Flexibly Constructing Secure Groups in Antigone 2.0*

Patrick McDaniel, Atul Prakash, Jim Irrer, Sharad Mittal, Thai-Chuin Thuang
Electrical Engineering and Computer Science Department
University of Michigan, Ann Arbor
pdmcdan{aprakash,irrer,mittals,thuang}@eecs.umich.edu

Abstract

Group communication is increasingly used as a low cost building block for the development of highly available and survivable services in dynamic environments. However, contemporary frameworks often provide limited facilities for the definition and enforcement of precise security policies. This paper presents the Antigone 2.0 framework that allows the flexible specification and enforcement of group security policies. Enforcement is achieved through the policy directed composition and configuration of sets of basic security services implementing the group. We summarize the design of the Antigone 2.0 architecture, its use, and the Application Programming Interface (API). The use of the API is illustrated through two applications built on Antigone; a reliable multicast system and host level multicast security service. We conclude with a description of current status and plans for future work.

1 Introduction

Group communication is increasingly used as an efficient building block for distributed systems. However, the cost and complexity of providing properties such as reliability, survivability, and security within a group is significantly higher than in peer communication. These costs are due to the additional number of failure modes, heterogeneity of the group members, and the increased vulnerability to compromise. Because of these factors, it is important to identify precisely the properties appropriate for a particular session.

The properties required by a session are defined through a group *policy*. Policy may be stated either explicitly through a policy specification or implicitly by an implementation. Contemporary group communication platforms operate from an largely fixed set of policies. These implicitly defined policies represent the threat and trust models appropriate for a set of target environments. However, an application and session whose security requirements are not directly addressed by the framework must implement additional infrastructure or modify their security model. Thus, these applications would benefit from frameworks allowing the explicit definition, distribution, and subsequent enforcement of security policies appropriate for the runtime environment.

In this paper we describe the design and use of the Antigone 2.0 system. Antigone provides flexible interfaces for the definition and implementation of security policies through the composition and configuration of security mechanisms. The set of services and protocols used to implement the group is developed from a systematic analysis of the properties appropriate for a given session in conjunction with operational conditions and participant requirements. The resulting session defining policy is distributed to all group participants and enforced uniformly at each host.

In Antigone, we define a group policy as the specification of all security relevant properties of the session. Thus, a group policy states how security directs behavior, the entities allowed to participate, and the mechanisms used to achieve security objectives. This view of policy affords a greater degree of coordination than found in extant systems; statements of authorization and access control, key management, data security, and other aspects of the group are defined within a single unifying policy.

Policy has been used in different contexts as a vehicle for representing authorization and access control [4, 7, 33, 28], peer session security [35], quality of service guarantees [6], and network configuration [31]. These approaches define a policy language or schema appropriate for their target problem domain. Antigone expands on this work by defining an approach in which policy is used to provision and regulate the services supporting communication. Furthermore, group participants can determine the compliance of the group definition with local requirements.

*This work is supported in part by the Defense Advanced Research Projects Agency (DARPA) and Air Force Research Laboratory, Air Force Materiel Command, USAF, under agreement number F30602-00-2-0508. The U.S. Government is authorized to reproduce and distribute reprints for Governmental purposes notwithstanding any copyright annotation thereon. The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of the Defense Advanced Research Projects Agency (DARPA), the Air Force Research Laboratory, or the U.S. Government.

Recent systems have adopted a more flexible definition of security policy. For example, the Security Policy System [35] provides interfaces for the flexible definition of security policies for IPSec [18] connections. These policies specify precisely the kinds of security mechanisms to applied to peer session. Similarly, the GSAKMP [14] protocol defines a *policy token* defining the specifics of a group session. The policy token is an exhaustive data structure (containing over 150 fields) stating precisely the kinds of security for a given group session. Group properties of authorization, access control, data security, and key management are defined precisely through the token. However, while these systems provide a great deal of flexibility in defining policy, the range of supported mechanisms and policies is largely fixed. Thus, addressing unforeseen or exceptional security demands requires additional application infrastructure.

The DCCM [9] system developed by Branstad et. al. allows the definition of flexible policies through Cryptographic Context Negotiation Templates (CCNT). Each template defines the types and semantics of the available mechanisms and parameters of a system. A principal aspect of the DCCM project is its use of policy as entirely defining the context in which a group operates. Policy may be negotiated or stated by an initiating member, and flexible mechanisms for policy representation and interpretation are defined. A DCCM policy focuses on the mechanisms implementing group security services; authorization and access control is defined independently of the derived group policy.

Mechanism composition has long been used as a building block for distributed systems [25, 26, 2, 3, 11, 24]. Composition-based frameworks specify the compile or run-time organization of sets of protocols and services used to implement a communication service. The resulting software addresses the requirements of each session. However, the definition and synchronization of specifications is largely relegated to system administrators and developers. Our approach seeks to extend compositional systems by defining an architecture and language in which security requirements are consistently mapped into a system configuration.

The remainder of this paper is as follows. The following section gives a brief overview of the Antigone group model, policy specification language, and architecture. Section 3 describes the Antigone Applications Programming Interface (API). Section 4 demonstrates the use of the API in two demonstrative applications. Section 5 considers a number of works relating to the management of group policies. We conclude in Section 6.

2 Architecture

This section presents a brief overview of the Antigone 2.0 system. The following subsection describes the Antigone system model. The remaining subsections describe the Antigone architecture and policy specification language. This section is not intended to provide an exhaustive tutorial on Antigone, but only to introduce its model and design. Significant detail about the Antigone 2.0 system and its predecessors can be found in [21, 20, 19].

2.1 System model

Depicted in Figure 1, an Antigone group is modeled as the collection of *participants* collaborating towards a set of shared goals. We assume the existence of a *policy issuer* with the authority to state session requirements. The issuer states the conditional requirements of future sessions through the *group policy*. Adherence of a group policy to a set of correctness principles (describing legal security policies) is assessed through the *analysis* algorithm. A group policy is issued only if the analysis algorithm determines that the policy conforms to these principles.

Each participant states its set of local requirements on future session through a *local policy*. Each participant trusts the issuer to create a group policy consistent with session objectives. However, a participant can verify a policy instance meets the requirements stated in their local policy through the *compliance* algorithm. Failure of the group policy to comply to the local policy can result in the modification of the local policy or the abstention of the participant from the session.

An *initiator* is an entity that generates a policy instance from group and local policies. The service used to acquire local policies prior to reconciliation is outside the scope of the current work. We view this service as part of the session announcement protocol [12], but may revisit this decision in the future. An instance is the result of the *reconciliation* of the group and local policies within the run-time environment. Through reconciliation, an instance identifies relevant session requirements, and defines how requirements are mapped into a configuration. The initiator is trusted to evaluate the group and local policies correctly.

Currently, Antigone does not provide an interactive policy negotiation protocol. However, each participant defines the range of acceptable policies through thier local policies. Reconciliation attempts to find a instance that is compliant with each local policy (see Section 2.4). Hence, Antigone provides implicit negotiation through the evaluation of local policies. We intend to investigate the extention of the reconciliation process to an interactive protocol in the future.

A policy instance defines the session configuration (*provisioning*) and the rules used for authorization and access

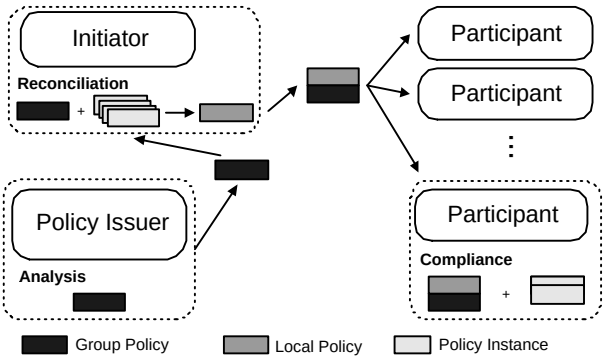

Figure 1: System Model - A session is a collection of *participants* collaborating towards some set of shared goals. A policy *issuer* states a group policy as a set of requirements appropriate for future sessions. The group and expected participant local policies are reconciled to arrive at a *policy instance* stating a concrete set of requirements and configurations. Prior to joining the group, each participant checks compliance of the instance with its *local policy*.

control. Provisioning of a group identifies the basic security requirements and the mapping of those requirements into a configuration of security-related services or mechanisms at member sites. Authorization and access control statements define how sessions regulate action within the group.

Participant software is modeled as collections of security *mechanisms*. Each mechanism provides a distinct communication service that is configured to address session requirements. Associated with a mechanism is a set of *configuration* parameters used to direct its operation. An instance defines precisely the set of mechanisms and configuration used to implement the session. For example, a *data security* mechanism implements transforms that enforce content security policies (e.g., message confidentiality, integrity, source authentication). The data security mechanism configuration identifies which transforms are used to secure application messages (see Section 3.2).

Similar to other secure group communication frameworks [13], the distribution of the policy instance is a two phase process. Potential group participants mutually authenticate themselves with the initiator (how authentication is performed is determined by the instance). The instance is distributed following authentication only if the initiator determines that the participant has the right to view policy (as determined by the instance access control policy). The member joins the group if the received instance is compliant with its local policy.

Note that because the instance defines the policy used throughout the lifetime of the group, no further policy synchronization is necessary. However, as described in [19], specialized `reconfig` events can trigger the policy re-evaluation. In this case, the group is disbanded and re-initialized under a newly established instance.

Antigone provides end-to-end group security service. In this, each participant acts as a policy enforcement point (PEP). Note that while many environments may benefit from the introduction of other non-participant PEPs (e.g., policy gateways, IPSec tunnels, etc.), we view these features as orthogonal to the current work. However, we plan to revisit this decision as Antigone matures and more environmental requirements become apparent.

2.2 Antigone Architecture

Described in Fig. 2, the Antigone architecture 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 (a brief overview of this API is given in Section 3.1). Group relevant actions such as join, send, receive, and leave are provided through simple C++ object methods. These actions are translated into events delivered to the other layers of Antigone. Group events (e.g., message received) are polled by the application through the API.

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 integrated with Antigone easily. Note that mechanisms need not only provide security services; other relevant functions (e.g., auditing, failure detection and recovery, replication) can be implemented through Antigone mechanisms. For example, the current implementation implements a novel secure crash failure detection mechanism [20].

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. Subsequently, the policy engine governs protected action through the evaluation of authorization 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.

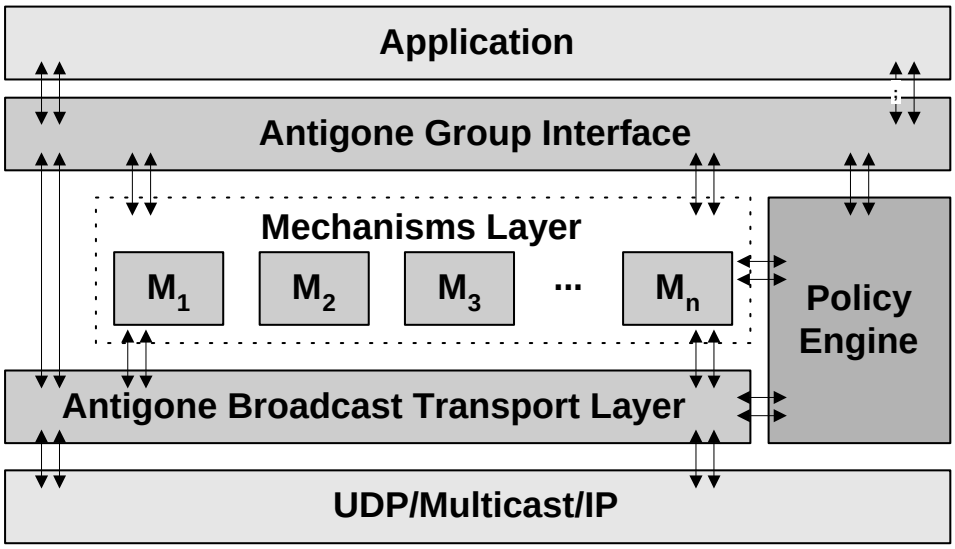

Figure 2: Antigone consists of four components; the group interface layer, the mechanism layer, the policy engine, and the broadcast transport layer. The group interface layer arbitrates communication between the application and lower layers of Antigone through a simple message oriented API. The mechanism layer provides a set 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.

2.3 Policy Language

Each group and local policy is explicitly stated through a policy specification. The prototype Ismene Policy Description Language (IPDL) defines the format and semantic of these specifications. Ismene is a subsystems defining a grammar and algorithms for the process of policy determiniation and analysis [19].

An IPDL policy is defined through a totally ordered set of *clauses*, where the ordering is implicitly defined by their occurrence in the specification. Each clause is defined by a tuple of *tags*, *conditions*, and *consequences*. Conditions test some measurable aspect of the operating environment, group membership, or presence of credentials. Consequences define what policies are to be applied to the group. Tags provide structure to the specification by directly defining the relations between sub-policies.

Figure 3 presents a subset of clauses from a typical group policy. The following example describes how a provisioning policy is derived from these clauses. The `key_management` clauses identify several key management policies appropriate for different operating environments. Initially, the initiator evaluates the conditionals associated with the first `key_management` clause. The `GroupIncludes` conditional tests whether a manager is expected to participate in the group. The `GroupSmaller` conditional tests whether the expected group will contain less than 100 members. Conditionals form a logical conjunction, where all conditionals must evaluate to true for the clause to satisfied. If a clause is satisfied, then the consequences are applied to the policy. In this example, if a manager is present and the group will contain less than 100 members, the `LKHKeyMnger` mechanism will be used with the identified configuration (i.e., `rekeyOnJoin=true` and `rekeyOnLeave=true`).

In the event the first clause is not satisfied, the second clause is consulted. This second clause represents a *default policy*; because it does not contain any conditions, it is always satisfied. Thus, where the first clause is not satisfied, the group falls back to a default Key-Encrypting-Key [15] key management policy. However, if the first clause is satisfied, the second clause is ignored.

The `data_handling` clause illustrates the use of the `pick` consequence. Pick consequences afford the initiator flexibility in developing the session. Semantically, the pick statement indicates that exactly one configuration must be selected. In the example, `pick` is used to state flexible policy; either DES or AES can be used to implement confidentiality, but not both or neither. The reconciliation process assesses the group and local policies to determine the most desirable configuration in the pick statement (see Section 2.4 below).

Authorization and access control are performed after the group has been provisioned. Typically, the evaluation of

```
% Key Management Provisioning
key_management:  GroupIncludes(Manager), GroupSmaller(100)
             ::  Config( LKHKeyMnger(rekeyOnJoin=true,rekeyOnLeave=true) );
key_management:  ::  Config( KEKKeyMnger(rekeytimer=300) );

% Data Handling Provisioning
data_handling:  ::  Pick( Config(adhdlr(conf=des)), Config(adhdlr(conf=aes)) );

% Join Authorization and Access Control
join:  Credential(Role=Manager,IssuedBy=$Trusted_CA) ::  accept;
join:  Credential(Role=SoftwareDesigner,IssuedBy=$Trusted_CA) ::  accept;
```

Figure 3: A subset of clauses define for an example IPDL group policy.

authorization requests test the presence of credentials proving a members right to perform some action (e.g., `join` the group). The simple `join` rules defined in Figure 3 state that any member who presents credentials issued by a trusted CA delegating the right to act as a `Manager` or `SoftwareDesigner` will be permitted into the group. Note that through the use of conditionals, a large number of complex authorization and access control models may be defined.

2.4 Reconciliation

The group policy is reconciled with the local policies of the expected participants to arrive at a concrete configuration. Thus, reconciliation determines which requirements are relevant to a session, and ultimately how the session is implemented. Ismene group policies are authoritative; all configurations and pick statements used to define the instance must be explicitly stated in the group policy. Local policies are consulted only where flexibility is expressly granted by the issuer through pick statements.

Reconciliation is the process by which configurations from pick statements in the group policy are selected. The selection process is guided by the configuration and pick statements in the local policies. Reconciliation appears on first viewing to be intractable. However, by restricting the structure and contents of IPDL policies, one can develop an efficient reconciliation strategy. [22] formulates the reconciliation problem and considers the complexity of the most general case. Several strategies are proposed and analyzed. This analysis lead to the efficient Prioritized Policy Reconciliation (PPR) algorithm used by the Antigone implementation.

For brevity, we have omitted many details of the IPDL construction, algorithms, and use. Interested readers are referred to [19, 22].

2.5 Implementing Policy

Inter-component communication in Antigone is event based. The observation of a security relevant event by any component is translated into an event object. Where policy decision is required, this event is posted to the policy engine event queue. If, based on the policy instance, the engine determines that further processing is warranted, the event is posted to the appropriate layer or application.

For example, consider an application wishing to broadcast a message to the group. The application initially makes the `SendMessage()` API call (see Section 3.1) with the data to be sent. The mechanism layer translates this call into a `SEND` event, which is posted to the policy engine event queue. The policy engine checks the policy instance, local credentials, and operational conditions to determine if the application has the right to send content to the group[1].

If permitted, the `SEND` event is posted to the mechanisms layer. The mechanisms layer allows each mechanism to process the event. In processing the event, the Data Security mechanism will perform a transform designed to provide the provisioned data security guarantees (e.g., confidentiality). The result is broadcast to the group via the transport layer.

Upon reception of the message, other participants translate the received message into a `RECV` event and post it to their local policy engine. The right of the sender to transmit data will be assessed with respect to the access control policy defined in the instance. If admitted, the reverse transform is performed by the Data Security mechanism on the received data. Once the original content is recovered, it is delivered to the application.

Note that the processing of a single event may trigger the

[1]Consulting authorization and access control policy on each relevant action may seriously affect performance. Antigone mitigates these costs by evaluating not only action acceptance or denial, but also the conditions under which the result should continue to be considered valid (i.e., invariant result, timed validity result, transient result). Therefore, authorization and access control policies need only be consulted when a valid previous result is unavailable.

enforcement of many policies. For example, a `NEW PARTICIPANT` event (representing a newly admitted member) may require the initiation of session rekeying, the creation of new process monitoring timers (for failure detection and recovery [20]), etc. The enforcement of each of these policies may lead to the generation of other events (e.g., `INIT REKEY`), authorization and access control decisions, and/or session traffic.

A central goal of Ismene (and Antigone) is the easy integration of additional services and conditionals. To this end, Antigone provides simple APIs for the creation of conditionals, mechanisms, and configurations. Developers create new mechanisms by constructing objects conforming to the `AMechanism` API. Developer stated unique identifiers (defining the mechanism and its configurations) can be added to IDPL policies, and are subsequently used as any other mechanism.

Application or mechanism specific conditionals can be implemented through the `APolicyImplementor` interface. `APolicyImplementor` objects define one or more conditionals to be used by Ismene. The unique identifiers associated with these conditionals can be immediately added to IDPL policies. Ismene performs an upcall to the implementor object upon encountering a defined conditional. The object is required to evaluate the conditional and return its result.

2.6 Policy Creation

Central to the security of any application is the definition of application policies. Each application, environment, and host can have unique requirements and abilities which must be reflected in the local and group policies. The `apcc` tool is used to assess Antigone policies with respect to these requirements.

`apcc` is a policy compiler; group and local policies are assessed to ensure *a)* the policy has the correct syntax (i.e., conforms the to the policy language grammar), and *b)* is consistent with a set of user supplied *assertions* (which define the correct usage principles discussed in Section 2). Any policy specification not conforming to the policy grammar is rejected by `apcc`.

Policy assertions define the correct usage of the underlying security mechanisms; dependencies and incompatibilities between different mechanisms are identified. For example, the following assertion identifies a dependency between security mechanisms;

```
assert: config(lkhkeymgt()) ::
        config(membership(leave=explicit));
```

This assertion states that all systems implementing a Logical Key Hierarchy [30, 32] must also implement explicit (member) leaves. The analysis algorithm implemented by `apcc` determines if any possible instance resulting from reconciliation violates this assertion (i.e., the instance defines an LKH mechanism, but not enforce an explicit leave policy). The user is warned of any such possible violation. In addition, policies which are irreconcilable (i.e., policies which, due to their construction, will always cause the reconciliation algorithm to fail) are identified.

Once policies have been created, they can be stored in any available repository. For example, an LDAP [34] service can be used to store and retrieve group and local policies. This approach is useful where the local domain wishes to enforce a set of security policies for all applications, or where users do not have the desire or sophistication to state policy. Note that while the storage and retrieval of policies is outside the scope of Antigone, each policy is evaluated by Antigone for freshness, integrity, and authenticity prior to its use.

3 Applications Programming Interface

The Antigone API abstracts group operations into a small set of message oriented interfaces. Conceptually, an application need only provide group addressing information and security policies appropriate for the application (*see below*). Once the group interface is created, the application can transmit and receive messages as needed.

The current implementation of Antigone consists of approximately 30,000 lines of C++ source and has been used as the basis for several non-trivial group applications (see next section). All source code and documentation for the Antigone Policy Description language, the Antigone framework, and applications are freely available. The six libraries comprising Antigone are described in Figure 4.

The Antigone API separates group operation from the broadcast medium. This separation is reflected in the AGroup (*Antigone Group*) and ATransport (*Antigone Transport*) APIs. The following subsections give an overview of the design, implementation, and interfaces of these libraries. Interested readers are referred to [19] for further detail.

Figure 5 presents a simple example application using the Antigone APIs. The application creates a group object for a server if invoked with no parameters, or a client if invoked with the name of the server host. Each process sends one message and receives all application data arriving within 60 seconds. All line numbers cited in the following subsections refer to this example.

3.1 Antigone Group API

The AGroup object serves as a conduit for all communication between an application and the group. After this ob-

Directory	Name	Description
atk	Toolkit	basic set of objects implementing basic data and structures (e.g., queues, timers, strings, . . .) and cryptographic functions (e.g., keys, hash functions, digital certificates, . . .) used by the other libraries.
atrans	Transport Layer	interfaces for an abstract broadcast channel in varying network environments. This embodies the entirety of the transport library described in Subsection 3.2.
amech	Mechanism Layer	abstract interfaces and classes upon which specific secure group mechanisms are built, coordinates the operation of mechanisms as directed by the policy instance.
mechs	Mechanisms	collection of mechanisms defining the services under which a group can be constructed. Policies are enforced using these basic services.
apdl	Policy Description Language	provides interfaces for the definition and evaluation of policies. The lexical analyzer and all policy algorithms are implemented in this library.
agrp	Group - Main API	Antigone Applications Programming Interface for secure groups. Applications communicate with Antigone through this API directly.

Figure 4: Antigone Component Libraries

```
#include <stdlib.h>
#include <AGroup.h>
int main (int argc, char **argv) { // usage: simple [ host_name_of_server ]
  if (getenv ("NAME") == NULL) setenv ("NAME", "unknown", 1); // set up id

  AGroup *group; // group object, policy files
  String locPol = "local.apd", grpPol = "example.apd", polList = "";

  // Setup the transport layer address - multicast address and port
  IPAddress *groupIp = IPAddress::IPAddressFactory("224.1.1.27", 9000);
  // specify server and port (argv[1] is the host name of the server)
  IPAddress *serverIp =
      IPAddress::IPAddressFactory(argc==1?"224.1.1.27":argv[1], 9001);
  // Construct transport layer
  ATransport *transport =
      new ATransport(groupIp, serverIp->Port(), ATransport::AT_SYMMETRIC);

  if (argc == 1) // server constructor for group - 5 parameters
      group = new AGroup(transport, grpPol, locPol, polList, NULL);
  else           // client constructor for group - only 3 parameters
      group = new AGroup(transport, locPol, NULL);
  (void)group->Connect();

  // Set up a buffer and send it
  String msg;
  msg.sprintf ("Hello World from %s\n", getenv("NAME"));
  Buffer *buf = new Buffer();
  (*buf) << msg;
  group->sendMessage(buf);

  AtkTimer timer(60 * 1000); timer.reset();   // wait for up to 60 seconds
  while (group->readMessage(&buf, &timer)) {  // read messages from group
      (*buf) >> msg;                          // extract message from buffer
      cout << " Received: " << (char*)msg;    // print message
      delete buf;
  }
  group->Quit(); // Leave, shutdown interface to the group
  exit (0);
}
```

Figure 5: Example Application

```
% File : example.apd
% Description : Example Antigone Group Policy
% Attributes Section
issr:= < iQBVAw ... >;

% Provisioning Section
provision: :: authentication, membership,
              keymgmt, datmgmt;
authentication: :: config(OpenSSL());
membership: :: config(amember(retry=3));
keymgmt: :: config(lkhkey(sens=memsens));
datmgmt: :: config(adhdlr(guar=conf,conf=desx)),
            config(adhdlr(guar=intg,intg=md5));

% Authorization/Access Control Policies
init: Credential(&cert,iss=$issr,
                 subj.CN=$joiner) :: accept;
join: Credential(&cert,iss=$issr,fs=$fsys,
                 subj.CN=$joiner) :: accept;
rekey: Credential(&key,key=$lkhKey) :: accept;
send: Credential(&key,key=$sessKey) :: accept;
eject: Credential(&key,key=$sessKey) :: accept;
leave: :: accept;

% Policy Verification
signature := < sdD5aR ... >;
```

Figure 6: Example Group Policy

ject is created (see below), all transmissions and receptions, state changes, and status probing are performed through AGroup member methods. The three phases of a group object include: initialization, operation, and shutdown.

The initialization of an AGroup object requires the member specify the appropriate policies and supply a transport object (lines 21 and 23 in Figure 5). The server constructor (line 21) supplies group and local policies which are reconciled to arrive at the session defining policy instance. Although not used in the example, the `polList` parameter identifies the list of local policies to be considered by the reconciliation algorithm. The client constructor (line 23) supplies its local policy and defers to the server for the instance. The `Connect` call (line 24) initializes the proper interfaces, joins the group, and retrieves or derives (through the reconciliation algorithm) the policy instance. Failures (either at the transport or group layers) generate an exception.

Subsequent sending, receiving, and processing of the messages during operation is achieved through an API similar to Berkeley Sockets [29] (e.g., `sendMessage` - line 31, `readMessage` - line 34). `sendMessage` sends and eventually deletes buffers. `readMessage` creates a buffer object for each incoming message. The `Buffer` object simplifies the tasks of memory management and message marshaling. `Buffer` objects handle translations between machine bit formats, automatically resize as needed, and maintain an internal heap of message structures. These objects allow Antigone to reduce the cost and simplify message memory management, translate between hardware and operating system platforms, and optimize message processing (e.g., reduce buffer copying).

```
% File : local.apd
% Description : Example Antigone Local Policy
issr:= < iQBVAw ... >;

% Requirements
provision: :: authentication, data_security;
authentication: :: config(OpenSSL());
data_security: :: config(adhdlr(guar=conf));

% No local policy regarding access control
join: :: accept; rekey: :: accept;
send: :: accept; leave: :: accept;
```

Figure 7: Example Local Policy

The interface to the group is shutdown through the `Quit` API call. This call exits from the group (explicitly sending a leave message as dictated by policy), destroys sensitive information (e.g., keys, messages), and cleans up all internal data.

An example policy appropriate for the above application is presented in Figure 6. This policy states a basic set of mechanisms are to be configured for the group; an `OpenSSL` mechanism for authentication, the Antigone `imember` membership management mechanism, a Logical Key Hierarchy key distribution mechanism, and the `adhdlr` data handler mechanism. The key management mechanism is configured to rekey after each membership change (e.g., member join or leave). The data handler mechanism is configured to provide confidentiality by encrypting all application traffic using DESX, and to provide integrity through keyed HMACs generated using the MD5 hash algorithm. The authorization and access control model for the group states that an appropriate certificate must be presented to gain access to the group, and that subsequent action is predicated on proof of knowledge of the appropriate session or key management keys.

An example local policy is presented in Figure 7. This local policy states that the local entity will only participate in groups that enforce a policy requiring OpenSSL authentication and which provide confidentiality of application traffic. The local policy states no requirements for group authorization (i.e., the local member accepts any authorization and access control model defined by the group policy).

3.2 Antigone 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, 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. 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. We note that a number of techniques can be used to vastly reduce the costs our implementation [17]. We plan to investigate these and other approaches in the near future.

In [1], we describe our experiences with the deployment of the *Secure Distributed Virtual Conferencing* (SDVC) application. This video-conferencing application is based on an early version of Antigone. 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 several institutions 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 session leader). The lack of bi-directional connectivity was attributed to limitations of the reverse routing of multicast packets. We present significant technical detail of this issue in [1].

The limited availability of bi-directional multicast on the Internet coupled with the costs of point-to-point multicast emulation lead us to introduce *asymmetric multicast*. This mode allows for messages emanating from the session leader to be multicast, and all other message 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, we reduce the costs associated with point-to-point group emulation to a unicast followed by a multicast. The increasing popularity of single source multicast make this a likely candidate for future use.

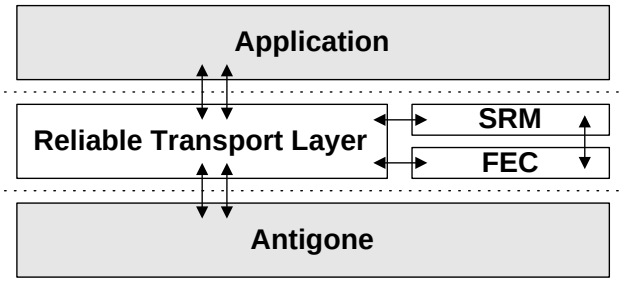

Figure 8: Reliable Transport Layer

The transport API requires the application supply the multicast or unicast addressing information appropriate for the environment and transport mode. IP addresses are specified through the creation of encapsulating `IPAddress` objects (lines 11, 15). These objects and the enumerated transport mode are passed to the constructor (line 18) of the transport object constructor, which is ultimately passed to the AGroup object upon its construction (lines 21 and 23). The transport object is not directly accessed after being passed to the AGroup object; all communication with the group is performed through the AGroup object.

4 Applications

This section briefly describes the use of Antigone in two applications: a reliable broadcast layer and an secure multicast layer used to augment existing group applications. A number of other applications (such as a streaming media and filesystem mirroring service) have been developed using Antigone. For brevity, we omit their discussion.

4.1 Reliable Transport Layer

The Reliable Transport Layer (RTL) provides FIFO delivery of application traffic delivered by an Antigone group. Depicted in Figure 8, RTL uses a combination of Forward Error Correction (FEC) and the approach used in the Scalable Reliable Multicast (SRM) protocol [10] to detect and recover from lost packets. A mechanism implementing each approach is layered between the application and Antigone.

The FEC mechanism uses a modified version of the approach described in [27]. In our implementation, FEC produces an additional q redundant packets for each p original packets. All $p+q$ packets are transmitted to the group. Because of the properties of packet construction, all original packets can be recovered from any p packets. However, a receiver encountering q or more losses cannot recover. Where enabled, these failures are repaired using the SRM mechanism.

The SRM mechanism implements the general approach implemented by Scalable Reliable Multicast protocol. Receivers detecting a lost packet broadcast a retransmission request to the group. Sender implosion is avoided by randomly delaying the retranmission request. To simplify, all receivers suppress requests corresponding to previously requested packets. If no such request is observed prior to the expiration of a random interval, the request is broadcast. This approach can effectively provide full reliable data delivery. However, the FEC mechanism can be used to reduce the number of retransmission requests.

Based on policy, an application can select either FEC, SRM, or both. Furthermore, policy can be used to parameterize their operation based on administrative considerations and operating conditions. For example, the FEC mechanism may wish to increase redundancy (e.g., larger values for q) where observed loss rates are high, and decrease redundancy where rates are low. Note the RTM policy can be specified directly in the Antigone group policy. RTM probes Antigone for the appropriate configuration during its initialization, and appeals to the application for direction where a configuration is not specified.

4.2 End-Host Security

The proliferation of group multicast applications (e.g., VIC [23]) has raised awareness of the need for secure multicast services. However, re-architecting applications to take advantage of security services is often difficult. Thus, it is highly desirable to use the end-host (transport) level services for security. Towards this end, we have developed the `Socket_s` library. `Socket_s` redirects multicast traffic to a user-space instantiation of Antigone. Existing applications can integrate with Antigone with only minor source code modification through this library.

Illustrated in Figure 9, the `Socket_s` library acts as a "bump in stack" by inserting Antigone between the application and the standard network interfaces. Each socket related call in the application is replaced with the appropriate `Socket_s` call. For example, each `bind` call is replaced with `bind_s`. The "_s" calls direct multicast related traffic towards Antigone, and non-multicast traffic to the standard library.

Local policies are managed at the host level; configuration and policy files are placed in a well-known directory, and are accessed by `Socket_s` as needed. Each domain has a known session leader who initiates and maintains groups for each active session occurring within its administrative scope. The identity and location the session leader is configured at each host.

Users initiate communication with an Antigone group through the `socket_s` and `setsockopt_s` (IGMP join [8]) calls. A background thread created during the

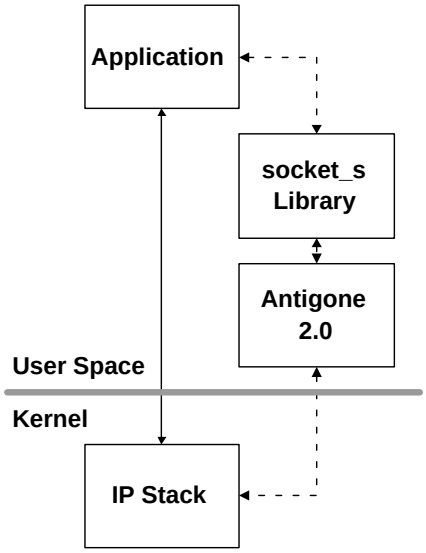

Figure 9: The Socket_s Library acts as a "bump in the stack" by redirecting all multicast traffic towards the Antigone interfaces.

`socket_s` call receives and sends all Antigone specific communication (Authorization requests, Rekey messages, etc..). The thread establishes a local connection with the parent application. Data received from the (Antigone) group is directed to the application through the local connection. The parent process receives this data from the local connection as with any normal socket.
The key interfaces to `Socket_s` include:

- *socket_s* : creates a "socket" endpoint for communication. If the desired socket is of the type SOCK_DGRAM (UDP), it initializes an Antigone group object and returns a "socket" filehandle. Note that (un-secured) unicast sockets can be created by accessing the `libc socket` call.
- *bind_s* : creates and initializes an Antigone Transport object (see Section 3.2) using the supplied address and port number.
- *setsockopt_s* : set socket parameters. The following two socket options are currently supported:
  - `IP_MULTICAST_LOOP` - enables/disables local host multicast loopback
  - `IP_ADD_MEMBERSHIP` - joins the Antigone group as described above
- *sendto_s* : send a message to the multicast group. The host network address and message data is transmitted using Antigone `sendMessage` interface (see Section 3.1).

- *recvfrom_s* : receive data from multicast group. The local connection associated with the socket is checked for data. If data is available, it is retrieved and copied to the (application) local buffer.

Antigone does not currently support point to point communication. Thus, all communication directed towards unicast addresses is redirected to the standard `libc` socket calls. Where needed, other transport layer security services (such as IPSec [18]) can be used to secure unicast communication. However, this has the disadvantage of decoupling all unicast traffic from the (Antigone) policies governing the application. An alternative is to use Antigone to create an additional group (containing only two members) for each peer session. However, these two party groups would pay the unnecessary cost of group management. These costs can be mitigated by provisioning the Antigone mechanisms with policies optimized for two member groups. We are currently investigating these and other approaches.

5 Related Work

Several recent group communication systems, including DCCM [9], GSAKMP [14], and Antigone 1.0 [21], support the notion of security policies defining detailed security service provisioning. In all these systems, generally, the range of group security policy is *static*. In that sense, the policy instance generated from Antigone can be considered as the policy input to these group communication systems. Antigone 2.0 extends these systems by stating the conditions under which certain policies should be enforced. In addition, Antigone 2.0 expresses policies that involve aspects of both provisioning and access control (support for the latter is limited in the above systems).

The problem of reconciling multiple policies in an automated manner is only beginning to be addressed. In the two-party case, the emerging Security Policy System (SPS) [35] defines a framework for the specification and reconciliation of local security policies for the IPSec protocol suite [18]. To handle a similar situation in Antigone, two local policies for the two ends of the IPSEC connection can be specified. These policies will be resolved against a group policy that leaves the choice of mechanisms open.

In the multi-party case, DCCM system [9] provides a negotiation protocol for provisioning. The first phase of the protocol involves the initiator sending a policy proposal to each potential member and receiving counter proposals. Subsequently, the initiator declares the 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. Antigone, if desired, can be easily adapted to use the DCCM's negotiation protocol. Antigone is more expressive because it can be used to state conditions under which various configurations can be used and when configurations need to be reconsidered in response to actions. The authorization and access control model is also more general in Antigone.

Language-based approaches for specifying authorization and access control have long been studied [4, 7, 33, 5, 28], but they generally lack support for provisioning. Because of the vast earlier work in this area and to simplify the language design, Antigone does not attempt to be as expressive for stating complex access control rules. Instead, Antigone is designed to leverage the expressive power of other access control systems via external authorization services.

The PolicyMaker [4] and KeyNote [5] 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 Antigone conditionals to leverage their expressive power and extend their use to group communication systems.

In [16], KeyNote has been used to define a distributed firewall application. The technique is to use conditional authorizations, 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 Antigone can help address these problems.

6 Conclusions

In this paper we have given an overview of the Antigone 2.0 architecture and API. Antigone provides flexible interfaces for the definition and implementation of security policies through the composition and configuration of security mechanisms. The set of services and protocols used to implement the group is developed from a systematic analysis of the properties appropriate for a given session in conjunction with operational conditions and participant requirements. The resulting session defining policy instance is distributed to all group participants and enforced uniformly at each host.

The Antigone Application Programmer Interface (API) allows the simple integration of group based applications with a flexible policy infrastructure. Developers are free to use the available policies and mechanisms, if they are satisfactory for its purpose, or build its own using the high-level mechanism API.

We have demonstrated the use of the API with two ex-

ample applications. The reliable group communication system provides an additional layer upon which reliable groups can be built. The host level multicast security application demonstrates how existing applications may be integrated with Antigone with only minor modifications.

Source code for the Antigone 2.0 system and applications are currently available in an alpha release. Updates and bug fixes for the Antigone source, documentation, and applications will be made available frequently. We maintain an open forum to which questions, bugs, and enhancement requests can be posted. Links to these resources and Antigone related publications are available at

`http://antigone.eecs.umich.edu/`

7 Acknowledgements

We would like to thank Peter Honeyman for his advice and guidance in the early stages of Antigone and the anonymous reviewers for their many insightful comments.

References

[1] A. Adamson, C.J. Antonelli, K.W. Coffman, P. D. McDaniel, and J. Rees. Secure Distributed Virtual Conferencing. In *Communications and Multimedia Security (CMS '99)*, pages 176–190, September 1999.

[2] Philip A. Bernstein. Middleware: A Model for Distributed System Services. *Communications of the ACM*, 39(2):86–98, February 1996.

[3] Nina T. Bhatti, Matti A. Hiltunen, Richard D. Schlichting, and Wanda Chiu. Coyote: A System for Constructing Fine-Grain Configurable Communication Services. *ACM Transactions on Computer Systems*, 16(4):321–366, 1998.

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

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

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

[7] 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.

[8] S. Deering. Host Extensions for IP Multicasting. *Internet Engineering Task Force*, August 1989. RFC 1112.

[9] 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.

[10] S. Floyd, V. Jacobson, C. Liu, S. McCanne, and L. Zhang. A Reliable Multicast Framework for Lightweight Sessions and Application Level Framing. In *Proceedings of ACM SIGCOMM '95*, pages 342–356. ACM, August 1995.

[11] 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.

[12] M. Handley and V. Jacobson. SDP: Session Description Protocol. *Internet Engineering Task Force*, April 1998. RFC 2327.

[13] T. Hardjono, R. Canetti, M. Baugher, and P. Dinsmore. Secure Multicast: Problem Areas, Framework, and Building Blocks (*Draft*). *Internet Engineering Task Force*, October 1999. `draft-irtf-smug-framework-00.txt`.

[14] 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`.

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

[16] Sotiris Ioannidis, Angelos D. Keromytis, Steve Bellovin, and Jonathan M. Smith. Implementing a Distributed Firewall. In *Proceedings of Computer and Communications Security (CCS) 2000*, pages 190–199, 2000. Athens, Greece.

[17] J. Janotti, D. Gifford, K. Johnson, Kaashoek M, and O'Toole 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.

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

[19] P. McDaniel and A Prakash. Ismene: Provisioning and Policy Reconciliation in Secure Group Communication. Technical Report CSE-TR-438-00, Electrical Engineering and Computer Science, University of Michigan, December 2000.

[20] 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.

[21] 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.

[22] Patrick McDaniel. *Policy Management in Secure Group Communication*. PhD thesis, University of Michigan, Ann Arbor, 2001. Chapter 4 - Policy Representation and Analysis, (*Draft*).

[23] Network Research Group, Lawrence Berkeley National Laboratory. `vic` - Video Conferencing Tool, July 1996. `http://www-nrg.ee.lbl.gov/vic/`.

[24] P. Nikander and Arto 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.

[25] 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.

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

[27] L. Rizzo. Effective Erasure Codes for Reliable Computer Communication Protocols. In *ACM Computer Communications Review, vol. 27, n.2*, pages 24–36. ACM, April 1997.

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

[29] W. R. Stevens. *Unix Network Programming*. Prentice Hall, ISBN 0 13 949876 1, Second edition, 1998.

[30] Debby M. Wallner, Eric J. Harder, and Ryan C. Agee. Key Management for Multicast: Issues and Architectures (*Draft*). *Internet Engineering Task Force*, September 1998. `draft-wallner-key-arch-01.txt`.

[31] A. Westerinen, J. Schnizlein, J. Strassner, Mark Scherling, Bob Quinn, Jay Perry, Shai Herzog, An-Ni Huynh, and Mark Carlson. Policy Terminology (*Draft*). *Internet Engineering Task Force*, July 2000. `draft-ietf-policy-terminology-00.txt`.

[32] 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.

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

[34] W. Yeong, T. Howes, and S. Kille. Lightweight Directory Access Protocol. *Internet Engineering Task Force*, March 1995. RFC 1777.

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