authentication and Access Control Policy

Language-based approaches for specifying authentication and access control have a rich literature [WL93, BFL96, CC97, WL98, BFIK99b, RN00]. These approaches govern access by mapping identities, credentials, and conditions onto a set of allowable actions. The flexibility and granularity with which access control is stated is defined by the policy representation. Several authentication and access control languages currently used in distributed environments illustrate the issues and design trade-offs of these languages.

An authentication policy is expressed in the GACL authentication framework through the GACL language [WL98]. The design of GACL is based on a strong theoretical foundation; that is, with respect to the semantics of the language definition, each policy is guaranteed to be evaluated correctly. Requests for authentication are evaluated within the context of the current system state and credentials provided by clients. To simplify, all GACL policies are expressed as objects and lists of expressions defining access rights. Clients are authorized if, based on available information, the lists indicate that access should be granted. The evaluation of GACL lists depends on the evaluation ordering stated in its definition. The following GACL rule is *ordered*:

`my.exe`	**declare**	**ordered**
	list	$< Alice, Bob >, [-execute]$,
		$< [Dept], [execute] >$,
		$highload \rightarrow < [*], [-execute] >$,
		inherit `my.doc` $::< [*], [write] >$

The evaluation of an authentication request for object `my.exe` begins at the first clause. If the first clause (stating that both Alice and Bob should not be allowed to execute the `my.exe`) does not reject the action, the second clause (stating any person in the `Dept` group is allowed to execute) is assessed. The analysis proceeds through each clause until a definitive answer is obtained or no more clauses are defined. Unlike traditional ACLs where denial is always assumed, if no definitive answer is obtained the *failure* is returned. The third clause (designated as *highload*) tests system state through pre-defined predicates. The fourth clause defines an **inherit**(ed) authentication. This clause states that *execute* authentication is granted (or denied) only where it is indicated that write access is allowed to `my.doc`. Unordered rules are assessed by evaluating all clauses simultaneously. Failures are generated when conflicting authentications are detected.

In GACL authentication services do not by themselves have sufficient context to assess access control in distributed environments. This widely accepted philosophy implies that application context must play a role in the evaluation of access control policy. This realization is not unique to GACL; the vast majority of recent languages and frameworks provide interfaces for similar language independent condition evaluation [SCR96, BFL96, CC97, IKBS00, RN00].

The GAA API [RN00] defines a representation and programming interface for the specification and evaluation of access control and authentication in heterogeneous distributed systems. Policies are represented in the GAA API as tuples defining conditions under which rights should be granted to authenticated parties. Each clause, called an Extended Access Control List (EACL), is logically defined as a tuple consisting of an access identity, a grantor identity, a set of access rights, and a set of conditions. The access identity is the entity to be granted the access rights defined in the EACL. The grantor is the entity stating the rights. The set of conditions identify when the access rights are to be granted. Identities are stated abstractly; a user can assume the identity stated in the EACL based on authentication provided by another EACL. Conditional statements are either generic or specific. Generic conditions are expressed as parameterized applications of statically defined operators. The

Clause 1	
`Token Type:`	*authentication_mechanism*
`Authority:`	*security_office_manager*
`Value:`	*Kerberos.V5*

Clause 2	
`Token Type:`	*time_window*
`Authority:`	*pacific_tzone*
`Value:`	*6AM-8PM*

Clause 3	
`Token Type:`	*printer_queue_length*
`Authority:`	*printer_manager*
`Value:`	*¡ 10*

Clause 4	
`Token Type:`	*pos_access_rights*
`Authority:`	*network_manager*
`Value:`	*PRINTER_a1:submit_job*

Figure 3.3: GAA API clause definition - these example clauses define an authentication and access control policy for network printer stating: if the printer queue associated with `PRINTER_a1` contains fewer than 10 entries and the current time is between 6AM and 8PM, then any entity authorized by the local Kerberos KDC may submit jobs.

meaning of each of these operators is globally understood, so each conditional is internally evaluated by the GAA API. Conversely, specific conditions test the environment through application defined functions.

The use of the GAA API is illustrated in Figure 3.3, which defines an authentication and access control policy for the network printer `PRINTER_a1`. The policy for this printer is evaluated at the point at which a user submits a job. In making this request, the local environment possesses authentications associated with the requesting entity. If one of these authentications is from the local Kerberos 5 server, then Clause 1 is satisfied. Clause 2 is evaluated by checking the current time. Clause 3 is application specific. Thus, via an upcall, the application is asked if the current queue length has fewer than 10 entries. Finally, if Clause 1, 2, and 3 are satisfied, then the access rights stated in Clause 4 are given to the entity. In this case, the entity originating the request is granted the right to submit the print job.

An important contribution of GAA API is its abstract interface for authentication and access control. The heterogeneity of current networks requires that the assessment of both credentials and conditions be flexible. Hence, the generalized interfaces provided by GAA API eliminate the need to modify applications to make use of environment specific authentication and access control services. Moreover, the use of a single specification language across applications promotes a more complete and consistent definition of environmental security.

Developed within the larger DCCM architecture, the Policy-Based Cryptographic Key Release Project (KRP) [BB00] defines a language and architecture used to state and enforce

key release policies. Key release (e.g., access to cryptographic material) is defined as sets of access control and authentication rules. Policies are modified by events occurring in the system. Access to keys is allowed when the current state (identified by monitoring processes) satisfy the set of rules associated with the key.

KRP policies are organized into a tree defining organizational structures. For example, a policy stated by the University of Michigan supersedes an Electrical Engineering and Computer Science departmental policy, which in turn supersedes a policy stated by `pdmcdan`. Access is allowed (or denied) based on the traversal of the rules from the root of the tree to leaf keys. Thus, each policy must be satisfied on the path from the root towards the leaf nodes.

The KRP language is defined formally. The authors assess the *correctness* of a policy specification using the Prototype Verification System (PVS) [SCR96]. PVS is given a set of assertions about the semantics of stated policies. If the assertions hold for all possible instantiations of the policy, then it is said to be correct. Each policy is also evaluated for *consistency* and *completeness*. A policy is consistent if there is no environment in which an access is both granted and denied. A policy is complete if there exists some occurrence of events where access to the key will be granted.

The ability to analyze the correctness of a policy is essential to any solution. Languages providing formal semantics (e.g., GACL) seek to ensure correctness by construction, while others (e.g., KRP) perform analysis on the policies themselves. While the former approach does not rely on informed users for a statement of correctness (and hence is less error prone), it does not allow correctness to be defined in terms of environmental constraints.

In [CC97], Cholvy and Cuppens develop a language and formalism used to specify and analyze access control and authentication policies. A central contribution of this work is the identification of a formal definition of policy consistency. This definition states that if there exists any world (i.e. operational environment) that both allows and denies access to an object for some principal, then it is inconsistent. The authors present a mathematical framework in which the consistency of policies written in the proposed language can be evaluated.

The Cholvy and Cuppens language, denoted throughout as the CC language, is based on the Standard Deontic Logic (SDL)[2]. CC extends the traditional access control and authen-

[2]Deontic logic defines logical relationships among propositions that assert actions that are deemed oblig-

Rule	Rule Type	Specification
1	Nominative	$\forall f, \forall A, File(f) \wedge Public(P) \wedge Play(A, User) \rightarrow Read(A, f)$
2	Nominative (with obligation)	$\forall p, \forall A, Passwd(A, p) \wedge Old\_Passwd(A) \wedge Play(A, User) \rightarrow O\ Change\_Passwd(A)$
3	Descriptive	$\forall A, Play(A, User) \equiv \exists level, Login(A, level)$
4	Domain Constraint	$\forall f, forallA, Read(A, User) \rightarrow Access\_System(A)$

Figure 3.4: CC Policy - nominative, descriptive and domain constraint policy rules in the Cholvy and Cuppens specification language.

tication languages by introducing *obligation*. Rules defining obligation indicate operations actions that are required of principals. The language provides machinery to reason about principals who fail to meet obligations. For example, a user who fails to change its password can be barred from accessing its files. In this case, the obligation to change the password is stated in one rule, and the consequence of not changing the password in another.

To simplify, the CC language is based on the development of three types of rules. Similar to traditional access control statements, *normative rules* define the conditional mapping of *roles* to potential obligatory access rights. *Descriptive rules* define the set of conditions under which principals will assume a role. *Domain constraint* rules define the basic set of constraints assumed by the domain in which the rules are interpreted. Figure 3.4 presents four rules defined in the CC language.

The nominal rule (rule 1 in figure 3.4) indicates that any principal should be allowed read access to public files only if they have assumed the $User$ role. The predicate $Play(p, r)$ asserts states that the principal p has assumed the role r. The consequence action (right side of implication, $Read(A, f)$) is true if the conjunction of predicates of the left side of the equation evaluates to true. The universal quantifiers state that this rule should be applied to all principals who wish to gain access to public files. The nominal with obligation rule (2) states that principals assuming the $User$ role are required to change their password when it becomes "old". Similar to trust management approaches GAA API, the semantics and evaluation of predicates is outside the scope of the language.

A descriptive rule defines the conditions under which a principal assumes a role. The descriptive rule (3) states that any user who has logged into the system accepts the role of

atory, permissible, or not permissible.

User. Finally, the domain constraint rule (4) states that read access can only be allowed until the principal accesses the system. Similar to nominal rules, a definition of the semantics of "accessing the system" is outside the scope of the specification.

One language extension allows users to define a total (or partial) ordering of the roles. The ordering is used to resolve specification conflicts. For example, a principal assuming both the *User* and *Root* roles may not be allowed write access to a password file. (based on a *User* nominal rule), but be given the write privilege (based on a *Root* nominal rule). This conflict is resolved by roles ordering; the higher ordered roles will take precedence.

An observation made by the authors of KRP states,

> Policies must be specified in a language understandable by people to be useful, and in a language understandable by computers to be enforceable. [BB00]

This statement highlights a fundamental tension of language-based approaches. Natural languages are best suited for expressing policy. However, because they are often ambiguous, automated interpretation and enforcement of these languages is difficult. Semantically rich languages represent another extreme; while operators such as global and existential quantifiers provide mathematical rigor and expressiveness (e.g., in CC), they increase the complexity of specification.

The PolicyMaker [BFL96] and KeyNote [BFIK99b] systems provide a powerful and easy to use framework for the evaluation of credentials. Generally, support for provisioning and resolving multiple policies is not the focus of these systems. When desired, these systems can be invoked in Ismene conditionals to leverage their expressive power and extend their use to group communication systems.

Blaze et al. introduced a unified approach for the specification and evaluation of access control in the PolicyMaker system [BFL96]. At the core of any trust management system is a domain independent language used to specify the capabilities, policies, and relationships of participant entities. Applications implementing trust management consult an evaluation algorithm (engine) for access control decisions at run-time. The engine evaluates the access control request using pre-generated specifications and environmental data. Therefore, applications need not evaluate access control decisions directly, but defer analysis to the trust management engine. Through rigorous analysis, the PolicyMaker [BFL96] trust management engine has been proven to be correct. Thus, with respect to access control, any

application using PolicyMaker is guaranteed to evaluate each decision correctly. However, enforcement is left to the application. Several other systems (e.g., KeyNote [BFIK99b] and REFEREE [CFL+98]) have extended the trust management architecture to allow easier integration with user applications and a minimal set of enforcement facilities.

In [IKBS00], KeyNote has been used to define a distributed firewall application. The technique is to use conditional authentications, where conditions involve checking port numbers, protocols, etc. However, it still remains problematic to construct a configuration, based on multiple local policies, or for determining the correctness of a configuration. The provisioning clauses and legal usage assertions of Ismene can help address these problems.

Modification of access rights in traditional authentication and access control frameworks by resource owners has been historically difficult. Policies are frequently stated though singular specifications created, distributed, and evaluated by the authentication framework. Hence, any rights modification must be mediated by the service, and not by the resource owner. Trust management approaches the intermediate step by allowing resource owners to arbitrarily create and delegate rights. This is stark contrast to Antigone approach, where group rights are the result of the coordinated reconciliation of member and issuer requirements.

3.2 Group Communication

While group communication is often used in distributed computing, no single definition of a group has been universally accepted. In common usage, the definition of a group is typically derived from the application domain in which the group is used. Antigone defines a group as ...

> ... the set of entities that maintain some form of shared context. Within a networking environment (which is the only instance within the scope of this chapter), the participants may be hosts, processes, network devices, other groups, or any other addressable entity.

Groups can be *open* or *closed*. In an open group, any member or non-member may transmit a message to the group. In a closed group, only members of the group may transmit messages to the group. The vast majority of secure group communication systems provide a closed group.

Centralized groups identify one or more distinct *group controller* participants.[3] A group controller is the logical leader of the group the manages session initialization, access control, and other aspects of the group. Groups containing more than one group controller can distribute group management tasks to single controllers, delegate to a subordinate controllers, or share duties among peer controllers.

Peer groups (which do not contain any group controllers) manage the group through consensus. These groups typically share state via an ordered reliable broadcast mechanism [RBM96]. Decisions of access control, the introduction of new members, and key management within peer groups have been studied extensively. However, because of the need for consensus building, peer groups are particularly costly to implement.

This thesis focuses primarily on the closed, centralized groups implemented by the vast majority of group communication frameworks and applications. However, Antigone is in no way restricted to centralized or closed groups. The investigation of the mechanism and policy requirements of these groups is left to future work.

3.2.1 Secure Reliable Group Communication

Much of the early work in group communication was targeted to the development of protocols providing delivery guarantees (e.g., total, casual, FIFO, etc.). Later works investigated infrastructures supporting reliable and secure groups existing within diverse environments. The following considers the design, facilities, and models of several representative group communication systems, and considers the trade-offs between reliable and unreliable group communication.

Often cited as the genesis of current group communication technologies, the ISIS [Bir93] and later HORUS [RBM96] frameworks provide interfaces for the construction of group architectures. Using these frameworks, developers can experiment with a number of protocol features through the composition and configuration of protocol modules. One important feature introduced by the HORUS system was a comprehensive security architecture. A central contribution of this architecture was the identification of a highly fault-tolerant key distribution scheme. Process group semantics are used to facilitate secure communication. A single session key is used throughout each HORUS session. Hence, HORUS groups are

[3] No single term denoting the logical group leader has emerged in the literature. In different contexts, a group leader is known as a session leader [MPH99], a sequencer [KT91], or group agent [Mit97].

vulnerable to past or future members of the group.

Reiter further explored the highly robust security and group management in the RAMPART system [Rei94]. The RAMPART system provides secure group communication in the presence of actively malicious processes and Byzantine failures. Protocols in RAMPART rely heavily on distributed consensus algorithms to reach agreement on the course of group action. Secure channels between pairs of members are used to ensure message authenticity. Authenticity guarantees are used to ensure the accuracy of the group views[4] constructed through membership protocols. The security context is not changed through shared session keys, but through the secure distribution of group views.

A limitation of reliable group communication is cost; the consensus protocols used to provide strong delivery guarantees frequently require each member to actively participate in delivery through message acknowledgments. More recent work has investigated security within best effort delivery groups. For example, virtual private networks provide an abstraction in which applications designed for (logically) local network traffic can be executed across physically larger networks. The Enclaves system [Gon96] extends this model to secure group communication. Enclaves secures the group content from previous members by distributing a new group key after any member leaves the group. Emerging group systems increasingly accept this latter model; group applications are infrequently willing to pay the cost of group reliability in networks with low or highly variable throughput (e.g., the Internet). A number of other unreliable group systems are identified in Section 3.3.

3.2.2 Membership Management

Historically, the facilities used to manage and distribute information about the membership of the group is called Process Group Management [Cri91, RVR93, Bir93]. These services typically use reliable group communication facilities to ensure that the membership information is distributed in a consistent and timely manner. However, as with reliable group communication, the cost of providing this consistency is high. Hence, a number of approximation techniques have been used to mitigate these costs. Chapter 2 identifies a range of guarantees associated with these techniques.

[4] A group *view* is the set of identities associated with members of the group during a period where no changes in membership occur. When the membership changes (a member joins, leaves, fails, or is ejected), a new view is created. This is a similar concept to Birmans's group view [Bir93].

3.2.3 Failure Detection and Recovery

The failure of a group member can have an immediate effect on both the security of the group and the operation of the supported application [Rei94, AS98]. For example, the failure of a leader in a centralized group can lead to undetected events which would normally require the modification of the group session key, prevent access to the group, or delay the ejection of failed or compromised members. Similarly, some applications cannot progress without the presence of critical members (e.g. the sender in a video broadcast). Thus, depending on policy and application requirements, groups often require a mechanism that detects and recovers failed processes.

Failure detection is implemented by one or more *failure monitor* processes. Each failure monitor (which may or may not be a member of the group) actively monitors the state of a subset of the group membership. A member that has failed is denoted as *failed*, and denoted *live* otherwise. Depending on the crash model and available facilities, the failure monitor chooses the correct course of action to recover from the failure. Often, this simply requires the monitor notify the group of the failure.

A failure monitor may require periodic proof that monitored processes are operating correctly. This proof typically takes the form of a member-generated statement indicating its continued correct operation. These statements must be authenticated for a failure detection mechanism to be secure; some information proof that the stated member generated the message is necessary. If authenticating information is not included, an adversary can mask failures by generating counterfeit statements. In groups where the members are not completely trusted (or resiliency to member compromise is required), the authentication information must uniquely identify the sender (see source authentication 3.3.2).

Traditional detection of failed process requires the encryption and transmission of a periodically generated freshness indicator under a known key [FKTT98]. The freshness indicators may take the form of timestamps (where some secure source of time is globally available) or nonces. Known as *heartbeats* or *keep-alives*, the indicator messages are generated by any party wishing to state its continued presence in the group, and validated by failure monitors. In high throughput, dynamic, or large groups, the costs of processing heartbeat messages from each member can be prohibitive [MP00].

Many of the techniques used for the detection of group member failures were developed within the context of reliable communication. Typically, these systems detect failures by the

presence or absence of message acknowledgments. A central property of reliable multicast is *safety*; a message is either received by every non-failed member or by none. Thus, systems providing safety must receive an acknowledgment from each member before committing it. Because these acknowledgments implicitly state a non-failed state, they serve a dual purpose as message acknowledgments and heartbeats. For example, members of RAMPART groups are removed (detected as failed) from the current group if they are deemed unresponsive (i.e. fail to respond to group messages). More recent systems (i.e. TRANSIS [DM96], Ensemble [RBH+98], and CACTUS [HJSU00]) rely on similar reliability or process group management protocol mechanisms to detect failures.

Similar to the traditional heartbeats, group members in the Iolus system [Mit97] are required to periodically re-assert their presence in the group. However, rather than using these messages as periodic proof of a process's continued presence, they provide a means by which a group member can `REFRESH` its membership in the group. Each member is assigned a membership expiration time during the join process. The member may refresh its membership prior to the expiration time. Any member who does not refresh its membership before the expiration time is ejected from the group. While this approach has the advantage of being sender-driven, it requires a refresh acknowledgment mechanism. Without an acknowledgment, the refresh can be lost and the member incorrectly ejected from the group.

The way in which groups should recover from failures is largely dependent on the group threat model and session context. Several possible ways in which the group can react to the detection of a member failure include: a) the group disbands or suspends operation until the group member recovers and re-joins the group (in the case where the failed process is essential to the group mission), b) the group can purge the member and continue, or c) ignore the failure.

The reliable group communication Transis System [DM96] provides services supporting continued group operation in the presence of network partitions. To simplify, each partition establishes a new group when the failure is detected. The new groups corresponding to the partitions record all group messages. After the partition is repaired, the recorded messages are transmitted to the members of the other partitions.

3.3 Secure Group Communication

Although solutions for security in peer-to-peer communication are well understood, group security has been less forthcoming. This is due in large part to the complexity of establishing and maintaining a security context in groups with large and potentially dynamic membership. Beyond issues of complexity, performance and scaling requirements often limit the quality of solutions [SSV01].

The focus of the majority of the work in group security has centered on the investigation of *group key management*, and *data services*. A group key management service is used to establish and maintain session keys (e.g., key distribution). Data services provide security guarantees over group messages (e.g., confidentiality). Thorough treatments of the issues and design alternatives of secure group communication services are presented in [CP00, CGI+99]. The following subsections consider a number of seminal works used to provide security within these services.

3.3.1 Group Key Management

A central determinant to the quality of security provided to a group is the means by which session key are established and replaced. Session keys are typically used to provide other security guarantees (e.g., confidentiality). Hence, the ability to restrict access to session keys is critical to group security. To simplify, group key management can be defined as:

Given group G with membership M at time t

Goal Each $m^i \in M$ at time t can obtain session key SK

Constraint Each $m^k \notin M$ at time t is can not obtain SK.

However, the degree to which the constraint is enforced limits the cost of key management [SSV01]. Many key management approaches relax the constraint in different ways to achieve a more efficient solution. Each relaxation represents a set of threats accepted by the group.

Another area limiting the performance of a key management solution is the cost of initial group access. Any solution must provide a means by which members can initially make contact and receive key management data. However, in large or highly dynamic groups, centralized services can become overloaded with requests for key data. Thus, the means by which group exchanges are accomplished will have an affect on the scalability of the group.

Key-Encrypting-Key (KEK) management approaches remove all timing constraints. This implies that any m_i that is in (or ever will be) M has (will have) access to the session keys. Hence, the group is not protected from past and future members. For example, in the Group Key Management Protocol (GKMP) [HM97b, HM97a], newly joined members receive a KEK under which all future session keys are delivered. A limitation of this approach is that misbehaving members can only be ejected by the establishment of a new group. GKMP reduces the costs of authentication by introducing a peer-to-peer review process in which potential members are authenticated by active members of the group. Existing members assert the joining members's authenticity.

In an approach implementing a service similar to GKMP, the Scalable Multicast Key Distribution (SMKD) [Bal96] implements key management at the transport layer. Developed for the Core-based Multicast Routing Protocols [BFC93], SMKD uses the router infrastructure to distribute session keys. As the multicast tree is constructed, leaf routers obtain the ability to authenticate and deliver session keys to joining members. Thus, the overheads associated with member authentication and key management can be distributed among the leaf routers.

Often referred to as the simple group key distribution method, the Antigone 1.0 [MPH99] system implements a key distribution approach using pair-wise secure channels established during member authentication (e.g., a *pair-key* known only to the controller and the member). New session keys are distributed to each member independently through a key distribution block encrypted under the pair-key. This results in either n unicasts, or a broadcast of size n times the size of the key distribution block. Thus, because of its linear growth, this approach does not scale well to large groups. Similarly, the state held at the controller (pair-keys) grows linearly with group size.

Mittra categorizes the effect of the key management constraint as a *1-effects-n* failure, where a single membership change event can affect the entire group [Mit97]. Mittra's Iolus system addresses this limitation through locally maintained subgroups. Each subgroup establishes and replaces its own session key through simple group key management. A second meta-group key is established and maintained between subgroup controllers. Hence, the inherent cost of simple group key distribution is mitigated by localizing rekeying to subgroups. However, this approach introduces other latencies. Group messages must be translated as it crosses the subgroup to meta-group and meta-group to subgroup boundaries.

Subgrouping has been used to establish key management domains mapping onto autonomous systems. For example, Hardjono et al. propose a scalable framework for group key management in [HCD00]. The framework consists of two planes defining a group management service; a *network infrastructure plane* and a *key management plane*. The network infrastructure plane consists of the protocols and entities providing the broadcast (multicast) medium. The key management plane consists of the protocols and entities providing a group security context. Hardjono's framework is targeted for large groups spanning potentially many autonomous systems. Thus, the authors sought to support the heterogeneous protocols and infrastructures of each AS by introducing a region of meta-key management. This region, called a *trunk region*, maintains a region key under which participating key managers secure communication. *Leaf regions* consisting of group members transmit data or security related requests to a border key manager. The framework has the same advantages (i.e., localized key management) and disadvantages (i.e., message translation) as Iolus.

Logical Key Hierarchies [WHA98, WGL98] (LKH) provide an efficient alternative to subgrouping in achieving scalable, secure key distribution. A key hierarchy is a singly rooted n-ary tree of cryptographic keys. The session leader assigns all interior node keys. Each leaf node key is a secret key shared between the session leader and a single member. Once the group has been established, each member knows all the keys between their leaf node key and the root. As changes in membership occur, rekeying is performed by replacing only those keys known (required) by the leaving (joining) member. Rekeying without membership changes can be achieved by inexpensively replacing the root key. Thus, the total cost of rekeying in key hierarchies scales logarithmically with group size. Many incremental variants of LKH have been proposed [Per97, MS98, CGI+99, CEK+99, WCS+99]. These improvements seek to reduce the costs by reducing message overhead or state held at the group controllers and members.

Similar in construction to LKH, one-way function trees [MS98] (OFT) implement a structure used to establish and replace session keys. OFT key trees are established by assigning keys to the leaf nodes representing members. All interior node keys are generated by combining hashes of the two keys below in the hierarchy. OFT based key management behaves in all other respects as LKH.

The VersaKey system [WCS+99] extends the LKH algorithm by converting the key tree into table of keys. Each group controller maintains a $2xl$ table of Key Encrypting

Keys (KEK), where $l = log_2(n)$ for a group of n members. The table contains entries for each 0 and 1 bit of user identifiers (e.g. the key for bit 3, value 0 is indexed $k_{3,0}$). Each member receives keys associated with its identifier and a traffic encrypting key (TEK) (e.g. a member with binary identifier 1011 would receive keys $k_{0,1}$, $k_{1,0}$, $k_{2,1}$, $k_{3,1}$). Thus, based on the identifier, each member maintains a unique combination of keys from the table. When a member is to be ejected, only those keys associated with the ejected member identifier and traffic encrypting key are replaced and distributed. In the case of a member join, each key associated with the identifier of the new member and the traffic encrypting key is modified by applying a one way hash function to its random data. Thus, in the case of joins, no key distribution to current members is necessary.

The VersaKey approach significantly reduces the amount of state held by group controllers ($O(2n)$ in LKH to $O(2log_2(n)$ in VersaKey). This allows key management to scale to very large groups. However, VersaKey is vulnerable to collusion of ejected members. For example, two colluding members with complimentary identifiers cannot be ejected without simultaneously replacing the entire table. The authors present an algorithm for simultaneously replacing arbitrary numbers of group members. A second variant supporting environments containing multiple group controllers is described.

Independent of group changes, the frequency with which the group is rekeyed has an effect both on the security and efficiency of the group. By enforcing a lower bound on rekeying frequency, the key management constraint can be relaxed by limiting the granularity with which time t is measured. Setia et al. challenge the need (and measure the effect) of arbitrarily small measurements of t in Kronos system [SKJH00]. The Kronos key management system promotes the use of limited lifetime rekeying for distributing keying material. Using analytical techniques, the authors of the Kronos system showed that the large or highly dynamic groups quickly become limited by the speed at which they can perform rekeying. Using this as justification, they introduce a limited lifetime rekeying system based on the subgrouping techniques found in Iolus. A distinct member of each subgroup is responsible for distributing keys to all members within the subgroup. However, unlike previous approaches, rekeying occurs only at policy defined intervals.

In [SKJH00], the performance of Kronos, LKH, and Iolus systems were compared via simulation. It was found that periodic rekeying provided the lowest latency for group content, and that the leave and join latencies were acceptable (a 1 second rekeying interval

was tested). For large groups, it was shown that periodic rekeying was the only approach where rekeying was not the bottleneck.

Egalitarian groups require key management to be participatory. In *decentralized keying* or *key-agreement* systems, a number (> 1) of group members contribute to the establishment of the session key. Proposed participatory key management solutions [SSDW98, CC89, FN93, BD96, BW98, AST00] are largely applications of the n-party Diffie-Hellman key exchange [DH76] under a given set of assumptions and constraints.

A secure lock [CC89] protocol introduced by Chiou and Chen is representative of key-agreement approaches. Secure locks allow the distribution of secret information to an arbitrary number of members without prior establishment of group keys. This approach is used to quickly (in 1 message) distribute a session key to the membership of a group.

Systems implementing secure locks assume each member of an universe of members U (where $u_i \in U$, for all $i \mid 1 \geq i \geq m$) have established a publicly known and pair-wise relatively prime number (N_i). Additionally, each u_i establishes an encryption key known to the sender (k_{u_i}). Each message M is encrypted under a symmetric one-time key ($\hat{k}$). A secure lock X for message M and group G (where $G \subseteq U$) is generated by solving the following system of equations using the Chinese Remainder Theorem [Sti95]:

$$\left.\begin{array}{r} X \equiv R_1 \bmod N_1 \\ \cdots \\ X \equiv R_i \bmod N_i \\ \cdots \\ X \equiv R_m \bmod N_m \end{array}\right\} \quad \textit{where} \quad \begin{array}{l} \text{for all } u_i \in G \\ R_i = \{\hat{k}\}_{k_{u_i}} \\ N_i = \text{publicly known value for } u_i \\ m \text{ is the size of } G \end{array}$$

Upon receiving the message u_i computes ($R_i = X \bmod N_i$) to obtain $\{\hat{k}\}_{k_{u_i}}$. Using its secret key, u_i obtains $\hat{k}$ and decrypts the message. In the case of key distribution, M will contain the group session key.

Secure locks are attractive because the size of each distribution message is constant. However, the costs associated with the generation of a secure lock grow linearly with the group size. The generation of each R_i requires an encryption of the one time key. Additionally, finding the X based on the system of equations requires n computationally expensive modular exponentiations.

3.3.2 Data Services

Data transforms are used to provide security over application or control data. A transform defines a manipulation of data meeting some set of security requirements. Requirements

are satisfied by the application of cryptographic algorithms over a set of data and keys. For example, consider the following simplified data transform,

$$D \rightarrow E(SK, D), E(SK, H(D))$$

where SK is a session key known only to the current group members, D is the data to be transmitted, E is an encryption function accepting key and data parameters, and H is a collision-resistant and non-invertible hash function. This transform guarantees group authenticity, integrity, and confidentiality. These properties inferred from the fact that SK is only known to the members of the group; only an entity in possession of SK can generate the transformed data. The implicit assumption made by this construction is that it is computationally infeasible to obtain the key or data from $E(SK, D)$ or invert H. While there has been considerable debate on validity of these and similar assumptions in the general case, investigation of the strength of cryptographic algorithms is outside the scope of this thesis.

Transforms are not only used for application content. Based on the threat model, each group service requires some set of guarantees be preserved over the control data. For example, key management services often require that key distribution data remain confidential. Transforms similar to those defined in this section are used to provide these guarantees. However, the algorithms and keys used to implement the transforms will be driven by service requirements and available resources.

The following considers the canonical data security guarantees required by groups. Note that there are other guarantees which may be necessary in some environments (e.g., non-repudiation, anonymity, etc.). For brevity, a description of approaches providing these guarantees are omitted.

Confidentiality, Integrity, and Group Authenticity

Confidentiality guarantees that no member outside the group may gain access to session content. Although typically implemented through encryption under a symmetric key, other techniques may be used to limit content exposure. For example, confidentiality may be achieved though the use of stenography [AP98], or through encryption of only critical portions of messages.

Integrity guarantees that any modification of a message is detectable by receivers. As

they are fundamentally insecure, one cannot trust underlying reliable communication (point to point TCP [J81], reliable group communication [FJL+97]) to guarantee integrity. Sequence numbers, checksums, and other components of these protocols can be trivially altered by adversaries to manipulate message content. The use of keyed message authentication codes (MAC) [Sch96, KBC97] is an inexpensive way to achieve message integrity.

Group authenticity guarantees that a received message was transmitted by some member of the group, and is typically a byproduct of other data security policies. In many cases, proof of the knowledge of the session key (as achieved through most integrity guarantees) is sufficient to establish group authenticity.

Source Authentication

Source authentication approaches uniquely identify the sender of a message. Efficiently providing sender authenticity (or source authentication) in groups is an area of active research. However, the inherent complexity and cost of providing source authentication has been found to be daunting. The following considers a number of constructions providing source authentication.

In an obvious approach, a group establishes secret keys between each pair of senders and receivers. The pair-wise secret keys are subsequently used to generate an unique keyed hash (i.e., HMAC [KBC97]) delivered to each receiver. The authenticity of the message is inferred from knowledge of the secret key. In the general case, the generation of the potentially n^2 pair-wise secret keys can be problematic. Moreover, if such keys were available, each message would require the generation and transmission of up to $n-1$ HMACs. Thus, the costs associated with a single transmission grow linearly with group size.

Several attempts have been made to apply symmetric message authentication codes (MACs) to the problem of source authentication. For example, Canetti et al. define a source authentication mechanism resilient to k collaborating members in [CGI+99]. In one proposed construction, ℓ keys are initially created (where $\ell = \mathcal{O}(w\ log(1/q))$, w is the number of collaborating members from which the group is to be resilient, and q is the desired probability that the collaborating members share all the keys known by some non-collaborating member[5], for that non-collaborating member (e.g., it is suggested should

[5]Trivially, a collaboration which knows all keys known to a non-collaborating member can generate a counterfeit message that will be accepted by the non-collaborating member. q embodies the likelihood that

be $q \cong 2^{-20}$). Each receiver u is given a subset of the keys $R_u \subset R, R = \{k_1, \ldots, k_\ell\}$. Each source s is given the set of *second generation keys* $S_s = \{f_{k_1}(s), \ldots, f_{k_\ell}(s)\}$, where f is some pseudo-random, non-invertible function. Each message M creates a specialized MAC with one output bit for all $\text{MAC}(f_{k_i}(s), M)$ such that $i | 1 \leq i \leq \ell$. The ℓ output bits are transmitted with the message. Each receiver computes $f_{k_i}(s)$ and $\text{MAC}(f_{k_i}(s), M)$ for each one k_i it knows. If all the bits verify correctly, then M is deemed authentic. Because of the way the keys are assigned, no collaboration of receivers of k or less can (with probabilistic certainty) generate counterfeit messages. As each sender knows a unique set of authentication keys, no amount of sender collaboration will provide more information than the set of individual sender keys. k resilient schemes provide probabilistic security. The cost of the construction is a direct result of the desired strength of its protection (i.e., the assignment of k).

Digital signatures [DH76, RSA78] have been used in many contexts to provide authenticity guarantees to large or indeterminate sets of receivers. Source authentication can be achieved in groups by simply signing each message. Any member with access to a sender's authenticated public key (e.g., via some certificate distribution service [HFPS99]) can accurately determine the source of a message. A limitation of signature based source authentication is cost; asymmetric algorithms can be 1000 times more expensive than symmetric algorithms [Sch96]. Thus, high throughput groups using a signature solution will be limited by the speed at which a sender (receivers) can generate (validate) signatures.

Off-line signatures [EGM96, GR97] mitigate the costs of signature generation and validation by removing public key operations from the critical path of message marshaling. For example, the off-line signature approach defined [GR97] defines a table of size $m + log_2(m)$ random values is constructed for each message using the following construction. For a message M (of size m in bits[6]), a random signing key (sk) and resulting public key (pk) are created:

$$sk = x_1, \ldots, x_{m+log_2(m)} \in 0, 1^k$$
$$pk = f(x_1), \ldots, f(x_{m+log_2(m)})$$

where k is the size of the output of some one way hash function f. The public key is signed

any such situation occurs. Thus, it is important to assign a statistically insignificant value for q.

[6]Typically, M is a hash of the data to be sent. Hence, m is the number of bits of output of f.

off-line using a normal asymmetric private key. A signing process begins by the signer appending a binary representation of the number of zero bits in M to M. The signature $s = s_1 \ldots s_{m+log_2(m)}$ consisting of $m + log_2(m)$ values is generated:

$$s_i = x_i \text{ if bit } i \text{ of message } m \text{ is 0, and}$$
$$s_i = f(x_i) \text{ if bit } i \text{ of message } m \text{ is 1.}$$

s, and M are transmitted to the receivers. pk can be distributed prior to transmission or in conjunction with the signature. Verification requires the receiver validate that the last $log_2(m)$ values encode the correct number of zeros. Because conversion of 1 to 0 in the first m bits would cause a bit to flip from a 0 to a 1 in the last $log_2(m)$ bits, M cannot be forged without inverting f.

Off-line signatures are limited by bandwidth consumption. A table and signature is required for each message (e.g. often exceeding 1kb of signature data). Thus, this approach is not likely to meet the needs of groups in limited bandwidth environments. Hybrid schemes [Roh99] reduce off-line signature costs by trading off computational resources or security for decreased bandwidth usage.

In another attempt to mitigate the costs of signature generation, Gennaro and Rohatgi introduced stream signatures in [GR97]. Rather than authenticating each packet individually, stream signatures amortize signing costs over blocks of messages. Two schemes are proposed in [GR97]; *off-line* and *on-line* stream signatures.

Off-line stream signatures require the transmission data be available prior to the session. Initially, the transmission data B is broken into k blocks $(B_1, \ldots, B_k)$. $k + 1$ blocks are generated and transmitted using the following construction:

$$\begin{aligned} B'_0 &= pk_s, \{f(B'_1, k)\}_{pk_s} \\ B'_i &= B_i, H(B'_{i+1}) \\ B'_k &= B_k, 0000 \end{aligned}$$

The first packet (B'_0) contains a signature generated using the private key of an asymmetric key pair pk_s of the sender. The signature is calculated over the hash of the first data packet and the total number of packets in the transmission. Each subsequent packet contains a hash of the following packet of the transmission. The single signature validates all data packets by chaining the hashes across the entire transmission. However, if any packet is lost, the chain is broken and no further packets can be authenticated.

The on-line stream signatures operate in essentially the same way, with the exception that it is not assumed all data packets are known prior to transmission. In this case, packets are buffered by the sender as data is generated. When enough packets are available, a stream signature is generated and the data is transmitted to the group. In this way, the sender trades off authentication costs with the latencies caused by buffering. Wong and Lam soften the reliability requirements of on-line signatures using forward error correction techniques in [WL99].

Source authentication is frequently relevant in non-group environments. For example, Chueng presents the Optimistic Link State Verification (OLSV) protocol in [Che97]. The OLSV target environment consists of a number of routers exchanging link state information from which routing tables are generated. OLSV augments existing link state distribution protocols by authenticating updates. The authors note that previous approaches were insufficient for high speed networks. Establishing a single symmetric key shared by all the routers would not prevent a compromised router from creating bogus updates. For reasons cited above, the costs of an approach based solely on digital signatures or that establishes pair-wise secrets was deemed equally undesirable.

To simplify, OLSV uses a commit-release approach. Each update is signed[7] using a symmetric key generated from a hash chain. The last value in the hash chain itself is signed using the sender's private key and distributed to all interested routers. The values of the hash chain (keys) are subsequently exposed and used by receivers to validate MAC values. However, OLSV requires that an update be received prior to the release of the associated key. Failure to preserve this property could result in forged updates (i.e., an adversary could generate an arbitrary update using the released key). OLSV is optimistic; all updates are accepted as authentic until the key is released. If it is later found that an update was forged or altered, the other perform a recovery protocol removing the misbehaving routers or hosts.

Timed Message Authentication Codes [PSTC00] (TMAC) provide group source authentication in a manner similar to OLSV. Each packet contains a keyed hash computed under a key to be released at a later time. If a receiver can determine that a message was received before the associated key was released, the packet is deemed authentic. However, like OLSV, the security of the this scheme is directly related to security of the timing information

[7] More properly, a message authentication code (MAC) is calculated using the symmetric key.

held at both the sender and receiver. The authors propose a group timing synchronization algorithm used in support of TMACs in [PCB+00].

A limitation of Timed MAC is their inability to provide non-repudiation. Also defined in [PSTC00], the Efficient Multi-Chained Stream Signature (EMSS) addresses this limitation by defining a tiered stream signature. In one construction, each packet P_i in EMSS contains a hash of the packets P_{i-1}, P_{i-2}, P_{i-4}, P_{i-8}. Later, a signature packet (P_j, where $j > i$) signing the hash of packets P_{j-1}, P_{j-2}, P_{j-4}, P_{j-8} is sent. When P_j is received, the authenticity of all previous packets can be asserted by following the hash chains backward from the signed values. Because the packet was signed with a private key, non-repudiation is assured. An optimization of this scheme uses error correcting codes to reconstruct missing hash values. Simulations of EMSS found that environments with high loss rates (60%) can achieve robustness (i.e., authenticity can be assessed $> 95\%$ of the time).

3.4 Component Systems

Monolithic protocol stacks are often difficult to construct, debug, and maintain. Component-based systems limit the complexity associated with stack implementation and maintenance by decomposing services into software modules. Protocol stacks are synthesized at compile or run-time from ordered collections of software components implementing the desired functionality. Hence, each service can be implemented and tested in isolation. Moreover, the cost of developing new stacks is low; uniform interfaces allow components to be reorganized quickly. Mechanism composition has long been used as a building block for distributed systems [OOSS94, RBM96, Ber96, BHSC98, FKTT98, NK98]. However, the definition and synchronization of stack specifications (e.g., component organization graphs) in these systems is largely relegated to system administrators and developers.

An early seminal work used to rapidly construct protocol stacks from protocol components was the x-kernel [HP94]. Course grained *protocol objects* in x-kernel implement a single protocol (e.g., TCP [J81], UDP [Pos80], IP [Pos81]). Protocol objects are composed at compile time to implement a particular stack as directed by the developer. Each protocol in the x-kernel is represented by a directed acyclic graph defining the data flow of through the layers of the stack. A *session object* explicitly states the path through the protocol graph (e.g., A RPC call packet would traverse nodes associated with RPC, TCP, IP, and

physical layer objects).

The ADAPTIVE system [SFS93] extended the x-kernel philosophy to the construction of protocols from a specification language. The Protocol Machine Specification Language (PMSL) used by ADAPTIVE acts as a form of policy; protocols are constructed for the particular Quality of Service, performance, and reliability requirements of the target system. PMSL protocol specifications are translated in a task graph defining the implementation. To simplify, an ADAPTIVE protocol machine *instantiation* is constructed at runtime from intermediate PMSL specifications.

Isis [Bir93] and Horus [RBM96] applied the course grained protocol object approach to reliable group communication. These works define protocol components implementing group services (e.g., ordering guarantees, membership management), rather than the general-purpose transport protocols. Different protocol variants (with potentially different delivery semantics and guarantees) are composed to construct an efficient group communication service.

The use of component frameworks as a building block for security was considered by Orman et al. in [OOSS94]. The authors assert that basic cryptographic functions and protocols can be interposed into general-purpose protocol graphs to achieve end to end security. The authors further state that general-purpose security services (e.g., authentication) can be constructed in a similar manner. The authors demonstrate the viability of this approach by implementing client and server interfaces for Kerberos [NT94] within the x-kernel. As illustrated through their implementation of Kerberos, sessions providing specific security guarantees are implemented through the decomposition of a protocol's transport and security goals, and realized through an appropriate protocol graph. Where multiple protocol variants are required (as is the case where support for different security policies is required), (de)multiplexing components are added to the graph.

A limitation of the original x-kernel and similar systems system was the granularity of its protocol objects. Each protocol object often implements a general-purpose protocol, which may be complex and inefficient. For example, TCP is a complex protocol, with many options and boundary cases. The Coyote [BHSC98] system investigated the use of fined-grained micro-protocol components in the construction of end to end protocols [HS98]. Micro-protocols implement properties, rather than services, as separate modules. For example, while HORUS may implement group FIFO delivery, Coyote may implement separate

micro-protocols for reliable delivery and ordering. Protocols in Coyote are configured from collections of micro-protocols implementing the protocol semantic required by an application. Similar to Antigone, communication between Coyote micro-protocols modules is implemented through generic event interfaces (see Chapter 5).

Component composition must be restricted to only those compositions preserving semantic correctness and interface compatibility. A number of works investigating techniques for *configuration programming* have been studied (e.g. Polylith [Pur94], DARWIN [MKM94]). Hiltunen investigates a specification language for configuration programming in [Hil98], which he subsequently applied to the construction of data security services in Cactus [HJSU00]. Hiltunen's specification language describes the legal system constructions through the conflict, dependency, independence, and component relations, where conflicting (dependent) components must (must not) be used in conjunction, and independent components may or may not be used in conjunction. These relations are used at the build time to ensure correctly operating implementations.

The ADAPTIVE system builds executable Protocol Machine Instantiation from a two phase process; a PMSL specification is complied into intermediate Protocol Machine Task Graph Language by the Protocol Machine Configurator. This process translates protocol requirements into a graph representing orderings relationships between the protocol tasks (e.g., CRC, sequencing). An executable instantiation is created by the Protocol Machine Synthesizer by mapping the various tasks to protocol components. Legal usage relations between components are enforced by knowledge internal to the configurator and synthesizer. This has the disadvantage of requiring additional knowledge be programmed into these tools when new components are introduced into the system.

3.5 Broadcast Communication

A prerequisite to efficient group communication is a broadcast transport media. Historically, the predominate transport mechanism used for group communication has been IP multicast [Dee89]. IP multicast provides a connectionless point-to-multipoint communication service. Membership of a multicast group is not bounded by location or restricted in any way. A host may be connected to any number of groups simultaneously, and need not explicitly join prior to sending data. Messages sent by member applications are deliv-

ered as best effort traffic to all members that have explicitly indicated their interest in the group (e.g., via IGMP [Dee89]). The difficulties inherent to multicast routing in wide-area networks have spawned a significant body of research [WPD88, Moy94, Bal97b, Bal97a, EFH+98, Moy98, DEF+99]. As evidenced by the multitude of multicast routing protocols, the issues surrounding multicast routing are difficult to solve with singular solutions.

Due to a number of technical limitations [AAC+99], no single broadcast transport service has been universally adopted. *Overlay Networks* [Fra99, BCG+00, CRZ00, JGJ+00, CMB00] address the limitations of IP multicast routing protocols by providing a broadcast transport at the application layer. These networks implement a broadcast channel through host-level self-organization and routing. Member hosts or dedicated servers act as multicast routers by constructing a virtual network over point-to-point communication. The topology of the overlay network is modified as the group membership or characteristics of the physical network change. Thus, the group attempts to converge on the lowest cost (as defined be any number of metrics) broadcast tree. Overlay networks have the advantage of allowing the group membership to control the broadcast media. Hence, issues of scaling, inter-domain communication, and addressing are all within the purvey of the participating hosts.

CHAPTER 4

POLICY REPRESENTATION AND ANALYSIS

A consideration of any policy management infrastructure is the means by which policy is represented. Previous policy management systems have sought to identify a structure or schema allowing the specification of all germane aspects of a session context appropriate for the target environments. The Ismene Policy Language extends this work by considering a language under which policies for a large number of application domains, system models, and environments can be defined.

To motivate the goals of Ismene, the following presents simplified security requirements for an example group teleconferencing application, `tc`. The `tc` application is to be deployed within a company *widget.com*. *widget.com*'s organizational policy for `tc` requires the following:

- the confidentiality of all session content must be protected by encryption using DES or AES (provisioning requirement)
- participation in a session is restricted to *widget.com* employees (access control requirement)

Now suppose *Alice* wishes to sponsor a session of application `tc` that meets her following local policy:

- Alice wishes to use only AES cryptographic algorithm only (provisioning requirement); and
- she wishes to restrict the session to the $BlueWidgets$ team (access control requirement)

A basic requirement on the policy language is that it must be able to specify provisioning and access control policies for each group member and resolve them into a specific session policy instantiation. In the above example, the result of such resolution is that Alice's

session is restricted to members who are in both *BlueWidgets* and *widget.com* (access control requirement), and the cryptographic modules must be configured so that all content is encrypted using *AES* (provisioning requirement).

In general, security requirements can be more complex. For example, Alice may wish to restrict access to certain hours of the day, require that the session be rekeyed when new members join or leave, etc. Furthermore, other members in the group may have their own local security policies. A member must be able to check whether the session's policy satisfies the member's local policy. If the local policy is not satisfied, the member can choose to abstain from the group rather than compromise its security policy.

Policy must also be responsive; changes in membership or the execution of a security-relevant action can affect the session configuration. Conversely, the session must be able to make access control decisions based on the use and configuration of security mechanisms.

Ismene, thus, has the following primary goals:

- *flexible provisioning* - Ismene must allow the provisioning of groups based on an assessment of environmental conditions and the local policies of the members.

- *action-dependent provisioning* - Ismene must allow the modification of session configuration based on changes in membership or the execution of a security-relevant action.

- *authentication and access control* - The authentication and access control embodied by a session must be explicitly, but flexibly, stated in Ismene. Authentication and access control should not only be based on operating conditions, but also on the current session configuration.

- *policy compliance* - Any member participating in a group must be able to assess compliance of a configuration with its local security policy.

- *legal usage analysis* - Ismene must be able to determine whether a configuration represents the proper usage of the underlying security mechanisms.

Contemporary policy languages for group communication focus on the development of security schema meeting the needs of existing group applications (e.g., [HM97a, HC01, DBH+00, HCD00]). While entirely appropriate for the target environments, the introduction of new security requirement necessitates modification of the not only the language, but

also of all frameworks supporting the group [BBD+99]. This is in direct contrast with the goals of this thesis; a flexible policy management infrastructure must address unforeseen requirements (and integrate new infrastructure) without modification to the architecture or supporting policy language [HCH+00].

Authentication and access control has been studied extensively for 25 years [BL73]. Recent approaches have accepted that access control, and security in general, is contextual [WL93, CC97, WL98, RN00]. As such, recently proposed access control languages map not only users and objects, but also environmental context to access rights. However, these languages do not regulate how access is provided. Trust Management approaches extend context to encompass service provisioning [BFL96, BFIK99b]. Trust management access policies typically state the configurations under which access is granted. Access is stateless in these languages, which limits the ability to construct and enforce a coordinated access control policy throughout the lifetime of a session.

The languages cited above largely assume policies are completely specified prior to session initialization by a single authority. Hence, these languages do not adapt to participant requirements at run-time. Existing languages supporting reconciliation (e.g., DCCM [DBH+00], MSME [PCKS01]) are limited in scope, or construct policies appropriate for peer-to-peer communication. These limitations led to the design of Ismene, which builds significantly upon the design and philosophy of these languages.

The remainder of this chapter considers the Ismene policy model, the construction of the Ismene Policy Language, and the design of the algorithms supporting its use. Hence, this chapter addresses two of the primary objectives of this thesis identified in Chapter 2; *Flexible Representation* and *Multiparty Determination*. The following section begins by further motivating the goals of the Ismene policy language.

4.1 System Model

An Ismene policy specifies the central components of a group communication security context: *provisioning* and *authentication and access control*. Both aspects are specified in group and local polices. A *group policy* defines the entirety of security-relevant properties, parameters, and facilities used to support a group session. Each group participant states the set of local requirements on group sessions through a *local policy*. The group and member local

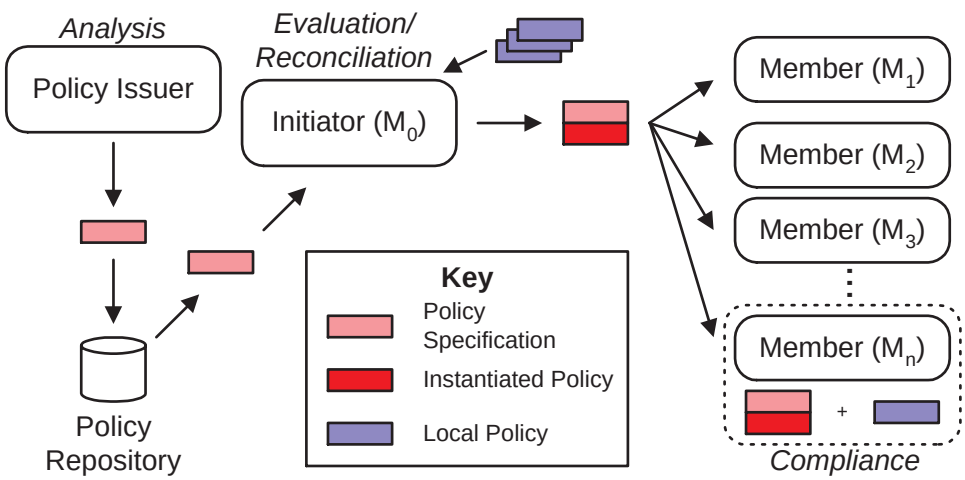


Figure 4.1: System Model - A session is a collection of *participants* collaborating towards some set of shared goals. A policy *issuer* states a group policy as a set of requirements appropriate for future sessions. The group and expected participant local policies are evaluated to arrive at a *policy instantiation* stating a concrete set of requirements and configurations. Prior to joining the group, each participant checks compliance of the instantiation with its *local policy*.

policies are *reconciled* within an environment prior to the establishment of each session.

A group provisioning policy identifies the security requirements of the group. These policies requirements are mapped into a configuration of security-related services or mechanisms. Ismene group and local policies are reconciled to arrive at a specific configuration for a session. Potential participants of a session verify that the session's configuration is *compliant* with their local policy before participating. A group policy is tested against a set of legal usage assertions through *analysis* to ensure that any configuration resulting from reconciliation will not introduce undesirable side effects.

Authentication and access control defines how sessions regulate action within the group. The authentication and access control implemented by a group is explicitly stated in its configuration. Ismene is limited to expressions of positive criteria under which access is allowed, but permits the integration of other authentication frameworks where more expressive power is required.

4.2 Approach

Depicted in Figure 4.1, a group is modeled as the collection of *participants* collaborating towards a set of shared goals. The existence of a *policy issuer* with the authority to state

session requirements is assumed. The issuer states the conditional requirements of future sessions through the *group policy*. Each member states the set of local requirements on future sessions through a *local policy*. Each participant trusts the issuer to create a group policy consistent with session objectives. However, a participant can verify the compliance of a policy *instantiation* with their *local policy*.

An *initiator* is an entity that generates an *policy instantiation* from group and local policies. The initiator may or may not be a participant of the group. An instantiation concretely defines the provisioning and authentication and access control rules to be enforced. The instantiation is the result of the *evaluation* and *reconciliation* of the group and local policies within the run-time environment. Provisioning identifies the relevant session requirements, and defines how requirements are mapped into a configuration. The initiator is trusted (by the group members) to evaluate and reconcile the group and local policies correctly.

Ismene policies are collections of totally ordered provisioning and action clauses. *Provisioning clauses* identify configuration. Participant software is modeled as collections of security *mechanisms*. Associated with a mechanism is a set of *configuration* parameters used to direct its operation. Each mechanism provides a distinct communication service that is configured to address session requirements. A provisioning clause explicitly states configuration through a set of mechanisms and parameters. Authentication and access control rules are defined in *action clauses*. An action clause defines the conditions under which a protected action should be allowed.

Each provisioning and action clause is defined as the tuple:

```
<tag> : <conditionals> :: <consequences>;
```

Tags are used to provide structure to the policy. Intuitively, tags represent session requirements or identify a protected action. The organization of tags dictates the relationships between clauses, and ultimately guides policy reconciliation. *Conditionals* contain zero or more predicates describing the conditions under which the consequences are to be enforced. Predicates are Boolean functions used to test the operating environment, session configuration, local or global state, or the presence of credentials. The result of the evaluation of a predicate is *true* where the represented condition holds and *false* otherwise. *Consequences* identify provisioning and authentication. Each consequence states a session requirement, a configuration, or the acceptance of an access request. The complete Ismene grammar is presented in Figure 4.2.

```
<policy> := <statements>
<statements> := <statement> ";" ["," <statements>]
<statement> := <attribute> | <prov_clause> | <action_clause> | <assertion>

<attribute> := <identifier> ":=" "<" <value> ">"  |
               <identifier> ":=" "<" <value list> ">"
<value list> := "{" <value> "}" ["," <value list>]
<identifier> := word
<value> := string

<prov_clause> := <tag> ":" [<conditionals>] "::" <consequences>
<tag> := <identifier>
<conditionals> := <var> "=" <value> ["," <conditionals>] |
                  <predicate> "(" [<args>] ")" ["," <conditionals>]
<var> := "$" <identifier>
<predicate> := <identifier>
<args> := <var> | <identifier>  ["," <args>]
<consequences> := <pick> | <config> |  <tag> ["," <consequences>]
<pick> := "pick" "(" <config> "," <config> ["," <configs>] ")"
<configs> := <config> ["," <configs>]
<config> := "config" "(" [ <cfgstmts> ] ")"
<cfgstmts> := <mechanism> [ "(" <params> ")" ] [, <cfgstmts>]
<mechanism> := <identifier>
<params> := <identifier> "=" <value> ["," <params> ]

<action_clause> := <tag> ":" [<action_conditionals>]
                           "::" "accept" [, "reconfig"]
<action_conditionals> := <action_condition> ["," <action_conditionals>]
<action_condition> := <conditionals> | <credential>  | <config> | <pick>
<credential> :=  Credential "(" <bind var> "," <cred_args> ")"
<bind var> := "&" <identifier>
<cred_args> := <identifier> "=" <value> ["," <cred_args>] |
               <identifier> "=" <var> ["," <cred_args>] |
               <identifier> "=" <bind var> ["," <cred_args>]

<assertion> := "assert" ":" [<assert_conds>] "::" <assert_args>;
<assert_args> := [!] <pick> | [!] <config> ["," <assert_conds>]
```

Figure 4.2: The Ismene Policy Language Grammar. A *word* represents a string of non-whitespace alphanumeric characters. A *string* is a string of alphanumeric characters (i.e., may contain newline and whitespace characters).

The semantic of a clause is as follows; if all the conditionals in the clause evaluate to **true**, then the consequences need to be enforced. For example, a clause containing conditionals $c_1 \ldots c_n$ and consequences $q_1 \ldots q_m$ represents the logical expression $(c_1 \wedge \ldots \wedge c_n) \Rightarrow (q_1 \wedge \ldots \wedge q_m)$. Thus, the result of an evaluation over a set of clauses is a conjunction of consequences. The collection of consequences defines precisely how the session's security parameters are allowed to be configured.

One or more clauses may be defined with a tag. Clauses associated with a single tag are evaluated in the order in which they are defined in the policy. Evaluation of a tag stops when a clause evaluates to true (i.e., the conditionals hold). In this case, the consequences are enforced and all other clauses associated with the tag are ignored. If a clause evaluates to false then the next clause associated with the tag is evaluated. If, for a particular tag, no clause evaluates to true, then the policy is rejected. A rejected policy indicates that the stated security requirements cannot be met in the current environment. All interested parties are notified of the failure, and the session is aborted.

Returning to the *widget.com* example described in the previous section, the network administrator responsible for setting application policy at *widget.com* acts as the policy issuer. The administrator generates the following group policy for the `tc` application:

```
provision : :: config(idhdlr()),
              pick( config(idhdlr(conf=des)), config(idhdlr(conf=aes)) );
join : Credential(&cert,iss=$CA,subj.O=widget.com,subj.CN=$joiner) :: accept;
```

The `idhdlr` defined in the `provision` clause is the *mechanism* providing security guarantees over application content. The `conf=des` and `conf=aes` are *configuration parameters* applied to the `idhdlr` mechanism stating how confidentiality is to be provided. The `pick` configuration is used to state flexible policy; either DES [Nat80] or AES [DR98, DR00] can be used to implement confidentiality, but not both or neither. The `join` action[1] clause states that only entities supplying a certificate credential with subject organization of *widget.com* and issued by a (known and trusted) CA should be admitted to the group.

Local policies are used by each participant to describe local requirements on future sessions. Alice defines her local policy as follows:

```
provision : :: config(idhdlr()), config(idhdlr(conf=aes));
join : Credential(&cert,iss=$CA,subj.O=BlueWidgets,subj.CN=$joiner) :: accept;
```

[1] The `join` action in this example is not a Ismene reserved word; the set of actions defined in each policy should those understood by the provisioned mechanisms.

Through this local policy, Alice states that any session must implement a confidentiality policy using the `idhdlr` mechanism implementing AES. The `join` clause states the session must enforce a policy that requires participants supply a certificate issued by a trusted CA with a subject organization of *BlueWidgets*.

Alice acts as the initiator in sponsoring the `tc` session. She acquires the local policies from the expected session participants. For this example, only the group policy and Alice's local policy are used. The instantiation resulting from evaluation and reconciliation is as follows:

```
provision : :: config(idhdlr()), config(idhdlr(conf=aes));
join : Credential(&cert,iss=$CA,subj.O=widget.com),
       Credential(&cert,iss=$CA,subj.O=BlueWidgets,subj.CN=$joiner) :: accept;
```

The reconciliation of the `tc` group and Alice's local policies attempts to find a configuration that is consistent with both policies. In this case, the configuration `config(idhdlr(conf=aes))` is selected. The use of two credential conditionals in the `join` clause represents a conjunction; credentials fulfilling the criteria for each condition must be supplied to gain access to the group[2].

The $CA descriptor in the above examples identifies an *attribute* defining the public key of a known and trusted certificate authority. An *attribute* describes a single or list-valued variable. For example, the following attributes define a single-valued version number and list-valued ACL:

```
version := < 1.0 >;
JoinACL := < {bob}, {john}, {george} >;
```

The occurrence of the symbol "$" in any clause signifies that the attribute should be replaced with its value. The *attribute set* is the set of all attributes. An application can add to the attribute set by passing name/value (list) pairs to the Ismene algorithms. Hence, the attribute set can be used to supply environmental context not representable by Boolean-valued conditionals. The attribute set serves the same function as the action environment in KeyNote [BFIK99a].

Prior to their use in any session, group policies are *analyzed* to determine if they represent legal configurations. Legal configurations are stated through *assertions*. Assertions represent invariant properties required by all instantiations. Logically, these statements

[2]Note that this does not imply a unique credential must be supplied for each `Credential` test. As well may be the case in the example above, a single credential may satisfy several credential tests.

identify illegal and mandatory configurations. Assertions are created by mechanism developers and security personnel, and provide a means by which provisioning correctness principles are guaranteed over all sessions. For example, the assertion:

```
assert: config(keymgt(mem=leavesens)) :: config(membership(leave=explicit));
```

states that a key management mechanism configured with leave sensitivity [MPH99] requires (is dependent on) the membership mechanism configured to provide explicit leaves. Thus, for this assertion to hold, any instantiation defining the leave sensitive key management must also define the membership mechanism with explicit leaves. Through *analysis*, Ismene guarantees that no instantiation resulting from the reconciliation of the group policy with any set of local policies will violate policy assertions.

Each potential participant acquires the policy instantiation prior to joining a group. The participant determines the consistency of the instantiation with its local policy through a *compliance* algorithm. A *late joiner* (i.e., a member whose local policy was not considered during the creation of the instantiation) is free to participate if the instantiation complies with their local policy. Participants in the `tc` session check the compliance of the instantiation received from Alice, and if successful, can participate in the session. Note that any participant whose local policy is used in the initial reconciliation phase is trivially compliant.

The following sections describe the format and use of the two types of clauses in Ismene; provisioning clauses and actions clauses.

4.3 Provisioning Clauses

Provisioning clauses are used to develop a policy instantiation from conditional statements. Each provisioning clause identifies zero or more *conditionals* used to define when the consequences are applied to the instantiation. *Configuration*, *tag*, and *pick* consequences define how the instantiation is derived and defined.

Environment conditionals test the session environment. Each environment conditional is defined as a (possibly parameterized) predicate assessing a measurable aspect of the environment. However, the evaluation of environmental conditionals is outside the scope of the Ismene language. The environment in which Ismene is used is required to provide an interface for the evaluation of predicates. This approach separates the definition of relevant conditionals from the process of policy evaluation. Similar to other authentication and ac-

cess control [RN00], Ismene defers condition evaluation to domain specific implementations (upcalls).

Configuration consequences define how the requirements of a session are realized through configuration. Each such consequence identifies either a mechanism or mechanism configuration to be added to the policy instantiation. For example, consider the following clauses defining secrecy and integrity policies:

```
confidentiality : privateGroup() :: config(idhdlr()), config(idhdlr(guar=conf));
integrity : RmteScope($mcaddr) :: config(idhdlr()), config(idhdlr(intg=rfc2104,hash=md5));
```

The `confidentiality` clause states confidentiality should be provided when a group is private (as indicated by the `privateGroup` predicate, but does not state how this is achieved (i.e., assumes the specification of a cryptographic algorithm occurs elsewhere). The second clause indicates that integrity is enforced through MD5-based HMACs [Riv92a, KBC97] if the session multicast address is remotely scoped (e.g., multicast traffic will extend beyond the local network). The application or infrastructure using Ismene is required to provide interfaces for evaluating the `privateGroup` and `RmteScope` conditionals.

Pick consequences afford the initiator flexibility in developing the session. Semantically, the pick statement indicates that exactly one configuration must be selected. Information in the local policies guide reconciliation towards the most desirable configuration (see Section 4.5.2).

Tag consequences describe the organization of the group policy. The structure defined by the organization of tags defines the dependencies between sub-policies. Each tag consequence requires the evaluation of other clauses, which may lead to the introduction of further configurations and tags.

Consider the following policies appropriate for public and private sessions in a conferencing application:

```
provision : private($addr,$pt) :: config(idhdlr()), config(idhdlr(guar=conf)),
                                  strong_key_mgmt, confidentiality;
provision :  :: config(idhdlr()), config(idhdlr(guar=conf)),
                weak_key_mgmt, confidentiality;
strong_key_mgmt : :: config(lkh_rekeying()), secrecy;
secrecy : ManagerPresent($group) :: config(lkh_rekeying(sens=mem));
secrecy : :: config(lkh_rekeying(sens=leave));
weak_key_mgmt : Audio(), Video() :: config(kekkey()), config(kekkey(rekeyperiod=240));
weak_key_mgmt : Video() :: config(kekkey()), config(kekkey(rekeyperiod=120));
weak_key_mgmt : :: config(kekkey()), config(kekkey(rekeyperiod=60));
confidentiality : sensitive($subject) :: pick( config(idhdlr(encr=3des)),
                                                config(idhdlr(encr=desx)) );
confidentiality :  :: config(idhdlr(encr=des));
```

This policy states that private groups should be configured with strong key management and confidentiality, and non-private groups with weak key management and confidentiality. Initially, the conditionals associated with the first provision clause are evaluated. If the group is private (as determined by the address), then the `idhdlr` mechanism is configured to enforce confidentiality and the `strong_key_mgmt` and `confidentiality` tags are evaluated. If this fails, the evaluation falls to the next clause defining a policy for non-private groups by applying the `idhdlr` configuration and evaluating the `weak_key_mgmt` and `confidentiality` tags.

The evaluation of the strong key management requirement illustrates how a session provisioning can be responsive to membership. The (unconditional) strong key management clause states that a LKH (key management) mechanism should be used. However, in applying the *secrecy* tag consequence, the instantiation will arrive at a backward or membership rekeying policy, depending on a manager being expected to participate. Thus, groups with managers are afforded greater protection from non-members through a membership-sensitive policy that rekeys following any membership change.

The `weak_key_mgmt` clauses describe how the quality of service provided by key management can be determined by the types of data being transmitted (e.g, audio and video groups rekey least frequently, followed by video only, followed by other groups). Thus, through similar policies, configuration can be a reflection of the available resources or the demands made on surrounding infrastructure.

Pick consequences are useful in environments where the issuer wishes to set standards for operation, but does not wish to mandate an implementation. The first confidentiality clause states that, for groups with sensitive subjects, the `idhdlr` mechanism can (only) be configured to use either the 3DES [Nat80] or DESX [KR96] algorithms to implement a strong confidentiality policy. If the subject is not sensitive, then the group will implement confidentiality by DES encryption.

To illustrate this process, the following configuration is the result of evaluation of a public, but sensitive, Audio/Video session;

```
config(idhdlr()), config(idhdlr(guar=conf)),
config(kekkey()), config(kekkey(rekeyperiod=240))
pick( config(idhdlr(encr=3des)), config(idhdlr(encr=desx)) );
```

Local policies are evaluated exactly as the group policy. Described in Section 4.5.2, the reconciliation algorithm resolves each pick statement in the evaluated group policy based

on evaluated local policies. Hence, the result of provisioning reconciliation is a completely defined session configuration (consisting of only mechanism and mechanism configuration consequences).

4.4 Action Clauses

The action clauses defined in an instantiation identify the authentication and access control policy enforced by the group. Action clauses can contain configuration, credential, and environmental conditionals (i.e., tag consequences are not allowed) and are restricted to *accept* and *reconfig* consequences. An accept consequence indicates that an action should be allowed. The *reconfig* consequence represents the need for a re-evaluation of the group and member provisioning policies.

Ismene represents a *closed world* in which denial is assumed. An action is allowed only if the evaluation of an associated action clause leads to an *accept* consequence. The tag of an action clause identifies the action to be considered (e.g., `join`, `send`). The set of protected actions are defined by the issuer, and assumed known *a priori* by the security mechanisms (see Chapter 5. Ismene is consulted for acceptance when any protected action is undertaken.

Configuration conditionals test the presence of configurations in an instantiation. A configuration conditional returns *TRUE* when the configuration is defined in the instantiation. The semantics of a pick conditional is the **or** of the configurations; the conditional returns *TRUE* if any one of the configurations described in the pick are contained in the instantiation.

Credential conditionals test the characteristics of authentication information associated with a protected action. A credential is modeled in Ismene as a set of attributes. For example, an X.509 certificate [HFPS99] can be modeled as attributes for subj.O (subject organization), issuer.CN (issuer canonical name), etc. To illustrate, consider the following action clause:

```
join : Credential(&cert,sgner=$ca,subj.CN=$joiner) : accept;
```

The first argument of a credential conditional (denoted with "&" symbol) represents binding. The credential test binds the matching credentials (see below) to the (&cert) attribute. This binding is scoped to the evaluation of a single clause. Conditionals are evaluated left to right.

The second and subsequent parameters of a credential conditional define a matching of credential attributes with attribute or constant values. The above example binds the credentials that were issued by a trusted CA (`sgner=$ca`) and have the subject name of the joining entity (`subj.CN=$joiner`) to the `&cert` attribute. The conditional returns true if a matching credential can be found. The assertion of valid and appropriate credentials is outside the scope of Ismene. Hence, it is up to the enforcement architecture and application to supply the set of validated credentials associated with an action.

Consider the following set of action clauses:

```
join : config(idhdlr(encr=des)), In($JoinACL,$joiner),
       Credential(&cert,sgner=$ca,subj.CN=$joiner) : accept;
join : Credential(&cert,sgner=$ca,delegatejoin=true),
       Credential(&tocert,sgner=&cert.pk,subj.CN=$joiner) :: accept;
eject : sensitive($subject),
        Credential(&cert,sgner=$ca,role=X,subj.CN=$ejector) :: accept;
send : Credential(&key,key=$sessky), pick( config(idhdlr()),
                                           config(gdhdnlr()) ) :: accept;
```

The first `join` describes an ACL-based policy for admitting members to the group. The member is admitted to the group if she is identified in the `JoinACL` list attribute, she can provide an appropriate credential, and the session is encrypting traffic using DES.

Action clauses are evaluated like provisioning clauses. The second `join` is consulted only when the conditionals of first clause do not evaluate to *TRUE*. The second `join` clause describes a delegation policy. The first credential conditional binds `&cert` to the set of credentials delegating join acceptance (in this case, the set of certificates from the CA delegating join acceptance). The second conditional tests the presence of any credential signed with a delegated public key. Ismene is restricted to explicit delegation chains; each link in the chain must be explicitly stated as a credential conditional.

The `eject` clause describes basic role-based authentication and access control. This clause states that the `eject` action will be allowed only if a credential stating the requester's right to assume the role `X` can be found. If the credential is found, the requester, acting in role `X`, is allowed to eject another member. A similar clause can be defined for each action role `X` is authorized to perform.

Credentials can be used to test knowledge of session specific keys. For example, the `send` action clause describes the conditions under which an application message should be accepted. The clause states that the right to send a message in sessions configured with the `idhdlr` or `gdhdnlr` mechanism is predicated only on proof of the knowledge of the current session key (matching `$sesskey`).

4.4.1 Reprovisioning the Group

The `reconfig` consequence provides a means by which the group may advise the environment of a need to re-evaluate the session configuration. This reconfig consequence indicates to the group that some action representing a fundamental change in the group is about to occur. The following example illustrates one such change.

Consider the following action clauses that define a group policy requiring re-provisioning before members belonging to the `SpecialUsers` group are admitted:

```
prejoin : In($SpecialUsers, $joiner),
          Credential(&cert,sgner=$ca,subj.CN=$joiner) :: accept, reconfig;
join : In($SpecialUsers,$joiner), config(idhdlr(encr=des)),
       Credential(&cert,sgner=$ca,subj.CN=$joiner) :: accept;
provision : SpecialUsersPresent() :: config(idhdlr(encr=aes));
provision : :: config(idhdlr(encr=des));
```

To support re-provisioning, joining the group becomes a two phase process. Initially the member will perform a *prejoin*, after which the environment will be requested to re-provision the session. The `provision` tags specify that AES encryption should be configured if `SpecialUsers` are present and DES otherwise. In this policy, the reconfig consequence signals to the group that security requirements need to be re-assessed when a SpecialUser member joins.

`reconfig` only causes notification; actual re-provisioning is outside the scope of Ismene. Re-provisioning may or may not be successful. However, the non-trivial task of transitioning a group to a new policy has yet to be fully investigated. The prejoin step was introduced to handle the possibility that re-provisioning could fail or take substantial time. Only after the session is successfully re-provisioned to use AES, a member belonging to the `SpecialUsers` is admitted.

4.4.2 Integrating Ismene with External Authentication Frameworks

Ismene provides a simple model and language for stating group authentication and access control. However, it is often the where more expressive power is required. Moreover, many environments will require enterprise-internal authentication services be used across applications. A central design goal of Ismene is to allow the delegation of authentication and access control decisions to external services.

Ismene desires to take advantage of widely deployed or more expressive authentication frameworks (e.g., Kerbeos [NT94], PolicyMaker [BFL96], KeyNote [BFIK99b], GAA API [RN00], Akenti [TJM+99]). In this way, Ismene need not replace existing approaches,

but augment them. The following action clause describes how KeyNote can be used within an Ismene policy:

```
join: KeyNote($joiner,$attrset,$grppol,$creds) :: accept;
```

This clause states that a member should be admitted to the group only if KeyNote can generate a proof of compliance stating authentication to join the group. The conditional states that requestor ($joiner), action description ($attrset), policy to be enforced ($grppol), and credentials ($creds) be passed to KeyNote. This is precisely the set of information used to evaluate a KeyNote policy.

4.5 Policy Processing

The following subsections describe the use of the Ismene policy algorithms to flexibly enforce group security policy. The construction and complexity of these algorithms are considered in detail in Section 4.5.4.

4.5.1 Evaluation

The Provisioning Evaluation algorithm (PEVL) is used to determine the provisioning that is appropriate for the run-time environment. The evaluation algorithm recursively assesses clauses defined over the set of tags, conditionals, and consequences. The special *provision* tag is the start symbol for reconciliation of provisioning. Obviously, group and local policies are required to have at least one clause defined with the *provision* tag. The evaluation of provisioning clauses is described in detail in Section 4.3.

The group and local policies are evaluated prior to reconciliation to arrive at an evaluated policy. The evaluated policy contains a set of mechanisms, mechanism configurations, and pick statements. The following restrictions are placed on the Ismene policies. These restrictions allow efficient policy reconciliation and compliance checking.

> **Restriction 1** - A mechanism can only be stated in at most one pick statement in an evaluated policy.
>
> **Restriction 2** - A mechanism configuration can only be stated in at most one pick statement in an evaluated policy.

Note that if the result of evaluation is unconstrained (i.e., does not conform to these restrictions), reconciliation of even a single policy becomes intractable[3]. The tangible result of these restrictions is that the pick statements identifying a particular mechanism or configuration are mutually exclusive (in terms of evaluation). These restrictions allow the reconciliation of pick statements to be independent of others within the same policy.

The Authentication and Access Control Evaluation algorithm (AEVL) determines acceptance or denial of action request within the run-time environment by assessing the action clauses defined in the instantiation. Hence, unlike provisioning clauses, the action clauses are evaluated each time an action is undertaken. The clauses defined in an instantiation are the result of the reconciliation process (see next section).

The AEVL algorithm evaluates clauses in the same manner as provisioning policy evaluation. However, because accept and reconfig are the only legal consequences, only singular clauses associated with the particular action (e.g., join) are consulted. If all conditionals of a clause associated with the action evaluate to true, then action is accepted, else it is rejected. The evaluation of action clauses and credential conditionals is described in Section 4.4.

4.5.2 Reconciliation

The evaluated group policy is reconciled with the evaluated local policies of the expected participants to arrive at a concrete configuration. Thus, reconciliation determines the requirements that are relevant to a session, and ultimately how the session is implemented. Reconciliation assumes that all policies have been previously evaluated. For this reason, the "evaluated" policy qualifier is omitted from the following discussion.

Ismene group policies are authoritative; all configurations and pick statements used to define the instantiation must be explicitly stated in the group policy. Local policies are consulted only where flexibility is expressly granted by the issuer through pick statements. Hence, the group policy acts as a template for the session. Local policies are used to further refine the template towards a concrete instantiation.

The local policy of an expected participant guides the resolution of *pick* statements to the most desirable configuration. To simplify, if a configuration in the pick is in the evaluated local policy, it is selected. If the local policy provides no such guidance, the pick

[3] A thorough treatment of unconstrained reconciliation is presented in Section 4.6.2

is left unresolved and the other local policies are consulted.

Conflicts may arise when consulting multiple local policies. For example, consider a group policy pick statement defining configurations for A and B. A conflict occurs when some local policies require A and others require B. The resolution of the pick statement determines who can participate in the session.

One potential resolution algorithm, Largest Subset Reconciliation (LSR), attempts to find a configuration compliant with the largest number of local policies. However, this approach has the undesirable property that it may fail to allow the participation of required members (for example, by excluding the video source in a video conference). Moreover, as shown in Section 4.6.2, LSR is intractable.

A second algorithm, Prioritized Policy Reconciliation (PPR), establishes an ordering of local policies. Higher prioritized policies representing more important members are considered first; lower priority policies are considered only when higher priority policies provide no guidance. The restrictions defined in the previous section allow this algorithm to be efficient. The current implementation of Ismene uses this solution. The following group and local policies illustrate prioritized reconciliation (where the local policy l_1 is ordered before l_2):

group policy	`config(A), pick(config(B), config(C)), pick(config(D), config(E))`
local policy l_1	`config(A), config(B)`
local policy l_2	`config(B), config(D)`

Initially, the group policy and l_1 are reconciled first. The instantiation is initially defined with the configuration A (A is a mandatory configuration in the group policy). Next the algorithm attempts to reconcile the pick for (B, C). $config(B)$ would be selected from the first pick statement because l_1 requires it. l_1 provides no guidance for the second pick statement. l_1 is completely reconciled at this point, and other policies are considered. l_2 is consulted and D selected from (D,E), after which l_2 and the group policy are reconciled. Thus, the instantiation contains A, B, and D. Note that the introduction of other local policies or requirements may lead to an irreconcilable local policy. For example, if l_1 also required `config(E)`, the algorithm would arrive at the instantiation A, B,and E. In this case, the requirement for D in l_2 cannot be satisfied, and the member associated l_2 would not to participate in the group.

The authentication and access control policy enforced throughout the session is the result of the reconciliation of action clauses stated in the group and local policies. The

reconciliation algorithm is designated as the Authentication and Access Control Policy Reconciliation Algorithm (AACR).

The set of clauses determining authentication and access control in an instantiation is defined by the intersection of the group and local policies. For example, consider group policy that defines the action clauses (t_i : c_1 :: *accept*;) and (t_i: c_2 :: accept;). Further, a local policy defines an action clause as (t_i: c_3 :: accept;), and another local policy defines the action clause (t_i: c_4 :: accept;). Authentication reconciliation algorithm takes the logical **and** of these policies; the action clause is for t_1 is logically defined as

$$t_1 : ((c_1 \vee c_2) \wedge c_3 \wedge c_4) :: accept$$

Authentication reconciliation constructs an action clauses for each action t_i defined in the group and local policies. `reconfig` consequences listed in any of the policies are added to the action clause in the instantiation. For convenience, an action clause is added for each conjunction of the disjunctive normal form (DNF) of the conditional expression. In this example, two clauses would be introduced for t_1 with the conditionals ($c_1 \wedge c_3 \wedge c_4$) and ($c_2 \wedge c_3 \wedge c_4$).

4.5.3 Compliance

Not all participant local policies are required to (or can) be consulted during reconciliation. Hence, a participant must be able to check the compliance of an instantiation with its local policy prior to participating in a session. Compliance is successful if all requirements stated in the local policy are satisfied by the instantiation. There are two phases of compliance; provisioning and authentication.

The Provisioning Compliance (PC) algorithm compares an evaluated local policy with a received policy instantiation. Each configuration and pick statement must be satisfied by the instantiation. A configuration is satisfied if it is explicitly stated in the instantiation. A pick statement is satisfied if exactly one configuration from the list is contained in the instantiation. Thus, provisioning compliance is as simple as testing the satisfaction of the evaluated local policy by the instantiation.

Participants may wish to place requirements on the kinds of authentication and access control enforced by the group. Ismene defines authentication and access control compliance as;

> The authentication and access control policy stated in the instantiation must be no more permissive than the local policy.

More precisely, compliance determines if, for any action and set of conditions, an action accepted by policy instantiation would also be accepted by the local policy. This emobodies a conservative approach to compliance, where any action that would be denied by the local policy **must** be denied by the instantiation. Hence, compliant instantiations always respect the limitations stated in the local policy. Gong and Qian consider a more restrictive definition of authentication and access control compliance between a composition of two policies in [GQ94]. Gong and Qian's definition of *secure interoperability* requires that both the principle of security as well as autonomy be preserved in policy composition. The principle of security is identical to the above compliance definition; the composition must be no more permissive than either policy. The principle of autonomy requires that any action accepted by one policy must be accepted by the composition (is no less permissive).

The Authentication and Access Control Compliance algorithm (AAC) assesses whether the instantiation logically implies the local policy. Given an expression e_1 describing the conditionals of action clauses in an instantiation, and a similar expression describing a local policy e_2, it is intellectually easy to check compliance between the policies by testing whether the expression $e_1 \Rightarrow e_2$ is a tautology. To illustrate, consider the action clauses defined in the following instantiation and local policies:

policy instantiation	$X : (c_1 \wedge c_2) \vee c_3 :: accept;$
local policy A	$X : c_1 :: accept;$ $X : c_3 :: accept;$
local policy B	$X : c_1, c_3 :: accept;$

The policy instantiation is compliant with the local policy A because the policy is less permissive (e.g., $(c_1 \wedge c_2) \vee c_3 \Rightarrow c_1 \vee c_3$). The group policy is not compliant with local policy B because the group policy is more permissive (e.g., $(c_1 \wedge c_2) \vee c_3 \not\Rightarrow c_1 \wedge c_3$). General purpose tautology testing is intractable [Coo71]. However, the lack of negative conditionals in Ismene allows efficient compliance testing. These factors are explored in depth in Section 4.6.3.

4.5.4 Analysis

It is important to restrict instantiations to legal configurations. Thus, Ismene must be able to describe the acceptable usage and configuration of the security mechanisms. *Assertion clauses* are used to describe the legal and required relations between mechanisms and mechanism configurations. Semantically, an assertion indicates that a configuration must or must never be true in any instantiation. Assertions are independent clauses (i.e, the clause does not contain tag consequences). Positive (negative) assertions must (not) be satisfied by any policy instantiation.

A number of systems have investigated techniques guaranteeing correct and efficient construction of software from components [Hil98, LKvR+99]. These approaches typically describe relations defining compatibility and dependence between components. A configuration is deemed correct if it does not violate these relations. For example, Hiltunen [Hil98] defines the conflict, dependency, containment, and independence relations. The following describes assertions representing these relations (where independence is assumed):

conflict (A *is incompatible with* B)	`assert : :: ! config(A()),config(B());`
dependency (A *depends on* B)	`assert : config(A()) :: config(B());`
containment (A *provides* B)	`assert : config(A()) :: ! config(B());`

An analysis algorithm attempts to assess whether a policy can or an instantiation does violate the assertions. The Online Policy Analysis algorithm (ONPA) assesses an instantiation with respect to a set of assertions. This algorithm simply tests the configurations of the instantiation against the relations described in the assertions. If no violation is found, the instantiation can be used.

The Offline Policy Analysis algorithm (OFPA) algorithm attempts to determine if **any** instantiation resulting from a group policy violates a set of assertions. In the worst case, this requires the generation of all possible instantiations (there may an exponential number of them). Offline policy analysis is performed by the issuer, and thus does not affect the efficiency of session initialization or operation. Moreover, most reasonable configurations exhibit a degree of *independence*; the introduction of a configuration is largely the result of the reconciliation of a few clauses. Hence, the evaluation of an assertion can be reduced to the analysis of only those clauses upon which the configurations stated in the assertions are dependent. An algorithm for OFPA is presented in Section 4.6.4, but optimization of the algorithm is left to future work.

Assertions can be used by the issuer as sanity checks on future instantiations. For

example, the issuer may wish to assert a completeness property [JSS97, BB00] that any instantiation resulting from reconciliation enforces confidentiality over the application data. Thus, knowing in advance that the *Ismene*, *generic*, and *xor* data handler mechanisms configured with confidentiality are the only available means by which this property can be provided, the issuer states the following completeness assertion:

```
assert :  :: pick( config(idhdlr(guar=conf)), config(gendhlr(guar=conf)),
                   config(xordhlr(guar=conf)) );
```

The analysis algorithm rejects any policy failing to preserve this requirement.

4.6 Algorithm Analysis

This section considers the construction and complexity of the algorithms used to process Ismene policies. The central result of this analysis is that all algorithms used over the lifetime of a session are tractable. Hence, with respect to policy, Ismene instantiations can be efficiently generated, distributed, and enforced. Table 4.1 summarizes the results of this analysis. The remaining sections defines the Ismeme policy processes and algorithms. Table 4.2 describes the notation used throughout the remainder of this chapter.

The analysis presented in the following section identifies complexity bounds on policy evaluation. A key question to be answered is whether the use of non polynomial bounded algorithms makes policy management infeasible. Clearly, for trivial policies, such algorithms will be reasonable (as is true trivial input to a large number of NP algorithms). However, real world policies can contain hundreds of clauses and configurations (see Chapter 6). Hence, such input strongly suggests that the use of NP algorithms is not feasible.

4.6.1 Evaluation

The evaluation algorithm is used to determine which configurations (or actions) are appropriate for (accepted by) the run-time environment. The result of the provisioning evaluation (called an evaluated policy) is a conjunction of configuration and pick statements. This conjunction is later used as input to the provisioning reconciliation algorithm. The result of authentication and access control evaluation is the acceptance or denial of an action. The clauses used to direct the authentication evaluation process are defined by the authentication and access control reconciliation algorithm. See Section 4.6.2 for further details of

Type	Algorithm	Complexity
Evaluation	Provisioning Evaluation (PEVL)	P*
	Authentication/Access Control Evaluation (AEVL)	linear*
Reconciliation	Generalized Policy Reconciliation (GPR)	NP
	Prioritized Policy Reconciliation (PPR)	P*
	Largest Subset Reconciliation (LSR)	NP
	Authentication/Access Control Reconciliation (AACR)	P*
Compliance	Provisioning Compliance (PC)	P*
	Authentication and Access Control Compliance (AAC)	P*
Analysis	Online Policy Analysis (ONPA)	P*
	Offline Policy Analysis (OFPA)	coNP*

Table 4.1: Policy Algorithm Complexity - asymptotic time complexity of the algorithms used for policy processing. All algorithms denoted with (*) are used within the current implementation.

reconciliation. The following describes efficient algorithms for provisioning and authentication and access control evaluation.

Algorithm 4.1 (Provisioning Evaluation (PEVL)) **Given**: *An unevaluated policy P.* **Find**: *A set of configuration and pick statements $R \subseteq E^p$ defined by the assessment of conditional statements in P.*

Provisioning evaluation recursively assesses policy clauses defined over the set of tags, conditionals, and consequences. Evaluation begins by testing the conditionals associated with the first *provision* clause in the group policy. If all conditionals evaluate to true (or no conditionals are defined for that clause), the consequences are applied to the instantiation. If not, then the next clause associated with the tag is evaluated. If no clause evaluates to true, then the policy cannot be successfully evaluated and the policy is rejected.

Applied configuration and pick statement consequences are added to the evaluated policy result. Tag consequences indicate the need for further evaluation. All applied tag consequences are added to the ordered set of tags that must be evaluated. Clauses associated with these tags are recursively evaluated as described above. The evaluation algorithm terminates when the set of tags to be evaluated is empty. The following states the algorithm more formally.

Note: in the description below, Q is an ordered set tags $\subset T$, and $Q.first$, is the first element of Q. All conditionals $c_i^{d_j}$ are modeled as Boolean valued variables (where the

Symbol	Description
g	a group policy, where $\|g\|$ is the number of clauses $\in g$
L	the set of local policies considered during reconciliation, where $\|L\|$ is the number of local policies $\in L$
l_i	a single local policy $\in L$, where $\|l_i\|$ is the number of clauses $\in l_i$
C_p	the set of all clauses defined in a policy p. C is used where it is unambiguous. $\|C\|$ is the number of clauses $\in C$
c_i	a provisioning clause $\in C$
c_i^t	the tag of c_i
c_i^d	a set of conditions $\in c_i$, where $c_i^{d_j}$ is a single conditional $\in c_i^d$
F	the set of all possible configurations (mechanism and configuration parameters)
c_i^q	a set of consequences $\in c_i$, where $c_i^{q_j}$ is a single consequence $\in c_i^q$
T^p	the set of all tags defined by a policy p. T is used where it is unambiguous
E^p	the set of all mechanism, configuration parameter, and pick statements defined by a policy p. E is used where it is unambiguous
e^p	the set of all mechanism, configuration parameter, and pick statements defined by an evaluated policy p. e is used where it is unambiguous
e_i^p	a single mechanism, configuration parameter, or pick statement defined by an evaluated policy p. e_i is used where it is unambiguous
$e_{i,j}$	a single configuration parameter for mechanism i, parameter j
I	a policy instantiation
i_j	a single pick statements defined by the Instantiation I, ($\|i_j\| = 1$)
A^p	the set of all action clauses in policy p
$A^p_{t_i}$	the set of all action clauses in policy p for action t_i
$A^p_{t_i,j}$	a single action clauses in policy p for action t_i indexed by j
S	a set of assertions
s_i	a single assertion $s_i \in S$
s_i^d	the conditions of assertion $s_i \in S$
s_i^q	the consequences of assertion $s_i \in S$

Table 4.2: Notation - notation used throughout the algorithm analysis presented in Section 4.6.

Boolean value is the value returned by its run-time result).

$Q =$ "**provision**"
repeat
    select $c_i | c_i^t = Q.first, minimum(i), c_i \in P$
    if no such c_i *exists* **reject**
    if $\forall c_i^{d_j} \in c_i^d, c_i^{d_j} = TRUE$ {
        foreach $c_i^{q_l} \in c_i^q$ {
            if $c_i^{q_l} \in T$
                append $c_i^{q_l}$ *to* Q
            else
                $R = R + c_i^{q_l}$
            }
        $Q = Q - Q.first$
    }
    $G = G - c_i$
until $|Q| = 0$

Each policy represents a graph whose nodes are clauses and edges are tag consequences. The removal of clauses during evaluation ensures that the evaluation graph is acyclic (by removing possible cycles). Logically, this prevents recursively defined requirements. This ensures that no reconciliation of policy can lead to the introduction of a previously evaluated clause. Note that no clause or condition is visited more than once. Hence, assuming conditionals are evaluated in $O(1)$, evaluation is P-time computable in the number of clauses (or actually linear in the number of clauses)[4].

Algorithm 4.2 (Authentication and Access Control Evaluation (AEVL))
Given: *A policy instantiation* I. **Find**: **Accept** *if the evaluation of an action clause in* I *is satisfied by the run-time environment (as represented by conditional statement), and* **deny** *otherwise.*

Statements of authentication and access control are represented by action clauses. Each clause defines a conjunction of conditionals which are evaluated to determine acceptance of action. The conditionals of each clause associated with the request action are tested. If for any clause, all conditions returns `TRUE`, then action is allowed. Hence, assuming conditionals are evaluated in $O(1)$, AEVL is trivially linear time in the number of action clauses in the instantiation.

[4]Evaluation is actually bounded linearly with the number of clauses. This section is primarily concerned with classification of algorithms as P or NP. Hence, the stated bound on algorithm is P.

Note that the evaluation of *Credential* conditions is also linear time; a given set of credentials is scanned for the appropriate attributes. If such a credential is found, the condition returns `TRUE`, and `FALSE` otherwise.

4.6.2 Reconciliation

The process of reconciliation attempts to find a policy instantiation that is consistent with the group policy and all local policies. Unrestricted reconciliation of (even) a single Ismene policy is intractable in the most general case. The following illustrates this point by reducing 3SAT [Coo71] to Generalized Policy Reconciliation (GPR). Note that an alternate, albeit significantly less illustrative, reduction from ONE-IN-THREE SAT [Sch78] to GPR also exists.

Reduction 4.1 (Generalized Policy Reconciliation (GPR))

The following definitions state specifics of 3SAT and GPR.

Definition 4.1 (Generalized Policy Reconciliation (GPR))

Given: A group policy g. **Question**: Is there a selection of configurations from picks statements that satisfies g?

Definition 4.2 (3SAT)

Given: The set U of variables and a collection C of clauses over U such that each $c \in C$ has $|c| = 3$. **Question**: Is there a truth assignment for U that satisfies C?

The following construction reduces 3SAT to GPR in polynomial time. Create g by defining a pick statement for for each $x \in U$ representing a variable or its negation (where $\bar{i}$ represents the negation);

$$pick(config(x_i), config(\bar{x}_i))$$

and for $c_i \in C, c_i = (x_1 \vee x_2 \vee x_3)$, create the following pick statement

$$pick(M_{123}, M_{12\bar{3}}, M_{1\bar{2}3}, M_{1\bar{2}\bar{3}}, M_{\bar{1}23}, M_{\bar{1}2\bar{3}}, M_{\bar{1}\bar{2}3}),$$

where $M_{123} = config(x_1, x_2, x_3)$.

A non-deterministic algorithm can simply guess the satisfying assignment of configuration statements for g, and verify its correctness in P-time. Thus, GRP is in NP. Assume a polynomial time algorithm exists for GPR. The configuration returned by GPR g is a

satisfying truth assignment for U, where each x_i (or $\bar{x}_i$) represents the truth assignment for x_i. The reduction is completed by returning *TRUE* where such a configuration exists, and *FALSE* otherwise. Thus, as 3SAT is a known NP complete problem, GPR is NP complete. □

This result indicates the reconciliation of the most general form is intractable. The language restrictions defined in Section 4.5.1 were introduced to address precisely this problem. These restrictions do not allow the specification of the pick statements described in the previous reduction. Hence, the restrictions lead to the following efficient reconciliation algorithm.

Algorithm 4.3 (Prioritized Policy Reconciliation (PPR)) **Given**: *An evaluated group policy g and an evaluated local policies L.* **Find**: *A set of configurations $g_{n+1} \subseteq E_g$ (where $n = |L|$), $\hat{L} \subseteq L$ such that g_{n+1} satisfies both g and all $l_i \in \hat{L}$.*

The PPR algorithm attempts to reconcile each local policy with the group policy according to priority. The highest priority local policies are reconciled first. If reconciliation with the local policy is successful, the group policy is reduced modulo the reconciliation. If not, the local policy is marked as irreconcilable. This process is repeated until an attempt to reconcile all local policies has been completed. The following assumes that the local policies have been placed in L in priority order (i.e., l_1 is the highest priority, followed by l_2, etc.). For ease of exposition, configuration statements are modeled below as single valued pick statements.

Step 1: Set $g_1 = g$.

Step 2: Reconcile local policy. for $i = 1 \ldots |L|$. If g_i is completely reconciled (all pick statements contain a single configuration), then add l_i to $\hat{L}$ only if the provisioning defined by g_i is compliant (as determined by the PC algorithm, Section 4.6.3). i.e.,

$$g_{i+1} = g_i$$
$$\text{if } PC(g_i, l_i) = \textbf{compliant} \text{ then } \hat{L} = \hat{L} + l_i$$

If g_i is not completely reconciled do steps 3 through 7.

Step 3: Extract irrelevant statements from the group policy. Remove all pick statements (containing solely) configurations $\in g_i$, but not $\in l_i$, and place them

in g_{i+1}. For example,

$$\begin{array}{ll} g_i = e(x_1, x_2), e(x_3), e(x_4), e(x_5, x_6) & \\ l_i = e(x_2, x_7), e(x_1, x_3) & \rightarrow \end{array} \begin{array}{l} g_i = e(x_1, x_2), e(x_3) \\ g_{i+1} = e(x_4), e(x_5, x_6) \end{array}$$

Step 4: Collapse equivalent configuration sets. A set of equivalent configurations is a set of two or more configurations contained within the same pick statement in both g_i and l_i. For example, if $e(x_1, x_2, x_3) \in g_i$, and $e(x_1, x_3, x_5, x_6)$ in l_i, then x_1, x_3 is an equivalent configuration set. By restriction 1, any sets of equivalent configurations A and B, $A \cap B = \emptyset$. Thus, equivalent sets of configurations can be treated as single values (because any configuration x_i in a satisfying selection for g_i and l_i can replaced with any other equivalent configuration x_j any retain satisfiability). For this reason, common sets are replaced with a new single (meta) configuration.

Step 5: Reduce both g_i and l_i by the single valued pick statements; i.e., for each $|e_j^{g_i}| = 1$ and $|e_j^{l_i}| = 1$ add the (single) configuration to g_{i+1} and reduce the pick statements containing the single value. *All* values of the reduced pick statement must be removed from both policies (because the selection of a configuration requires the others in the pick statement not be selected). For example, consider the following expressions

$$\begin{array}{c} g_i = e(x_1, x_2), e(x_3) \\ l_i = e(x_2, x_3), e(x_1, x_4) \end{array}$$

then $e(x_3)$ is added to g_{i+1} and reduce g_i and l_i modulo x_3 and x_2. Hence, the result is

$$\begin{array}{c} g_i = e(x_1) \\ l_i = e(x_1, x_4) \\ g_i = x_3 \end{array}$$

which can lead to further reduction. g_i and l_i would reduce to (x_3, x_1).

If, as in this example, this leads to a completely reconciled instantiation (all pick statements are single valued), the algorithm adds l_i to $\hat{L}$ and returns to step 2 (for the next l_i). If any pick statement in g_i or l_i reduces to zero values, the local policy is marked as irreconcilable and the algorithm aborts further reconciliation. In this case l_i is marked as irreconcilable, and g_{i+1} is set to the original g_i (as defined prior to step 3), and the algorithm returns to step 2.

Step 6: Remove all configurations not shared by both policies; delete any configuration in g_i but not l_i or in l_i but not g_i. No such configuration can be selected

(because it would not be consistent with the other policy). Thus, it is safe to remove all such configurations. For example, given the following policies

$$g_i = e(x_1, x_2), e(x_3, x_5)$$
$$l_i = e(x_1, x_4), e(x_3, x_6)$$

x_4, x_5 and x_6 are not shared, so they are removed from each expression resulting in;

$$g_i = e(x_1, x_2), e(x_3)$$
$$l_i = e(x_1), e(x_3)$$

If, as it is in this example, it is the case that any expression is reduced to a single configuration, the algorithm returns to step 5. If any pick statement reduces to zero values, the local policy is marked as irreconcilable and the algorithm aborts further reconciliation (of l_i) as defined in step 5. If neither of these cases occur, then the algorithm continues to step 7.

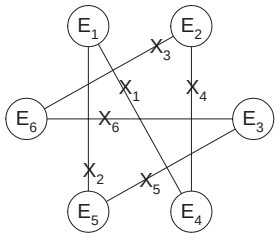
Step 7: Selection of satisfying configurations. At this point, every configuration in the g_i and l_i occurs *exactly* twice (once in g_i and once in l_i). This follows from restriction 1 and step 5. Construct an undirected graph $R = (V, E)$ as follows:

1. (V) Create a vertex for each $e_j^{g_i} \in e^{g_i}$ and $e_j^{l_i} \in e^{l_i}$ (below as labeled e_i).
2. (E) For each configuration $x_m \in g_i$ or l_i, create an edge $\{e_i, e_j\}$, where e_i and e_j are the two expressions in which x_m occurs.

For example, the graph resulting from this construction over the e^{g_i} and e^{l_i} defined below would be;

$$e^{g_i} = e_1(x_1, x_2), e_2(x_3, x_4), e_3(x_5, x_6)$$
$$e^{l_i} = e_4(x_1, x_4), e_5(x_2, x_5), e_6(x_3, x_6)$$

$\rightarrow$



Note that the edge cover EC (or perfect matching [GHR95]), if it exists, is a satisfying assignment of configurations for g_i and l_i. By definition, any edge cover will have at least one end-point of a selected edge on each vertex. Thus, each expression e_i has at least one selected x_i. This follows from the definition of edge cover where the number of vertices is even ($|g_i| = |l_i| \Rightarrow |g_i + l_i| \bmod 2 = 0$).

Step	Description	$O()$
1	This is a copy of g, linear time.	$\|g\|$
2	This set is executed $\|L\|$ times. If, at some i the PC algorithm is triggered the cost is $\|l_i\| * \|g_i\|$, which in all cases is less than or equal to cost of executing steps 3-7. Thus, for the purposes of complexity analysis, we can assume this case never occurs.	$\|L\|$.
3	The extraction of irrelevant statements requires the scanning of l_i $\|g_i\|$ times.	$\|l_i\| * \|g_i\|$
4	Detection and equivalent requires, scanning of l_i, $\|g_i\|$ times. Replacement is linear time ($\|g_i\| + \|l_i\|$).	$\|l_i\| * \|g_i\|$
5	Reduction of single values require linear time for detection ($\|l_i\|$ or $\|g_i\|$), and linear time for reduction ($\|l_i\| + \|g_i\|$). However, each reduction may lead to other singular values. This may occur at most $O(\|l_i\| + \|g_i\|)$ times (one for each clause).	$\|g\| * (\|l_i\| + \|g\|)$
6	Detection and equivalent requires scanning of l_i $\|g_i\|$ times (once per variable).	$e\|l_i\| * \|g_i\|$
7	Graph construction is linear in the number of clauses and variables ($\|l_i\| + \|g_i\| + \|E^g\|$). EC is polynomial ($n^2$) in the number of edges (number of configurations in g, E^g)	$\|E^g\|^2$
8	Executed in linear time by scanning $g_{\|L\|+1}$	$\|g_{\|L\|+1}\|$

Table 4.3: Complexity of PPR - time complexity of each step in the Prioritized Policy Reconciliation algorithm.

> Because no more than one edge $x_i \in EC$ is incident to a vertex, no more than one x_i is selected for any e. Thus, EC is a satisfying solution for g_i and l_i. Add all x_i's in EC to the configurations in g_{i+1}. Now replace each selected equivalent configuration sets with pick statements containing their original definition.

Step 8: Arbitrarily reconcile the pick statements in $g_{|L|+1}$ by selecting their first value. The result is the completely reconciled policy g_{n+1} and reconciled local policies $\hat{L}$.

The complexity of each step of the PPR algorithm is defined in table 4.3. Using the data presented in the table, PPR executes in[5]

$$O(|g| + |L| * (4 * (|l_i| * |g|) + |g|^2 + |E^g|^2) + |g_{|L|+1}|) = O(|g|^2 + |E^g|^2)$$

Hence, PPR is polynomial in the number of clauses and configurations in g. □

An attractive reconciliation solution is to identify the largest subset of local policies that can be satisfied by the group policy. However, finding the largest subset is intractable. The

[5] As $|g|$ forms an upper bound for $|g_i|$, all occurrences of g_i are replaced with g in the complexity expression.

following subsection illustrates the intractable property of such an algorithm by reduction from the known NP complete MAX2SAT problem [GJ79, GJS76][6].

Reduction 4.2 (Largest Subset Reconciliation (LSR))

The following definitions state specifics of LSR and MAX2SAT.

Definition 4.3 (Largest Subset Reconciliation (LSR))

Given: A group policy g and a set of local policies to be considered by reconciliation L. **Question**: What is the largest $\hat{L} \subseteq L$ such that g and all policies $l_i \in \hat{L}$ are satisfied?

Definition 4.4 (MAX2SAT)

Given: The set U variables, collection C clauses over U such that each $c \in C$ has $|c| = 2$, and a positive integer $K \leq |C|$. **Question**: Is there a truth assignment for U that simultaneously satisfies at least K of the clauses in C?

The following construction reduces MAX2SAT to LSR. Assume $U = \{x_1, x_2, \ldots, x_n\}$. Construction: For each $c_i \in C, c_i = (x_1 \vee x_2)$, create three local policies as follows:

$$l_{c_1 1} : pick(x_1), pick(\bar{x_2}), pick(x_3, \bar{x_3}), \ldots, pick(x_n, \bar{x_n})$$
$$l_{c_1 2} : pick(\bar{x_1}), pick(x_2), pick(x_3, \bar{x_3}), \ldots, pick(x_n, \bar{x_n})$$
$$l_{c_1 3} : pick(x_1), pick(x_2), pick(x_3, \bar{x_3}), \ldots, pick(x_n, \bar{x_n})$$

Note that each policy describes *mandatory* configurations (pick statements containing only one configuration). Negative variables are inverted. For example, the expression $c_1 = (a \vee \bar{b})$ generates the following local policies;

$$l_{c_1 1} : pick(a), pick(b), \ldots$$
$$l_{c_1 2} : pick(\bar{a}), pick(\bar{b}), \ldots$$
$$l_{c_1 3} : pick(a), pick(\bar{b}), \ldots$$

Create the group by creating a pick statement for each variable in U as follows;

$$g = \forall v_i \in U : pick(v_i, \bar{v_i})$$

For example, if $U = \{a, b, c\}$, then

$$g = pick(a, \bar{a}), pick(b, \bar{b}), pick(c, \bar{c}).$$

Note by this construction, g can only satisfy 0 or 1 of the clauses associated with each c_i. Each local policy represents the (mutually exclusive) ways in which each clause c_i can be satisfied, and the reconciliation of g with L is simply a truth assignment for U.

[6] It has been shown that there exists a polynomial time algorithm for this MAX2SAT where $K = |C|$ [EIS76].

Assume a polynomial time algorithm exists for determining the largest satisfiable subset of all considered local policies. Answering MAX2SAT simply becomes the process of evaluating the policies resulting from the construction. If $|\hat{L}| \geq K$, then MAX2SAT returns *true*, and *false* otherwise. Thus, because MAX2SAT is a known NP complete problem, LSR is NP complete. □

Reconciliation of authentication and access control is achieved by performing a logical *AND* of the group and local policies (see Section 4.5.2). This is achieved through the P-time AACR algorithm described below. The result of this algorithm is a set of action clauses used by AEVL to enforce authentication and access control. For reasons described in Section 4.6.3, it is convenient for the resulting action clauses to be stated in disjunctive normal form (DNF).

Algorithm 4.4 (Authentication and Access Control Reconciliation (AACR))
Given: *A group policy g and and set of local policies L.* **Find**: *A set of action clauses A^I representing the logical* **and** *of g and L.*

step 1: Repeat steps 2 through 5 for each $t_i \in T^g$.

step 2: Drop impossible actions. If $|A^{l_j}_{t_i}| = 0$ for some $l_j \in L$, return to step 1 (next t_i).

step 3: Construct initial action clause expression. Create an expression e_0 by logically ORing the conditionals of each $A^g_{t_i,k} \in A^g_{t_i}$. E.g.,

$$e = \bigvee_{k=1}^{|A^g_{t_i}|} A^g_{t_i,k}$$

Mark each conjunction associated with a clause containing a reconfig consequence.

step 4: Refine each clause with action clauses defined in each local policy. For each local policy, repeatedly apply the conditions of each relevant action clause to the conditionals conjuncts of e. E.g.,

$$\forall l_i \in L, \forall \textit{ conjuncts } c_i \in e_{i-1}, \forall A^{l_i}_{t_i,k} \in A^{l_i}_{t_i} \quad e_i = e_i \vee (c_i \wedge A^{l_i}_{t_i,k})$$

Mark each conjunction associated with a clause containing a reconfig consequence. Propagate such marks from previous rounds.

step 5: Add an action clause to the instantiation for each conjunction of $c_i \in e_{|L|}$. Add a reconfig consequence to each clause associated with a marked conjunction.

E.g.,

$$A^I = A^I + t_i : c_i :: accept;$$
$$A^I = A^I + t_i : c_i :: accept, reconfig; \quad (where\ marked)$$

To illustrate, the following describes intermediate and final results of the reconciliation of a group policy with two local policies for an action a_1 (where u, v, w, x, y, and z are conditionals, and * denotes a reconfig mark);

Policy	**Intermediate Result**
Group Policy	
$a_1 : u, v :: accept;$	
$a_1 : w :: accept;$	$e = (u \wedge v) \vee w$
Local Policy 1	
$a_1 : x, y :: accept, reconfig;$	
$a_1 : z :: accept;$	$e = (u \wedge v \wedge x \wedge y)^* \vee (u \wedge v \wedge z) \vee (w \wedge x \wedge y)^* \vee (w \wedge z)$
Local Policy 2	
$a_1 : u :: accept;$	$e = (u \wedge v \wedge x \wedge y)^* \vee (u \wedge v \wedge z) \vee (u \wedge w \wedge x \wedge y)^* \vee (u \wedge w \wedge z)$
Result	$a_1 : u, v, x, y :: accept, reconfig;$
	$a_1 : u, v, z :: accept;$
	$a_1 : u, w, x, y :: accept, reconfig;$
	$a_1 : u, w, z :: accept;$

Note that most reasonable policies will exhibit significant overlap in the action clause conditions. The example above describes an extreme case which is unlikely to occur frequently.

AACR is a P-time algorithm. The internal loops will be executed once per distinct combination of policy and action clauses. Because each clause is defined for exactly one action, the worst case execution of AACR is,

$$O(\textstyle\sum_{j=1}^{|T^g|} |A^g_{t_j}| * |A^{l_1}_{t_j}| * \ldots * |A^{l_{|L|}}_{t_j}|),$$

where $|A^g_{t_j}|$ and $|A^{l_k}_{t_j}|$ is the number of action clauses defined by the group and local policy for t_j, respectively. Hence, AACR is polynomial in the number of action clauses in I and L. □

4.6.3 Compliance

Compliance determines whether a received policy instantiation is consistent with the local policy. One must check the compliance of both the provisioning and authentication and access control policies.

The provisioning policy compliance algorithm tests to see if the received instantiation satisfies the evaluated local policy. The definition of provisioning compliance states that all configuration and pick statements in the evaluated local policy must be satisfied by the instance. Hence, the algorithm simply tests satisfaction.

Algorithm 4.5 (Provisioning Compliance (PC)) **Given**: *A policy instantiation I and a local policy l.* **Find**: *Is I consistent with l evaluated within the run-time environment?*

step 1: Evaluate local policy. Evaluate L (using PEVL) to arrive a pick statements E.

step 2: Remove all compliant policies. Repeat 3 until $|E| = 0$.

step 3: Remove individual compliant policies. Pick $e_k \in E$. If $|e_k \cap I| = 1$, remove e_k from E. Otherwise return **non-compliant**.

step 4: Return **compliant**.

The evaluation algorithm used in step 1 is in P. Step 3 is repeated a maximum of $|E|$, times, each of which executes in $O(|I|)$ (by scanning I). Step 4 executes in constant time. In the worst case, the algorithm executes $O(O(PEVL) + (|E| * |I|)$, and PC is polynomial in the number of configuration values in E and I.

In general, determining compliance of authentication and access control policies has been found to be intractable. Gong and Qian found that the closely related problem of determining interoperability between general purpose authentication policies is NP complete [GQ94]. However, by disallowing negative conditions, Ismene can compute compliance in P-time.

As identified in Section 4.5.3, the process of determining the compliance of an instantiation and a local policy can be reduced to a tautology test. However, the set of connectives used to describe authentication and access control represents a truth-functionally incomplete fragment of Boolean logic. The set authentication clauses resulting from the PEVL algorithm logically contain ($\wedge$) and ($\vee$) operators, and inference ($\rightarrow$) is required for testing compliance. The lack of negation ($\neg$) makes the fragment incomplete. From this property and [Lew78], it can be inferred that a P-time algorithm exists for determining compliance. The following describes one such algorithm.

Algorithm 4.6 (Authentication and Access Control Compliance (AAC))

Given: *A policy instantiation I and a local policy l.* **Question**: *Is I no more permissive than l (i.e., $I \rightarrow L$)?*

For each $t_i \in T^I$ {
  For each $A^l_{t_i,j} \in A^l_{t_i}$ {
    if $|A^l_{t_i}| = 0$ **or** $\exists\ A^I_{t_i,k} \in A^I_{t_i} | A^l_{t_i,j} \not\subseteq A^I_{t_i,k}$
      return **non − compliant**
  }
}
return **compliant**

The AAC algorithm simply determines for each $t_i \in T^I$, if some $A^l_{t_i,j}$ is not completely contained within all clauses of $A^I_{t_i}$ (or no $A^l_{t_i,j}$ exists). If this test fails, then there is a set of conditions under which an action would be allowed by I but not L (which is the definition of non-compliance). AAC scans each $A^I_{t_i}$ once for every $A^l_{t_i,j}$, and AAC executes in $O(|A^I| * |A^l|)$. Hence, AAC is polynomial time in the number of action clauses in l and I.

4.6.4 Analysis

Analysis tests a policy (or in the case of on-line analysis, a policy instantiation) against a set of correctness assertions. Each assertion is a conditional statement representing the legal relations between configurations. Online analysis is performed immediately following the completion of reconciliation as a sanity check of the resulting instantiation. However, this step is not necessary if offline analysis was performed by the policy issuer. A group policy which was deemed consistent by offline analysis cannot be reconciled (with any set of local policies) to an instantiation violating the assertions. The following considers the complexity and construction of analysis algorithms.

Algorithm 4.7 (Online Policy Analysis (ONPA)) **Given**: *A instantiation I and a set of assertions S?* **Find**: *Is I consistent with the assertions S?*

Online analysis tests whether a policy instantiation is compliant with the set of assertions. I defines a truth assignment for F (where each configuration $x \in F$ is true if $x \in I$ and false otherwise). Online analysis is the process of evaluating the truth of each expression $s_i \in S$ over this truth assignment. As the ONPA algorithm is obvious and trivially in P, further details are omitted.

Offline analysis attempts to determine if any set of conditions and local policies can lead to an instantiation violating the assertions. Unlike authentication and access control clauses, negated conditionals are allowed in assertions. Negation is required for the specification of incompatibility, and thus essential to defining relations between configurations. As offline analysis seeks a *succinct disqualifer* (counter-example) for a functionally complete fragment of Boolean logic, it is in coNP. The following reduction confirms this fact.

Reduction 4.3 (Offline Policy Analysis (OFPA))

The following definitions state specifics of OFPA and VALIDITY.

Definition 4.5 (Offline Policy Analysis (OFPA))

Given: A group policy g and set of assertions S. **Question**: Would any possible reconciliation of g violate an assertion in S?

Definition 4.6 (VALIDITY)

Given: An arbitrary Boolean expression e defined over the universe of variables U.

Question: Is e valid?

Clearly, a non-deterministic Turing machine can guess a violating set of conditions for OFPA, so OFPA is in coNP. The following construction reduces VALIDITY to OFPA in polynomial time. Create g by defining a *provision* clause containing a tag consequence (l_1) for the first variable $x_1 \in U$. Create four clauses for each variable $x_i \in U$ as follows;

$$
\begin{array}{ll}
provision : :: l_1; & l_2 : x_2, \overline{x_2} :: fail; \\
l_1 : x_1, \overline{x_1} :: fail; & \ldots \\
l_1 : x_1 :: l_2; & \nearrow \; l_i : x_i :: p; \\
l_1 : \overline{x_1} :: l_2; & \quad\;\; l_i : \overline{x_i} :: p; \\
l_1 : :: fail; & \quad\;\; l_i : :: fail;
\end{array}
$$

Note that the last set of clauses for $x_i \in U$ references a tag to the clauses for p. Convert e into DNF. For each conjuncts $c_i \in e$, create the clause $p : c_i :: r;$, where the conditionals enumerate are the (possibly negated) variables of c_i, and r is a arbitrary configuration. Complete g by appending the default clause containing a single f configuration clause ($p : \; :: f;$), and a fail clause ($fail : \; :: t;$). Complete the construction for by creating a single assertion ($assert : ::!f;$).

To illustrate, an expression $(a \wedge b \wedge c) \vee (\overline{a} \wedge \overline{b} \wedge d) \vee (\overline{c} \wedge \overline{d} \wedge e)$ would result in the following g and S;

$$
\begin{array}{rll}
g = & provision :: l_1; & l_e :: fail; \\
& l_a : a, \overline{a} :: fail; & p : a, b, c :: t; \\
& l_a : a :: l_b; & p : \overline{a}, \overline{b}, d :: t; \\
& l_a : \overline{a} :: l_b; \quad \nearrow & p : \overline{c}, \overline{d}, e :: t; \\
& l_a : :: fail; & p : :: f; \\
& l_b : b, \overline{b} :: fail; & fail : :: t; \\
& \ldots & \\
S = & assert : :: \quad !f; &
\end{array}
$$

Now consider the possible evaluations of g. Each positive or negative assignment of variable $x_1 \in U$ is defined as a unique condition. The evaluation of the clauses l_{x_1} has two possible results; if the condition x_1 and $\overline{x_1}$ are both true or neither is, the evaluation algorithm will immediately drop to the $fail$ clause which defines a single condition t. In this case, the assertion test will trivially be satisfied by this evaluation. If exactly one of the conditions x_1 and $\overline{x_1}$ is TRUE, then the clauses associated with x_2 are consulted. This process repeats until either the $fail$ clause or the first clause associated with p is reached. If the p clause is reached, then the conditions represent a legal truth assignment for U. Moreover, it is clear that no legal truth assignment for U arrives at $fail$.

Now, consider the evaluation of the clauses of p. Because e is represented in DNF, any truth assignment for U must satisfy at least one conjunct for e to be valid. The evaluation of some p clause will arrive at configuration t if any conjunct is satisfied by the truth assignment for U, and f otherwise. If e is valid, the default clause for p will never be reached (because no legal truth assignment will not satisfy at least one conjunct of e). The assertion will never be violated by OFPA. Because any invalid expression will violate some not satisfy all p clauses for some truth assignment, the assertion will violated by OFPA. Because VALIDITY is a known to be in coNP, so is OFPA. □

Even though offline analysis is intractable, it provides an essential service. The following algorithm reduces the cost over a naive approach by identifying independence between configurations and conditions. This allows analysis to be performed only over the set of conditions relevant to an assertion.

Algorithm 4.8 (Offline Policy Analysis (OFPA)) **Given**: *A group policy g and set of assertions S.* **Question**: *Would any possible reconciliation of g violate an assertion*

in S?

step 1: Construct an expression for each configuration identified in an assertion describing the conditions under which may be reached. This is accomplished by traversing g backward from any clause in which the configuration is a consequence. For example,

$$\begin{array}{l} provision : d_1, d_2 : t_1; \\ provision : t_2; \\ t_1 : d_3, d_4 :: f_2; \\ t_1 : :: t_3; \\ t_2 : d_5 :: f_1; \\ t_2 : :: f_2; \\ t_3 : d_5 :: f_1; \end{array} \quad \rightarrow \quad \begin{array}{l} f_1 : (d_1 \wedge d_2 \wedge (\overline{d_3} \vee \overline{d_4}) \wedge d_5) \vee ((\overline{d_1} \vee \overline{d_2}) \wedge d_5) \\ f_2 : (d_1 \wedge d_2 \wedge d_3 \wedge d_4) \vee ((\overline{d_1} \vee \overline{d_2}) \wedge \overline{d_5}) \end{array}$$

The resulting expression for each configuration $d_i \in S$ is denoted as e^{d_i}

step 2: Construct assertion expressions. For each assertion $s_i \in S$, construct an expression representing the relation described. For example, the assertions

$$\begin{array}{l} assert : config(d_1), config(d_2) : !config(d_3), config(d_4); \\ assert : pick(config(d_1), config(d_2)) : config(d_3); \end{array}$$

would result in the following expressions, respectively;

$$\begin{array}{l} (e^{d_1} \wedge e^{d_2}) \rightarrow ((\overline{e^{d_3}}) \wedge e^{d_4}) \\ (e^{d_1} \vee e^{d_2}) \rightarrow e^{d_3} \end{array}$$

The resulting expression for each configuration $s_i \in S$ is denoted as e^{s_i}

step 3: Detect assertion violations. Analysis is the process of determining if each e^{s_i} is valid. Any truth assignment making e^{s_i} false represents a set of conditions under which the assertion is violated. Hence, a general purpose validity tester is used for analysis and violations reported.

Note that the e^{d_i} expressions constructed by offline analysis can also be used to detect potential violations of the language restriction 1 defined in Section 4.5.1. An expression e^{d_i} represents a disjunction of conditions leading to each distinct occurrence of $d_i \in g$. The language restriction requires that each such occurrence be mutually exclusive. Hence, violations of the language restriction can be detected by testing the satisfiability of the logical *and* of the pair-wise disjunctions. A satisfying truth assignment of conditions would represent an evaluated policy in which a configuration occurs twice, and hence be a violation of the restriction.

CHAPTER 5

POLICY ENFORCEMENT IN ANTIGONE

Network security has historically suffered from poorly coordinated services. Developers must often construct applications from diverse and largely disconnected security services. Moreover, the introduction of additional application requirements (e.g., fault tolerance) requires a re-assessment of the use of security. Subtle interactions between services introduced by naive implementations can lead to undetected vulnerabilities.

Traditional monolithic security architectures seek to address all aspects of security in a single, tightly integrated implementation [FKTT98]. Applications built on these architectures often defer all aspects of security to the architecture. The flexibility with which trust and threat models appropriate for an application or session are defined is frequently limited. As a result, applications must pay a performance penalty (in implementing unnecessary security services) or accept undesirable vulnerabilities (where security requirements are unaddressed).

Antigone extends previous work in component based security through a *policy enforcement architecture*. Previous systems have sought to compose protocols or collections of services from compile- or configuration-time specifications. However, the delivery, correctness, and synchronization of specifications is outside the scope of their definition, and few systems have meaningfully addressed authentication and access control. Hence, while these frameworks allow the efficient construction of services, little support for the management of the run-time session context is provided. Antigone builds on these works by providing not only a service construction framework, but by regulating the construction and use of the service using run-time distributed security policies.

This chapter details the design and operation of the Antigone policy enforcement architecture. Antigone is a middleware layer [Ber96] enforcing group security policy through the initialization, configuration, and coordination of services needed to implement the group. Group and service activities are directed by a policy instantiation provided by an external

policy determination architecture. Each end system security service is implemented through a *mechanism* conforming to uniform *signal* interfaces. As is the promise of component based systems, different services (supporting potentially different policies) can be composed freely.

Antigone assumes that a policy determination architecture (e.g., Ismene) is available. However, Antigone is not dependent on Ismene. Other group policy specifications (e.g., GSAKMP policy token [HCH+00], DCCM Cryptographic Context [BBD+99, DBH+00]) can be used to direct the Antigone services. However, the use of these policy specifications requires the creation of software compliant with the Antigone policy interfaces (see Section 5.3).

The following sections address several primary objectives of this thesis identified in Chapter 2; *Flexible Policy Enforcement*, *Efficient Enforcement*, and through the broadcast transport layer, *Transport Agnostic* communication. The following section presents an overview of the Antigone policy enforcement approach and architecture. Section 5.2 describes the operation of the group interface layer. Section 5.3 gives an overview of the integration of a policy determination architecture through the policy engine interfaces. Section 5.4 presents the mechanism layer, and details the design and operation of several example mechanisms. Section 5.5 presents the broadcast transport layer. Section 5.6 concludes with a description of several architectural constructs used to optimize policy enforcement.

5.1 Policy Enforcement

Enforcement is the process whereby the semantics of a policy are realized in software. Policy can be defined by separate, but related, aspects of policy *representation*, system *provisioning* and session *authentication and access control*. The following considers the goals of Antigone with respect to these facets of policy.

A policy representation determines the form and semantics of policy. Each environment may have different systems for determining and evaluating policy. Hence, as no single policy representation is likely to be applicable to all environments, enforcement should not be dependent on any policy determination architecture.

Provisioning defines the services and configurations used to support communication. However, the static provisioning found in monolithic security architectures is not appropriate for all environments. The requirements of an application may differ for each session. Hence,

communication provisioning should be made in accordance with the run-time requirements dictated by policy. The effort required to integrate security services addressing new security requirements should be low.

Authentication and access control determines whom and in what capacity processes may participate in a session. A singular model or service for authentication access control is unlikely to meet the requirements of all environments. Hence, Antigone should support a variety of authentication and access control services. Note that while the enforcement of authentication and access control must be performed by Antigone, the interpretation of policies (decision making) is deferred to the policy determination architecture.

The remainder of this section defines how these goals are addressed through a policy enforcement architecture.

5.1.1 Mechanisms

An Antigone *mechanism* defines some basic service required by the group. Each mechanism is identified by its type and implementation. A type defines the kind of service implemented. Antigone currently supports six mechanism types; authentication, membership management, key management, data handling, failure detection and recovery, and debugging. A mechanism implementation defines the specific service provided. For example, there are currently three key management implementations; Key-Encrypting-Key, Implicit Group Key Management, and Logical Key Hierarchy. These categories are not exhaustive; new types (e.g., congestion control) or implementations (e.g., One-Way Function Tree Key Management) can be integrated with Antigone easily. Associated with each mechanism is a set of configuration parameters (or just configurations). Configurations are used to further specify the behavior of the mechanism. For example, a data handling mechanism providing confidentiality may be configured to use triple-DES. Details of the current mechanisms are detailed in Section 5.4.

The set of mechanisms and configurations used to implement the session (provisioning) is explicitly defined by policy. The policy determination architecture is consulted at session initialization (or following policy evolution) for a provisioning policy. This policy is enforced by the creation and configuration of the appropriate mechanisms.

Unlike traditional protocol objects in component protocol systems [SFS93, HP94, BHSC98], mechanisms are not vertically layered (e.g., layered services of TCP/IP stacks). This does

not imply that an implementation be defined by monolithic or course-grained component protocol stacks. Each mechanism implements an independent state machine, which itself may be layered. For example, the Cactus-based membership service defined in [HS98] can be used as a membership mechanism within Antigone. In this case, the mechanism configuration determines the protocol graph constructed at run-time.

5.1.2 Signals

Internally, group operation is modeled in Antigone as *signals*. Each signal indicates that some relevant state change has occurred. Policy is enforced through the observation, generation, and processing of signals. Antigone defines event, timer expiration, and message signals.

Events signal an internal state change. An event is defined by its type and data. For example, send events are created in response to an application calling the sendMessage API. This event signals that the application desires to broadcast data to the group. A send event has the type `EVT_SEND_MSG` and its data is the buffer containing the bytes to be broadcast. A table of the basic events defined by Antigone is presented in Table 5.1. Note that mechanisms are free to define new events as needed. This is useful where sets of cooperating mechanisms need to communicate implementation specific state changes.

A *timer expiration* indicates that a previously defined interval has expired. Timers may be global or mechanism-specific; all mechanisms are notified at the expiration of a global timer, and the creating mechanism is notified of the expiration of a mechanism specific timer. Similar to events, a timer is defined by its type and data. For example, a join request retry mechanism timer may signal that a request has timed out. The timer data identifies context-specific information (a nonce) required to generate a join request. Timers are registered with a global timer queue (ordered by expiration). Timers may be unregistered (removed from the queue) or reset prior to expiration.

Messages are created upon reception of data from the underlying broadcast transport service (i.e., broadcast transport layer, see Section 5.5). Messages are specific to (must be marshaled/processed by) a mechanism. Every message m is defined by (and is transmitted with a header including) the tuple $\{m_t, m_i, m_g\}$, where m_t identifies a (one byte) mechanism type, m_i identifies a (one byte) mechanism implementation, and a (two byte) m_t defining the message type. For example, the header {`KEY_MECH`, `KEK_KEY_MECH`, `AKK_REKEY`} header

Event	Meaning	Data
EVT_AUTH_REQ	Authentication request	*none*
EVT_AUTH_COM	Authentication complete	join nonce
EVT_AUTH_FAL	Authentication failed	*none*
EVT_JOIN_REQ	Join request	join nonce
EVT_JOIN_COM	Join complete	None
EVT_JOIN_MEM	New user in group	member identifier
EVT_REJN_MEM	Member attempting to rejoin	member identifier
EVT_LEAV_REQ	Request to leave	*none*
EVT_EJCT_REQ	Request member ejection	member identifier
EVT_MEM_EJCT	A member has been ejected	member identifier
EVT_EJCT_COM	Member ejection	Boolean (TRUE = successful)
EVT_MEM_LEAV	A Member has left the group	member identifier
EVT_LEFT_GRP	Local left group	*none*
EVT_NEW_GRUP	New group ID accepted	*none*
EVT_SEND_MSG	Send message	application data
EVT_SENT_MSG	A message has been broadcast	application data
EVT_DAT_RECV	Data message received	received application data
EVT_KDST_DRP	Lost key distribution message	*none*
EVT_GROP_LST	Group communication lost	*none*
EVT_PRC_FAIL	Process failure	member identifier
EVT_CRECOVER	Client recover request	member identifier
EVT_POL_RCVD	Policy received	policy
EVT_NGRP_POL	New Group Policy	*none*
EVT_POL_EVGP	Policy Evolution	policy
EVT_SHUTDOWN	Group Shutdown	shutdown the interfaces to the group
EVT_SHUT_COM	Group Shutdown Complete	shutdown complete
EVT_INFO_MSG	Informational Event	information string
EVT_ERRORED	Signal Unrecoverable error	error description string

Table 5.1: Basic Antigone Events - events signal a change of state in the group. Mechanisms are free to define new events as needed.

identifies a key management, KEK implementation, rekey message. Type, implementation, and message identifiers are used to partition the message identifier space. Header information is later used to route the message to the appropriate implementation for unmarshaling and processing (see below).

The interfaces used to create and deliver signals are presented in Figure 5.1. Each signal type uses a process function to deliver the signal to the mechanism. Events are created and queued via the post event interface. Timers are created and placed in the timer queue via the register timer interface. Messages are sent to the group using the send message interface.

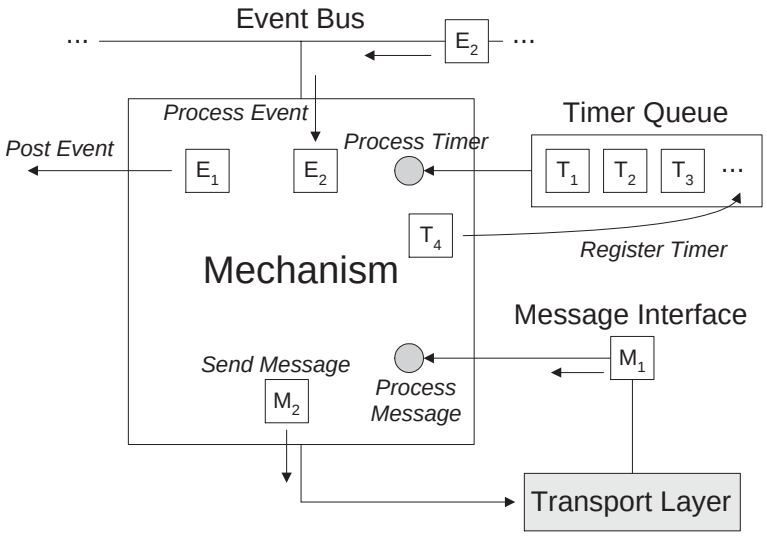


Figure 5.1: Mechanism Signal Interfaces - Policy is enforced through creation and processing of *events*, *timers*, and *messages*. To simplify, events are posted to and received via the event bus. The expiration of timers registered to the timer queue is signaled to the mechanism through the process timer interface. Messages are sent to the group via the send message interface, and received through the process message interface.

5.1.3 Group Interface

The group interface arbitrates communication between the application and mechanisms of Antigone through a simple message oriented API. Actions such as join, send, receive, and leave are provided through simple C++ object methods. These actions are translated into events. Group state (e.g., received messages) are polled by the application through API calls.

The group interface implements event and timer signal processing functions. The group interface implementation does not directly send or receive messages. All communication with other processes is performed indirectly through mechanisms. However, the group interface acts as a de-multiplexer for received data. Messages received from the group are forwarded to the appropriate mechanisms based on header information. Mechanisms subsequently unmarshal and process received messages.

5.1.4 The Event Bus

The *event bus* directs the delivery of events to mechanisms. Depicted in Figure 5.2, the event bus defines the interface between the group interface and mechanisms. To simplify, all communication between these layers and between mechanisms is through the event bus.

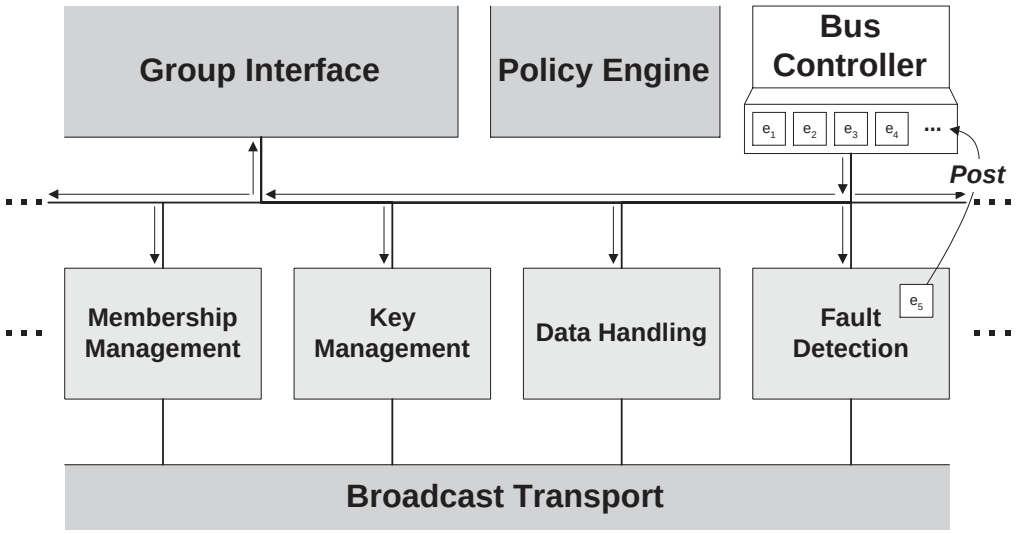


Figure 5.2: The Event Bus - the event bus manages the delivery of events between the group interface and mechanisms of Antigone. Events are *posted* to the bus controller event queue. Events are subsequently broadcast to all software connected to bus in FIFO order. Note that the event bus is implemented in software and is completely independent of network broadcast service supported by the broadcast transport layer.

During initialization (see Section 5.3), the set of mechanisms defined by an instantiation are created and logically connected to the event bus. Mechanisms are removed from the bus when the group is destroyed or reprovisioned during policy evolution.

The bus controller is a software service that implements ordered delivery of events. The group interface and mechanisms *post* events to the bus controller. Posted events are subsequently delivered in FIFO order. Critical events (e.g., group errored) are placed at the head of the queue through a *priority post*.

Logically, the event bus is a broadcast service. All posted events are delivered to every mechanism and the group interface. Each mechanism processes events received on the bus in accordance with its purpose and configuration. After that, the mechanism signals the bus controller that the event has been processed. Unprocessed events are logged.

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 group interface before any other event (including a priority event) is processed. These guarantees are preserved by mechanism acknowledgement of event processing completion. The event bus provides no guarantees on the ordering of mechanisms to which the event is delivered. This places additional requirements on event processing.

For example, consider a data handling service that transmits a message in response to a send event, and a group congestion control service [MJV96] that wishes to place an upper bound on transmissions per quanta. A naive implementation of a data handler would simply transmit data upon reception of a send event, and the congestion control mechanism would queue messages when the local member's fair share (of bandwidth) is exceeded. The naive implementation would thus (incorrectly) both transmit and queue the data. Several solutions to this problem exist. First, congestion control and data handling may be integrated into the same mechanism (which in many cases may not be possible or convenient). Second, one could require that all policies configuring congestion control must also configure the data handler to be cognizant of congestion control (e.g., through policy assertions). In this case, the data handler would ignore sent events, and only transmit in response to a `congest_send` event posted by the congestion control mechanism.

In general, dependencies between events are few [MPH99]. Hence, the response of a mechanism to a particular event is largely independent of other mechanisms. However, careful analysis of the effect of an event on all possible mechanisms is necessary. The composition mechanisms should be restricted (e.g., through assertions in Ismene) to only allow compatible mechanisms and configurations.

5.1.5 Attribute Sets

State is shared by the components of Antigone through the *group attribute set*. Similar to the KeyNote action environment [BFIK99b], the attribute set maintains a table of typed attributes. Attributes are defined through a {name, type, value} tuple. Mechanisms and the group interface are free to add, modify, or remove attributes from the table. Attributes are defined over basic data types (e.g., strings, integers, Boolean), identities (e.g., unique identifier), and credentials (e.g., keys, certificates). The group attribute set defines the current context of the group. For example, groups using a symmetric session key maintain the current session key through the SessionKey attribute. Mechanisms access the key by acquiring it from the group attribute set.

Authentication and access control decisions are deferred to the Policy Engine (See Section 5.3). However, mechanisms must supply information describing the context under which a particular action is attempted. The mechanism testing an action constructs an action set (which is frequently a subset of the group attribute set) from relevant informa-

Action	Meaning
group_auth	member authentication of group
member_auth	group authentication of member
acquire	a potential participant policy acquisition
join	a member access to the group
view_dist	accept a view distribution
eject	request the ejection of another member
leave	accept a leave request
leave_resp	accept a leave response
key_dist	accept a key distribution
rekey	accept a group rekey
send	send data to the group
content_auth	source authenticate data
group_mon	accept a group monitor information
member_mon	accept member monitor information
accept_policy	accept a policy instantiation
reconfig	initiate policy evolution
shutdown	accept a shutdown message

Table 5.2: Basic Antigone Actions - actions under which Antigone authentication and access control policy is defined. New actions may be introduced by mechanisms and applications as needed.

tion. The context primarily consists of the credentials used to prove identity and rights. All cryptographic material (e.g., keys, certificates) are modeled as credentials. Mechanisms provide the set of credentials and attributes associated with the action being performed through the action set. For example, a certificate provided by a joining member may be used as a credential to gain access to the group. The mechanism must decide, based on information provided, on the appropriate set of attributes to provide to the policy engine. For example, acceptance of an incoming packet encrypted under a current session key implies knowledge of the session key. Hence, the session key can be used as credential when assessing acceptance. The action being attempted is defined through the action attribute. A table of basic actions used in the current implementation are presented in Table 5.2.

5.1.6 Policy Enforcement Illustrated

This section briefly illustrates how the group interface, policy engine, event controller, and mechanisms work in concert to enforce policy. The following example demonstrates the enforcement of data security, failure detection, and authentication and access control

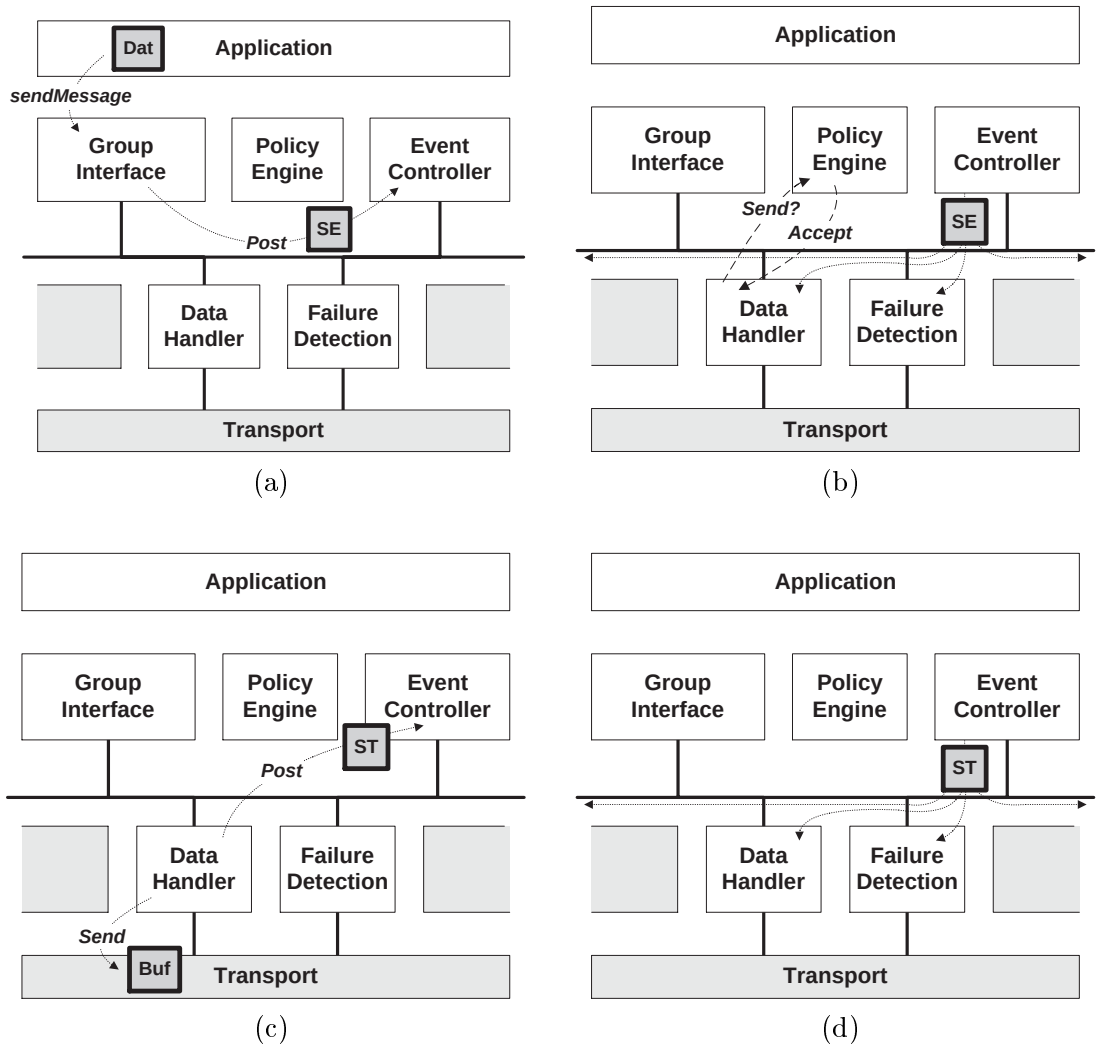


Figure 5.3: Policy Enforcement Illustrated - an application `sendMessage` API call is translated into a *send event* delivered to all mechanisms (a). This triggers the evaluation of an authentication and access control policy via upcall (b), and ultimately to the broadcasting of the application data (c). The send triggers further event generation and processing (d). Note that the policy engine does not listen to or create events.

policies associated with the sending of an application message. The policy under which this example is defined requires application content confidentiality. Furthermore, the policy requires failure detection to be supported through a timed heartbeat detection mechanism (see Section 5.4.5). Figure 5.3 and the following text illustrate how this policy is enforced (where the letters a, b, c and d correspond to the labeled figures);

a) The application attempts to broadcast data to the group via the `sendMessage` API call. The call is translated into an `EVT_SEND_MSG` event (SE) by the group interface,

which is posted to the event controller. The application data (Dat) is encapsulated by the send event.

b) The event controller delivers the send event to all mechanisms. The data handler tests the *send* action in response to the delivery of this event by an upcall to the policy engine. Credentials supplied by the local user are passed to the policy engine. For this example, the policy engine accepts the send action.

c) The data handler mechanism encrypts the application data using a session key obtained from the attribute set. A *confidentiality only* message is constructed by placing the appropriate headers and encrypted data into a buffer (Buf). The buffer is then broadcast to the group via the transport layer. An `EVT_SENT_MSG` (ST) event containing the sent buffer is posted to the event queue following the transmission.

d) The sent event is posted to all mechanisms. The failure detection mechanism, using the send as an implicit heartbeat message, resets an internal heartbeat transmission timer.

Other policies may dictate very different behavior. For example, the kinds of data transforms and the reaction of mechanisms to sent data may be very different. This is the promise of policy driven behavior; an application can specify precisely the desired behavior through the definition of group provisioning and authentication and access control.

5.1.7 Architecture

Described in Fig. 5.4, the Antigone architecture consists of four components; the group interface layer, the mechanism layer, the policy engine, and the broadcast transport layer. As described in Section 5.1.3, the group interface layer arbitrates communication between the application and mechanism layer.

The mechanism layer provides a set of mechanisms used to implement security policies. The mechanisms and configuration to be used in a session are defined by the policy instance. While the Antigone implementation currently provides a suite of mechanisms appropriate for many environments, new mechanisms can be developed and easily integrated with Antigone. Note that mechanisms need not only provide security services; other relevant functions (e.g., auditing, failure detection and recovery, replication) can be implemented. For example, Antigone implements a novel crash failure detection mechanism [MP00].

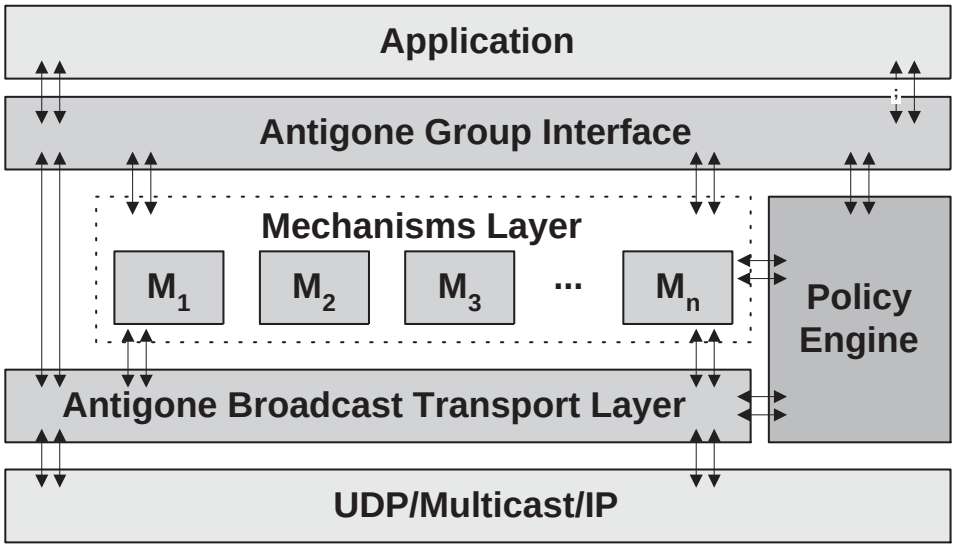


Figure 5.4: Antigone consists of four components; the group interface layer, the mechanism layer, the policy engine, and the broadcast transport layer. The group interface layer arbitrates communication between the application and lower layers of Antigone through a simple message oriented API. The mechanism layer provides a set of software services used to implement secure groups. The policy engine directs the configuration and operation of mechanisms through the evaluation of group and local policies. The broadcast transport layer provides a single group communication abstraction supporting varying network environments.

The policy engine directs the configuration and operation of mechanisms through the evaluation of policies (i.e., reconciliation and compliance checking). Initially, as directed by the policy instance, the policy engine provisions the mechanism layer by initializing and configuring the appropriate software mechanisms. The policy engine subsequently governs protected actions through the evaluation of authentication and access control policy.

The broadcast transport layer defines a single abstraction for unreliable group communication. Due to a number of economic and technological issues, multicast is not yet globally available. Thus, where needed, Antigone emulates a multicast channel using the available network resources in the transport layer.

5.1.8 Alternative Architectures

While many aspects of the Antigone architecture are present in previous works, the unique requirements of policy enforcement made the direct use of existing component frameworks inappropriate. Centrally, the need to compose re-configurable, tightly coupled, and fine-grained protocol components dictated the development of infrastructure not present in

extant systems.

A number of recent works have investigated the construction of flexible and efficient distributed systems from components [HP94, SFS93, OOSS94, BHSC98]. Components conforming to uniform interfaces are composed in different ways to address application requirements. Hence, new requirements can be quickly addressed by altering the composition of underlying components. This approach has been successfully extended to security [MQRG97, NK98, HJSU00, Wra00], where services and protocols addressing a specific set of security requirements are built from components. These works significantly constrain system organization; largely motivated by protocol stack designs, components are organized into vertical or hierarchical message processing pipelines. Hence, these frameworks are suitable for the creation of tightly coupled protocol state machines. Antigone, in contrast, composes loosely coupled services. Each Antigone service transmits messages, processes timers, and monitors state independently of other services. Hence, the traditional model of layered services (e.g., TCP/IP) is inconsistent with the needs of Antigone. Moreover, the interfaces over which state is communicated in traditional protocol component systems are typically restricted to connection management and data handling information. Note that while these architectures are not well suited to Antigone, they may be useful in creating flexible implementations of individual mechanisms within Antigone.

Typically used in the construction of complex distributed systems in heterogeneous environments, configuration programming frameworks specify component interfaces through a language-agnostic module interconnection language (MIL) [Kra90, Pur94]. Distributed systems are constructed by developers from component interconnection specifications. The framework translates and routes all communication between the components defined by the developer. As these systems are designed to support communication between largely autonomous and distributed components, shared state is explicitly forbidden. In contrast, the mechanisms of Antigone are required to share a significant amount of state (e.g., keys, timers, attributes, etc.). Hence, the loose coupling and translation overheads make these frameworks inappropriate for end-host policy enforcement.

Software buses have traditionally been used to construct distributed object architectures [HWC95, Sch95, Ses97, Vin94, Wal99, OKP00]. Components in these frameworks are typically used to define interfaces to database, compute, or user-interface services. Communication between components is handled via standardized marshaling interfaces. Hence,

tool-kits of diverse components can be used to flexibly construct distributed systems. Components in these systems represent course-grained and possibly distributed services. Hence, the overheads associated with inter-component communication (i.e., marshaling and inter-process communication) are in conflict with the needs of high-performance protocol stacks.

5.2 Group Interface

The group interface acts as a conduit for communication between the application and the Antigone mechanisms, and performs the high level direction of the policy management. These duties include the translation of application requests into events, the coordination of mechanism initialization and operation, and the queuing of incoming and outgoing data.

As detailed in Section 5.3, the group interface consults the local policy for an initial configuration (prior to receiving the policy instantiation). This policy (minimally) defines a service used to initiate communication with the group and acquire the instantiation. Once the group interface and mechanisms are initialized, the application is required to call the blocking `Connect` API. This call is translated into an `EVT_AUTH_REQ` event posted to the event controller. The various mechanisms will perform authentication in response to this event (see authentication mechanism Section 5.4.1). The completion of the authentication process is signaled through the `EVT_AUTH_FAL` (authentication failed) or `EVT_AUTH_COM` (authentication successful) event. If authentication fails, an error is reported to the application. If authentication is successful, an `EVT_POL_RCVD` event identifying the instantiation is posted by the authentication mechanism. The group interface passes the opaque policy structure associated with the event to the policy engine. The policy engine deactivates the initial configuration and configures the mechanisms layer as dictated by the instantiation. Once the policy engine completes this task, an `EVT_NGRP_POL` initialization event is posted, and the `Connect` call returns.

The group interface provides a simple message oriented API. However, an application desiring to view the current membership, obtain the current group state, or access policy is free to use advanced interfaces. Each relevant API call is translated into an event by the group interface. For example, an `EVT_SEND_MSG` event is created in response to an application sendMessage API call. The event is posted to the event controller and ultimately delivered to the mechanisms via the event bus.

Similarly, relevant events delivered to the group interface over the event bus are signaled to the application. The means by which this signaling is achieved is event-specific. Upon reception of an `EVT_DAT_RECV` event, the group interface places the message buffer associated with the event on the *receive queue*. The application can determine the state of the receive queue through the `messagePending` API. Messages are extracted from the receive queue via the `readMessage` API. Typically, an application polls the receive queue, acquiring message buffers as they become available[1].

`EVT_ERRORED` events signal to the group interface that an unrecoverable error has occurred. An `EVT_SHUTDOWN` event is posted following the observation of an error event. The group interface waits for an `EVT_SHUT_COM` event. This latter event indicates that the local mechanisms have cleaned up their internal state and the group interface may be destroyed.

An application exits the group via the blocking `Quit` API call. The `Quit` call posts an `EVT_LEAV_REQ` handled by the appropriate mechanisms. The `EVT_LEFT_GRP` event is used to signal the completion of the member leave. Antigone is then deactivated through the shutdown events as described above.

5.3 The Policy Engine

Depicted in Figure 5.4, the policy engine acts as the central enforcement agent in Antigone. All interpretation of policy occurs within the policy engine. This has the advantage of allowing the integration of other policy approaches. For example, a group desiring to enforce the policy defined by a GSAKMP *policy token* [HCH+00] would simply replace the current Ismene policy engine with a GSAKMP policy engine. As is true for Ismene policy instantiations, the token would be distributed by the authentication mechanisms as opaque data. Subsequent enforcement of the policy is relegated to the replacement policy engine.

There are three central tasks of a policy engine; initialization, authentication and access control policy evaluation, and group evolution. The policy engine directs the initialization and configuration of mechanisms upon reconciliation or reception of the policy instantiation. The initiator interprets the provisioning policy by creating a mechanism object for each mechanism defined in the policy instantiation. Mechanisms are configured using the

[1] The group interface provides timed or indefinitely blocking receive methods, and `select` and file descriptor set utility methods. Hence, Antigone can be quickly integrated with existing applications using standard network programming techniques.

configuration statements in the instantiation immediately following their creation.

Non-initiator participants are faced with a dilemma prior to contacting the group; they do not possess the instantiation with which they can initialize Antigone. This is solved by using an instantiation resulting from the self-reconciliation of the local policy (which in Ismene, for any correctly constructed policy, is guaranteed to terminate successfully[2]). However, several requirements are placed on this local policy. First the local policy must specify an *authentication* mechanism used to contact the group and acquire the instantiation. Secondly, an access control policy stating from whom an instantiation can be accepted must be defined. The instantiation defined by the local policy is used to initialize the set of services used to contact the group. Once the instantiation is received, the member determines its compliance with the local policy. If compliant, the mechanisms and configuration defined by the local policy are discarded, and Antigone is re-initialized using the configurations defined in the instantiation.

The enforcement of authentication and access control is performed by the policy engine throughout the session. Each mechanism is cognizant of the actions to be protected by policy (i.e., hard-coded in implementation). For example, a *membership* mechanism consults the policy engine when a participant attempts to join the group. The policy stating the requirements to gain access to the group (i.e., the conditions and credentials) are stated in the *join* authentication and access control clauses. The Ismene policy engine performs the Authentication and Access Control Evaluation algorithm (AEVL) defined in Section 4.6.1 to arrive at an acceptance decision. Note that how a participant joins the group is largely independent of evaluation. Policy engines implementing other languages behave in essentially the same way, with the exception of the evaluation of conditions and authentication statements. Antigone supports a range of basic actions protected by policy through the current mechanism implementations. However, mechanisms are free to define new protected actions. All policies must acknowledge the existence of the action through the definition of authentication and access control clauses.

Policy evolution occurs when a `reconfig` consequence (or similar construct in other policy languages) is enacted by the policy engine. `reconfig` signals to the group some aspect of the group has fundamentally changed, and that this change requires the group

[2]Not all policy languages implement local policies. In this case, some other means of communicating an initial configuration must be found. For example, a simple configuration file can be used to state a local policy.

re-assess its provisioning and authentication and access control (policy evolution). The group disbands in response to the observation of the `reconfig` event. At this point, the initiator performs reconciliation (potentially under a new set of local policies), and the group is initialized as before. Note that initiation of the this process is in itself a protected action. Left unprotected, a malicious member of the group may mount a denial of service by continually signaling reconfiguration.

5.4 Mechanisms

Policy in Antigone is enforced through the software modules called *mechanisms*. Each Antigone mechanism consists of a set of behaviors and associated protocols designed to perform some service within the session. The current mechanisms layer defines six types of mechanisms; authentication, membership, key management, data handling, failure detection and recovery, and debugging.

The mechanisms layer coordinates the construction of mechanisms. Mechanisms are created from a repository of implementations by the *mechanism factory* as directed by the policy engine. The factory maps unique mechanism identifiers onto an implementation. Once created, the mechanism is configured and attached to the event bus.

This section describes mechanisms for a centralized group. Centralized groups contain a distinct member performing policy distribution and authentication (known throughout as the authentication service), membership management (admittance entity), key distribution (group key controller), and failure detection (failure monitor). For simplicity, the following assumes the initiator is the central entity for all these functions. However, new mechanisms and policies may be introduced to distribute the various centralized functions to one or more members of the group. In the extreme case, such as in *participatory key management*, all members collaborate to provide a function. The following text describes the requirements, interfaces, and operation of Antigone mechanisms.

5.4.1 Authentication Mechanisms

Authentication mechanisms provide facilities for potential group members (requestors) to initiate communication with the group. All authentication mechanisms implement protocols performing mutual authentication and acquiring the policy instantiation. This typically

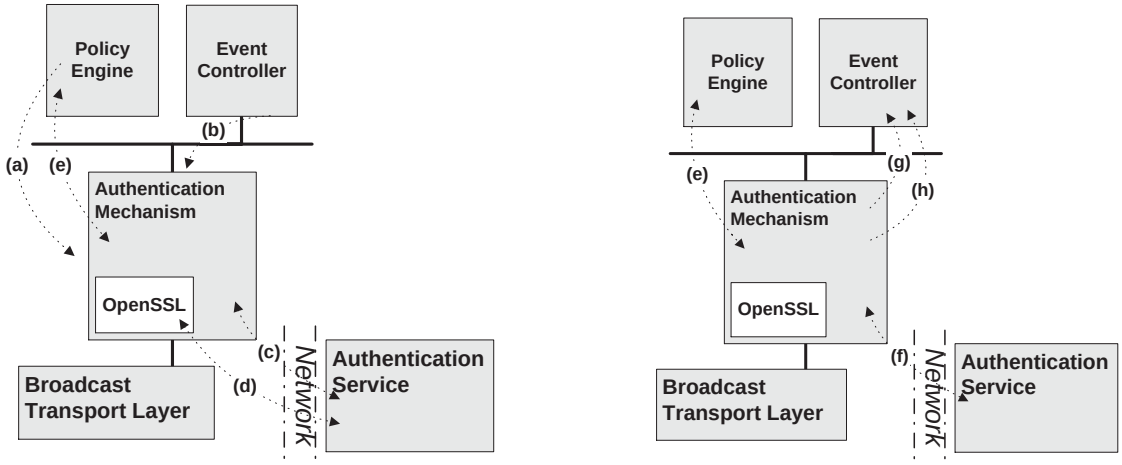


Figure 5.5: Authentication Mechanism - The authentication mechanism is initialized by the policy engine (a), after which authentication request event is received. The mechanism responds by locating the authentication service and establishing a secure channel (b,c,d). After authenticating the group (e), the channel is used to exchange policy and session state (f). The authentication process is completed by posting a policy received and authentication complete event (g,h) to the event controller.

requires an authentication and key exchange protocol between the member and an authentication service. The authentication mechanism implements both the requester (joining member) and service (initiator processing authentication requests) sides of the authentication.

As with any mechanism, the authentication mechanism is created by the policy engine when the application is initialized. The local policy is evaluated to arrive at a set of mechanisms and their configurations. The mechanisms are created by calling the appropriate mechanism constructor functions which are passed the configuration parameters. The newly initialized mechanisms then wait for events.

The requestor initiates an authentication protocol after receiving an `EVT_AUTH_REQ` event (emitted after completion of the mechanism initialization) . How the mechanism proceeds is dependent on its implementation, its configuration, and statements of authentication and access control. Typically, the requestor will initiate an exchange with the authentication service. For example, the Leighton-Micali key exchange [LM94] protocol was used in an early version of Antigone [MPH99]. Alternately, mutual authentication can be established via some external authentication service, (e.g., Kerberos [NT94]). Antigone currently implements three authentication mechanisms; a null authentication mechanism, an OpenSSL [Gro00] based mechanism, and a Kerberos mechanism. The following text and

Figure 5.5 describe the operation of the OpenSSL based authentication mechanism. However, independent of an implementation, the operation of each of these mechanisms is largely similar.

The mechanism is created and initialized as directed by the evaluated local policy (a). Upon reception of the `EVT_AUTH_REQ` event, the OpenSSL mechanism initiates communication by establishing a mutually authenticated secure channel. The means by which the authentication service is identified is external to Antigone (it is currently implemented by the broadcasting of a locator message to the group). The authentication service responds with an address and port to which the requestor may connect (b). However, other implementations are free to use other mechanisms (anycast, expanding ring searches, session announcements, etc.).

The certificate used to prove authenticity of the local entity is explicitly stated in the local policy through a configuration parameter. The associated certificate file is read from the local disk and passed to OpenSSL (c). The SSL implementation performs the handshake protocol [Gro00], which receives an authenticated public key certificate for the service (d). The certificate is translated into an Antigone credential, and provided to the policy engine for evaluation of the *group_auth* action (e). A positive result signals that the local policy states the certificate is sufficient to prove the authenticity of the authentication service. The authentication request is aborted on a negative result.

If the service authentication is successful, the authentication mechanism obtains the policy instantiation, a join nonce, and a group public key[3], and to establish a pair-key. This is accomplished through a single request-response exchange over the OpenSSL connection. The local member creates and transmits a `pair key` (for a configured algorithm), and the server responds with the nonce, group public key, and policy instantiation (f). The SSL connection is closed, and the local entity places nonce and group public key in the (mechanism) group attribute set. An `EVT_POL_RCVD` event containing the instantiation is posted to the event controller. The mechanism signals the completion of the authentication process by posting an `EVT_AUTH_COM` (h) event.

The authentication service performs the server side of the exchange. The *member_auth* action is evaluated upon completion of the OpenSSL connection establishment. A positive

[3]The group public/private key pair is generated by the initiator during initialization of some centralized groups. The private key is later used to guarantee the (source) authenticity of broadcast data (e.g., rekey messages, failure detection messages).

evaluation signals that the client side certificate is sufficient to prove rights to access the instantiation, and the exchange is completed as described. The pair key is placed in the authentication service's attribute set.

Note that a number of error conditions can occur during the authentication process. A retry timer is registered when the authentication mechanism begins initialization (the length of which is defined through a configuration parameter). Any exchange not completing prior to expiration is retried and a retry count incremented. If the (configurable) maximum retry count is reached, a fatal error is generated and the authentication is aborted. Similarly, any denial of a *group_auth* or *member_auth* action fatally errors the authentication attempt.

5.4.2 Membership Mechanisms

Membership mechanisms provide facilities for a previously authenticated member to join and leave the group, to request the ejection of other group members, and for the distribution of group membership lists. The original Antigone implemented these facilities in the Join, Leave, and Rekey/Group Membership mechanisms. However, it was found that the separation of these membership tasks among different mechanisms limited flexibility; modification of membership services required changes across several mechanisms. This conflicted with the component philosophy, and hence led to the new structure. Antigone currently implements a single membership mechanism (the Antigone membership mechanism).

The membership mechanism implements both client (member joining group) and server (admittance entity) services. The client implementation initiates the join protocol in response to the `EVT_JOIN_REQ` event posted by the group interface in response to the `EVT_AUTH_COM` event. The client simultaneously registers a join retry timer and sends a join request message to the admittance entity. The completion of the join is signaled by a key management mechanism through the `EVT_NEW_GRUP` event, after which all timers are unregistered.

While in general any cryptographic material may be used to prove the authenticity of the joining member, the current implementation uses the pair-key established by the authentication mechanism. Some environments may desire to separate the authentication of members from the join process [BHHW01]. Hence, other implementations may require a second authentication protocol to join the group.

The admittance entity, through the policy engine, evaluates the *join* action upon recep-

tion of the join request. If the action is permitted, a join accept message is broadcast to the group, and a join reject otherwise. The admittance entity posts an `EVT_JOIN_MEM` if the member was not previously in the group, and an `EVT_REJN_MEM` if the member is currently in the group. The latter event signals that the joining member's state is stale and should be refreshed.

The `EVT_REQ_LEAV` event signals that the local member desires to leave the group. The membership mechanism broadcasts a member leave message and posts `EVT_MEM_LEAV` event. As configured by policy, the local member may or may not wait for a leave response before exiting.

`EVT_EJCT_REQ` events signal that the local entity wishes to eject another member. The event data identifies the member to be ejected. The associated text identifier is placed in the ejection request message broadcast to the group. The ejection is either accepted or denied by the admittance entity, the result being reported in the ejection response message. The positive or negative result of the ejection is reported through the `EVT_ECJT_COM` event. The admittance entity restricts access to the ejection through the evaluation *eject* action. Based on configured policy, eject requests may either be encrypted using the pair key or digitally signed. In the latter case, the certificate itself is included with the request. Hence, the right to eject members can be explicitly granted through an issued certificate.

Policy determines when membership lists are distributed. For example, the current membership mechanism supports policies for none, best-effort, positive, negative, or perfect membership. Based on the policy, a sequenced and signed (with the group private key) membership list is distributed following every member leave (`EVT_MEM_LEAV` and `EVT_PRC_FAIL` events) (positive), every member join (`EVT_JOIN_MEM`) (negative), or on all membership events (perfect). In all cases except a none policy, the membership list is broadcast to the group periodically. Members failing to receive membership lists can request the membership list via the membership request message.

Membership lists contain two sequence numbers. The *interval identifier* states the current interval (which increases by one per configurable quanta). The *view identifier* sequences the membership changes between intervals. Because the intervals are fixed, the accuracy of membership information is bounded by the configured announcement periodicity (quanta). The current interval and view sequence numbers are reported to a joining member during the join request/response protocol.

5.4.3 Key Management Mechanisms

Key management mechanisms are used to establish and replace the ephemeral keys used to secure the group. While Antigone currently implements a Key Encrypting Key (KEK) [HM97b], Authenticated Group Key Management (AGKM, see below), and Logical Key Hierarchy (LKH) [WHA98, WGL98] key management mechanisms, others are possible. For example, the current interface can be used to implement participatory key management (e.g., Cliques [AST00]). The following assumes that the KEK mechanism managing a single symmetric session key is used to secure the group. Key management implements two distinct operations; key distribution and rekey management.

The group key controller (GKC) creates a key encrypting key and a traffic encrypting key (TEK) for the configured cryptographic algorithm upon reception of the `EVT_NGRP_POL` event. A key distribution message encrypted with the member pair-key and containing the KEK, TEK, and a *group identifier* is subsequently sent to a joining member in response to the reception of each `EVT_JOIN_MEM` or `EVT_REJN_MEM` event. A member receiving the message places the KEK and TEK in the local attribute set and posts an `EVT_NEW_GRUP` event signaling that the join has been completed.

The group identifier uniquely identifies the session context. A group identifier is the concatenation of a text identifier and nonce value. The text identifier is an eight byte, null terminated name string that uniquely identifies the session. The nonce is a four byte nonce value. The group identifier is used by all mechanisms to identify under which context (e.g., key) a message was sent. The nonce is incremented by one each time the group is rekeyed.

Policy determines when the group is rekeyed. Similar to membership management, the group rekeying is defined over time, leave, join, and membership sensitive policies. These policies indicate that the group is rekeyed periodically, after member leaves, joins, or all membership events, respectively. However, only time sensitive policies are meaningful in KEK based schemes. KEK mechanisms are required to be time-sensitive. Hence, a timer is created with a configured period at initialization. A group rekey message containing a new TEK encrypted with the KEK is distributed following each timer expiration. Clients receiving a group rekey message install the new group identifier and TEK, and post an `EVT_NEW_GRUP` as described above.

`EVT_KDST_DRP` events signals that the local member has missed a rekey message. These events are posted by any mechanism receiving a message containing a group identifier for

which a corresponding key has not been received. A naive implementation would simply immediately transmit a key request. However, message loss caused by network congestion may be exacerbated by the simultaneous generation of retransmit requests by many members. Known as *sender implosion*, this problem is likely to limit the efficiency of key distribution in large groups or on lossy networks. A retransmit mechanism similar to SRM [FJL+97] addressing this limitation is used. The member sets a random timer before sending a key request message. If another key request is received prior to expiration of the timer, the request is suppressed. The GKC retransmits the last rekey message upon reception of a key request message.

Authenticated Group Key Management (AKGM)

The *Authenticated Group Key Management* (AGKM) mechanism implements a variant of KEK key management. With respect to Antigone, it processes signals as described above. However, TEKs are calculated rather than distributed. Hence, because any member receiving seeding data can calculate session keys, much of the complexity associated with key management can be avoided.

Described in Figure 5.6, AGKM provides a sequence of authenticated session keys preserving the advantages of KEK-based solutions. This approach provides session key independence; knowledge of a session key provides no information with which other session keys can be determined (without inverting $h()$). This approach also suffers from some of the disadvantages of KEK-based key management; it is not possible to eject members without replacing all keying material.

AGKM offers several advantages over traditional KEK approaches. Every key is authentic; proof of its origin can be obtained from the authenticator values. Moreover, membership forward secrecy is guaranteed only by distributing the most recent seed values to a joining member. Because a malicious member of the group cannot generate validation information for future session keys, new keys can only be released by the group key controller. Hence, a member can only use those keys that have been released.

An alternate use of AGKM places a header containing the current validator information on each transmitted message. Members receiving any message are able to directly calculate the session key. Hence, much of the cost of explicit key distribution is avoided. This approach is useful in large groups (e.g., as one might find in large scale multimedia appli-

Configuration Parameters		Initial Values Generated by the GKC	
l	length of key chain	k_0	random key seed of size $\|h()\|$
$h()$	collision resistant hash function	v_0	random authenticator seed of size $\|h()\|$
		g^+, g^-	group public key pair

Construction

$v_i = h^{l-i}(v_0)$	authenticator values
$k_i = h^i(k_0)$	key seed values
$SK_j = h(k_i \oplus v_i)$	session key

1. The session keys are used in index order (e.g., SK_0, SK_1, ..., SK_p). Hence, the session key SK_i is valid only during the interval i, and is replaced periodically (see Section 5.4.3) through the rekeying process.

2. A member joining during interval i receives the i, v_i, k_i, and g^+. These values are transmitted under a pair key known only to the GKC and the joining member.

3. The group is rekeyed by incrementing i and transmitting i and v_i (in cleartext). When the values of v_i are exhausted (e.g., $i = p$), a reseed message is broadcast to the group. The reseed message has the following structure;

   $$\{\overline{v_p}, \{0, k_0, v_0\}_{\overline{k_p}}\}_{SIG(g^-)}$$

   Where $\overline{v_p}$ and $\overline{k_p}$ are the last validator and key seed values from the previous chains, and $SIG(g^-)$ is a digital signature generated using the group private key g^-. The new values of k and v are used to seed the new key chain.

4. Any member who does not receive a rekey or reseed value can request it directly from the GCK. However, the GCK will never broadcast past values of k or v (e.g., 0, k_0, and v_0 are replaced with the current interval values – i,k_i, and v_i). In all cases, the session key is calculated from the header information and known values of v and k.

5. Any v^i can be authenticated by evaluating the truth of the expression $v_{i-1} = h(v^i)$. More generally, any member who has received the key distribution data for some index $j < i$ can validate v_i by applying $h()$ the appropriate number of times to the initially authenticated (via digital signature) v_j.

Figure 5.6: The AGKM construction - members are distributed seed information from which session keys are calculated. Session keys can only be calculated after authenticating information is disclosed by the GKC.

cations); providing reliability for rekeying information can lead to *sender implosion*. The cost of this construction is header size; assuming a 16 byte hash function, AGKM requires 18 bytes of header per message.

AGKM is not the first implicit key management approach. The NARKS [BF99] and MARKS [Bri99] systems use a seeded hierarchy to implement implicit key management for pay-per-view video. However, AGKM's construction is significantly less costly. Receivers in NARKS and MARKS must maintain state that grows logarithmically with the number

of session keys supported (as opposed to the constant amount of state required by AGKM).

5.4.4 Data Handling Mechanisms

The data handling mechanism provides facilities for the secure transmission of application level messages. The security guarantees provided by the current Antigone data handler mechanism include: *confidentiality*, *integrity*, *group authenticity*, and *sender authenticity*. The mechanism is configured to provide zero or more of these properties. A data transform is defined for each unique combination of properties.

Upon reception of an `EVT_SEND_MSG` event, the data handler evaluates the *send* action via the policy engine. This tests whether the local member has the proper credentials to send a message. If a positive result is returned, the data handler mechanism performs the appropriate transform and broadcasts the data via the broadcast transport layer. Once the message is sent, an `EVT_SENT_MSG` event is posted to the event queue.

The mechanism receiving the message performs the reverse transform and evaluates the *send* action (using the context supplied in the message rather than local credentials). If a positive result is returned, an `EVT_DAT_RECV` event identifying the received data is posted. Note that a received message may require a session key that the local member has not yet received (or never will). In this case, recovery is initiated by the posting of an `EVT_KDST_DRP` event. The key management mechanism is required to recover by attempting to acquire the key. Messages associated with unknown keys are dropped.

Confidentiality is achieved by encrypting the application data under the session key. The algorithm used for encryption is defined by a policy. Note that the key management and data handling mechanisms must be configured to use compatible cryptographic algorithms. This is stated as a policy requirement in Ismene policies through assertions (see Chapter 4).

Integrity is achieved through *Keyed Message Authentication Codes* (HMAC) [KBC97]. To simplify, an HMAC is generated by XORing a hash of the message with the session key. A receiver determines the validity of an HMAC by decrypting and verifying the hash value. If the hash is correct, the receiver is assured that the message has not been modified in transit by an adversary external to the group. Group authenticity is a byproduct of integrity. Two constructions supporting source authentication are currently available. In either construction, the content is accepted if the message and authenticating information is properly formed and the *content_auth* action evaluates successfully.

The *packet signing* source authentication construction implements source authentication through digital signature [DH76]. The signature is generated using the private key exponent associated with the sender's certificate. Receivers obtain the sender's certificate and verify the signature using the associated public key.

Due to the computational costs of public key cryptography, the use of per-message digital signatures to achieve sender authenticity is infeasible in high throughput groups. Several efforts have identified ways in which these costs may be mitigated [EGM96, GR97, WL98]. While the speed of these algorithms is often superior to strictly on-line signature solutions, their bandwidth costs make them infeasible in high throughput groups.

The *online* construction implements source authentication through a custom variant of Gennaro and Rohatgi online signatures [GR97]. In this approach, outgoing data is buffered for a configurable period. A online digital signature is applied when a configurable threshold of data (called a frame) is buffered or the period expires. The signature performs a forward chaining approach in which a packet signs (contains a hash) of the immediate succeeding packet. The first packet is digitally signed, all data is transmitted to the group. Receivers can validate the first and subsequent packets as they arrive. However, all packets following a lost packet are dropped; the chained signature is broken (however the mechanism is resilient to packet re-ordering). This makes this approach inappropriate for networks with significant packet loss. An investigation of the feasibility of both the online and packet signing is presented in Chapter 7.

5.4.5 Failure Detection and Recovery Mechanisms

Failure detection mechanisms provide facilities for the detection and recovery of process or communication failures. The current *chained failure detection* (CFD) mechanism detects crash failed processes. However, other mechanisms (e.g., partition detection and recovery [DM96]) may be integrated through the event interfaces. The remainder of this section assumes a CFD failure detection mechanism.

An application's threat model may require that the system tolerate attacks in which an adversary prevents delivery of rekeying material. Thus, without proper failure detection, members who do not receive the most recent session information will continue to transmit under a defunct session key. Additionally, the accuracy of membership information is in part determined by the ability of the session leader to detect failed processes. Thus, in

support of the other guarantees, the goal of CFD is to determine *a)* which members are operating, and *b)* that each process has the most recent group state (session keys and group view).

Failure detection in CFD is symmetric. The failure monitor is a centralized service monitoring all current members of the group. Conversely, each member monitors the group via communication with the failure monitor. As dictated by policy, members who are deemed failed are removed from the group. A member detecting the failure of the monitor assume the group has failed and attempts recovery. If recovery fails, the member notifies the application and errors the communication. Each member and the failure monitor periodically transmit heartbeat messages. CFD detects failed processes through the absence of correct heartbeats. If a policy stated threshold of contiguous heartbeats is not received, the member or monitor is assumed failed. The current group context (e.g., group and view identifiers) is included in each heartbeat. Hence, heartbeats are used to detect when current group state is stale.

The CFD mechanism creates a heartbeat transmission timer during initialization. A heartbeat is transmitted and the timer reset at its expiration. Members associate a timer with the failure monitor after being admitted to the group (e.g., on a `EVT_JOIN_COM` event). If no valid failure monitor heartbeat is received before the expiration of this timer, the group failure is signaled through the `EVT_GROP_LST` event.

Upon reception of an `EVT_JOIN_MEM`, the failure monitor creates a timer for the joining entity (this timer is reset on an `EVT_REJN_MEM` event). The timer is reset on valid heartbeats received from the member. If the timer expires, then the member is assumed to have failed and an `EVT_PRC_FAIL` event is posted. Member timers are deactivated on `EVT_MEM_LEAV` events, and all timers are reset on `EVT_NEW_GRUP` events.

A member detecting a stale state or lost heartbeat messages can initiate recovery by sending a client recovery message. This message indicates to the session leader that the member requires the most recent group state. The reception of this message triggers an `EVT_KDST_DRP` event at the failure monitor, which ultimately leads to the re-distribution of the current session state.

Chained Failure Detection

CFD uses *secure heartbeats* to detect failed or disconnected processes. The presence of a sequence number in the heartbeat ensures that it is fresh. Group and view identifiers are used to verify that the sending process has the most recent state.

With respect to group members, the goal of the current failure protection mechanism is the reliable detection of a session leader's failure, not its recovery. As needed, additional mechanisms can be introduced in the future that implement recovery algorithms using primary backup, replication, or voting protocols to establish a new failure monitor, group key controller, and admittance entity.

Hash chains [Lam81] are used to amortize the cost of heartbeat generation over many messages[4]. A hash chain is the sequence of values resulting from the repeated application of a secure hash function (f) on some initial value. For example, given an initial value x and chain of length $k+1$, the hash chain is: $\{f^0(x) = x, f^1(x), f^2(x), \ldots, f^k(x)\}$. Because, by definition, (even partial) inversion of f is not feasible, knowledge of $f^i(x)$ gives no meaningful information to derive $f^{i-1}(x)$, for some $i, 0 < i < k$. By revealing $f^k(x)$ securely, the remaining values can be used in reverse order as proof of the knowledge of x. This is useful in authentication schemes (one-time passwords) because only a person who has knowledge of x can generate the intermediate values.

Heartbeats are generated as follows. Initially, a sending process A generates a random value x of length equal to the output of the hash function (e.g., MD5 has a 128 bit output). Freshness is asserted through monotonically increasing heartbeat sequence numbers (S_A^i), the first of which is selected at random (S_A^0). A applies f k times to generate the following hash chain (where f and k are configured through configuration parameters):

$$\delta^0 = x, \delta^1 = f(x), \delta^2 = f^2(x), \ldots, \delta^k = f^k(x)$$

A generates a *heartbeat validation block* containing the group identifier g, her identity A, the first heartbeat sequence number S_A^0, the last value in the hash chain $f^k(x) = \delta^k$, and an HMAC generated using the pair key of A, PK_A:

$$g, A, S_A^0, \delta^k, \{H(g, A, S_A^0, \delta^k)\}_{PK_A}$$

[4]This use of hash chains is similar to those found in one-time password authentication systems [Hal94, Rub96].

A heartbeat message is generated by concatenating the current sequence number ($S_A^i = S_A^0 + i$) and the next value in the hash chain (in reverse order, δ^{k-i}) with the validation block. Because encryption is only required when creating the validation block and the hash chain itself is cached, heartbeat generation is fast. When the values of a chain are exhausted ($i > k$) or the session is rekeyed, the member generates a new hash chain and the associated validation block. The failure monitor performs the same construction, save the HMAC is replaced with a digital signature calculated under the group private key.

Heartbeats are validated by checking the HMAC (or signature) and testing the relation:

$$f^{S_A^i - S_A^0}(\delta^{k-i}) = \delta^k.$$

If the relation holds, then the heartbeat is valid. The heartbeat is authentic because of the use of the pair key (or group private key) in the validation block generation. The heartbeat is fresh because of the presence of the next value in the hash chain. After receiving and validating the initial heartbeat for a hash chain, subsequent validation can be achieved by byte comparison of a validation block of a previously validated heartbeat. Thus, heartbeat validation is fast.

5.4.6 Debugging Mechanisms

Debugging mechanisms are used to view the internal state of the group through the observation of events. Depicted in Figure 5.7, the currently implemented Antigone Scope mechanism logs the progress of the group and records the throughput and latencies characteristics of the application content. Which information is recorded is defined by the policy instantiation. The scope mechanism does not currently post events, but only passively observes events posted to the event bus. As a result, one can debug event processing by analyzing the type, data, and ordering of posted events.

The specialized `EVT_INFO_MSG` is used by mechanisms to post information to debugging mechanism. This event specifies a single string containing some information of import to the mechanism, and is frequently used to indicate state changes not reported through events.

5.5 Broadcast Transport Layer

Multicast services have yet to become globally available. As such, dependence on multicast would likely limit the usefulness of Antigone. Through the broadcast transport layer,

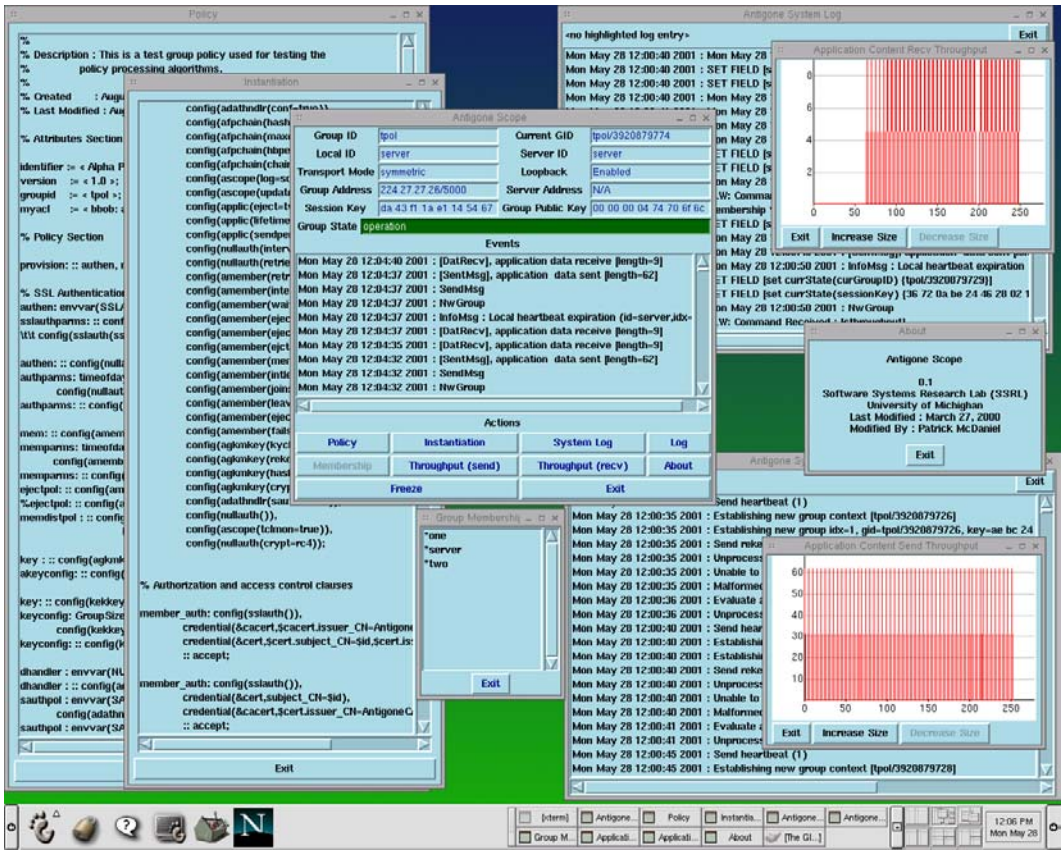


Figure 5.7: Antigone Scope Interface - the scope mechanism records and displays all state changes signaled via the event bus.

Antigone implements a single group communication abstraction supporting environments with varying network resources. Applications identify at run time the level of multicast supported by the network infrastructure. This specification, called a *broadcast transport mode*, is subsequently used to direct the delivery of group messages. The broadcast transport layer implements three transport modes; *symmetric multicast*, *point-to-point*, and *asymmetric multicast*.

The symmetric multicast mode uses multicast to deliver all messages. Applications using this mode assume complete, bi-directional multicast connectivity between group members. In effect, there is no logical difference between this mode and direct multicast.

The point-to-point transport mode emulates a multicast group using point-to-point communication. All messages intended for the group are unicast to the session leader, and relayed to group members via UDP/IP [Pos80]. As each message is transmitted by the session leader to members independently, bandwidth costs increase linearly with group size. This approach represents a simplified Overlay Network, where broadcast channels are emulated

over point to point communication. Note that a number of techniques can be used to vastly reduce the costs our implementation [CRZ00, JGJ+00].

In [AAC+99], the experiences with the deployment of the *Secure Distributed Virtual Conferencing* (SDVC) application are reported. SDVC is a video-conferencing application based on an early version of Antigone (LSGC [MHP98]). The deployed system was to securely transmit video and audio of the September 1998 Internet 2 Member Meeting using a symmetric multicast service. The receivers (group members) were distributed at various cites across the United States. While some of the receivers were able to gain access to the video stream, others were not. It was determined that the network could deliver multicast packets towards the receivers (group members), but multicast traffic in the reverse direction was not consistently available (towards the source). The lack of bi-directional connectivity was attributed to limitations of the reverse routing of multicast packets [AAC+99].

The limited availability of bi-directional multicast on the Internet coupled with the costs of point-to-point multicast emulation lead to the design of the *asymmetric multicast* mode. This mode allows for messages emanating from the session leader to be multicast, and all other messages to be relayed through the session leader via unicast. Members unicast each group message directly to the session leader, and the session leader retransmits the message to the group via multicast. Thus, the costs associated with point-to-point groups are reduced to a unicast followed by a multicast. The increasing popularity of single source multicast [Gro01] make this a likely candidate for future use.

5.6 Optimizing Policy Enforcement

The architecture described throughout differs significantly from the initial Antigone design [MPH99]. Several lessons learned from the initial architecture drove the design of the modified Antigone. First, the introduction of a formal policy language required fundamental changes in the way in which policy is enforced (e.g., through repeated evaluation of authentication and access control policy). Secondly, any flexible policy infrastructure should provide simple, yet powerful, interfaces for implementing the many required protocols. Finally, care must be taken in designing efficient structures upon which policy is enforced. The following describe several optimizations addressing these design considerations.

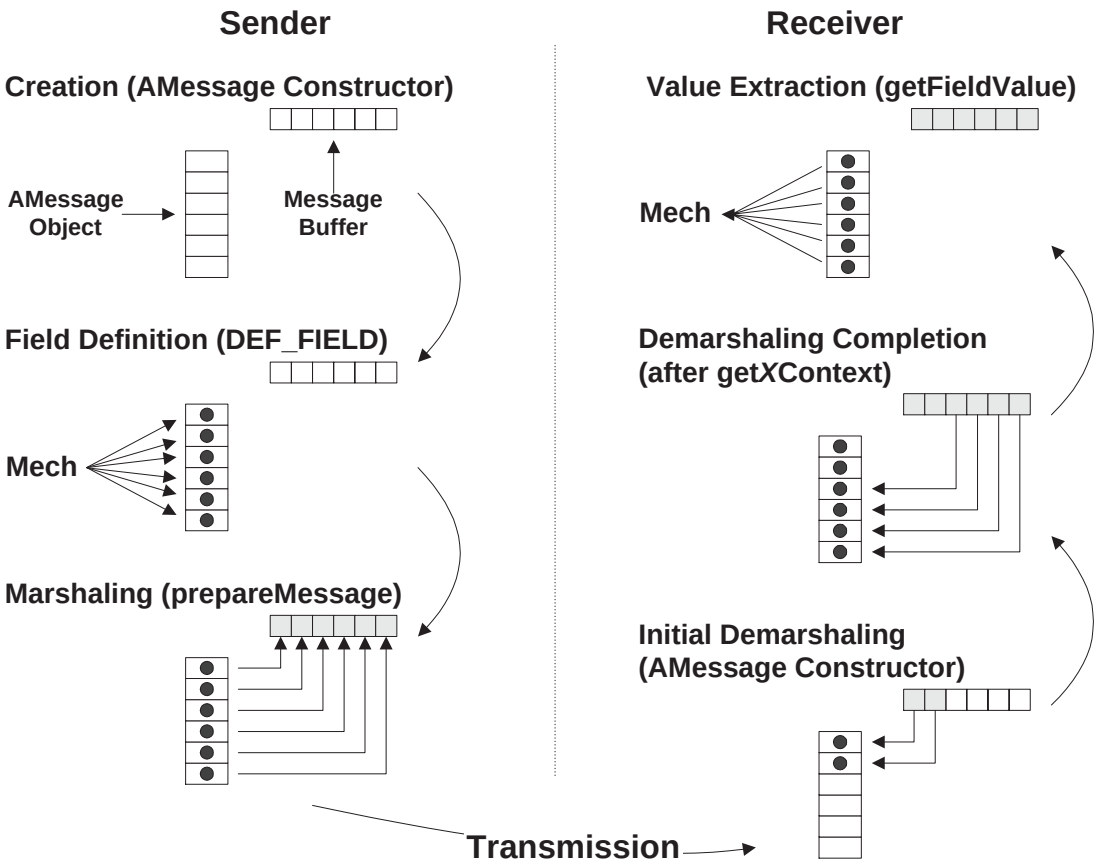


Figure 5.8: Generalized Message Handling (GMH) - GMH abstracts the complex tasks of data marshaling. Senders associate data with each field defined in a (*AMessageDef*) message template object. GMH marshals the data as directed by the template using the supplied information. Receivers reverse the process by supplying additional context (such as decryption keys) based on previously unmarshaled fields. In the figure, shaded boxes represent marshaled or unmarshaled data (at the sender and receiver respectively), and dots represent known field values.

5.6.1 Generalized Message Handling

By definition, a flexible policy enforcement architecture must implement a large number of protocols, messages, and data transforms. However, correctly implementing these features requires the careful construction of marshaling code. Marshaling is widely accepted as a difficult, time consuming, and error prone process. This belief was reinforced by difficulties encountered while developing and debugging the first version of Antigone.

The *Generalized Message Handling* (GMH) service is designed to address the difficulties of protocol development. This service abstracts marshaling by allowing the flexible definition of message formats. GMH uses this information in conjunction with user sup-

plied context to marshal data. Encryption, padding, byte ordering, byte alignment, and buffer allocation and resizing are handled automatically by GMH. Hence, the development costs associated with implementing new protocols are reduced and bugs associated with marshaling are largely eliminated.

An `AMessageDef` object defines the structure of a message. Typically, a static `AMessageDef` object is defined for each message type implemented by a mechanism. For example, the ADataHandler mechanism defines a definition object for each unique combination of data security policies (e.g., confidentiality, integrity, etc.). The central attribute of each message definition object is the `msgDef` string. This alphanumeric string defines the typed ordering and encapsulation of fields. For example, the following defines a simplified key distribution message:

$$msgDef = \text{“}\texttt{LTH[H[E[DT]]}''$$

Each alphanumeric character in the definition represents a field (data fields) or operation spanning fields (encapsulation fields). The latter field types identify the scope of operations using bracket symbols. In the above example the characters `L`, `T`, and `D` represent a long integer (group identifier), string (identity), and data block (key). The symbols `H`[...] and `E`[...] represent HMAC and encryption operations.

Described in Figure 5.8, a message is marshaled in three steps. First, an `AMessage` object is constructed as directed by the associated `AMessageDef` object. Next, the values for each field are assigned through the type checking `DEF_FIELD` macro (indexed by field number). Data fields are passed the values to place in the message. Encapsulation fields are passed `Key` objects (for encryption) or `Key` and `HashFunction` objects (for HMACs). Once all field values have been assigned, the `prepareMessage()` method is called to perform the marshaling. The marshaled data is accessed through the `MsgBuf` access method after the `prepareMessage()` method returns. This buffer is used to transmit the marshaled message to the group.

Upon reception of the message, receivers reverse this process through an `AMessage` constructor accepting the received buffer. There may not enough information at the time of reception to completely unmarshal the message. For example, the parent mechanism may not know *a priori* the key that was used to encrypt a message. Hence, the mechanism must determine the context under which a message was sent. The GMH service unmarshals as much data as is possible, and calls the `getEncryptionContext()` or `getHMACContext()`

method on the mechanism object. The mechanism can call `getFieldValue()` on every field that has been unmarshaled within the context method. Fields values are used to determine the appropriate context, and the appropriate keys and algorithms are reported to GMH based on this information. Once the constructor completes, all fields values may be accessed through the `getFieldValue()` method on the message object.

5.6.2 Caching Authentication and Access Control

Authentication and access control policy is consulted on every regulated action. Some actions are undertaken frequently. For example, a video conferencing application may send many packets per second. Thus, evaluating policy prior to the transmission of every packet may negatively affect performance.

Antigone provides a two level cache for authentication and access control. The first level cache stores the result of *condition evaluation*. As described in Chapter 4, the right to perform an action may be predicated on measurable state. The measurement of state is tested using special purpose functions implemented by mechanisms, the group interface, or the application itself through the `PolicyImplementor` API. This API requires that each condition evaluation return not only the positive or negative evaluation of the condition, but must indicate the period during which the result should be considered valid. There are three indicators associated with the reported period; *transient*, *timed*, and *invariant*. Transient results should be considered valid for only the current evaluation. Timed results explicitly state a discrete period during which the result should be considered valid. Invariant results are considered valid for the lifetime of the session. The cache is consulted during the evaluation of any authentication and access control policy.

A second level cache stores the results of *policy evaluation*. This cache stores the relevant context under which an action was considered (e.g., credentials and conditions used during evaluation). Entries in the cache are considered valid for the minimum of the reported condition evaluations. Hence, any member testing the same conditions and credentials (as would be the case in frequently undertaken actions) would simply access a cached result. Both caches are flushed following policy evolution.

5.6.3 Memory Management

The cost of acquiring and releasing frequently used objects via the C++ `new` and `delete` calls significantly contributed to the message processing overheads of the initial Antigone implementation. For example, the management of `Buffer` objects are used to store messages, keys, and identities negatively impacted system performance. These costs were exacerbated by requirements for additional processing prior to object deletion. It is unsafe for Antigone to release buffers to the free store without zeroing their contents. Zeroing prevents the exposure of potentially sensitive information to other processes acquiring the previously released memory [Pro00]. While some implementations of C++ automatically initialize (zero) allocated data, other methods do not (e.g., `malloc` [Pagb]).

Antigone mitigates the costs of free store management and memory zeroing by creating internal heaps for the most frequently used objects (e.g., typically known as slab allocation [Bon94]. The `Buffer` and `String` classes define a stack of `IntlBuf` objects as static data. `IntlBuf` objects are resizable memory blocks used to implement both buffers and strings. The buffer and string allocation methods (as called through either the free store `new` or local variable stack allocation) consult the class specific `IntlBuf` stack for available objects. If such an object exists, it is associated with the object being constructed. If no such object is available, a new `IntlBuf` object is created. Buffer and string destructors release these objects back onto the appropriate stack. Note that the internal heaps are shared by all objects in the same address space. The `Buffer` class implements a service similar to the x-kernel *buffer manager* [HMPT89] through the manipulation of attributes and use of internal heaps.

Each stack will grow to a high-water mark. Once reached, no further memory allocation for the objects is needed. Hosts with limited resources may wish to place an upper bound on the size of these stacks, and garbage collect when the bound is exceeded. This is accomplished by assigning the `Buffer::maxBufferHeapBytes` and `String::maxStringHeapBytes` static attributes to the desired threshold (measured in bytes). Any deallocation that causes the stack to exceed the threshold is handled by zeroing and freeing the buffer directly. In all cases, the stacks are flushed (and buffers zeroed and released) at process termination.

CHAPTER 6

CASE STUDIES: VIRTUAL PRIVATE FILESYSTEMS IN AMIRD

The increasingly distributed nature of computing has heightened concerns over the security of services used to share information. Software has historically been built upon *ad hoc* collections of security mechanisms. The resulting designs address singular views of performance and security. Hence, the resulting service is inappropriate for environments with differing requirements. Users must accept performance penalties (where unnecessary security infrastructure is provided) or highly undesirable vulnerabilities (where required security is not provided).

This chapter presents the AMirD filesystem replication system. AMirD is used to efficiently and securely replicate filesystems across agents. The Ismene policies defining and driving the security services used to protect replication are constructed from run-time conditions. These conditions can be a reflection of participant abilities and requirements, the available resources, or any other measurable aspect of the environment. Hence, AMirD determines an appropriate security model at run-time through the evaluation of policy. This approach allows network administrators and users, rather than developers, to construct and control the security infrastructure via the systematic enforcement of policy.

This chapter serves not only as an investigation of group-based filesystem replication, but also as an exploration of the power of policy-based security. The four environments discussed in the latter sections of this chapter represent distinct operating environments, Each environment embodies a different set of security and performance requirements. The presented policies demonstrate how the vastly different needs of each environment can be met through a *flexible policy representation*. Ismene not only defines a set of security requirements, but also defines under what conditions each should be considered relevant.

Independent of security, the use of broadcast communication can significantly reduce the

costs associated with content replication [JGJ+00]. For example, the semantics of AMirD make it ideal for mirroring (e.g., website, source code repositories). AMirD *filesystems* are replicated at specified intervals or on demand. Collections of mirror sites receive the scheduled broadcast updates in unison. Hence, updates can be scheduled to occur only during periods of low use or during maintenance windows. Another scenario investigates the use of mirroring in mobile environments. Mobile users have intermittent connectivity with highly variable throughput. Thus, these users can refresh the local filesystem only when it is convenient and desirable. Other environments (local LAN, coalition) implement replication within one or more enterprises. These latter scenarios principally illustrate the need for reconciliation of diverse security policies.

The remainder of this chapter is organized as follows. Section 6.1 describes the design of AMirD. Section 6.2 describes policies appropriate for a number of operating environments. Section 6.3 concludes with details on the configuration and management of AMirD agents.

6.1 AMirD

AMirD is a filesystem replication service based on secure group communication. An AMirD agent is placed at each host participating in the distribution of files. The contents of each filesystem are periodically identified and files updated at each agent as necessary. AMirD is not a distributed filesystem in the traditional sense. No formal read/write semantics are provided [Tan95]. Hence, the replication service avoids the inherent complexities and costs associated with distributed filesystems (e.g., file locking). AMirD is focused on the reliable and secure distribution of files across large and widely distributed communities. However, as needed in the future, the design can be extended to provide stronger consistency guarantees.

AMirD separates advertisement from distribution. As a result, consistency management and content distribution can operate under different security models. This differs from previous replication services that enforce a fixed security (if any) over all communication. This affords a greater degree of flexibility in meeting the needs of interested parties.

An *AMirD filesystem* is a securely replicated directory tree. Each filesystem is identified by its local root pathname on the *exporter* host. An exporter subsequently directs both the identification (announcement) and subsequent distribution (download) of files within the exported filesystem. Filesystem contents are communicated through periodic announcements.

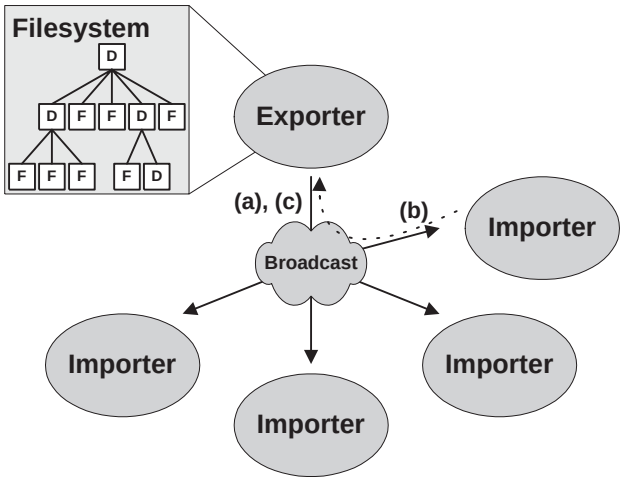


Figure 6.1: The Control Group - filesystem announcements are broadcast by *exporters* to the control group at a policy dictated periodicity (a). *Importers* noting missing or stale content request updates via download requests (b). Exporters identify the location of download groups (associated with a single file download) through a download group announcement (c).

Importers receiving announcements request downloads for each locally stale or non-existent file. The files are reliably broadcast to all interested and authorized importers under a run-time-generated security policy. Note that any agent can simultaneously act as the importer or exporter for zero or more filesystems.

The *control group* is used for the distribution of filesystem announcements and download requests. *Download groups* are used to reliably broadcast the contents of files. The following subsections summarize the operation and use of these groups within AMirD.

6.1.1 Control Group

The control group is created by an *initiator*. The initiator creates a policy instantiation through the reconciliation of a group policy and the local policies of each expected exporter and importer (see Chapter 4)[1]. The policy instantiation is used to initialize the appropriate services and establish the group context. Interested importers and exporters attempt to gain access to the group. Entities that are successfully authenticated (and permitted access by the authentication and access control policy) are admitted to the group.

Any number of exporters and importers can exist within a single control group. However,

[1]The means by which group and local policies are acquired is outside the scope of AMirD. It is currently assumed that all such policies are made available to the initiator prior to the creation of the group. However, future revisions may provide an additional service from which these policies may be acquired.

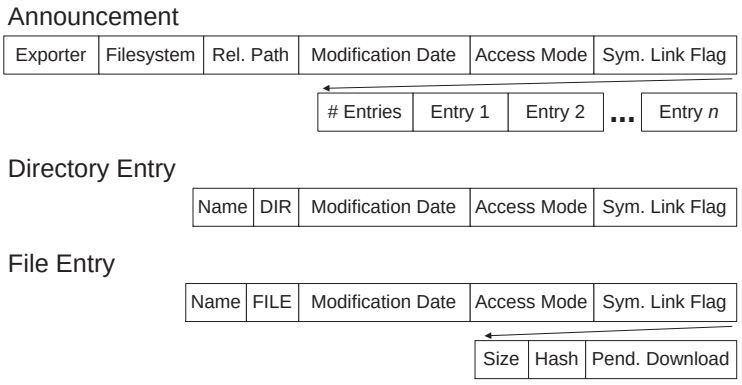


Figure 6.2: Filesystem Announcement - AMirD filesystem content announcements are fragmented into per-directory sub-announcements. Each sub-announcement contains the pertinent information regarding the files and directories of the announcement directory.

it may be desirable to limit the number of filesystems governed by a control group. This is important where filesystem announcements are themselves sensitive, or where importers wish to the limit processing of announcements in which they are not interested. The current AMirD agent supports only a single control group. Hosts must execute a separate agent for each control group managing distinct sets of filesystems.

Depicted in Figure 6.1, exporters describe the contents of filesystems through periodic broadcasts. Export announcements are fragmented into sub-announcements identifying the directories in the filesystem. Depicted in Figure 6.2, sub-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 [Riv92a]). Announcements are tagged with the exported filesystem and a unique exporter identifier.

Importers compare received announcements against the local filesystem. Directories found to be inconsistent with an announcement are updated (or created) 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.

Importers indicate the need for updates through download requests. These single file requests are broadcast to the group. Note that in a naive approach, *sender implosion* can result from the many simultaneous request broadcasts resulting from an announcement. The sudden burst of requests can overwhelm the exporter, cause congestion, and ultimately

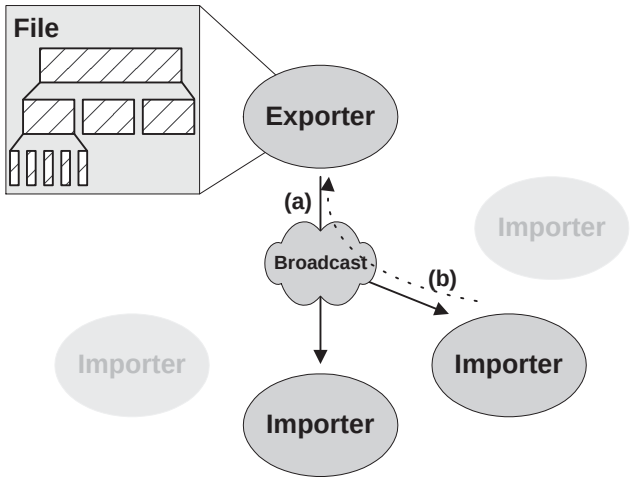


Figure 6.3: The Download Group - The file associated with the download group is broadcast through windowed protocol (a). Selective acknowledgments (b) received by the exporter are used to direct transmissions. The group is disbanded when the file transfer is complete.

delay updates. Similar to SRM [FJL+97], implosion is mitigated by randomly delaying requests (for 100-1000 milliseconds). Duplicates of requests received prior to the expiration of the delay timer are suppressed.

Non-redundant download requests are queued for processing by the exporter. Exporters initiate a download group for each requested file (in the order that the associated requests are queued, round-robin by filesystem). The exporter broadcasts a download group announcement at the point at which the group is initialized. This announcement contains file and network addressing information for the spawned download group.

6.1.2 Download Group

The exporter, acting as the initiator, creates a policy instantiation through the reconciliation of the filesystem specific download group policy and the local policies of the control group members. Consistency management and synchronization are separated through the enforcement of the distinct download policy. Hence, the security used to protect file content can be derived from the download file itself, the expected importers, or any other measurable aspect of the environment. Moreover, the authentication and access control policies used to govern the group can be used to place controls over the individual files or sub-trees within an exported filesystem (see Section 6.2.5).

Depicted in Figure 6.3, the download group reliably distributes the file to admitted im-

porters. The current download protocol ensures content reliability through a one-to-many *selective acknowledgment scheme* [FF96]. The use of other reliability techniques [Bir93, RBM96, FJL+97, RBH+98] 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 [Riz97] and SRM [FJL+97]) within Antigone is currently being investigated [MPI+01].

The exporter allows a configured period for importers to join, after which no further members are accepted into the group. The download is configured with window (w) and block (b) sizes. The configuration parameters are gleaned from the instantiation, or where not specified, default values are used (see Section 6.3.1).

The exporter begins the download by broadcasting w packets of size b containing the first blocks of the file. Upon reception of the w^{th} packet, each importer broadcasts a w-width bitmask acknowledgment. If an acknowledgment is not received from each importer within a specified interval, the exporter re-transmits the w^{th} packet. Re-transmissions are broadcast as directed by the importer bitmasks once all acknowledgments have been received (or the non-reporting import have been deemed failed and ejected from the download group). This process is repeated until the window is fully acknowledged. This window acknowledge protocol is repeated for each window. After completion of the download, the group is disbanded.

In general, reliable broadcast is an expensive enterprise. The period waiting for explicit acknowledgments represents a lost opportunity to transmit data. Furthermore, the efficiency of the group is governed by the slowest and highest loss rate importers. Hence, to make better use of the available resources, an exporter can initiate a policy driven number of simultaneous download groups.

6.2 Policy

This section considers the construction of AMirD policies addressing the requirements of four environments. The scenarios illustrate the use of policy to support secure replication under diverse conditions. The policies designed for these environments represent singular views of environmental requirements. A number of other interpretations may be entirely appropriate and realizable in AMirD. Such is the promise of policy defined behavior; al-

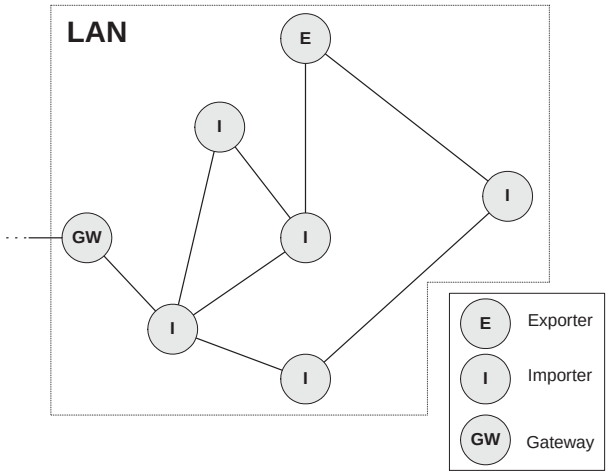


Figure 6.4: Scenario 1 - Members and services in this LAN environment are mutually trusted. Access to filesystem content is predicated on local filesystem access rights.

ternate interpretations leading to other application requirements can be addressed through flexible policy definition and enforcement.

As described in Chapter 2, group security can be decomposed into provisioning and authentication and access control. Provisioning policies can be further decomposed into the basic sub-policies for authentication, membership, key management, data handling, and failure detection. Note that while many security and system requirements can be met through the definition of policies along these dimensions, many others exist (e.g., reliability). These issues are largely orthogonal to the current work, and for brevity are omitted from the following discussion.

6.2.1 Scenario 1 - Local LAN

Local LAN environments characterized by this scenario exist within a single administrative domain. Depicted in Figure 6.4, an *enterprise internal network* supports users and services within a small geographic area (a single building). The network itself is protected with standard devices (i.e., firewalls, intrusion detection, etc.). The users within this community largely trust the local services and each other. However, users should have (read) access rights to the files prior to obtaining their content. AMirD provides (and suffers from the limitations of) a service similar to NFS [SGK+85] in this environment.

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 do not need protection. Similarly, requiring users to be authorized before being allowed to participate in the control group is not necessary. This is reflected in the minimal authentication, cleartext data handling policy. A simple key management service configured with a (near) infinite rekey period implements an essentially static session key. As users are largely trusted, membership information is not distributed.

The distribution of files within LAN should be protected, but users participating in the distribution trust each other not to interfere with the exporting of files. Thus, the costs of provided cryptographic guarantees such as integrity and authenticity of the files can be avoided. 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.

Access to exported files should be predicated on rights assigned by the local network administrators. As a local authentication service is almost certainly to be used for a range of existing services, it is highly desirable that this same service be used within the group. Once admitted, no further authentication is required. The current authentication mechanism transfers the UNIX user identifier (UID) during authentication, and are used to govern access to download groups. However, this approach has similar limitations as NFS; forging identifiers is trivial. Where needed, stronger mechanisms may be introduced.

6.2.2 Scenario 2 - Mobile Users

Mobile users[2] place a number of unique security and performance requirements on AMirD. These members exist in untrusted environments with often inconsistent and limited connectivity to their *home enterprise*. AMirD can be used in this context as a means by which remote users synchronize content between a mobile host and resources at the home enterprise. Depicted in Figure 6.5, the synchronization service is implemented by a gateway machine at the border of the home enterprise. Participants are expected to connect to the home enterprise for short periods during which 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

[2]This scenario defines a mobile user as a participant operating in remote, untrusted environments. The use of wireless technologies for communication is neither assumed nor precluded.

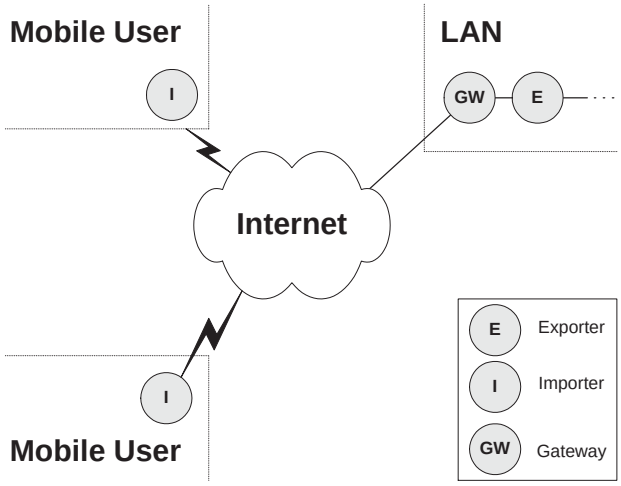


Figure 6.5: Scenario 2 - The mobile users in this environment use AMirD to synchronize mobile devices with filesystems in their home environment. As the device is executing in an untrusted network, the control and download information must be protected.

file content exposes little (such as email cache), confidentiality is not a concern. In all cases, the (group) 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 that low cost mechanisms for achieving these guarantees should be employed.

Similar to the control group, download groups are primarily driven by the sensitivity of the data being transmitted and the costs and availability of computational and network resources. 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 download group protocol parameters can be tuned for smaller groups.

Mobile users are first class 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 the relevant file read access. Where remote users are acting as exporters, it is likely that only the gateway machine and the remote users should be allowed into the download sessions.

A problem arises when dealing with user identities and credentials. If different credentials are used for gaining access to the group and for exporter local file access, some mapping

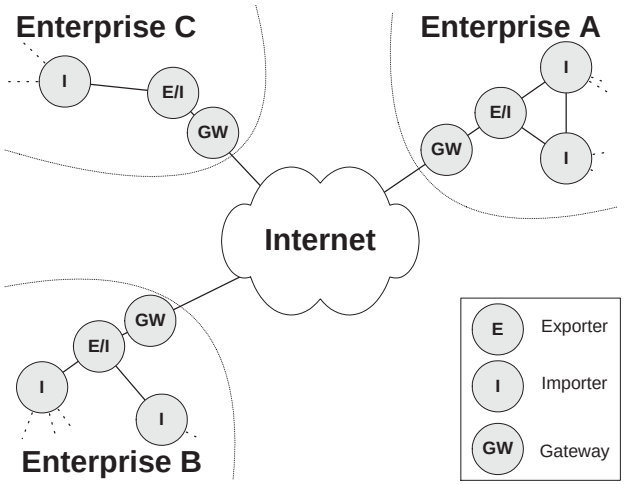


Figure 6.6: Scenario 3 - The enterprises comprising the coalition have conditional and fluid trust. The policy under which the control and download content is distributed is a direct reflection of these conditions.

must be used to enforce authentication and access control. AMirD currently implements a credential mapping function between the two identities (Certificate common names and UNIX identifiers). Members who have rights to the relevant file (as determined by the mapping) are allowed access to download groups.

6.2.3 Scenario 3 - Coalition Networks

A coalition network allows independent enterprises to share information in a secure and controlled manner [PCKS01]. Depicted in Figure 6.6, the example coalition contains three 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. Ejection of misbehaving members is supported through an appropriate keying policy.

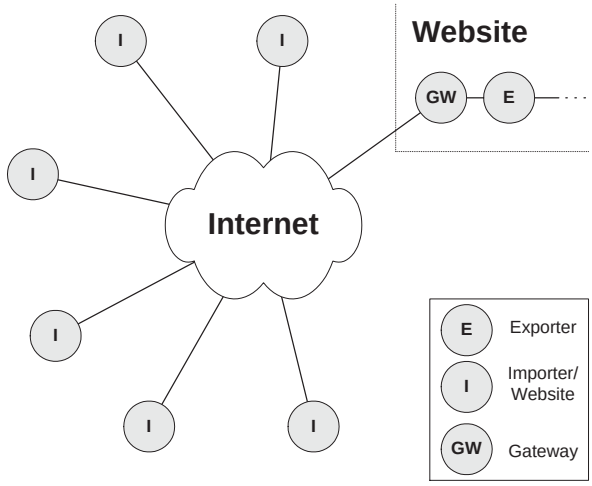


Figure 6.7: Scenario 4 - The contents of a website are synchronized to a large body of largely untrusted mirror sites. The authenticity of the content is of paramount importance.

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. Thus, control over filesystem access should be enforced. 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 Scenario 1. 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.

6.2.4 Scenario 4 - Site Mirroring

Website replication is increasingly being used to reduce client latency and Internet backbone load. This scenario demonstrates how AMirD can implement a replication service efficiently and securely. Described in Figure 6.7, 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). Emergency updates are initiated without prior announcement (see Section 6.3.2).

Because the content announcements of a mirrored website are unlikely to be sensitive, secrecy of the content 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 the sessions can be statically defined in the group policy.

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, Similarly, one must ensure authorized mirror sites are allowed to import the web filesystems.

6.2.5 Illustrating Policy

This section describes the representation and meaning of an example policy meeting the scenario goals set forth in the previous sections. The policy cited in the following text is summarized in Table 6.1 and presented in its entirety in Appendix A. A study of the performance of AMirD under these policies is presented in Chapter 7. Note that these policies represent one way to achieve the stated goals. In practical use, the construction and content of Ismene policies will be a function of the application goals and issuer preferences.

As is true of any Ismene policy, evaluation of the AMirD policy begins with the `provision` clause. This default clause states three sub-policies must be reconciled; `antigone`, `monitor`, and `application`. Evaluation of the policy is largely driven by the `grouptype` predicate. The following clauses direct reconciliation to the appropriate group provisioning:

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

Policy	Control Group		Download Group	
	mechanism	**configuration**	**mechanism**	**configuration**
Scenario 1 - 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*	
Scenario 2 - Mobile Users				
Authentication	OpenSSL		OpenSSL	
Membership	Antigone	no membership	Antigone	no membership
Key Management	AGKM	RC4 or Blowfish	AGKM	RC4 or Blowfish
Data Handling	Antigone	integ/(conf)	Antigone	integ/(conf)
Failure Detection	*none*		*none*	
Scenario 3 - Coalition Networks				
Authentication	OpenSSL		OpenSSL	
Membership	Antigone	membership	Antigone	no membership
Key Management	LKH	leave/eject/fail sens	KEK or AGKM	RC4 or Blowfish
Data Handling	Antigone	sauth/integ/conf	Antigone	sauth/integ/conf
Failure Detection	Chained FP		*none*	
Scenario 4 - 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 6.1: Scenario Provisioning Policy Summary - policies appropriate for the environments described in Section 6.2.

Implemented by the application, `grouptype` identifies the environment and purpose of the current AMirD agent. The last line (beginning with a '%' symbol[3]) 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.

Scenario 1 enforces a statically defined policy similar to those available in contemporary group communication services (Local LAN, `lanprov` clauses). As is appropriate for the target environment, the control group is largely unprotected. Members are not authenticated (and are trusted to provide their real identity), and all group text is sent in the clear. No authentication and access control is enforced over the control group; anyone on the local

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

network is free to join. This egalitarian policy is represented through authentication and access control clauses of the form:

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

These clauses state that any member should be allowed access to Local LAN control groups. However, the membership is free to augment these policies with additional restrictions through local policies (See Chapter 4).

Download access (join) in scenario 1 is partially regulated by the `hasReadAccess` predicate. Semantically, the predicate determines whether the identity of the member has rights to access the download file. The user identity ($id) download ($file), and filesystem ($fsys) are *application attributes* asserted by the agent at run-time. The predicate implementation maps the user identities to UNIX a 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 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.

Scenario 2 (Mobile Users, `mobileprov` clause) requires a stronger security policy. Members are authenticated using the local OpenSSL [Gro00] service. As determined by the `isSensitiveFile` and `hasSensitiveFilesystem` predicates, the content may be confidential. The selection of a cryptographic algorithm used to protect content is driven by local policies. In the clause,

```
mkey : :: config(agkmkey(kychlen=64,rekeyperiod=60,hash=sha1)),
          pick(config(agkmkey(crypt=rc4)),config(agkmkey(crypt=blowfish)));
```

the issuer states that either the RC4 or Blowfish encryption algorithms are acceptable. Through reconciliation, the selection of a single algorithm is the result of the member desires stated in the associated local policies.

All keys, certificates, and tickets used by Antigone are modeled as credentials. The right to perform an action is partially derived from the credentials asserted by the member. Credential conditions test the attributes associated with credentials. Actions are *accept*ed where the asserted credentials and conditions are sufficient to meet at least one authentication and access control clause.

Access to the control and download groups in scenario 2 is partially predicated on the assertion of X.509 certificates [HFPS99]. The `credential` test in the clause,

```
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 member must provide a certificate issued by the known issuer `Antigone_SSL_CA`. Note that the Antigone enforcement engine will only assert a certificate after it has been validated internally; a certificate is deemed valid only if a certification chain rooted by a locally stored CA certificate can be found[4]. In this case, validation will be based on the locally stored `Antigone_SSL_CA` public key certificate.

The statically defined ACL attribute,

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

defined in the policy further specifies group access. This ACL attribute (as processed by the Antigone internal `inlist` predicate) explicitly enumerates the identities that have the right to access the group.

Access to other group action requires the assertion of the appropriate keys. The policy assumes two types of keys are established and maintained by Antigone. Negotiated during initial authentication, *pairkeys* are ephemeral shared secret keys known only to the initiator and a single member. *Session* keys are created by the key management mechanism and replaced as directed by policy. These keys are used throughout the lifetime of the group as authorizing information. For example, the clauses

```
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

[4]Revocation, and certificate management in general, presents a number of difficult challenges [MR00]. As such, Antigone currently does not make use of any online certificate acquisition or revocation service. A mechanism based certificate service is being considered, and may be introduced in future revisions of Antigone.

the current session key, and the credential test determines whether the session key was used to generate the message sent to the group.

The environment of Scenario 3 (Coalition Network, `coalprov` clause) consists of a large number of independent enterprises. However, the services available to each enterprise may be very different. Hence, AMirD must allow groups to converge on an acceptable and interoperable set of policy implementing mechanisms. This ability is demonstrated through the evaluation of clauses associated with key management provisioning. The clause,

```
ckey : :: pick(config(agkmkey(crypt=rc4)),config(kekkey(crypt=rc4)));
```

states that either AGKM or KEK mechanism may be used for key management. The decision of mechanism will be determined by the local policies. Moreover, access to the group will be predicated on this determination; members whose local policy is satisfied by the resulting decision will be free to participate. However, any member who requires the mechanism that is not selected is precluded from participation.

Members must assert a certificate credential issued by an authority indicating the right to participate in the control group. The clause,

```
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. Hence, binding allows policy to specify trust relationships. In this case, the SSL authority is trusted to regulate group access through certificate issuance.

The clauses,

```
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 policy of Scenario 4 (Site Mirroring, `webprov` clauses) demonstrates a commonly used content distribution policy. Members are allowed into the group if they can provide an appropriate access certificate. Subsequent access to group functions is entirely predicated on the initial authentication. All communication has integrity, but management of membership or failures is deemed unnecessary.

Authenticity of group content is paramount in website mirroring. The clause,

```
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 packet or online content signature. However, it is not realistic to assume that any entity will ultimately be the authenticating body (CA) for all websites. Moreover, Antigone cannot determine the correct authority; the application must identify an authenticating body based on the content. The application signifies this judgment by identifying an authority through `$authorid` attribute. Hence, a member will only be able to mirror a website after obtaining and locally storing the appropriate public key certificate.

Similarly, the clause,

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

indicates that the application will assert the desirable provisioning 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.

Note that, in this scenario, there are no clauses associated with the `eject` action. The AGKM keying mechanism does not provide backward secrecy (see Chapter 2). Hence, an

ejection would not prevent a member from continuing to view application content. Because the key management mechanism does not support ejection, the action is invariantly rejected.

Monitoring and application policies are relevant to (enforced by) all scenarios. The monitoring policy states that the `ascope` debugging mechanism be used. The Antigone `envvar` predicate tests whether the `TKSCOPE` environment variable is set to TRUE. If so, the scope user interface is presented at each local host (see Chapter 5).

Application polices allow the application configuration to be driven by policy evaluation. The reserved mechanism designator `applic` is interpreted by Antigone as application policy. All parameters defined in the instantiation for the `applic` mechanism are stored in the group attribute set. Applications obtain the relevant policy by querying the group attribute set at run-time. To illustrate, the group policy defines the clauses,

```
application : :: config(applic(followsymlinks=true)), apsauth;
apsauth : grouptype(locallan) ::
          config(applic(maxexportsubgroups=20, maximportsubgroups=20));
apsauth : :: config(applic(maxexportsubgroups=10, maximportsubgroups=10));
```

which indicate that symbolically linked files and directories should be exported. The latter two clauses place a maximum on the number simultaneous export or import groups in which a member may participate. This ceiling is placed at 10 groups where a computationally expensive sender authenticity policy is enforced (as determined by other evaluation), and 20 elsewhere.

6.3 Implementation

This section considers the configuration and use of AMirD. The following subsection discusses the means by which the AMirD filesystems and policies are specified through a configuration file. Section 6.3.2 concludes with a brief discussion of the use of UNIX signals to modify AMirD behavior at run-time.

6.3.1 AMirD Configuration

The configuration of an AMirD agent is specified through a run-time specified file located on the local host (e.g., `amird.conf`). This configuration file specifies the filesystems, parameters, and security policies to be used to direct the action of the local agent. The

```
1  # Configuration section
2  [config]
3  grppol = amird_scen.apd
4  locpol = amird_lscen.apd
5  exporter=initiator
6
7  # Exports section
8  [Exports]
9  # <filesystem root> <group policy> <local policy> [<authority file>]
10 /usr/local/amird amird_scen.apd amird_lscen.apd antigone_ca
11
12 # Imports section
13 [Imports]
14 # <exporter:filesystem root> <mount point> <local policy> [<authority file>]
15 antigone:/usr/local/import /usr/local/antigone amird_lscen.apd antigone_ca
```

Figure 6.8: An example AMirD configuration file

configuration file consists of three sections; a *configuration section*, an *exports section*, and an *imports section*. Figure 6.8 shows a subset of an example AMirD configuration file, and table 6.2 enumerates all configuration file parameters.

Designated with the `[config]` label, the configuration section defines the operational parameters of the AMirD agent. This includes addressing information for the various control and download groups, transport parameters, and behavioral parameters. Required only by the initiator, the policy used to define the control group is specified through the `grppol` parameter. The control group local policy is specified in the `locpol` parameter. The exporter identifier is specified through the `exporter` parameter.

The exports section (`[exports]`) defines the filesystems to be exported by the local agent. Each exported filesystem is identified by a one line definition consisting of three fields; the local path to the root of the exported filesystem, and the group and local policies to be used when initiating the associated download groups. If specified, the authority file is used to identify the public key of authority issuing content authenticating certificates for the exported filesystem.

The imports section (`[imports]`) defines the filesystem to be imported by the local agent. Each import definition consists of three fields; the exported filesystem identifier, a path to the root of the replicated filesystem (e.g., mount point), and the local policy to be used when participating in download groups associated with the imported filesystem. The exported filesystem is uniquely identified by the exporter identifier and exported path filename. The imported filesystem content authority file is optionally specified.

General Configuration	
exporter	exporter identifier
isserver	local initiate control group flag
updateperiod	frequency of AMirD advertisements
minupdateperiod	minimum advertisement frequency
mcloopback	flag enabling multicast loopback
Control Group Configuration	
grppol	control group policy
locpol	control group local policy
authoritycert	control group admittance authority certificate
transport	control group transport mode
mcaddr	control group multicast address
mcport	control group multicast port
srvraddr	control group server address
srvrport	control group server address
Download Group Configuration	
sgtransport	subgroup transport mode
sgaddr	subgroup address
sgport	subgroup multicast port
sgsrvraddr	subgroup server port (asymmetric, overlay) modes
sgsrvrport	subgroup port (starting)

Table 6.2: AMirD configuration - parameters stating the policies and configuration of an AMirD agent.

6.3.2 Signal Handling

AMirD is designed to be executed as a background process. However, UNIX signals [Pagc] delivered through `kill` [Paga] (or similar service) can be used to trigger AMirD action. The following signals are currently supported by AMirD;

- `SIGHUP` - reinitialize. This terminates participation in all groups, destroys sensitive state, and re-reads the configuration and policy files. Configuration information is used to direct the initialization of all identified filesystems, and to re-join the control group.

- `SIGUSR1` - re-synchronize filesystems. Announcements are immediately broadcast (requested) for each exported (imported) filesystem.

- `SIGINT`, `SIGQUIT` - gracefully terminate the local agent. AMirD immediately exits control and download groups, destroys sensitive state, and terminates.

Signals are useful in managing the mirrored filesystems. For example, a (importer) user desiring an immediate update uses the `SIGUSR1` signal to indirectly trigger an announcement. However, the ability to arbitrarily trigger announcements can lead to congestion or introduce latencies in the download process. The `minupdateperiod` configuration parameter is used to mitigate this problem by placing a lower bound on consecutive filesystem announcements.

CHAPTER 7

PERFORMANCE

A prerequisite to the acceptance of any policy infrastructure is an understanding of its inherent cost. Any solution that significantly impedes the achievement of application goals will not likely be useful. This chapter seeks to identify the fundamental costs of policy determination and enforcement within Antigone. The processes and mechanisms defined by the Antigone architecture are measured and analyzed. It is through this analysis that the limitations of group policy infrastructures are illuminated, and specific enhancements to Antigone identified.

This chapter considers the degree to which Ismene and Antigone satisfy the goals for *Efficient Multiparty Determination* and *Efficient Enforcement*. This evaluation of the Antigone policy infrastructure described throughout is partitioned into three areas. The first area studies the costs associated with policy determination. A stated goal of this thesis is the identification of approaches that allow efficient policy processing. Section 7.2 considers the run-time costs associated with Ismene policy evaluation, reconciliation, compliance checking, and analysis under a number of diverse policies. A second area of investigation presented in Section 7.3 considers whether Ismene policies can be efficiently enforced. This study evaluates the throughput and latency characteristics of Antigone under a range of group policies. The study further investigates direct enforcement costs by profiling the dominant factors associated with communication. Finally, an investigation of application performance is presented in Section 7.4. The AMirD filesystem mirroring application operating under the four scenario policies presented in Chapter 6 is measured, and the effects of several security and application policies explored.

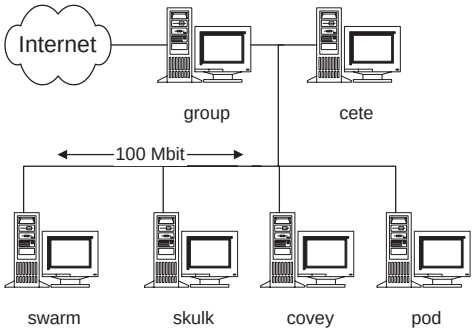


Figure 7.1: Experimental Environment - all tests described in this chapter were executed in within the Antigone cluster. Group tests were executed over five member group containing the hosts; cete (session leader), swarm, skulk, covey, and pod.

7.1 Implementation and Experimental Setup

The Antigone enforcement architecture and Ismene Policy Description Language have reached maturity. The current implementation of Antigone consists of approximately 58,000 lines of C++ code in 133 classes, and has been used as the basis for several non-trivial group applications. All source code and documentation for Antigone, Ismene, and applications are freely available from the Antigone website (`http://antigone.eecs.umich.edu/`).

Interfaces to the Ismene policy determination support libraries are defined though the Ismene Applications Programming Interface (API). The API defines interfaces for the creation, parsing, reconciliation, and analysis of Ismene policies. A number of tools used to support the development of Ismene policies have been constructed. The Ismene policy compiler, `apcc`, validates the syntax of group and local policies, and performs the policy algorithms defined in Chapter 4. This tool is used extensively in the following section to evaluate efficiency of the Ismene policy processing.

The Antigone enforcement architecture provides two programming interfaces; the Antigone application and mechanism APIs. The former defines a set of interfaces used to integrate applications with Antigone [MPI+01]. The latter API is used to integrate user defined mechanisms within Antigone and Ismene.

All experiments described in this chapter were performed in the Antigone cluster environment depicted in Figure 7.1. This environment used five 750 megahertz IBM Netfinity servers running the Redhat 7.1 distribution of Linux kernel 2.2.14-5. Each host contains 256

Symmetric Algorithms			Asymmetric Algorithms				
Algo.	Bits	MB/Sec	Algo.	Bits	Enc KB/Sec	Dec KB/Sec	Sig/Sec
DES	56	14.12	RSA Pub.	512	273.00	285.27	2423
RC4	128	51.17	RSA Priv.	512	28.82	28.66	274
Blowfish	160	24.30	RSA Pub.	1024	94.02	98.13	804
MD5	128	107.24	RSA Priv.	1024	5.37	5.36	45
SHA-1	160	51.05	RSA Pub.	2048	28.70	29.5	254
			RSA Priv.	2048	0.88	0.89	8

Table 7.1: Cryptographic Algorithm Performance - performance of algorithms used by Antigone on an unloaded host. The bits field indicates the performance of the algorithm under the given key (or hash) length. All symmetric algorithms were tested under 1 kilobyte blocks of randomly generated data. All asymmetric algorithms were tested by encrypting 53-bit blocks (the size of a 512-bit RSA signature).

megabytes of RAM and a 16 gigabyte disk. The Antigone cluster is connected by a 100Mbit Ethernet LAN. Antigone and all applications defined in this chapter were complied using the GNU G++ compiler version `egcs-2.91.66`. The communication experiments were executed over the isolated LAN, and all hosts were unloaded during testing.

The cryptographic functions used by Antigone are implemented by the `crypto` library distributed with the OpenSSL toolkit [Gro00]. This library has been in existence for several years and is widely used to support SSL-enabled applications. The current implementation has been optimized for the x86 architecture. Table 7.1 presents a performance analysis of the cryptographic functions implemented in the crypto library. Throughput for the various algorithms was established by measuring the processing time (using a local hardware clock) of a single application of the cryptographic algorithm on a block of randomly generated data. This process was repeated 300 times and throughput calculated from the average of the collected metrics.

The symmetric key algorithms used by Antigone (i.e., DES [Nat77], RC4 [Riv92b], and Blowfish [Sch94]) were evaluated over 1 kilobyte blocks[1]. Another battery of tests evaluated the algorithms under larger input block size (10 and 50 kilobytes). The results showed that the throughput DES and Blowfish is independent of block size (within 0.9% and 1.3%, respectively). However, for large input block sizes (100k), RC4 was up to 50% faster. This

[1]To simplify evaluation, all experiments described in this chapter reflect the fixed key sizes defined in Table 7.1. Where supported by the algorithm, however, variable length key sizes may be altered to suit the application.

is largely due to the setup costs; the initialization of RC4 S-Boxes significantly impacts throughput.

RSA [RSA78] is the only asymmetric key algorithm currently used by Antigone. The throughput of RSA under the varying key sizes was calculated from 53-byte input blocks (the largest input size allowed by the OpenSSL RSA implementation). Larger key size may accommodate larger input sizes. Hence, the throughput of RSA 1024 and 2048 may actually have throughput sizes twice or four times the reported throughput, respectively.

Asymmetric algorithms are rarely used to encrypt data in Antigone. These algorithms are used to sign data associated with some member or the group. Hence, the important metric for these algorithms is the throughput of signing operations. As is demonstrated in the latter sections of this chapter, the speed of signing operations can be a limited factor in groups enforcing sender authenticity.

7.2 Policy Determination

While Chapter 4 demonstrated the tractability of the algorithms used to construct and analyze Ismene policies, the run-time costs associated with Ismene policy determination will ultimately determine the feasibility of the approach. This section investigates these costs by benchmarking policy operations in a number of environments.

The `apcc` policy compiler was used in all experiments. `apcc` implements the predicates defined in any policy by invarienently returning FALSE. Hence, the algorithm performance cited below reflects low cost predicate evaluation; based on the semantic and implementation, application predicates may increase run-time costs. However, where possible, the predicate evaluation cache will mitigate these costs (See Chapter 5).

The policies used in the experiments below were randomly generated by the `mkpolicy` utility. `mkpolicy` creates group and local policies as dictated by command line arguments defining the number of clauses, configurations, and local policies. By construction, a specified subset of local policies are guaranteed to be compliant with any policy instantiation resulting from reconciliation with the group policy. Clauses in generated policies are constructed in a linear fashion, where three clauses are defined for each tag. The final clause defined for a tag is unconditional (a default clause), and contains a consequence directing evaluation to the subsequent tag. For example, a subset of clauses for a randomly gener-

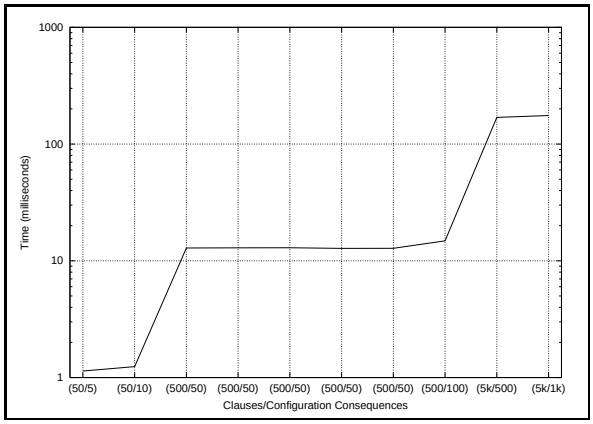


Figure 7.2: Evaluation Algorithm Performance - time to evaluate randomly generated policies containing fixed number of clauses and configuration consequences.

ated policy is defined as follows (where n is the number of clause and k is the number of configurations);

```
...
ti-1 : :: cnsqj(), ti;
ti : p3(), p4() :: unreach(); # Unreachable
ti : p5(), p5() :: unreach(); # Unreachable
ti : :: cnsqj(), ti+1;
...
tn : :: cnsqk;
```

Because all predicates will return FALSE, evaluation is guaranteed to traverse all clauses. The configurations are evenly split among pick and configuration consequences. `mkpolicy` creates an average of two predicates per non default clause.

All policies, whether group or local, are evaluated to determine the set of configurations potentially relevant to the group. Evaluation performs directed testing of predicates defined in the policy and records the resulting configuration consequences. The performance of the evaluation algorithm under randomly generated policies is presented in Figure 7.2. The figure describes the cost of the evaluation of a single policy containing the identified number of clauses and configurations.

The figure illustrates a central feature of evaluation; performance is largely independent of the number of configurations. The number of clauses traversed, and hence the number of predicates tested, will determine the cost associated with evaluation. Because the predicates used in this test simply return *invariant/false*, these results serve as a lower bound to

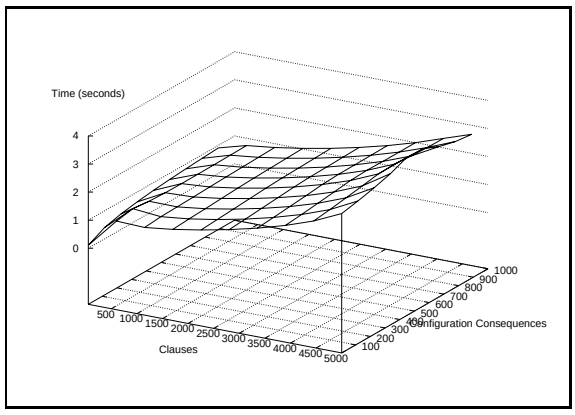


Figure 7.3: Reconciliation Algorithm Performance - performance of reconciliation under randomly generated policies containing a fixed number of clauses and configuration consequences.

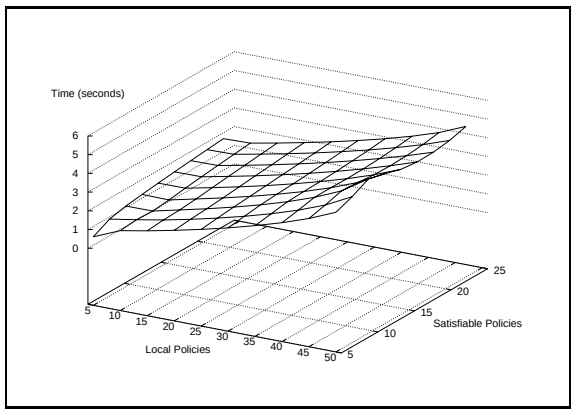


Figure 7.4: Local Policy Reconciliation - performance of reconciliation with a fixed number of satisfiable and unsatisfiable local policies.

evaluation. In practice, the cost of evaluation will largely be determined by the costs associated with predicate implementations.

The *Reconciliation* algorithm attempts to find an instantiation satisfying at the group policy and as many local policies as is possible. Reconciliation begins by evaluating the group and each local policy. The set of consequences resulting from group policy evaluation is used as a template; any instantiation will contain a subset of the configurations defined in the `config` and `pick` statements. Local policies are used to further refine the template (e.g., via selection of values of pick statements). Local policies (and any refinements resulting from its reconciliation) that cannot be satisfied by the instantiation are discarded.

The performance of the reconciliation algorithm under policies with a fixed number of clauses and configurations is presented in Figure 7.3. Note that these experiments measure not only the refinement process, but also the evaluation of each policy. Hence, the figure describes the total time required to reconcile a collection of policies. The figure illustrates, for the generated policies, that the costs associated with reconciliation increase gradually with the number of configuration consequences. Hence, the reconciliation of each consequence requires a constant, albeit small, cost. The number of clauses has a much larger affect on reconciliation. The evaluation of the group and (10) local policies used in these tests dominate measured costs.

While the previous experiments indicate that the number of local policies is a central determinant of the cost of reconciliation, all local policies were satisfiable. Figure 7.4 considers

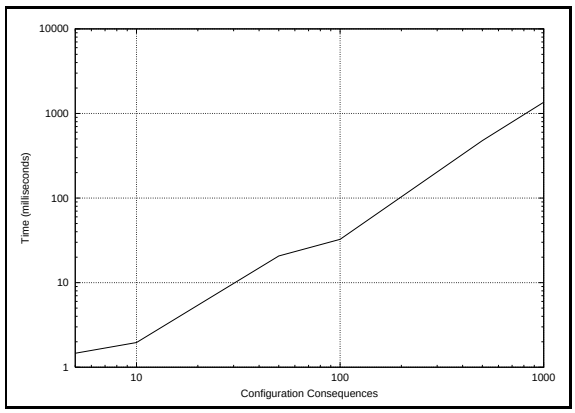


Figure 7.5: Compliance Algorithm Performance - time to test the compliance of a randomly generated local policy with an instantiation containing a fixed number of configurations.

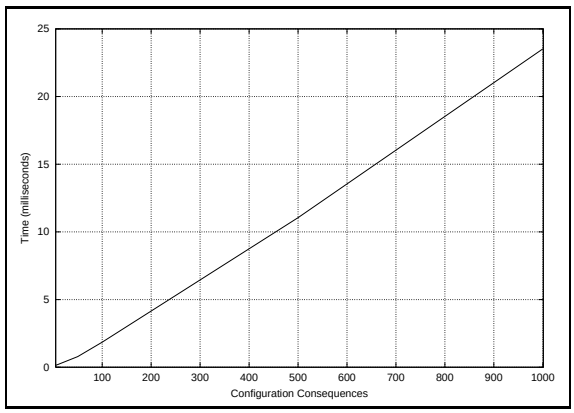


Figure 7.6: Online Analysis - Algorithm Performance - time to test the compliance of a randomly generated local policy with an instantiation containing a fixed number of configurations.

the cost of reconciliation where a subset of local policies are unsatisfiable (where the group and all local policies contain 500 clauses and 50 configuration consequences). The figure shows that the satisfiability of the local policy does not significantly impact performance. As satisfiability of a local policy is not known *a priori*, the reconciliation algorithm must attempt to find a satisfying instantiation. Because evaluation dominates the refinement process, the effect of non-compliance has little impact on performance.

The *Provisioning Compliance* algorithm tests whether a local policy is satisfied by a policy instantiation. This consists of searching the instantiation for configurations satisfying the consequences resulting from the evaluation of the local policy. Configuration consequences are satisfied if the instantiation defines the associated configuration. Pick consequences are satisfied if exactly one of the configurations is defined by the instantiation. Hence, to simplify, the provisioning compliance algorithm calculates the intersection of the instantiation and each local policy consequence. A consequence is satisfied when the size of the intersection is exactly one.

Figure 7.5 depicts the measured performance of the provisioning compliance algorithm. In all tests, the number of local policy consequences was equal to the number of configurations. The figure shows the linear relationship between the number of configurations/consequences and performance; there is a constant cost of performing policy intersection. As the number of consequences and configurations grows, so do the costs of performing the intersection.

An analysis algorithm determines whether a set of invariant configuration properties defined by an *assertion* clause are preserved by policy. `mkpolicy` automatically creates two assertions with each policy. By construction, the first assertion (consisting of a single configuration consequence) is guaranteed to not be violated by (any) reconciliation. While occurring frequently, the configuration consequence defining the second assertion is guaranteed to be violated.

Semantically similar to provisioning compliance, *Online analysis* evaluates assertions at run-time by analyzing an instantiation against the set of configuration assertions. This is performed by searching an instantiation for the (non)existence of the identified sets of configurations. Depicted in Figure 7.6, this simple search is performed quickly; even very large instantiations can be searched in a matter of milliseconds. As one would expect, the search costs grow linearly with the number of configurations to be searched.

Offline analysis attempts to determine whether, under a given policy, a set of assertions can ever be violated via reconciliation with a set of local policies. This intractable problem requires analysis search the space of possible instantiations. Variability (e.g., pick statements) of the group policy can exponentially increase this space. Tests of the offline analysis demonstrated this explosion; a policy containing pick consequences clauses took under 1 second to complete, and a policy with 20 clauses took over 1 hour. While a more efficient implementation of the *validity* test that forms the foundation of offline analysis may glean faster results, it is not clear that offline analysis of large, highly variable policies is feasible.

7.3 Policy Enforcement

An oft-cited liability of component and event based systems is cost. The added overheads of event management and generalized interfaces may slow group progress, and ultimately lead to implementations that are inappropriate for high-speed communication. This section considers the performance of Antigone under a number of policies. It is from this analysis that an understanding of the fundamental limitations of flexible policy enforcement can be gleaned.

The measured throughput and latency of Antigone under a number of data handling policies is given in Figures 7.7 and 7.8. These tests attempt to find the maximum data

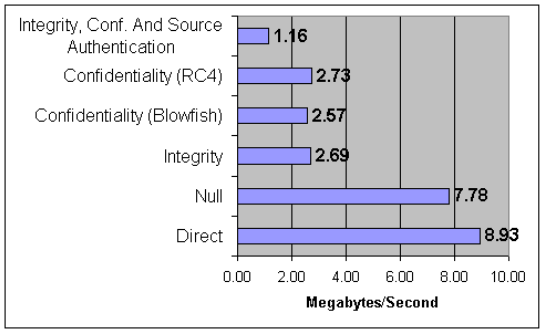


Figure 7.7: Antigone Throughput - throughput of Antigone under diverse data handling policies.

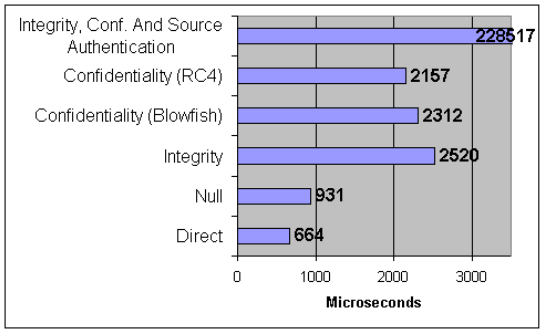


Figure 7.8: Antigone Latency - latency of Antigone under diverse data handling policies.

rate and average round trip latency of group communication under direct, null, integrity, confidentiality and an integrity, confidentiality, and source authentication policy. The direct test estimates the maximum throughput possible on the network through standard send/receive socket calls. The null experiment implements a group that transmits data in the clear (implements a cleartext policy). The integrity policy is enforced by attaching SHA-1/RC4 based HMACs to each packet. The confidentiality policies encrypt all data using the identified cryptographic algorithm. The final experiment implements a strong data handling policy where SHA-1, Blowfish and 1024 bit off-line signatures (frame size=20, see below) are used to ensure integrity, confidentiality, and source authentication. Note that the latency measurements calculate the total round trip time of a single message on an unloaded network. Hence, the latency metrics represent four traversals of the Antigone protocol stack.

The throughput of Antigone closely follows the strength of the enforced security policy. The direct and null data test identifies Antigone overheads. The environment is capable of transmitting up to 9 MBytes/Second, and Antigone is limited to just under 8. This represents an 11% reduction in throughput. A detailed investigation of Antigone overheads is presented below.

Integrity and confidentiality policies exhibit similar throughout. It is interesting that confidentially policy using the much slower Blowfish algorithm only marginally reduces throughput over confidentiality implemented with the fast RC4 algorithm. Both algorithms are significantly faster than the throughput of the network. Hence, unlike many previous systems, throughput is not limited by encryption, but on message marshaling. The dra-

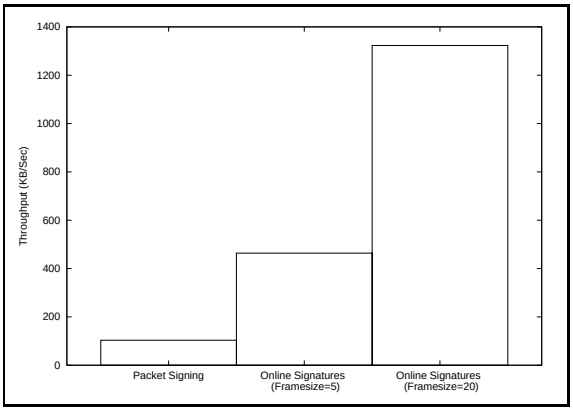


Figure 7.9: Source authentication throughput - throughput of Antigone under a set of source authentication policies.

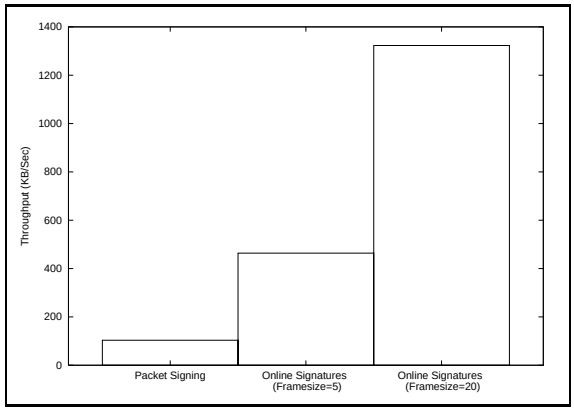


Figure 7.10: Source authentication latency - measured latency of Antigone under a set of source authentication policies.

matic differences in throughput between these policies and null policy can be attributed to encryption and marshaling; confidentiality and integrity transforms require an additional buffer copy and the setup of the appropriate cryptographic algorithms.

The integrity, confidentiality, and source authentication policy demonstrates the canonical strong group data handling policy. While it has been suggested that similar policies are difficult or impossible to implement in group communication [CP00], Antigone can achieve high data rates (1.16 Mbytes/second) through the proper application of off-line signatures and high-speed encryption. It is worth noting that the test environment made use of high-end hardware. Contemporary desktop machines may not be able to achieve these rates.

The latencies associated with these policies mirror throughput. The null and direct (differing by 10%), confidentiality and integrity policies (differing by at most 4%) exhibit similar latencies. Note that the latency of integrity, confidentially, and source authentication policy is dominated by a data forwarding timer (see below). This timer delays the packet transmission by 100 milliseconds in both directions. In all cases, the observed latencies were well within normal application tolerances.

A number of works have addressed the problem of source authentication in group communication [EGM96, GR97, Roh99, WL99]. The authors of these works frequently state that the cost of digital signature generation is in conflict with the abilities of high speed groups. However, experiments within Antigone have shown that this need not always be true. Figures 7.9 and 7.10 describe the throughput and latency of source authenticated data under three policies. Note that each experiment uses 1024-bit RSA keys and transmit

1024 bits packets.

The *packet signing* transform implemented by the current Antigone Data Handler mechanism signs each packet as it is transmitted to the group. Receivers validate each signature using a locally stored public key certificate. *Off-line signature* caches a policy determined number of packets (called the frame size). The packets are cached for a maximum period defined by the data forwarding timer (50 milliseconds in all tests). An off-line signature [EGM96, GR97] is generated for all packets in the frame when either the frame becomes full or the data forwarding timer expires. The off-line signature is calculated by appending a forward hash chain to all packets in the frame, and signing the first packet (see Chapter 3).

Figure 7.9 shows that reasonable data rates can be achieved with packet signing. Over 100 packets per second can be signed and processed. Moreover, this rate can be significantly improved with packet size increases; the cost of digital signature generation are largely independent of the size of the signed content. For example, the 3.5Mbps through required by a 30 frames per second MPEG-1 video stream [AAC+99] could be achieved through 4096 byte packets.

Off-line signatures act as rate multipliers. Throughput under source authentication policies are dominated by signature generation. Hence, by only signing every k^{th} packet (where k is the frame size), you can linearly increase the throughput of source authentication. This is demonstrated in the figure, a frame size of 5 provides nearly 5 times the throughput of packet signing. A performance ceiling is reached at a frame size of 12, where signature generation no longer dominates cost. In this case, other aspects of message handling (e.g., signature generation, marshaling) begin to affect throughput.

Off-line signatures are limited by latencies, reliability, and bursty transmission. Latencies are incurred when an entire frame of data is not sent. Off-line signatures also exacerbate message loss. Any packet received following a dropped packet (until the end of the frame) cannot be validated. Finally, there is a burst of packets following the completion of the frame or after the data forwarding timer expires. These limitations can be mitigated by shorter data forwarding timers and through the application of forward error correction [WL99] and rate-controlled transmission [ZF94].

While the previous experiments indicate the efficiency of Antigone, they do little to illuminate the direct costs of policy enforcement in Antigone. The following considers

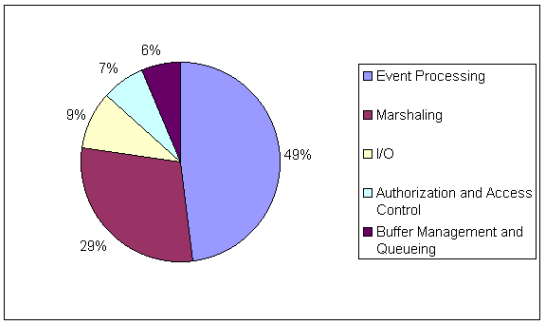


Figure 7.11: Receive Processing - breakdown of Antigone protocol stack operation involved in the receive of an application message.

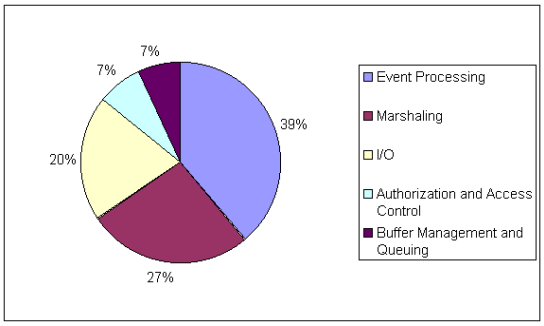


Figure 7.12: Send Processing - breakdown of Antigone protocol stack operations involved in the broadcasting of an application message to the group.

the costs associated with the Antigone protocol stack. These tests calculate the average of 100 measurements of Antigone protocol operations during application send and receive operations. All measurements were taken under a Blowfish confidentiality data handling policy

The costs associated with *event processing*, *marshaling*, *I/O*, *authentication and access control*, and *buffer management and queuing* were measured by collecting hardware timing information at many points in the Antigone protocol stack. Event processing metrics measure the time consumed in the creation, queuing, and delivery of events. Marshaling is the process of message parsing and unparsing. I/O measures the time consumed by `send()` and `recv()` socket calls. Authentication and access control measures the time consumed by the policy engine in evaluating action clauses and caching results. Buffer management and queuing measures the time taken to create and destroy buffers, and the management of message queues.

Figures 7.11 and 7.12 present the costs associated with the reception and transmission of a single packet, respectively. Note that almost half of the 93 microseconds[2] of receive overhead is consumed by event processing. This is due to event processing by other mechanisms; unlike send events, receive events may incur mechanism action. For example, the scope mechanism records the length of sent data. Hence, the application receive is delayed by the processing of the received data by other mechanisms. Similarly, event processing

[2]All metrics reflect the Antigone specific operations. Hence, the performance reported by the read/write breakdown does not include encryption costs. The time spent, including encryption and decryption costs, in the readMessage and sendMessage calls is 115 and 96 microseconds, respectively.

consists of about 40% of transmission overhead.

Event processing costs are dominated by the event creation and queuing. Unlike other event systems (e.g., Cactus [HJSU00]), event delivery is broadcast rather than directed by subscription. The costs associated with this design are small; event delivery (a function call) are dominated by the cost of event creation and event queue management. Note that subscription based approaches will have a set of costs not present in Antigone, where the event must be mapped onto the set of mechanisms to which it will be delivered.

About 30% of time spent in message processing is consumed by marshaling. While the generalized message handling interface significantly reduces the effort required to develop new message transforms, it comes at a cost. The interpretation of the message template structures presents additional overheads over hard coded message marshaling code. The generalized message marshaling implementation has not as yet been optimized. Hence, based on an evaluation of the current design, there may be several ways to mitigate marshaling costs. The average per cost message marshaling is about 27 microseconds for receives and 20 for transmissions.

An interesting observation made during these experiments indicates that the time spent in the `send` system call (15 microseconds) is about twice that of the `receive` call (8 microseconds). A test of `send` and `recv` calls external to Antigone resulted in similar results. Hence, the difference can be attributed to the Linux implementation of the IP protocol stack.

Enforcement of fine-grained authentication and access control has often been cited as too inefficient to be realizable in high speed communication. In Antigone, however, the use of the evaluation caching mechanism is demonstrated to mitigate the costs associated with content regulation. The "send" action used to govern broadcast data in the test environment is predicated (only) on knowledge of the session key. Therefore, the send action is evaluated on the first reception from a sender, and cached results are used until the session is rekeyed. Authentication and access control becomes the process of searching the cache. However, other policies requiring the evaluation of transient predicates may incur significantly higher evaluation costs.

The remainder of the costs associated with Antigone represent those required by any protocol implementation; buffer management and queuing. These costs are associated with the creation and destruction of send/receive/intermediate buffers and the management of

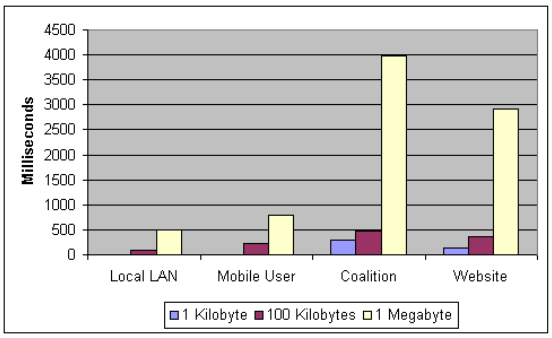


Figure 7.13: Reliable Broadcast Transfer - average transfer times for files of varying size under AMirD scenario policies.

the receive queue. With the notable exception of off-line signature data handling policies, all data is transmitted during the application sendMessage call. Therefore, no output queue is necessary.

7.4 End-to-end Performance

While throughput measurements provide insight into the costs of policy based communication, only through its use in meaningful applications can confidence in the Antigone approach be gained. This section considers the characteristics of the AMirD application under the scenario policies presented in Chapter 6. The results of file transfer experiments using the AMirD reliable transfer protocol are depicted in Figure 7.13. These tests measured the average time of 10 file transfers of 1-kilobyte, 100 kilobyte, and 1 megabyte files using AMirD under the scenario policies. All tests transmitted 1-kilobyte blocks and used an acknowledgment window of 100 packets.

For all file sizes, the local LAN and mobile user policies exhibited significantly shorter transfer times than the other, relatively costly policies. This is consistent with the throughput measurements presented in the preceding section. The confidentiality policy enforced by the LAN policy requires the encryption of data. However, encryption is sufficient to only marginally delay transmission. The mobile user policy implemented both confidentiality and integrity. Hence, the round trip times were delayed while the HMACs were calculated and additional marshaling performed.

The transmission of smaller (1k) and medium (100k) sized files is completed through a

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 7.2: AMirD Performance - time, in seconds, to synchronize two AMirD filesystems under the scenario policies.

single window of data. Hence, in all policies, the transmission is completed in less than 500 milliseconds. The difference between transfer rates can be attributed to receivers waiting for the data forwarding timers to expire.

While qualitatively similar, data forwarding and confidentiality policies affected coalition and website environments differently. Both the coalition and mobile user policies enforced confidentiality, integrity, and sender authenticity policies. The coalition policy implemented a 250 millisecond data forwarding timer. Therefore, the longer transfer times can be attributed to increased waiting periods associated with window acknowledgments. The short (50 millisecond) data forwarding timer and lack of confidentiality (no transferred files were deemed sensitive) in the mobile user policy resulted in faster transfer rates.

These results serve as a cautionary tale; one must be aware of the effects of the selected policies. In this case, the data forwarding timer used by off-line signatures limits the speed with which data can be transferred. Hence, it is incumbent on a policy issuer not only to be aware of the security required by an environment, but also to be cognizant of the secondary effects of enforcement.

Table 7.2 describes the total time required by an AMirD agent to synchronize two filesystems under the test environment. 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 period and a 3 second shutdown period. The setup protocol is required because the exporter does not know *a priori* the identity of all interested importers, and thus must allow ample time for members to join. The shutdown protocol is required to gracefully destroy the group.

Synchronization times are partially a reflection of the file transfer rates and application policies. Simple, efficient policies allow the file transfers to process at a rapid rate. Moreover,

the Local LAN policy allows twice as many 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. Such is the promise of policy behavior; through policy, the application can be responsive to the security policy appropriate for a given session.

Note that differences in reported measurements represent timing conflicts and dropped packets. A member who is delayed while processing other group content can either become aware of a spawned subgroup after the transfer has begun, or miss announcements entirely. In this case, the member will must request and subsequently await the re-scheduling of the missed transfer. Moreover, packet loss was exacerbated by the use of SSL. The expensive authentication protocol used by SSL consumed significant resources (the initiator performed this exchange for each member in every download group).

Note that, unlike Antigone, AMirD has yet to mature. The measurements and insights gleaned from these tests indicate areas of possible improvement. For example, a more efficient startup protocol may increase the rate at which filesystems can be updated. Similarly, delays incurred by the acknowledgment window have been demonstrated to severely limit throughput under off-line signature policies. A further improvement may allow the window to be adaptive; the acknowledgment window can grow where no packet loss is detected. Moreover, knowledge of the appropriate window can be persistent over time. In this latter strategy, the sender will use receiver profiles developed over the course of an AMirD session.

CHAPTER 8

CONCLUSIONS AND FUTURE WORK

This thesis has investigated technologies supporting policy management in secure group communication. The goals identified at the outset of this work have been largely addressed. Antigone realizes these goals through an architecture supporting flexible and efficient policy determination and enforcement. This chapter considers the contributions and future directions relating to Antigone. The following section considers how the goals presented in Chapter 2 are addressed. Section 8.2 presents several avenues of future investigation.

8.1 Goals

The goals presented in Chapter 2 identify the requirements of a general-purpose group policy management infrastructure. The following text revisits each of these goals.

The Ismene policy determination engine supports *flexible representation* through the specification of policies encompassing the entirety of the group security context. Policy issuers and members state the requirements for group provisioning, authentication, and access control. Provisioning policies identify not only the security guarantees provided to the group, but also the means by which they are achieved. Policies can be conditional and discretionary. Conditional policies state the environmental conditions under which a provisioning policy is relevant. Discretionary policies state a range of acceptable provisioning alternatives.

Authentication policies identify the members that are allowed to participate in the group. Fine-grained access control policies define the actions admitted members are permitted to perform. However, the actions to be governed by policy are not fixed; new actions can be defined as requirements arise. Moreover, access is not only predicated on the presentation of appropriate credentials, but also on environmental conditions.

The ability of Ismene to represent policies of real-world applications and environments

is demonstrated in Chapter 6. Policies appropriate for the Antigone-based AMirD content distribution service in diverse environments are represented through the Ismene language. The performance of AMirD under these policies is further considered in Chapter 7. As expected, these experiments show that the strength of security afforded by the group has a dramatic affect on performance.

A central requirement of this work is the definition of techniques for *multiparty determination*. Policy must be efficiently derived from the desires and abilities of all communication participants. The policy instantiation enforced by a group is the result of the reconciliation of the group policy and the local policies of each expected member. Members joining the group use the compliance algorithm to determine if the instantiation meets the requirements stated in their local policy.

Note that the security defined by a policy instantiation must adhere to a set of correctness principles. Failure to adhere to these principles may result in the creation of a group service that is insecure or non-functional. The analysis algorithms assess whether an instantiation (in the case of online analysis) or group policy (in the case of offline analysis) adheres to a set of policy-validating assertions. Hence, the security of a policy can be assessed prior to the creation of the group.

The efficiency of policy determination has been assessed both analytically and experimentally. The formal analysis presented in Chapter 4 shows that all algorithms used in the critical path of session creation and maintenance are tractable (in the time-complexity sense). Experiments in Chapter 7 further demonstrate that the policy determination algorithms are efficient; an instantiation resulting from the reconciliation of many large group and local policies can be executed in a matter of seconds. Moreover, policy reconciliation with more modest policies (such as those defined in Chapter 6) can be performed in microseconds.

Flexible policy enforcement is achieved through the Antigone component architecture. Security mechanisms are composed and configured as determined by policy. Policy is subsequently enforced by mechanisms through the observation of and reaction to relevant events. The unique requirements of policy management require infrastructure not present in existing protocol frameworks. A number of architectural optimizations reducing both development and run-time costs allow the easy integration and efficient use of new security mechanisms. The flexibility of Antigone mechanism interfaces is demonstrated in Chapter 5 through the

definition of a number of diverse security mechanisms.

The ability of Antigone to provide *efficient enforcement* is investigated in Chapter 7. Experiments show that the high-throughput, low latency communication can be supported by the architecture. The cost of policy enforcement in Antigone is small; in the test environment, an application message send and receive is completed in about 100 microseconds. These costs primarily represent those features present in any event-based protocol implementation; about three quarters of the processing overhead can be attributed to event handling and message marshaling.

Antigone is required to be *transport agnostic*. Hence, other means of broadcast communication must be supported where multicast is not readily available. Described in Chapter 5, the broadcast transport layer provides a single abstraction for multiparty communication in the presence of varying network services. The broadcast transport layer supports three modes. These include symmetric multicast (implementing a broadcast media over n-directional multicast), asymmetric multicast (using single source multicast), and point-to-point (using unicast).

8.2 Future Work

While this thesis has achieved its stated goals, it does not address the requirements of all possible environments. The following considers areas of investigation that may advance the state of the art in group policy management.

The Ismene `reconfig` consequence signals that some state change has occurred that requires policy evolution. Antigone currently disbands and reforms the group under the new instantiation. More efficient means of policy evolution may exist. It may be possible to transition to new mechanisms and policies without loss of security. However, the subtleties of mechanism transition warrant further investigation.

Participatory groups provide an egalitarian environment in which all members contribute the definition and enforcement of policy. Hence, for many applications (e.g., auctions), participatory groups may be highly desirable. While Antigone is not restricted to centralized groups, more investigation into the requirements and constraints of participatory groups is necessary. Such an investigation is likely to not only increase the scope of policy supported by Antigone, but also improve the quality and robustness of its infrastructure.

The cost of policy enforcement in real applications is likely to be dominated by the techniques and algorithms used to provide security guarantees (e.g., key management, source authentication, fault tolerance). While the performance of point solutions has been investigated [SKJH00], a comprehensive study of the trade-offs between security policies and mechanisms has not been documented. A thorough investigation of the techniques used to provide group security will guide the construction of future architectures, and is likely to accelerate the adoption of secure group communication services.

Group negotiation protocols have been studied in many contexts [KT93, MC94, WYL+99]. The use of these protocols in Antigone would allow participants to perform policy determination through coordinated communication, rather than through prior distribution of local policies. Moreover, negotiation allows members to more flexibly adapt local policies to the requirements stated by other participants.

Antigone can be used to define application policies. These policies allow developers and administrators to dictate application configuration through the policy determination process. However, the advantages of this facility have yet to be fully explored. The use of application policy may result in more coordinated behavior, and ultimately reduce the management burden placed on users.

Antigone is currently designed for unreliable group communication in high-speed networks. The integration of other services (e.g., such as reliable group communication, survivability) into Antigone may yield a more complete solution. Moreover, these services may be made more flexible through policy. Similarly, resource assumptions (i.e., bandwidth, computing power) may limit the effectiveness of Antigone in some environments (e.g., wireless environments and mobile devices). The study and realization of mechanisms and policies appropriate for these environments will expand the domains in which Antigone is useful.

A technical limitation of the current architecture is that all mechanism implementations must be available at build-time. It is often inconvenient to recompile Antigone and all its applications after introducing a new security mechanism. The use of dynamically loaded mechanism implementations is currently being investigated. Dynamic loading will allow security implementations to be distributed independently of applications. In the presence of such facilities, the mechanisms themselves can be distributed with a policy instantiation.

APPENDIX A

This section presents the Ismene policies used to control the AMirD application presented in Chapter 6 and evaluated in Chapter 7.

AMirD Group Policy

This following listing describes the group policy used in all cited AMirD test environments. A throughout discussion of the construction and use of this policy is presented in Section 6.2.5.

```
% Description : This is a AMirD group policy for the Chapter 6 scenarion.
%               policy processing algorithms.
%
% Created       : August 24, 2000
% Last Modified : May 29, 2001

% Attributes Section
groupid    := < amird >;

% Antigone Group-based Policy Derivation
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

% 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;
leave_resp : 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=rc4,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=rc4)),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=$id) :: accept;
leave_resp : grouptype(mobileuser), credential(&ky,name=$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=rc4,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=rc4)),config(kekkey(crypt=rc4)));
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=$id) :: accept;
leave_resp : grouptype(coalition), credential(&ky,name=$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),
```

```
99          credential(&cert,subject_CN=$id,issuer_CN=$ca.subject_CN) :: accept;
100 shutdown : grouptype(coalition), credential(&ky,name=$gid) :: accept;
101 key_dist : grouptype(coalition), config(kekkey()),
102            credential(&ky,name=kek) :: accept;
103 key_dist : grouptype(coalition), credential(&ky,name=$id) :: accept;
104 rekey : grouptype(coalition), config(kekkey()),
105         credential(&ky,name=kek) :: accept;
106 rekey : grouptype(coalition),credential(&ky,name=$gid) :: accept;
107 send : grouptype(coalition), credential(&ky,name=$gid) :: accept;
108 content_auth : credential(&ca,issuer_CN=Antigone_Content_CA),
109                credential(&cert,subject_CN=$id,issuer_CN=$ca.subject_CN) :: accept;
110 group_mon : grouptype(coalition), credential(&cred,name=$grp) :: accept;
111 member_mon : grouptype(coalition), credential(&cred,name=$id) :: accept;
112
113 % Scenario 4 - Website Mirroring
114 webprov : :: wath, wmem, wkey, wdat;
115 wath : :: config(sslauth(interval=10,retries=2,crypt=blowfish,cafile=$author));
116 wmem : :: config(amember(memdist=none,ejectenabled=false));
117 wkey : isControlGroup() :: config(agkmkey(kychlen=64,rekeyperiod=60,hash=sha1));
118 wkey : :: config(agkmkey(crypt=blowfish,hash=sha1));
119 wdat : isControlGroup() ::
120        config(adathndlr(integ=true,sauth=true,conf=false,satype=signpkt,hash=sha1));
121 wdat : isSensitiveSite($author) ::
122        config(adathndlr(integ=true,sauth=true,conf=true,hash=sha1)), sapl;
123 wdat : :: config(adathndlr(integ=true,sauth=true,conf=false,hash=sha1)), sapl;
124 sapl : :: config(adathndlr(satype=online,frmsize=15,datfwd=50));
125
126 member_auth : grouptype(website), credential(&ca,issuer_CN=$authorid),
127               credential(&cert,subject_CN=$id,issuer_CN=$ca.subject_CN) :: accept;
128 join : grouptype(website), credential(&ky,name=$id) :: accept;
129 leave : grouptype(website), credential(&ky,name=$id) :: accept;
130 leave_resp : grouptype(website), credential(&ky,name=$id) :: accept;
131 shutdown : grouptype(website), credential(&ky,name=$gid) :: accept;
132 key_dist : grouptype(website), config(kekkey()),
133            credential(&ky,name=kek) :: accept;
134 key_dist : grouptype(website), credential(&ky,name=$id) :: accept;
135 rekey : grouptype(website), config(kekkey()),
136         credential(&ky,name=kek) :: accept;
137 rekey : grouptype(website),credential(&ky,name=$gid) :: accept;
138 send : grouptype(website), credential(&ky,name=$gid) :: accept;
139 content_auth : grouptype(website), credential(&ca,issuer_CN=$authorid),
140                credential(&cert,subject_CN=$id,issuer_CN=$ca.subject_CN) :: accept;
141 group_mon : grouptype(website), credential(&cred,name=$grp) :: accept;
142 member_mon : grouptype(website), credential(&cred,name=$id) :: accept;
143
144 % Monitoring
145 monitor : envvar(TKSCOPE,TRUE) ::
146           config(ascope(tclmon=true,log=scope.log,update=250));
147 monitor : :: config(ascope(tclmon=false));
148
149 % AMirD Application Policies
150 application : :: config(applic(followsymlinks=true)), apsauth;
151 apsauth : grouptype(locallan) ::
152           config(applic(maxexportsubgroups=10, maximportsubgroups=10));
153 apsauth : :: config(applic(maxexportsubgroups=5, maximportsubgroups=5));
```

AMirD Local Policy

The following listing describes the local policy used by the initiator and all clients in the AMirD test environment.

```
%
% Description : This is a test local policy used for testing the policy
%               processing algorithms.
%
% Created       : August 22, 2000
% Last Modified : August 22, 2000

% Attributes Section

% Policy Section

% Scenario 1 - Local Lan

provision: grouptype(locallan)   ::
           config(nullauth()),config(amember());

% Scenario 2 - Mobile User

provision : grouptype(mobileuser) ::
            config(sslauth(crypt=rc4,cafile=ssl_ca)),
            config(amember(retries=5,interval=5)),
            config(agkmkey(crypt=blowfish)),
            config(adathndlr(hash=sha1));

% Scenario 3 - Coalition

provision : grouptype(coalition), isDownloadGroup()  ::
            config(sslauth(crypt=rc4,cafile=ssl_ca)),
            config(amember(retries=5,interval=5)),
            config(kekkey(crypt=rc4));
provision : grouptype(coalition), isControlGroup()  ::
            config(sslauth(crypt=rc4,cafile=ssl_ca)),
            config(amember(retries=5,interval=5));

wath : :: config(sslauth(interval=10,retries=2,crypt=blowfish,cafile=$author));
wmem : :: config(amember(memdist=none,ejectenabled=false));

% Scenario 4 - Website Mirroring

provision : grouptype(website) ::
            config(sslauth(crypt=blowfish,cafile=$author)), config(amember());

% Authorization and Access Control

group_auth : config(nullauth()) :: accept;
group_auth : grouptype(mobileuser), credential(&ca,issuer_CN=Antigone_SSL_CA),
```

AMirD Initiator Configuration File

The following listing describes the complete initiator AMirD configuration file used by all clients in AMirD testing. The initiator acts as the exporter for all filesystems in the test environment.

```
#
# File : initaitor.conf
#
# Description : Configuration file for AMirD initator
#
#

#
# Configuration section
[config]
grppol = amird_scen.apd
locpol = amird_lscen.apd
exporter=initiator
isserver=true
transport=symmetric
mcaddr=224.27.27.1
mcport=6500
srvraddr=192.168.27.4
srvrport=6400
updateperiod=60
authoritycert=ssl_ca
mcloopback=true
sgtransport=symmetric
sgaddr=224.27.27.2
sgport=6500
sgsrvraddr=192.168.27.4
sgsrvrport=7400

#
# Exports section
[Exports]
/usr/local/experiments/amird/exportsa amird_scen.apd amird_lscen.apd ssl_ca
/usr/local/experiments/amird/exportsb amird_scen.apd amird_lscen.apd ssl_ca

#
# Imports section
[Imports]
```

AMirD Client Configuration File

The following listing describes the complete client AMirD configuration file used by all clients in AMirD testing. Clients act as the importers for all filesystems in the test environment.

```
#
# File : member1.conf
#
# Description : Configuration file for AMirD client.
#
#

#
# Configuration section
[config]
grppol = amird_scen.apd
locpol = amird_lscen.apd
transport=symmetric
mcaddr=224.27.27.1
mcport=6500
isserver=false
updateperiod=60
mcloopback=true
srvraddr=192.168.27.4
srvrport=6400
sgtransport=symmetric
sgaddr=224.27.27.2
sgport=6500
sgsrvraddr=192.168.27.4
sgsrvrport=7400
authoritycert=ssl_ca

#
# Exports section
[Exports]

#
# Imports section
[Imports]
initiator:/usr/local/experiments/amird/exportsa \
    /usr/local/experiments/amird/importsa1 amird_lscen.apd ssl_ca
initiator:/usr/local/experiments/amird/exportsb \
    /usr/local/experiments/amird/importsb1 amird_lscen.apd ssl_ca
```

REFERENCES

[AAC+99] A. Adamson, C. Antonelli, K. Coffman, P. D. McDaniel, and J. Rees. Secure Distributed Virtual Conferencing. In *Proceedings of Communications and Multimedia Security (CMS '99)*, pages 176–190. September 1999. URL `http://www.eecs.umich.edu/ pdmcdan/docs/cms99.pdf`, katholieke Universiteit Leuven, Belgium.

[ABG+98] C. Alaettinouglu, T. Bates, E. Gerich, D. Karrenberg, D. Meyer, M. Terpstra, and C. Villamizer. Routing Policy Specification Language (RPSL). *Internet Engineering Task Force*, January 1998. RFC 2280.

[AN96] M. Abadi and R. Needham. Prudent Engineering Practice for Cryptographic Protocols. *IEEE Transactions on Software Engineering*, 22(1):6–15, January 1996.

[AP98] R. J. Anderson and F. A. P. Petitcolas. On the Limits of Steganography. *IEEE Journal on Selceted Areas in Communications*, 16(4):474–481, May 1998.

[AS98] Y. Amir and J. Stanton. The Spread Wide Area Group Communication System. Technical Report CNDS-98-4, The Center for Networking and Distributed Systems, The Johns Hopkins University, 1998.

[AST00] G. Ateniese, M. Steiner, and G. Tsudik. New Multi-Party Authentication Services and Key Agreement Protocols. *IEEE Journal of Selected Areas in Communication*, 18, March 2000.

[Bal96] A. Ballardie. Scalable Multicast Key Distribution. *Internet Engineering Task Force*, May 1996. RFC 1949.

[Bal97a] A. Ballardie. Core Based Trees (CBT) Multicast Routing Architecture. *Internet Engineering Task Force*, September 1997. RFC 2201.

[Bal97b] A. Ballardie. Core Based Trees (CBT version 2) Multicast Routing. *Internet Engineering Task Force*, September 1997. RFC 2189.

[BB00] D. Branstad and D. Balenson. Policy-Based Cryptographic Key Management: Experience with the KRP Project. In *Proceedings of DARPA Information Survivability Conference and Exposition (DISCEX '00)*, pages 103–114. DARPA, January 2000. Hilton Head, S.C.

[BBD+99] D. Balenson, D. Branstad, P. Dinsmore, M. Heyman, and C. Scace. Cryptographic Context Negotiation Template. Technical Report TISR #07452-2, TIS Labs at Network Associates, Inc., February 1999.

[BCG+00] A. Basso, C. Cranor, R. Gopalakrishnan, M. Green, C. Kalmanek, D. Shur, S. Sibal, C. Sreenan, and J. van der Merwe. PRISM, an IP-Based Architecture for Broadband Access to TV and Other Streaming Media. In *Proceedings of International Workshop on Network and Operating System Support for Digital Audio and Video (NOSSDAV)*. IEEE, June 2000.

[BD96] M. Burmester and Y. Desmedt. Efficient and Secure Conference Key Distribution. In *Proceedings of 1996 Cambridge Workshop on Security Protocols*, pages 119–129. April 1996.

[Bee97] T. Beever. Personal communication, November 1997. National Aeronautical and Space Administration, Kennedy Space Center.

[Ber96] P. A. Bernstein. Middleware: A Model for Distributed System Services. *Communications of the ACM*, 39(2):86–98, February 1996.

[BF99] B. Briscoe and I. Fairman. Nark: Receiver-based Multicast Non-repudiation and Key Management. In *Proceedings of E-commerce '99*. ACM, Denver, Colorado, June 1999.

[BFC93] T. Ballardie, P. Francis, and J. Crowcroft. Core Based Trees (CBT). In *Proceedings of ACM SIGCOMM '93*, pages 85–95. ACM, September 1993.

[BFIK99a] M. Blaze, J. Feigenbaum, J. Ioannidis, and A. Keromytis. The Role of Trust Management in Distributed Systems Security. In *In Secure Internet Programming: Issues in Distributed and Mobile Object Systems*, volume 1603, pages 185–210. Springer-Verlag Lecture Notes in Computer Science State-of-the-Art series, 1999. New York, NY.

[BFIK99b] M. Blaze, J. Feignbaum, J. Ioannidis, and A. Keromytis. The KeyNote Trust Management System - Version 2. *Internet Engineering Task Force*, September 1999. RFC 2704.

[BFL96] M. Blaze, J. Feigenbaum, and J. Lacy. Decentralized Trust Management. In *Proceedings of the 1996 IEEE Symposium on Security and Privacy*, pages 164–173. November 1996. Los Alamitos.

[BH99] D. C. Blight and T. 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.

[BHHW01] M. Baugher, T. Hardjono, H. Harney, and B. Weis. Domain of Interpretation for ISAKMP (*Draft*). *Internet Engineering Task Force*, February 2001. `draft-ietf-msec-gdoi-00.txt`.

[BHSC98] N. T. Bhatti, M. A. Hiltunen, R. D. Schlichting, and W. Chiu. Coyote: A System for Constructing Fine-Grain Configurable Communication Services. *ACM Transactions on Computer Systems*, 16(4):321–366, November 1998.

[Bir93] K. Birman. The Process Group Approach to Reliable Distributed Computing. *Communications of the ACM*, 36(12):37–53, December 1993.

[BL73] D. Bell and L. LaPadula. Secure Computer Systems: Mathematical Foundations and Model. Technical Report M74-244, MITRE Corperation, Bedford, MA, 1973.

[Bon94] J. Bonwick. The slab allocator: An object-caching kernel memory allocator. In *USENIX Summer*, pages 87–98. 1994. URL `citeseer.nj.nec.com/bonwick94slab.html`.

[Bri99] B. Briscoe. MARKS: Zero Side-Effect Multicast Key Management Using Arbitrarily Revealed Key Sequences. In *Proceedings of First International Workshop on Networked Group Communication*. November 1999.

[BW98] C. Becker and U. Wille. Communication Complexity of Group Key Distribution. In *Proceedings of 5th ACM Conference on Computer and Communications Security*. ACM, November 1998. San Francisco, California.

[CC89] G.-H. Chiou and W.-T. Chen. Secure Broadcasting Using the Secure Lock. *IEEE Transactions on Software Engineering*, 15(8):929–934, 1989.

[CC97] 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.

[CEK+99] I. Chang, R. Engel, D. Kandlur, D. Pendarakis, and D. Saha. Key Management for Secure Internet Multicast using Boolean Function Minimization Techniques. In *Proceedings of IEEE Infocom 1999*, volume 2, pages 689–698. IEEE, March 1999.

[CFL+98] Y. Chu, J. Feigenbaum, B. LaMacchia, P. Resnick, and M. Strauss. REFEREE: Trust Management for Web Applications. In *Proceedings of Financial Cryptography '98*, volume 1465, pages 254–274. Anguilla, British West Indies, February 1998.

[CGI+99] R. Canetti, J. Garay, G. Itkis, D. Micciancio, M. Naor, and B. Pinkas. Multicast Security: A Taxonomy and Efficient Constructions. In *Proceedings of IEEE Infocom 1999*, volume 2, pages 708–716. IEEE, March 1999. New York, New York.

[Che97] S. Cheung. An Efficient Message Authentication Scheme for Link State Routing. In *13th Annual Computer Security Applications Conference*, pages 90–98. 1997. San Diego, California.

[CMB00] Y. Chawathe, S. McCanne, and E. Brewer. RMX: Reliable Multicast in Heterogeneous Network. In *Proceedings of IEEE INFOCOM 2000*, pages 795–804. IEEE, March 2000. Tel Aviv, Israel.

[Coo71] S. Cook. The Complexity of Theorem-Proving Procedures. In *Proceedings of 3th Annual ACM Symposium on Theorey of Computing*, pages 151–158. ACM, 1971.

[CP00] R. Canetti and B. Pinkas. A Taxonomy of Multicast Security Issues (*Draft*). *Internet Engineering Task Force*, August 2000. `draft-irtf-smug-taxonomy-01.txt`.

[Cri91] F. Cristian. Reaching Agreement on Processor Group Membership in Synchronous Distributed Systems. *Distributed Computing*, (4):175–187, October 1991.

[CRZ00] Y. Chu, S. G. Rao, and H. Zhang. A Case For End System Multicast. In *Proceedings of ACM Sigmetrics*, pages 1–12. ACM, June 2000. Santa Clara,CA.

[DBC+00] 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.

[DBH+00] 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.

[Dee89] S. Deering. Host Extensions for IP Multicasting. *Internet Engineering Task Force*, August 1989. RFC 1112.

[DEF+99] S. Deering, D. Estrin, D. Farinacci, V. Jacobson, A. Helmy, D. Meyer, and L. Wei. Protocol Independent Multicast Version 2 Dense Mode Specification (*Draft*). *Internet Engineering Task Force*, June 1999. `draft-ietf-pim-v2-dm-03.txt`.

[DH76] W. Diffie and M. Hellman. New Directions in Cryptography. *IEEE Transactions on Information Theory*, IT-22(6):644–654, November 1976.

[DM96] D. Dolev and D. Malki. The Transis Approach to High Availibility Cluster Communication. *Communications of the ACM*, 39(4), April 1996.

[DR98] J. Daemen and V. Rijmen. AES Proposal: Rijndael. AES submission, June 1998. `http://csrc.nist.gov/encryption/aes/rijndael/Rijndael.prf`.

[DR00] J. Daemen and V. Rijmen. The Block Cipher Rijndael. In Quisquater and B. Schneier, editors, *In Proceedings of Smart Card Research and Applications*, volume LNCS 1820, pages 288–296. Springer, 2000.

[EFH+98] D. Estrin, D. Farinacci, A. Helmy, D. Thalerxs, S. Deering, M. Handley, V. Jacobson, C. Liu, P. Sharma, and L. Wei. Protocol Independent Multicast-Sparse Mode (PIM-SM). *Internet Engineering Task Force*, June 1998. RFC 2362.

[EGM96] S. Even, O. Goldreich, and S. Micali. On-Line/Off-Line Digital Signatures. *Journal of Cryptology*, 9(1):35–67, 1996.

[EIS76] S. Evan, A. Itai, and A. Shamir. On the Complexity of Timetable and Multicommodity Flow Problems. *SIAM Journal of Computing*, (5):691–703, 1976.

[FF96] K. Fall and S. Floyd. Simulation-Based Comparisons of Tahoe, Reno, and SACK TCP. *ACM Computer Communication Review*, 26(3):5–21, July 1996.

[FJL+97] 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.

[FKTT98] I. Foster, C. Kesselman, G. Tsudik, and S. Tuecke. A Security Architecture for Computational Grids. In *Proceedings of the 5th ACM Conference on Computer and Communications Security*, pages 83–92. ACM, 1998.

[FLP85] M. Fischer, N. Lynch, and M. Paterson. Impossibility of Distributed Consensus with One Faulty Process. *Journal of the ACM*, 32(2):374–382, 1985.

[FN93] A. Fiat and M. Naor. Broadcast Encryption. In *Proceedings of CRYPTO '93*, pages 480–491. 1993.

[Fra99] P. Francis. Yoid: Extending the Multicast Internet Architecture, September 1999. `http://www.yallcast.com`.

[Gan22] M. Gandhi, March 1922. Letter, 8 Mar 1922, to the general secretary of the Congress Party, India.

[GHR95] R. Greenlaw, H. Hoover, and W. Ruzzo. *Limits to Parallel Computation: P-Completeness Theory*. Oxford University Press, New York, Oxford, first edition, 1995.

[GJ79] M. R. Garey and D. S. Johnson. *Computers and Intractibility, A Guide to the Theory of NP-Completeness*. W. H. Freeman and Co., New York, NY, first edition, 1979.

[GJS76] M. R. Garey, D. S. Johnson, and L. Stockmeyer. Some Simplified NP-Complete Graph Problems. *Theoretical Computer Science*, (1):237–267, 1976.

[Gon96] L. Gong. Enclaves: Enabling Secure Collaboration Over the Internet. In *Proceedings of the Sixth USENIX Security Symposium*, pages 149–159. USENIX Association, July 1996.

[GQ94] L. Gong and X. Qian. The Complexity and Composability of Secure Interoperation. In *Proceedings of the IEEE Symposium on Research in Security and Privacy*, pages 190–200. IEEE, Oakland, California, May 1994.

[GR97] R. Gennaro and P. Rohatgi. How to Sign Digital Streams. In *Proceedings of CRYPTO 97*, pages 180–197. August 1997. Santa Barbara, CA.

[Gro00] T. O. Group. OpenSLL, May 2000. `http://http://www.openssl.org/`.

[Gro01] S.-S. M. W. Group. Source-Specific Multicast Charter, March 2001. Internet Engineering Task Force,
`http://www.ietf.org/html.charters/ssm-charter.html`.

[Hal94] N. Haller. The S/Key^{tm} One-Time Password System. In *Proceedings of 1994 Internet Society Symposium on Network and Distributed System Security*, pages 151–157. February 1994. San Diego, CA.

[HC01] H. Harney and A. Colegrove. The GSKKMP Security Policy Token with IPSec. *Proceedings of the Internet Society Symposium on Network and Distributed System Security*, February 2001. *(to appear)*.

[HCD00] T. Hardjono, B. Cain, and N. Doraswamy. A Framework for Group Key Management for Multicast Security (*Draft*). *Internet Engineering Task Force*, February 2000. `draft-ietf-ipsec-gkmframework-02.tx`.

[HCH+00] H. Harney, A. Colegrove, E. Harder, U. Meth, and R. Fleischer. Group Secure Association Key Management Protocol (*Draft*). *Internet Engineering Task Force*, June 2000.
`draft-harney-sparta-gsakmp-sec-02.txt`.

[HCM01] H. Harney, A. Colegrove, and P. McDaniel. Principles of Policy in Secure Groups. In *Proceedings of Network and Distributed Systems Security 2001*. Internet Society, February 2001. URL `http://www.eecs.umich.edu/ pdmcdan/docs/ndss01.pdf`.

[HFPS99] 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.

[Hil98] M. Hiltunen. Configuration Management for Highly-Customizable Software. *IEE Proceedings: Software*, 145(5):180–188, 1998.

[HJSU00] M. Hiltunen, S. Jaiprakash, R. Schlichting, and C. Ugarte. Fine-Grain Configurability for Secure Communication. Technical Report TR00-05, Department of Computer Science, University of Arizona, June 2000.

[HM97a] H. Harney and C. Muckenhirn. Group Key Management Protocol (GKMP) Architecture. *Internet Engineering Task Force*, July 1997. RFC 2094.

[HM97b] H. Harney and C. Muckenhirn. Group Key Management Protocol (GKMP) Specification. *Internet Engineering Task Force*, July 1997. RFC 2093.

[HMPT89] N. Hutchinson, S. Mishra, L. Peterson, and V. Thomas. Tools for Implementing Network Protocols. *Software - Practice and Experience*, 19(9):895–916, December 1989.

[HP94] N. Hutchinson and L. Peterson. The x-Kernel: An Architecture for Implementing Network Protocols. *IEEE Transactions on Software Engineering*, 17(1):64–76, January 1994.

[HS98] M. Hiltunen and R. Schlichting. A Configurable Membership Service. *IEEE Transactions on Computers*, 47(5):573–586, May 1998.

[HWC95] M. J. Handley, I. Wakeman, and J. Crowcroft. CCCP: Conference Control Channel Protocol: A Scalable Base for Building Conference Control Applications. In *Proceedings of ACM SIGCOMM '95*. ACM, 1995.

[IKBS00] S. Ioannidis, A. D. Keromytis, S. Bellovin, and J. M. Smith. Implementing a Distributed Firewall. In *Proceedings of Computer and Communications Security (CCS) 2000*, pages 190–199. 2000. Athens, Greece.

[J81] P. J. Transmission Control Protocol - DARPA Internet Protocol Program Specification. *Internet Engineering Task Force*, September 1981. RFC 793.

[JGJ+00] J. Janotti, D. Gifford, K. Johnson, K. M, and O. J. Overcast: Reliable Mutlicasting with and Overlay Network. In *4th USENIX Symposium on Operating System Design and Implementation (OSDI 2000)*, page (*to appear*). USENIX, October 2000. Santa Clara,CA.

[JSS97] S. Jajodia, P. Samarati, and V. Subrahmanian. A Logical Language for Expressing Authorizations. In *In Proceedings of the 1997 IEEE Symposium on Security and Privacy*, pages 31–42. IEEE, Oakland, CA, March 1997.

[KA98] S. Kent and R. Atkinson. Security Architecture for the Internet Protocol. *Internet Engineering Task Force*, November 1998. RFC 2401.

[KBC97] H. Krawczyk, M. Bellare, and R. Canetti. HMAC: Keyed-Hashing for Message Authentication. *Internet Engineering Task Force*, April 1997. RFC 2104.

[Ken93] S. Kent. Internet Privacy Enhanced Mail. *Communications of the ACM*, 36(8):48–60, August 1993.

[Koc98] P. Kocher. On Certificate Revocation and Validation. In R. Hirschfeld, editor, *Financial Cryptography*, volume 1465, pages 172–177. Springer, Anguilla, British West Indies, February 1998.

[KR96] J. Kilian and P. Rogaway. How to Protect DES Against Exhaustive Key Search. In *Proceedings of Crypto '96*, pages 252–267. August 1996.

[Kra90] J. Kramer. Configuration Programming - A Framework for the Development of Distributable Systems. In *Proceedings of IEEE International Conference on Computer Systems and Software Engineering (CompEuro 90)*, pages 374–384. May 1990. Tel-Aviv, Israel.

[KT91] M. Kaashoek and A. Tanenbaum. Group communication in the Amoeba Distributed Operating System. In *Proceedings of the 11th International Conference on Distributed Computing Systems*, pages 222–230. IEEE, May 1991.

[KT93] R. Kakehi and M. Tokoro. A Negotiation Protocol for Conflict Resolution in Multi-Agent Enviroments. In *In Proceedings of International Conference On Intelligent and Cooperative Information Systems*, pages 185–196. May 1993.

[LABW92] B. Lampson, M. Abadi, M. Burrows, and E. Wobber. Authentication in Distributed Systems: Theory and Practice. *ACM Transactions Computer Systems*, 10(4):265–310, November 1992.

[Lam81] L. Lamport. Password Authentication with Insecure Communication. *Commuications of the ACM*, 24(11):770–772, November 1981.

[Lew78] H. R. Lewis. Satisfiability Problems for Propositional Calculi, 1978. *Unpublished manuscript.*

[LKvR+99] X. Liu, C. Kreitz, R. van Renesse, J. Hickey, M. Hayden, K. Birman, and R. Constable. Building Reliable High-Performance Communication Systems from Components. In *Proceedings of 17th ACM Symposium on Operating Systems Principles (SOSP'99)*, volume 33, pages 80–92. ACM, 1999.

[LM94] T. Leighton and S. Micali. Secret-key Agreement without Public-Key Cryptography. In *Proceedings of Crypto 93*, pages 456–479. August 1994.

[MC94] C. W. M. Chang. A Speech-Act Based Negotiation Protocol: Design, Implementation and Test Use. *ACM Transactions on Information Systems*, 12(4), February 1994.

[MHC+00] P. McDaniel, H. Harney, A. Colegrove, A. Prakash, and P. Dinsmore. Multicast Security Policy Requirements and Building Blocks (*Draft*). Internet Research Task Force, Secure Multicast Research Group (SMuG), November 2000. URL `http://www.ietf.org/internet-drafts/draft-irtf-smug-polreq-00.txt`, (`draft-irtf-smug-polreq-00.txt`).

[MHDP00] P. McDaniel, H. Harney, P. Dinsmore, and A. Prakash. Multicast Security Policy (*Draft*). Internet Research Task Force, Secure Multicast Research Group (SMuG), June 2000. URL `http://www.ietf.org/internet-drafts/draft-irtf-smug-mcast-policy-00.txt`, (`draft-irtf-smug-mcast-policy-00.txt`).

[MHP98] P. McDaniel, P. Honeyman, and A. Prakash. Lightweight Secure Group Communication. Technical Report 98-2, Center for Information Technology Integration, University of Michigan, April 1998. URL `http://www.citi.umich.edu/techreports/reports/citi-tr-98-2.pdf`.

[Mit97] S. Mittra. Iolus: A Framework for Scalable Secure Multicasting. In *Proceedings of ACM SIGCOMM '97*, pages 277–278. ACM, September 1997.

[MJ00] P. McDaniel and S. Jamin. Windowed Certificate Revocation. In *Proceedings of IEEE INFOCOM 2000*, pages 1406–1414. IEEE, March 2000. URL `http://www.eecs.umich.edu/ pdmcdan/docs/info00.pdf`, tel Aviv, Israel.

[MJV96] S. McCanne, V. Jacobson, and M. Vetterli. Receiver-Driven Layered Multicast. In *Proceedings of ACM SIGCOMM '96*, pages 117–130. ACM, September 1996.

[MKM94] J. Magee, J. Kramer, and M.Sloman. Regis: A Constructive Development Environment for Distributed Programs. *Distributed Systems Engineering Journal*, 1(5):663–675, 1994.

[Moy94] J. Moy. Multicast Extensions to OSPF. *Internet Engineering Task Force*, March 1994. RFC 1585.

[Moy98] J. Moy. OSPF Version 2. *Internet Engineering Task Force*, April 1998. RFC 2328.

[MP00] P. McDaniel and A. Prakash. Lightweight Failure Detection in Secure Group Communication. Technical Report CSE-TR-428-00, Electrical Engineering and Computer Science, University of Michigan, June 2000. URL `docs/CSE-TR-428-00.pdf`.

[MPH99] 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. URL `http://www.eecs.umich.edu/ pdmcdan/docs/usec99.pdf`.

[MPI+01] 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. URL `http://www.eecs.umich.edu/ pdmcdan/docs/discex01.pdf`.

[MQRG97] M. Moriconi, X. Qian, R. A. Riemenschneider, and L. Gong. Secure Software Architectures. In *In Proceedings of the 1997 IEEE Symposium on Security and Privacy*, pages 84–93. May 1997.

[MR00] 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. URL `http://www.eecs.umich.edu/ pdmcdan/docs/finc00.pdf`, anguilla, British West Indies.

[MS98] D. McGrew and A. Sherman. Key Establishment in Large Dynamic Groups Using One-Way Function Trees. Technical Report TIS Report No. 0755, TIS Labs at Network Associates, Inc., May 1998. Glenwood, MD.

[Mul93] S. Mullender. *Distributed Systems*. Addison-Wesley, First edition, 1993.

[Nat77] National Bureau of Standards. Data Encryption Standard. *Federal Information Processing Standards Publication*, 1977.

[Nat80] National Bureau of Standards. DES Modes of Operation - FIPS PUB 81. *Federal Information Processing Standards Publication*, December 1980.

[Nat99] National Bureau of Standards. Data Encryption Standard (DES) - FIPS PUB 46-3. *Federal Information Processing Standards Publication*, December 1999.

[NK98] P. Nikander and A. Karila. A Java Beans Component Architecture for Cryptographic Protocols. In *Proceedings of 7th USENIX UNIX Security Symposium*, pages 107–121. USENIX Association, January 1998. San Antonio, Texas.

[NN98] M. Naor and K. Nassim. Certificate Revocation and Certificate Update. In *Proceedings of the 7th USENIX Security Symposium*, pages 217–228. January 1998.

[NS78] R. Needham and M. Schroeder. Using Encryption for Authentication in Large Networks of Computers. *Communications of the ACM*, 21:393–399, 1978.

[NT94] B. C. Neuman and T. Ts'o. Kerberos: An Authentication Service for Computer Networks. *IEEE Communications*, pages 33–38, September 1994.

[OKP00] J. Ott, D. Kutscher, and C. Perkins. The Message Bus: A Platform for Component-based Conferencing Applications. In *Proceedings of CBG2000: The CSCW2000 workshop on Component-based Groupware*. December 2000. Philadelphia, PA.

[OOSS94] H. Orman, S. O'Malley, R. Schroeppel, and D. Schwartz. Paving the Road to Network Security or the Value of Small Cobblestones. In *Proceedings of the 1994 Internet Society Symposium on Network and Distributed System Security*. February 1994.

[Paga] U. M. Page. `kill` man page. Linux 2.2.18, Section 1.

[Pagb] U. M. Page. `malloc` man page. Kernel 2.2.18, Section 3.

[Pagc] U. M. Page. `signal` man page. Linux 2.2.18, Section 7.

[PCB+00] A. Perrig, R. Canetti, B. Briscoe, D. Tygar, and D. Song. TESLA: Multicast Source Authentication Transform. *Internet Engineering Task Force*, July 2000. `draft-irtf-smug-tesla-00.txt`.

[PCKS01] 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*).

[Per97] R. Perlman. LKH+, 1997. Observation concerning LKH [WHA98] from the conference floor.

[Pos80] J. Postel. User Datagram Protocol. *Internet Engineering Task Force*, August 1980. RFC 768.

[Pos81] J. Postel. Internet Protocol. *Internet Engineering Task Force*, August 1981. RFC 791.

[Pow92] D. Powell. Failure Mode Assumptions and Assumption Coverage. In *Proceedings of 22nd International Symposium on Fault-Tolerant Computing (FTCS-22)*, pages 386–395. IEEE, July 1992. Boston, MA.

[Pro00] N. Provos. Encrypting Virtual Memory. In *Proceedings of the 9th USENIX Security Symposium*, pages 35–44. USENIX Association, August 2000. Denver, CO.

[PSTC00] A. Perrig, D. Song, D. Tygar, and R. Canetti. Efficient Authentication and Signature of Multicast Streams over Lossy Channels. In *2000 IEEE Symposium on Security and Privacy*, pages 56–70. IEEE, May 2000. Oakland, CA.

[Pur94] J. M. Purtilo. The POLYLITH software bus. *ACM Transactions on Programming Languages and Systems*, 16(1):151–174, January 1994.

[RBH+98] O. Rodeh, K. Birman, M. Hayden, Z. Xiao, and D. Dolev. Ensemble Security. Technical Report TR98-1703, Cornell University, September 1998.

[RBM96] R. V. Renesse, K. Birman, and S. Maffeis. Horus: A Flexible Group Communication System. *Communications of the ACM*, 39(4):76–83, April 1996.

[Rei94] M. Reiter. Secure Agreement Protocols: Reliable and Atomic Group Multicast in Rampart. In *Proceedings of 2nd ACM Conference on Computer and Communications Security*, pages 68–80. ACM, November 1994.

[Riv92a] R. Rivest. The MD5 Message Digest Algorithm. *Internet Engineering Task Force*, April 1992. RFC 1321.

[Riv92b] R. Rivest. The RC4 Encryption Algorithm. Technical Report Document No. 003-013005-100-000-000, RSA Data Security Inc., March 1992.

[Riz97] L. Rizzo. Effective Erasure Codes for Reliable Computer Communication Protocols. *ACM Computer Communications Review*, 27(2):24–36, April 1997.

[RN00] 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. DARPA, Hilton Head, South Carolina, January 2000.

[Roh99] P. Rohatgi. A Compact and Fast Hybrid Signature Scheme for Multicast Packet Authentication. In *Proceedings of 6th ACM Computer and Communications Security Conference*. ACM, November 1999. Singapore.

[RR98] M. K. Reiter and A. D. Rubin. Crowds: Anonymity for Web Transactions. *ACM Transactions on Information and System Security*, 1(1):66–92, 1998.

[RSA78] R. Rivest, A. Shamir, and L. Adleman. A Method for Obtaining Digital Signatures and Public-Key Cryptosystems. *Communications of the ACM*, 21(2):120–126, February 1978.

[Rub96] A. D. Rubin. Independent One-Time Passwords. *USENIX Journal of Computer Systems*, 9(1):15–27, February 1996.

[RVR93] L. Rodrigues, P. Verissimo, and J. Rufino. A Low Level Processor Group Membership Protocol for LANS. In *Proceedings of 13th International Conference on Distributed Computing Systems*, pages 541–550. 1993.

[SBCY97] R. Sandhu, V. Bhamidipati, E. Coyne, and S. G. C. Youman. The ARBAC97 Model for Role-based Administration of Roles: Preliminary Description and Outline. In *Proceedings of 2nd ACM Workshop on Role-Based Access Control*. November 1997. Fairfax, Virginia.

[SC98] L. Sanchez and M. Condell. Security Policy System (*Draft*). *Internet Engineering Task Force*, November 1998. `draft-ietf-ipsec-sps.txt`.

[SCFY96] R. S. Sandhu, E. J. Coyne, H. L. Feinstein, and C. E. Youman. Role-based access control models. *IEEE Computer*, 20(2):38–47, 1996.

[Sch78] T. J. Schaefer. The Complexity of Satisfiability Problems. In *Proceedings of 10th Annual ACM Symposium on Theorey of Computers*, pages 216–226. ACM, 1978. New York, New York.

[Sch94] B. Schneier. Description of a New Variable-Length Key, 64-Bit Block Cipher (Blowfish). In *Proceedings of Fast Software Encryption, Cambridge Security Workshop*, pages 191–204. Springer-Verlag, 1994.

[Sch95] H. Schulzrinne. Dynamic Configuration of Conferencing Applications using Pattern-Matching Multicast. In *Proceedings of 5th International Workshop on Net. and O.S. Support for Digital Audio and Video*, pages 231–242. 1995. Durham, New Hampshire.

[Sch96] B. Schneier. *Applied Cryptography*. John Wiley & Sons, Inc., second edition, 1996.

[SCP98] A. D. Santis, G. D. Crescenzo, and G. Persiano. Communication-Efficient Anonymous Group Identification. In *Proceedings of 5th ACM Conference on Computer and Communications Security*, pages 73–82. 1998.

[SCR96] D. Stringer-Calvert and J. Rushby. A Less Elementary Tutorial for the PVS Specification and Verification System. Technical Report CSL-95-10, Computer Science Laboratory, SRI International, August 1996.

[Ses97] R. Sessions. *COM and DCOM: Microsoft's Vision for Distributed Objects*. John Wiley and Sons, First edition, 1997. New York, NY.

[SFS93] D. Schmidt, D. Fox, and T. Sudya. Adaptive: A Dynmaically Assembled Protocol Transformation, Integration, and eValuation Environment. *Journal of Concurrency: Practice and Experience*, 5(4):269–286, June 1993.

[SGK+85] R. Sandberg, D. Goldberg, S. Kleiman, D. Walsh, and B. Lyon. Design and Implemention of the Sun Network Filesystem. In *Proceedings of the Summer USENIX Conference*, pages 119–130. USENIX, 1985.

[SKJH00] S. Setia, S. Koussih, S. Jajodia, and E. Harder. Kronos: A Scalable Group Re-keying Approach for Secure Multicast. In *2000 IEEE Symposium on Security and Privacy*, pages 215–218. IEEE, May 2000. Oakland, CA.

[SSDW98] D. Steer, L. Strawczynski, W. Diffie, and M. Wiener. A Secure Audio Teleconference System. In *Proceedings of CRYPTO '88*, pages 520–528. 1998.

[SSV01] J. Snoeyink, S. Suri, and G. Varghese. A Lower Bound for Multicast Key Distribution. In *Proceedings of IEEE Infocom 2001*. IEEE, April 2001. Anchorage, Alaska.

[Sti95] D. Stinson. *Cryptography: Theory and Practice*. CRC Press, first edition, 1995.

[SWM+99] M. Stevens, W. Weiss, H. Mahon, B. Moore, J. Strassner, G. Waters, A. Westerinen, and J. Wheeler. Policy Framework (*Draft*). *Internet Engineering Task Force*, September 1999. `draft-ietf-policy-framework-00.txt`.

[Tan95] A. S. Tanenbaum. *Distributed Operating Systems*. Prentice–Hall, Englewood Cliffs, New Jersey, first edition, 1995.

[TJM+99] M. Thompson, W. Johnson, S. Mudumbai, G. Hoo, K. Jackson, and A. Essiari. Certificate-based Access Control for Widely Distributed Resources. In *Proceedings of 8th USENIX UNIX Security Symposium*, pages 215–227. USENIX Association, August 1999. Washington D. C.

[Vin94] S. Vinoski. CORBA: Integrating Diverse Applications Within Distributed Heterogeneous Environments. *IEEE Communications Magazine*, 14(2), February 1994.

[Wal99] J. Waldo. The Jini Architecture for Network-centric Computing. *Communications of the ACM*, 3(4):76–82, July 1999.

[WCS+99] M. Waldvogel, G. Caronni, D. Sun, N. Weiler, and B. Plattner. The VersaKey Framework: Versatile Group Key Management. *IEEE Journal on Selected Areas in Communications, Special Issue on Middleware*, 17(8), August 1999.

[WGL98] C. K. Wong, M. Gouda, and S. S. Lam. Secure Group Communication Using Key Graphs. In *Proceedings of ACM SIGCOMM '98*, pages 68–79. ACM, September 1998.

[WHA98] D. M. Wallner, E. J. Harder, and R. C. Agee. Key Management for Multicast: Issues and Architectures (*Draft*). *Internet Engineering Task Force*, September 1998. `draft-wallner-key-arch-01.txt`.

[WL93] T. Woo and S. Lam. Authorization in Distributed Systems; A New Approach. *Journal of Computer Security*, 2(2-3):107–136, 1993.

[WL98] T. Woo and S. Lam. Designing a Distributed Authorization Service. In *Proceedings of INFOCOM '98*. IEEE, San Francisco, March 1998.

[WL99] C. Wong and S. Lam. Digital Signatures for Flows and Multicasts. *IEEE/ACM Transactions on Networking*, 7(4):502–513, August 1999.

[WPD88] D. Waitzman, C. Partridge, and S. Deering. Distance Vector Multicast Routing Protocol. *Internet Engineering Task Force*, November 1988. RFC 1075.

[Wra00] J. Wray. Generic Security Service API Version 2 : C-bindings. *Internet Engineering Task Force*, January 2000. RFC 2744.

[WSS+00] A. Westerinen, J. Schnizlein, J. Strassner, M. Scherling, B. Quinn, J. Perry, S. Herzog, A.-N. Huynh, and M. Carlson. Policy Terminology (*Draft*). *Internet Engineering Task Force*, July 2000. `draft-ietf-policy-terminology-00.txt`.

[WYL+99] X. Wang, X. Yi, K. Lam, C. Zhang, , and E. Okamoto. Secure Agent-Mediated Auctionlike Negotiation Protocol for Internet Retail Commerce. In LNCS, editor, *In Proceedings of the 3rd International Workshop on Cooperative Information Agents (CIA'99)*, volume 1652, pages 291–302. Springer, Stockholm, Sweden, July 1999.

[YHK95] W. Yeong, T. Howes, and S. Kille. Lightweight Directory Access Protocol. *Internet Engineering Task Force*, March 1995. RFC 1777.

[ZF94] H. Zhang and D. Ferrari. Rate-Controlled Service Disciplines. *Journal of High-Speed Networks*, 3(4), 1994.

[Zim94] P. Zimmermann. PGP User's Guide. Distributed by the Massachusetts Institute of Technology, May 1994.

[ZSC+00] 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. DARPA, Hilton Head, South Carolina, January 2000.