Framework for Authentication and Access Control of Client-Server Group Communication Systems *

Yair Amir, Cristina Nita-Rotaru, Jonathan R. Stanton
Department of Computer Science
Johns Hopkins University
3400 North Charles St.
Baltimore, MD 21218 USA
{yairamir, crisn, jonathan}@cs.jhu.edu

Abstract

Researchers have made much progress in designing secure and scalable protocols to provide specific security services, such as data secrecy, data integrity, entity authentication and access control, to multicast and group applications. However, less emphasis has been put on how to integrate security protocols with modern, highly efficient group communication systems and what issues arise in such secure group communication systems. In this paper, we present a flexible and modular architecture for integrating many different authentication and access control policies and protocols with an existing group communication system, while allowing applications to provide their own protocols and control the policies. This architecture maintains, as much as possible, the scalability and performance characteristics of the unsecure system. We discuss some of the challenges when designing such a framework and show its implementation in the Spread wide-area group communication toolkit.

1 Introduction

The Internet is used today not only as a global information resource, but also to support collaborative applications such as voice- and video-conferencing, white-boards, distributed simulations, games and replicated servers of all types. Such collaborative applications often require secure message dissemination to a group and efficient synchronization mechanisms. Secure group communication systems provide these services and simplify application development.

A secure group communication system needs to provide confidentiality and integrity of client data, integrity, and possibly confidentiality, of server control data, client authentication, message source authentication and access control of system resources and services.

Many protocols, policy languages and algorithms have been developed to provide security services to groups. However, there has not been enough study of the integration of these techniques into group communication systems. Needed is a scheme flexible enough to accommodate a range of options and yet simple and efficient enough to appeal to application developers. Complete secure group communication systems are very rare and research on how to transition protocols into complete systems has been scarce.

Secure group systems really involve the intersection of three major, and distinct, research areas: networking protocols, distributed algorithms and systems, and cryptographic security protocols.

*This work was supported by grant F30602-00-2-0526 from The Defense Advanced Research Projects Agency.

A simplistic approach when building a secure group system is to select a specific key management protocol, a standard encryption algorithm, and an existing access control policy language and integrate them with a messaging system. This would produce a working system, but would be complex, fixed in abilities, and hard to maintain as security features would be mixed with networking protocols and distributed algorithms.

In contrast, a more sophisticated approach is to construct an architecture that allows applications to plug-in both their desired security policy *and* the mechanisms to enforce the policy. Since each application has its particular security policies, it is natural to give an application more control not only on specifying the policy, but on the implementation of the services part of the policy too.

This paper proposes a new approach to group communication system architecture. More precisely, it provides such an architecture for authentication and access control. The architecture is flexible, allowing many different protocols to be supported and even be executing at the same time; it is modular so that security protocols can be implemented and maintained independently of the network and distributed protocols that make up the group messaging system; it allows applications to control what security services and protocols they use and configure; it efficiently enforces the chosen security policy without unduely impacting the messaging performance of the system.

As many group communication systems are built around a client-server architecture where a relatively small number of servers provide group communication services to numerous clients, we focused on systems utilizing this architecture. [1]

We implemented the framework in the Spread wide-area group communication system. We evaluate the flexibility and simplicity of the framework through six case studies of different authentication and access control methods. We show how both simple (IP based access control, password based authentication) and sophisticated (SecurID, PAM, anonymous payment, and group based) protocols can be supported by our framework.

Note that this paper is *not* a defense of any particular access control policy, authentication method or group trust model. Instead, it provides a flexible, complete interface to allow many such polices, methods, or models to be expressed and enforced by an existing, actively used group communication system.

The rest of the paper is organized as follows. Section 2 overviews related work. We present the authentication and access control framework and its implementation in the Spread toolkit in Section 3. We provide several brief case studies of how diverse protocols and policies can be supported by the framework in Section 4. Finally, we conclude and discuss future directions.

2 Related Work

There are two major directions in secure group communication research. The first one aims to provide security services for IP-Multicast and reliable IP-Multicast. Research in this area assumes a model consisting of one sender and many receivers and focuses on the high scalability of the protocols. Since the presence of a shared secret can be used as a foundation of efficiently providing data confidentiality and data integrity, a lot of work has been done in designing very scalable key management protocols. Examples of such protocols are: the Group Key Management Protocol (GKMP) [16], Multicast Key Management Protocol (MKMP) [14], the Scalable Multicast Key Distribution (SMKD) [7] approach based on the Core Based Trees [8], Intra-domain Group Key Management Protocol (IGKMP) [13], the VersaKey Framework [11] and the Group Secure Association Key Management Protocol (GSAKMP) [15].

[1] Some of the work may apply to network level multicast, but we have not explored that.

The second major direction in secure group communication research is securing application level multicast systems, also known as group communication systems. These systems assume a many-to-many communication model where each member of the group can be both a receiver and a sender, and provide reliability, strong message ordering and group membership guarantees, with moderate scalability. Initially group communication systems were designed as high-availability, fault-tolerant systems, for use in local area networks. Therefore, the first group communication systems ISIS [10], Horus [25], Transis [4], Totem [5], and RMP [29] were less concerned with addressing security issues, and focused more on the ordering and synchronization semantics provided to the application (the Virtual Synchrony [9] and Extended Virtual Synchrony [22] models).

The number of secure group communication systems is small. Besides our system (Spread), the only implementation of group communication systems that focus on security are the RAMPART system at AT&T [24], the SecureRing [18] project at UCSB and the Horus/Ensemble work at Cornell [26]. A special case is the Antigone [20] framework, designed to provide mechanisms allowing flexible application security policies. Most relevant to this work are the Ensemble and the Antigone systems. Ensemble focused on optimizing group key distribution, and chose to allow application-dependent trust models in the form of access control lists treated as replicated data within the group. Authentication is achieved by using PGP. Antigone instead, allows flexible application security policies (rekeying policy, membership awareness policy, process failure policy and access control policy). However, it uses a fixed protocol to authenticate a new member and negotiate a key, while access control is performed based on a pre-configured access control list.

We also consider frameworks designed with the purpose of providing authentication and/or access control, without addressing group communication issues. Therefore, they are complementary to our work. One of these frameworks is the Pluggable Authentication Module (PAM) [27] which provides authentication services to UNIX system services (like login, ftp, etc). PAM allows an application not only to choose how to authenticate users, but also to switch dynamically between the authentication mechanisms without (rewriting and) recompiling a PAM-aware application. Other frameworks providing access control and authentication services are systems such as Kerberos [19] and Akenti [28]. Both of them have in common the idea of authenticating users and allowing access to resources, with the difference being that Kerberos uses symmetric cryptography, while Akenti uses public-key cryptography to achieve their goals.

One flexible module system that supports various security protocols is Flexinet [17]. Flexinet is an object oriented framework that focuses on dynamic negotiations, but does not provide any group-oriented semantics or services.

3 General System Architecture

The overall goal of this work is to provide a framework that integrates many different security protocols and supports all types of applications which have changing authentication and access control policy requirements, while maintaining a clear separation of the security policy from the group messaging system implementation. In this section, after discussing some design considerations, we present the authentication and access control frameworks.

3.1 Why is a General Framework Needed?

When a communication system may only be used with one particular application, integrating the specific security policy and needed protocols with the system may make sense. However, when a communication system needs to support many different applications that may not always be cooperative, separating the policy issues which will be unique to each application from the enforcement mechanisms which must work for all applications avoids an unworkable "one-size-fits-all" security model, while maintaining efficiency.

Separating the policy implementation from both the application and the group communication system is also useful because in a live, production environment, the policy restrictions and access rules will change much more often than the code or system changes. So modifications of policy modules should not require recompiling or changing the application code.

The features of the general framework, as opposed to the features of a particular authentication or access control protocol, are:

1. Individual policies for each application.

2. Efficient policy enforcement in the messaging system.

3. Simple interface for both authentication and access control modules.

4. Independence of the messaging system from security protocols.

5. Many policies and protocols work with the framework, including: access control lists, password authentication, public/private key, certificates, role based access control, anonymous users, and dynamic peer-group policies.

We distinguish between authentication and access control modules to provide more flexibility. Each type of module has a distinctive interface which supports its specific task. The authentication module verifies that a client is who it claims to be. The access control module decides about all of the group communication specific actions a client attempts after it has been authenticated: join or leave a group, send an unicast message to another client or multicast a message to a group. It also decides whether a client is allowed to connect to a server (the access control module can deny a connection even if the authentication succeeded).

Supporting all possible policies is an important goal, however, this goal has some specific problems that make achieving it costly in performance and simplicity. In this version of the framework we chose to focus on supporting a wide class of polices, but not all polices, and preserving good scalability, minimal performance impact, and a simple abstraction of enforcement. The framework supports many of the commonly known examples of specific policies and protocols. Specifically, it supports all policies that are *State-independent* including both static and dynamic policies. *State-independent* policies are those that do not require knowledge of the current state of the group or messages sent to the group in order to make decisions. *State-dependent* policies do require such knowledge. An example of a *State-dependent* access control policy is one which prohibits one user from joining while another is a member, or a policy which bases decisions on whether certain messages have been sent to the group or not. The framework does not support all *State-dependent* policies as they require very tight inter-relationships between the particular implementation of a group system and how the policy is specified.

For example, establishing a protocol where the group-creator defines what the policy should be is difficult in a distributed group system with no central policy server. The difficulty comes from several issues:

- Several members may think they are the first member of the group and they will only know who was actually first after others have already started the joining process. So the system must be able to go back and redo access control decisions that were already made before the first member established the policy.

- Any policy defined by the group creator obviously has no way to control who can create the groups in the first place. Policies restricting who can create groups are needed both for security reasons and effective resource management. So some other, probably more static, policy must also be in effect.

- Since all the members of the group participate in establishing the access control they all must know what the policy should be in case they are the first member of the group. Or if all processes are not equal and only a few or only one are able to establish the groups, you lose one of the major advantages of peer-groups as opposed to one-to-many groups. This complicates the management of access control polices because all members must have current, accurate policies.

- In partitionable environments, some "meta-policy" must exist to handle the merging of groups from previously partitioned components. One of the most common ideas to solve this is to use the oldest policy based on real-life time, but that still has problems with unsynchronized clocks and changing policies.

The framework supports *dynamic* policies. The main challenge with such policies is to allow changes during execution. Since the framework itself does not have any knowledge of the actual policy, for example it does not cache decisions or restrict what form actual policies take, it is possible for the access control modules to change how they make decisions independently of server. The modules need to make sure they activate dynamic changes in a consistent way, by using synchronized clocks, or by using the group communication services to agree on when to activate changes.

The framework provides services to ease the implementation of correct dynamic policies. The most important service is allowing the policy modules to also act as clients of the group messaging system. They can then use the services of the system such as reliable multicast, ordered messages, and membership to simplify the handling of dynamic updates. The policies could then use the same synchronization and fault-tolerance techniques of regular applications. They could use total order messages to order changes to the policies and use the membership information to handle merging groups.

The performance cost of our general framework when compared with a custom implementation of the required authentication methods and access control policy is very small. The only notable cost is the difference between the overhead of a function call and the overhead of inline code for access control checks.

3.2 Framework Implementation in Spread

We implemented the framework in the Spread group communication system to give a concrete, real-world basis for evaluating the usefulness of this general architecture. Although we only implemented the framework within the Spread system, the model and the interface of the framework are actually quite general and the set of events upon which access control decisions can be made includes all of the available actions in a group-based messaging service (join, leave, group send, unicast send, connect).

3.3 The Spread Group Communication Toolkit

Spread [6, 3, 2] is a local and wide-area messaging infrastructure supporting reliable multicast and group communication. It provides reliability and ordering of messages (FIFO, causal, total ordering) and a membership service. The toolkit supports four different semantics: No membership, Closely Synchronous[2], Extended Virtual Synchrony (EVS) [1] and View Synchrony (VS) [12].

The system consists of one or more servers and a library linked with the application. The servers maintain most of the state of the system and provide reliable multicast dissemination, ordering of messages and the membership services. The library provides an API and basic services for message oriented applications. The application and the library can run on the same machine as a Spread server, in which case they communicate over IPC, or on separate machines, in which case the client-server protocol runs over TCP/IP.

Note that in order to implement our framework, we needed to modify both the Spread client library and the Spread daemon. When an application implements its own authentication and access control method, it needs to implement both the client side and the server side modules, however, it does not need to modify the Spread library or the Spread daemon.

In Spread each member of the group can be both a sender and a receiver. The system is designed to support small to medium size groups, but can accommodate a large number of different collaboration sessions, each of which spans the Internet. This is achieved by using unicast messages over the wide-area network and routing them between Spread nodes on an overlay network. Spread scales well with the number of groups used by the application without imposing any overhead on the network routers. Group naming and addressing is not a shared resource (as in IP multicast addressing), but rather a large space of strings which is unique to a collaboration session.

The Spread toolkit is available publicly and is being used by several organizations for both research and practical projects. The toolkit supports cross-platform applications and has been ported to several Unix platforms as well as Windows and Java environments.

3.4 Authentication Framework

All clients are authenticated when connecting to a server, and trusted afterwards. Therefore, when a client attempts actions, such as sending messages or joining groups, no authentication is needed. However, the attempted user actions are checked against a specified policy which controls which actions are permitted or denied for that user. This approach explicitly assumes that as long as a connection to the server is maintained, the same user is authenticated.

Figure 1 presents the architecture and the process of authentication. Both the client and the server implement an authentication module.

The change on the client side consists of the addition of a function (see Figure 2) that allows an application to set the authentication protocol it wishes to use and to pass in any necessary data to that protocol, before connecting to a Spread server. When the function that specifies the request of a client to connect to a server is called (`SP_connect`), the connection tries to use the method the application set to establish a connection.

The authentication method chosen by the application applies to all connections established by this application (i.e. by this process). If the application wishes to change the authentication method or data, it must call `SP_set_auth_method` or `SP_set_auth_methods` again. All subsequent connections established by `SP_connect` will use the new information. In a multi-threaded application the setting of new authentication methods or data must be synchronized by the application with any calls to `SP_connect`.

A server authentication module needs to implement the functions listed in the `auth_ops` structure (see Figure 3, line 10). Then the module should register itself with the Spread daemon by calling the `Acm_auth_add_method` function. By default, a module is registered in the 'disabled' state. The system administrator can enable the module when configuring Spread.

[2]This is a relaxed version of EVS for reliable and FIFO messages.

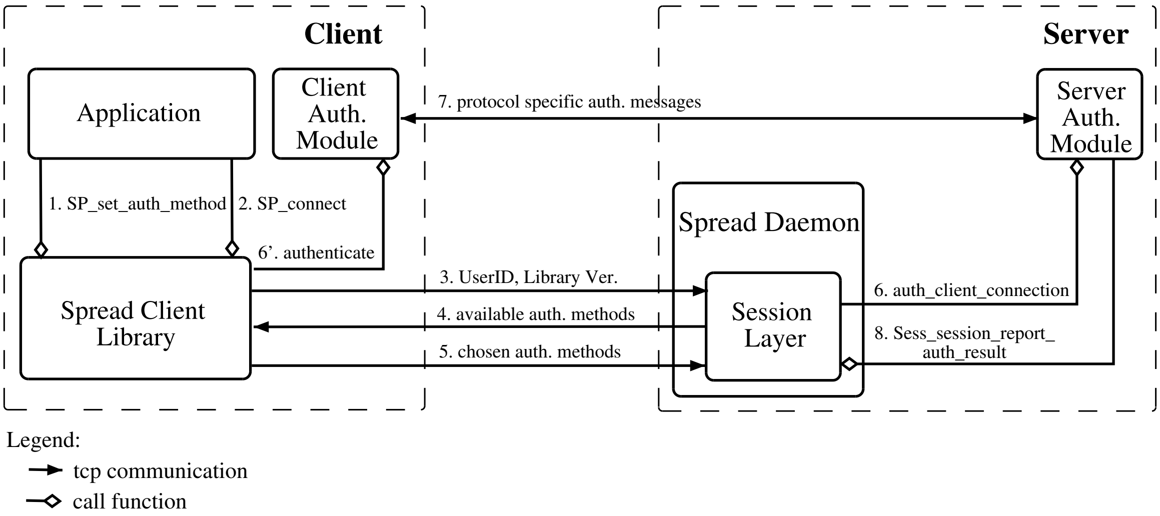

Figure 1: Authentication architecture and communication flow

```
int SP_set_auth_method( const char *auth_name, int (*authenticate) (int, void *), void * auth_data );
int SP_set_auth_methods( int num_methods, const char *auth_name[], int (*authenticate[]) (int, void *), void * auth_data[] );

/* declaration of authenticate function */
int authenticate(int fd, void * user_data_pointer);
```

Figure 2: Client Authentication Module API

The authentication process operates as follows: when the session layer of the daemon receives a connection request from a client (this happens when the client calls `SP_connect`, it first receives some basic information from the client such as the library version number and the client name. Then, the server sends to the client a list with all the available authentication methods. The client sends back to the server the name of the preferred authentication method. If the authentication method requested by the client is allowed by the server, the session module constructs a `session_auth_info` structure containing the list of authentication methods which must all complete successfully for the connection to be considered authenticated. The list of methods is made by including all of the methods the administrator listed in the `RequiredAuthMethods` field of the Spread configuration file as well as whatever method the client selected from the `AllowedAuthMethods` field. This structure is passed as a parameter to each authentication function and is used as a handle for the entire process of authenticating a client. The authentication function can use the `module_data` pointer to store any module specific data that it needs during authentication. This data pointer is only valid during a particular instance of the module and will be overwritten as soon as the module completes authentication. The session module passes control of the connection to each of the listed authentication modules in the order they were listed, by calling the `auth_client_connection` method and then "forgets about" the client connection. A minimal state about the client is stored, but no messages are received or delivered to the client at this point.

The `auth_client_connection` function is responsible for authenticating the client connection. If authenticating the client will take a substantial amount of CPU or real time, the function should not do the work directly, but rather setup a callback function to be called later (for example when

```
1  struct session_auth_info {
2      mailbox mbox;
3      void *module_data;
4      int num_required_auths;
5      int completed_required_auths;
6      int required_auth_methods[MAX_AUTH_METHODS];
7      int required_auth_results[MAX_AUTH_METHODS];
8  };
9
10 struct auth_ops {
11     void (*auth_client_connection) (struct session_auth_info *sess_auth_p);
12 };
13
14 struct acp_ops {
15     bool (*open_connection) (char *user);
16     bool (*open_monitor) (char *user); /* not used currently */
17     bool (*join_group) (char *user, char *group, void *acm_token);
18     bool (*leave_group) (char *user, char *group, void *acm_token);
19     bool (*p2p_send) (char *user, char dests[][MAX_GROUP_NAME], int service_type);
20     bool (*mcast_send) (char *user, char groups[][MAX_GROUP_NAME], int service_type);
21 };
22
23 /* Auth Functions */
24 bool Acm_auth_add_method(char *name, struct auth_ops *ops);
25
26 /* Access Control Policy Functions */
27 bool Acm_acp_set_policy(char *policy_name);
28 bool Acm_acp_add_method(char *name, struct acp_ops *ops);
```

Figure 3: Server Authentication and Access Control Module API

messages arrive from the client), and then it should return. Another approach is to fork off another process to handle the authentication. This is required because the daemon is blocked while this function is running.

The `auth_client_connection` function never returns a decision value because a decision may not have been reached yet. When a decision has been made the server authentication module calls `Sess_session_report_auth_result` and releases control to the session layer. The `Sess_session_report_auth_result` function reports whether the current authentication module has successfully authenticated the session or not. If more than one authentication method was required, the connection succeeds if all the methods succeed.

3.5 Access Control Framework

In our model, an authenticated client connection is not automatically allowed to perform any actions. Each action a client may request of the server, such as sending a message or joining or leaving a group, is checked at the time it is attempted against an access control policy module. The enforcement checks are implemented by having the session layer of the server call the appropriate access control policy module callback function (see Figure 3, lines 14-20) return a decision. The implementation of the check functions should be optimized as they have a direct impact on the performance of the system as they are called for every client action.

An interesting problem is rejecting actions in asynchronous systems. In asynchronous messaging systems a client can initiate messages to the system without waiting for any synchronous feedback indicating that the message was accepted or succeeded in being sent. This is a significant advantage of asynchronous systems as it allows them to pipeline requests and easily overlap messaging communication time with local computation. However, it prevents giving a client feedback about a particular event, such as denying that event, by simply returning an error code or having the "send" function return an error. We note that the way many networking protocols deal with this situation is to have the next action attempted on the connection return an error indicating something went wrong with a prior action. This can work when the error causes the entire connection to be reset (as in most TCP errors) but is very limited when errors are not fatal to the connection and identifying which application action resulted in the error can be difficult.

In our approach if the module chooses to allow the request, then the server handles it normally. In the case of rejection, the server creates a special "reject" message which will be sent to the client in the normal stream of messages. The reject message contains as much of the data included in the original attempt as possible. The application should be able to identify which message was rejected by whatever information it stored in the body of the message (such as an application level sequence number) and respond to it appropriately. That response could be a notification to the user, establishing a new connection with different authentication credentials and retrying the request, logging an error, etc.

The server can reject an action at two points, when the server receives the action from the client or when the action is going to take effect. For example, when a client joins a group the join can be rejected when the join request is received from the directly connected client, and when the join request has been sent to all of the servers and has been totally ordered. Rejecting the request the first time it is seen avoids processing requests that will later be rejected and simplifies the decision-making because only the server the client is directly connected to will make the decision. The disadvantage is that at the time the request is being accepted or rejected the module only knows the current state of the group or system and not what the state will be when the request would be acted upon by the servers. Since these states can differ, some type of decisions may not be possible at the early decision point.

4 Case Studies

To provide some intuition as to what building a Spread authentication module requires, this section discusses the implementation of several real-world modules: an IP based access control module, a password based authentication module, a SecurID or PAM authentication module, an anonymous payment authentication and anonymous access control module, and a dynamic peer-group authentication module. In this section, we provide actual implementation code instead of pseudocode to emphasize the real world simplicity and usability of our framework.

4.1 IP Access Control

A very simple access control method that does not involve any interaction with the client process or library, is one that is based on the IP address of the clients. The connection is allowed considering the IP address from which the client connected to the server. This module only restricts the `open_connection` (see Figure 3, line 15) operation.

Figure 4 shows the main code for the module. Note the decision of allowing access or not is made at line 40. The resulting decision is returned to the daemon by the return value being ACM_ACCESS_ALLOWED or ACM_ACCESS_DENIED. The check is done against an in memory list of access rules. For a small number of rules this is reasonable, however, more efficient data structures and a way to dynamicly reload the rules would be good enhancements in practice.

A configuration file specifies what IP address the system supports. For example, in the configuration file presented in Figure 5, UNIX and localhost sockets are supported, along with TCP connections from IP addresses between 192.168.1.0 and 192.168.1.255, inclusively.

4.2 Password Authentication

A common form of authentication uses some type of password and username to establish the identity of the user. Many types of password based authentication can be supported by our framework from passwords sent in the clear (like in telnet) to challenge-response passwords.

In Figures 6, 7, and 8 we show how a basic, telnet style, password protocol could be implemented. This protocol has both a client and a server module and demonstrates how an authentication

```
void ip_client_connection(char *user);

static struct  acp_ops IP_ops = {
    ip_client_connection,
    permit_open_monitor,
    permit_join_group,
    permit_leave_group,
    permit_p2p_send,
    permit_mcast_send,
};
struct ip_rule {
    int32u      network_address;
    int         prefix_length;
    struct ip_rule *next;
};
static struct ip_rule *Allow_Rules;

void ip_init(void)
{
    char        file_name[80];
    sprintf(file_name, "spread.access_ip");

    if (!Acm_acp_add_method("IP", &IP_ops))  {
        Alarm( EXIT, "ip_init: Failed to register IP. Too many ACM methods registered.\n");
    }

    /* load spread.access_ip file */
      ... removed code that loads the access_ip file in the Allow_Rules ...
}
void ip_client_connection(char *user)
{
    int32u client_ip, client_net;
    struct ip_rule *rule_p;
    bool allowed;
    int ses;

    ses = Sess_get_session(user)
    client_ip = Sessions[ses].address;
    rule_p = Allow_Rules;
    allowed = FALSE;
    /* Search allowed lists */
    while(rule_p) {
        client_net = (client_ip & ( (~0x0) << (32 - rule_p->prefix_length)));
            if(rule_p->network_address == client_net) {
            allowed = TRUE;
            break;
      }
      rule_p = rule_p->next;
    }
    if(allowed)
        return(ACM_ACCESS_ALLOWED);
    else
        return(ACM_ACCESS_DENIED);
}
```

Figure 4: IP Server Authentication Module

protocol can communicate with the client process. Figure 6 in lines 50 and 51 and Figure 7 in lines 8 and 9 also show how authentication modules can use the Events subsystem in Spread to wait for network events to occur and avoid blocking the Spread server while the user is entering its password or the client and server modules are communicating. Figure 6 shows the standard module setup and some support routines to store a list of usernames and passwords to allow quick lookups during authentication.

The client module consists of one function which is called during the establishment of a connection and returns either success or failure. The function can use the file descriptor of the socket over which the connection is being established and whatever data pointer was registered by the `SP_set_auth_method`. In this case the application prompted the user for a username and password and created a `user_password` structure. The `authenticate` function, sends the username and the password to the server and waits for a response, informing it of whether or not the authentication succeeded.

Note that this sample module is shown using blocking "recv" calls in lines 11 and 23 of Figure 7 which could cause the entire server process to block if the client did not send all the bytes the

```
# allow local unix domain socket connections
unix
# allow localhost TCP connections
local
# allow TCP connections from IP addresses between 192.168.1.0 and 192.168.1.255
192.168.1.0/24
```

Figure 5: IP Authentication configuration file

server is expecting. This is done for sake of simplicity, a correct implementation would read in non-blocking mode and store partially received data in a structure linked off the `module_data` pointer in the `session_auth_info` structure.

4.3 SecurID

A popular authentication method is RSA SecurID. The method uses a SecurID server to authenticate a SecurID client based on a unique randomly generated identifier and a PIN. In some cases the SecurID server might ask the client to provide new credentials. We do not discuss here the internal of the SecurID authentication mechanism (see [23] for more details), but focus on how our framework can accommodate this method.

The main difference from the previous examples is that in this case the Server Authentication Module needs to communicate with an external entity, the SecurID server.

In Figure 9 we present the architecture of a Spread system using SecurID as authentication mechanism. The Server Authentication Module needs to implement the `auth_client_connection`. As mentioned in Section 3.4, `auth_client_connection` should return immediately, and not block. Blocking can happen when opening a connection with a SecurID server and retrieving messages from it. Therefore, `auth_client_connection` forks another process responsible for the authentication protocol and then registers an event such that it will get notified when the forked process finished. The forked process establishes a connection with the SecurID Server and authenticates the user. When it finishes, the Server Authentication Module gets notified, so it can call the `Sess_session_report_auth_result` function to inform the Spread daemon that a decision was taken and to pass control back to it.

4.4 PAM

Another popular method of authentication is the modular PAM [27] system which is standard on Solaris and many Linux systems. Here the authentication module will act as a client to a PAM system and request authentication through the standard PAM function calls. To make authentication through PAM work, the module must provide a way for PAM to communicate and interact with the actual human user of the system, to prompt for a password or other information. The module would register an interactivity function with PAM that would pass all of the requests to write to the user or request input from the user over the Spread communication socket to the Spread client authentication module for PAM. This client module would then act on the PAM requests and interact with the user and then send the reply back to the Spread authentication module which would return the results to the actual PAM function.

4.5 Anonymous Payments

An interesting approach is providing access to anonymous clients in exchange for payment. Systems such as NetCash [21] provide support for transactions between a client and a merchant by means

```
struct user_password {
    char username[MAX_PWORD_USERNAME + 1];
    char crypt_pass[MAX_PWORD_CRYPTPASSWORD + 1];
    struct user_password *next;
};
static struct  auth_ops Pword_ops = {
    pword_auth_client_connection,
    pword_auth_monitor_connection,
};
static struct user_password *Users;

/* inserts username and password into Users list */
static void insert_user(char *username, char *crypt_password);
/* searches Users list and returns entry */
static bool lookup_user(char *username, struct user_password **user_h);
static bool check_password(char *username, char *clear_password);
static void auth_client_conn_read(int fd, int dummy, struct session_auth_info *sess_auth_p);

void pword_init(void)
{
    char        file_name[80];
    sprintf(file_name, "spread.access_pword");

    if (!Acm_auth_add_method("PWORD", &Pword_ops)) {
        Alarm( EXIT, "pword_init: Failed to register PWORD. Too many ACM methods registered.\n");
    }
    /* load spread.access_pword file */
           ... removed code that loads the access_pword file in the Users list ...
}
static bool check_password(char *username, char *clear_password)
{
    struct user_password *user_p;
    char *crypt_presented_pass;
    char salt[2];

    if(lookup_user(username, &user_p)){
        memcpy(salt, user_p->crypt_pass, 2);
        crypt_presented_pass = crypt(clear_password, salt);
        if(!strncmp(crypt_presented_pass, user_p->crypt_pass, 13)) {
            return(TRUE);
        } else {  /* password did not match */
            return(FALSE);
        }
    } else {  /* user not found */
        return(FALSE);
    }
}
void pword_auth_client_connection(struct session_auth_info *sess_auth_p)
{
    E_attach_fd(sess_auth_p->mbox, READ_FD, (void (*)(int,int,void *)) auth_client_conn_read, 0, sess_auth_p, LOW_PRIORITY);
    E_attach_fd(sess_auth_p->mbox, EXCEPT_FD, (void (*)(int,int,void *)) auth_client_conn_read, 0, sess_auth_p, LOW_PRIORITY);
    return;
}
```

Figure 6: Password Server Authentication Module (support code)

of digital cash. Dedicated servers called *currency servers*, can issue coins to clients. Later on, this digital cash can be used in transactions. By using cryptographic techniques, the system provides anonymity to the client and basic security services. In addition, the system addresses problems such as ensuring that a client will get a valid receipt for the transaction and guaranteeing that the money is not double spent. We do not detail the cryptographic details, but show how this method can be accommodated in our framework.

Our framework can provide access to the system in exchange for payment using an anonymous payment mechanism. We assume that a client previously obtained some digital coins from a *currency server*. When the client wants access to the system the Client Authentication Module implemented in the `authenticate` function, registered with the Spread library by the `SP_set_auth_method` function passes the coins over the network connection to the Server Authentication Module. To protect itself from accepting double-spent money, the Server Authentication Module contacts the *currency server* and checks the coins received from the client (if necessary another process will be forked as in the SecurID case). If the coins are valid, the Server Authenticated Module will generate a random identifier and register it with the access control policy as a paid user of the appropriate

```
static void auth_client_conn_read(mailbox mbox, int d, struct session_auth_info *sess_auth_p)
{
    char username[MAX_PWORD_USERNAME + 1];
    char clear_password[MAX_PWORD_PASSWORD + 1];
    int ioctl_cmd, ret;
    unsigned char response;

    E_detach_fd(mbox, READ_FD);
    E_detach_fd(mbox, EXCEPT_FD);

    ret = recv(mbox, username, MAX_PWORD_USERNAME, 0);
    if(ret < 0) {
        Alarm(ACM, "auth_client_conn_read: reading username failed on fd %d\n", mbox);
        Sess_session_report_auth_result(sess_auth_p, FALSE);
        return;
    }
    if(ret < MAX_PWORD_USERNAME) {
        Alarm(ACM, "auth_client_conn_read: reading username SHORT on fd %d\n", mbox);
        Sess_session_report_auth_result(sess_auth_p, FALSE);
        return;
    }
    username[MAX_PWORD_USERNAME] = '\0';
    ret = recv(mbox, clear_password, MAX_PWORD_PASSWORD, 0);
    if(ret < 0) {
        Alarm(ACM, "auth_client_conn_read: reading password failed on fd %d\n", mbox);
        Sess_session_report_auth_result(sess_auth_p, FALSE);
        return;
    }
    if(ret < MAX_PWORD_PASSWORD) {
        Alarm(ACM, "auth_client_conn_read: reading password SHORT on fd %d\n", mbox);
        Sess_session_report_auth_result(sess_auth_p, FALSE);
        return;
    }
    clear_password[MAX_PWORD_PASSWORD] = '\0';
    if(check_password(username, clear_password)) {
        response = 1;
        send(mbox, &response, 1, 0);
        Sess_session_report_auth_result(sess_auth_p, TRUE);
    } else {
        response = 0;
        send(mbox, &response, 1, 0);
        Sess_session_report_auth_result(sess_auth_p, FALSE);
    }
    return;
}
```

Figure 7: Password Server Authentication Module (communication code)

groups. Then, for as long as the payment was valid, the client will be permitted to access the groups they paid for and the server has no knowledge of the client's identity.

We note that in case the client gets disconnected, the client might get less service than he paid for because when the network connection was terminated the binding between connection and paid user was lost. In order to avoid this, the Server Authentication Module can generate a random token, together with a random userid and a timestamp and encrypt them and then pass the encrypted string back to the client. If the client gets disconnected it can present this encrypted string to the server and the server can decrypt the string and validate that the user is a currently paid user and give the client access immediately.

4.6 Group-Based Authentication

In all the previous authentication methods presented, the authentication of a client is handled by the server that the client connects to. In larger, non-homogeneous environments authentication may involve some or all of the group communication system servers. Although these protocols may be more complex, they can provide better mappings of administrative domains, and possibly better scalability.

An example of such a protocol is when a server does not have sufficient knowledge to check a client's credentials (for instance a certificate). In this case, it sends the credentials to all the servers in the configuration and each server then attempts to check the credentials itself and sends

```c
int pword_authenticate(int fd, void *data_p)
{
    struct user_password *user_p;
    int ret;
    char response;

    user_p = data_p;

    /* Send username and password */
    while(((ret = send( fd, user_p->username, MAX_PWORD_USERNAME, 0 )) == -1) &&  ((errno == EINTR) || (errno == EAGAIN)))
        ;
    if( ret != (MAX_PWORD_USERNAME)) {
        printf("pword_authenticate: unable to send username %d %d: %s\n", ret, MAX_PWORD_USERNAME, strerror(errno));
        return(0);
    }
    while(((ret = send(fd, user_p->password, MAX_PWORD_PASSWORD, 0)) == -1) && ((errno == EINTR) || (errno == EAGAIN)))
        ;
    if( ret != (MAX_PWORD_PASSWORD)) {
        printf("pword_authenticate: unable to send password %d %d: %s\n", ret, MAX_PWORD_PASSWORD, strerror(errno));
        return(0);
    }
    /* Receive response code */
    ret = recv(fd, &response, 1, 0);
    if(ret < 0) {
        printf("pword_authenticate: reading response failed on mailbox %d\n", fd );
        return(0);
    }
    if(ret < 1) {
        printf("pword_authenticate: reading response string SHORT on mailbox %d\n", fd );
        return(0);
    }
    if(response == 1)
        return(1);
    else
        return(0);
}
```

Figure 8: Password Client Authentication Module

an answer back. If at least one server succeeds, the client is authenticated. The particularity of such a protocol is that the servers need to communicate between them as part of the authentication process. Since all the servers can communicate between them in our system, the framework provides all necessary features that allows the integration of such a group-based authentication method.

The code demonstrating how this protocol could be implemented in the framework is provided in Figures 10 and 11. This module utilizes the Events system in Spread, just as the password module did, to listen for available messages from the network. In this case these messages may not only come from the client sending their initial credentials, but also from other group authentication modules communicating through Spread messages, as shown in the function `group_auth_mcast_read` in Figure 11. When the module is initialized, it connects to the Spread daemon just like an application client would, using the username “group_auth_mod” and then joins a group called “auth” with all the other group authentication modules. Two kinds of authentication messages are sent through Spread. First, creds_msg is sent by the server the client connected to to all of the other servers. When a server receives a creds_msg, it does a local check to see if it can authenticate the client with those credentials and sends its answer back to the initial server in a resp_msg. The first server collects all of the resp_msg messages and when it has received them all it locally computes the complete decision in the `check_responses` function and notifies the daemon.

4.7 Access Control

We realize that the above case studies are focused on authentication. Few standard access control protocols that we could use as case studies exist. To demonstrate the ability of the access control architecture we create a case study about an imaginary secure IRC system. Consider a set of users where some users are allowed to chat on the intelligence group, while others are restricted to the operations group. Some are allowed to multicast to a group but are not allowed to read the

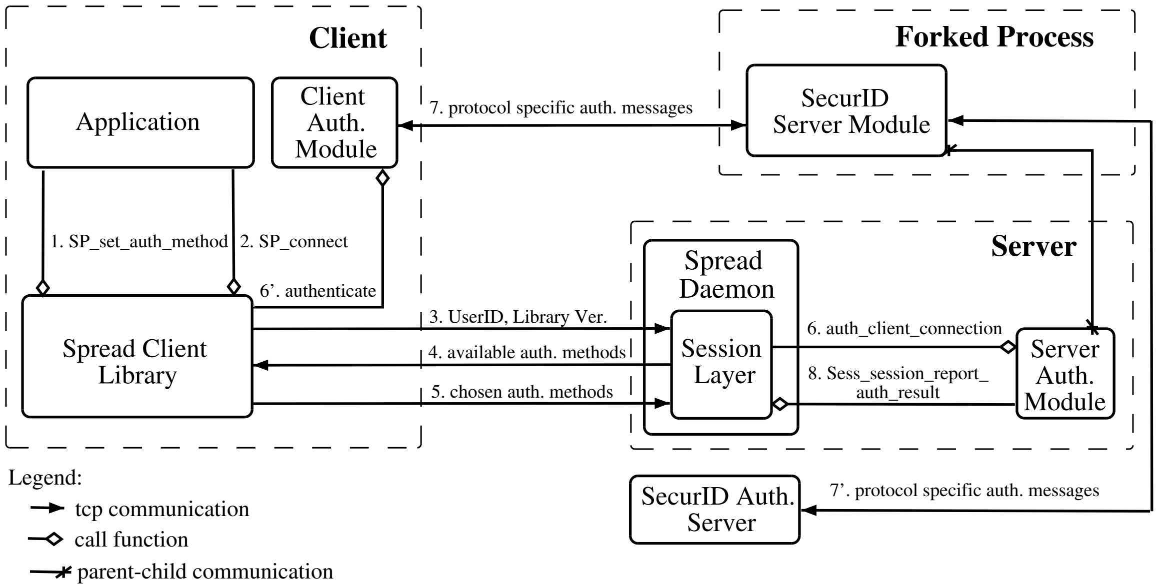

Figure 9: SecurID Authentication

group messages (virtual drop-box). Our framework supports these access control policies through appropriate implementation of the join and multicast hooks defined in Figure 3. Access control modules support identity based, role based, or credential based restrictions.

5 Conclusions and Future Work

We presented a flexible implementation of an authentication and access control framework in the Spread wide area group communication system. Our approach allows an application to write its own authentication and access control modules, without needing to modify the Spread client or server code. The flexibility of the system was presented by showing how a wide range of authentication methods can be implemented in our framework.

There are a lot of open problems that are subject of future work. These include: providing tools that allow an application to actually specify a policy, handling policies in a system supporting network partitions (for example merging components with different policies), providing support for meta-policies defining which entity is allowed to create or modify group policies, and developing dynamic group trust protocols for authentication.

References

[1] Amir, Y. *Replication using Group Communication over a Partitioned Network*. PhD thesis, Institute of Computer Science, The Hebrew University of Jerusalem, Jerusalem, Israel, 1995.

[2] Amir, Y., Awerbuch, B., Danilov, C., and Stanton, J. Flow control for many-to-many multicast: A cost-benefit approach. Tech. Rep. CNDS-2001-1, Johns Hopkins University, Center of Networking and Distributed Systems, 2001.

```
static struct  auth_ops GroupAuth_ops = {
    group_auth_client_connection,
    group_auth_monitor_connection,
};
struct creds_msg {
    mailbox client_mbox;
    unsigned char cred[MAX_CREDENTIAL_LEN];
};
struct resp_msg {
    mailbox client_mbox;
    unsigned char response;
};

static mailbox GA_SpreadConnection;
static char GA_SpreadName[MAX_PRIVATE_NAME];
static session_auth_info *GA_index[MAX_FD];

void group_auth_init(void)
{
    char file_name[80];
    sprintf(file_name, "spread.access_pword");

    if (!Acm_auth_add_method("GROUP", &GroupAuth_ops)) {
        Alarm( EXIT, "group_auth_init: Failed to register GROUP. Too many ACM methods registered.\n");
    }

    SP_connect( ‘‘4803’’, ‘‘group_auth_mod’’, 1, 1, &GA_SpreadConnection, GA_SpreadName);
    SP_join( GA_SpreadConnection, ‘‘auth’’ );
    E_attach_fd(GA_SpreadConnection, READ_FD, group_auth_mcast_read, 0, NULL, LOW_PRIORITY);
    E_attach_fd(GA_SpreadConnection, EXCEPT_FD, group_auth_mcast_read, 0, NULL, LOW_PRIORITY);
}

static void group_auth_client_conn_read(mailbox mbox, int dummy, struct session_auth_info *sess_auth_p)
{
    struct creds_msg creds;
    int ret,msg_len;

    E_detach_fd(mbox, READ_FD);
    E_detach_fd(mbox, EXCEPT_FD);

    ret = recv(mbox, creds.cred, MAX_CREDENTIAL_LEN, 0);
    if(ret < 0) {
        Alarm(ACM, "group_auth_client_conn_read: reading creds failed on fd %d\n", mbox);
        GA_index[sess_auth_p->mbox] = NULL;
        Sess_session_report_auth_result(sess_auth_p, FALSE);
        return;
    }
    /* creds should be validated here before being multicast */

    SP_multicast(GA_SpreadConnection, AGREED_MESS, ‘‘auth’’, CRED_MSG, sizeof(creds), creds);
    return;
}
void group_auth_client_connection(struct session_auth_info *sess_auth_p)
{

    GA_index[sess_auth_p->mbox] = sess_auth_p;

    E_attach_fd(sess_auth_p->mbox, READ_FD, (void (*)(int,int,void *)) group_auth_client_conn_read, 0, sess_auth_p, LOW_PRIORITY);
    E_attach_fd(sess_auth_p->mbox, EXCEPT_FD, (void (*)(int,int,void *)) group_auth_client_conn_read, 0, sess_auth_p, LOW_PRIORITY);
    return;
}
```

Figure 10: Group Authentication Module (initialization and client code)

[3] Amir, Y., Danilov, C., and Stanton, J. A low latency, loss tolerant architecture and protocol for wide area group communication. In *Proceedings of the International Conference on Dependable Systems and Networks* (June 2000), pp. 327–336.

[4] Amir, Y., Dolev, D., Kramer, S., and Malki, D. Transis: A communication sub-system for high availability. *Digest of Papers, The 22nd International Symposium on Fault-Tolerant Computing Systems* (1992), 76–84.

[5] Amir, Y., Moser, L. E., Melliar-Smith, P. M., Agarwal, D., and Ciarfella, P. The Totem single-ring ordering and membership protocol. *ACM Transactions on Computer Systems 13*, 4 (November 1995), 311–342.

```
static void group_auth_mcast_read(mailbox mbox, int dummy, void *dummy_p)
{
    char sender[MAX_PRIVATE_NAME];
    char groups[20][MAX_GROUP_NAME];
    char buffer[1000];
    int service, num_groups, endian;
    int16 mess_type;
    unsigned char response;
    struct group_auth_info *gauth_info;
    struct session_auth_info *sess_auth_p;
    struct creds_msg *creds;
    struct resp_msg *resp;

    assert(mbox = GA_SpreadConnection);

    SP_receive(GA_SpreadConnection, &service, sender, 20, &num_groups, groups, &mess_type, &endian, 1000, buffer);

    if (mess_type == CRED_MSG){
        creds = (cred_msg *) buffer;
        response = check_creds(creds->cred);
        if (!strcmp(sender, GA_SpreadName) ) {
            sess_auth_p = GA_index[creds->client_mbox];
            gauth_info = sess_auth_p->module_data;
            gauth_info->response[gauth_info->num_responses] = response;
            gauth_info->num_responses++;
        } else {
            struct resp_msg r;
            r.client_mbox = creds->client_mbox;
            r.response = response;
            SP_multicast(GA_SpreadConnection, RELIABLE_MESS, sender, RESP_MSG, 1, r);
        }
        return;
    }
    if (mess_type == RESP_MSG) {
        resp = (resp_msg *) buffer;
        sess_auth_p = GA_index[resp->client_mbox];
        gauth_info = sess_auth_p->module_data;
        gauth_info.responses[gauth_info.num_responses] = resp->response;
        gauth_info.num_responses++;
    }
    if (gauth_info.num_responses < cur_membership)
        return;
    if(check_responses(gauth_info->responses)) {
        response = 1;
        send(sess_auth_p->mbox, &response, 1, 0);
        GA_index[sess_auth_p->mbox] = NULL;
        Sess_session_report_auth_result(sess_auth_p, TRUE);
    } else {
        response = 0;
        send(sess_auth_p->mbox, &response, 1, 0);
        GA_index[sess_auth_p->mbox] = NULL;
        Sess_session_report_auth_result(sess_auth_p, FALSE);
    }
    return;
}
```

Figure 11: Group Authentication Module (Spread message handler)

[6] Amir, Y., and Stanton, J. The Spread wide area group communication system. Tech. Rep. 98-4, Johns Hopkins University, Center of Networking and Distributed Systems, 1998.

[7] Ballardie, T. Scalable multicast key distribution. RFC 1949, 1996.

[8] Ballardie, T., Francis, P., and Crowcroft, J. Core based trees: An architecture for scalable interdomain multicast routing. In *Proceedings of ACM SIGCOMM'93* (1993), pp. 85–95.

[9] Birman, K. P., and Joseph, T. Exploiting virtual synchrony in distributed systems. In *11th Annual Symposium on Operating Systems Principles* (November 1987), pp. 123–138.

[10] Birman, K. P., and Renesse, R. V. *Reliable Distributed Computing with the Isis Toolkit.* IEEE Computer Society Press, March 1994.

[11] Caronni, G., Waldvogel, M., Sun, D., Weiler, N., and Plattner, B. The VersaKey framework: Versatile group key management. *IEEE Journal of Selected Areas in Communication 17*, 9 (September 1999).

[12] Fekete, A., Lynch, N., and Shvartsman, A. Specifying and using a partitionable group communication service. In *Proceedings of the 16th annual ACM Symposium on Principles of Distributed Computing* (Santa Barbara, CA, August 1997), pp. 53–62.

[13] Hardjono, T., Cain, B., and Monga, I. Intradomain group key management protocol. Work in Progress, November 1998.

[14] Harkins, D., and Doraswamy, N. A secure scalable multicast key management protocol (MKMP). Work in Progress, November 1997.

[15] Harney, H., Colegrove, A., Harder, E., Meth, U., and Fleischer, R. Group secure association key management protocol (GSAKMP). draft-irtf-smug-gsakmp-00.txt, November 2000.

[16] Harney, H., and Muckenhirn, C. Group key management protocol (GKMP) specification. RFC 2093, July 1997.

[17] Hayton, R., Herbert, A., and Donaldson, D. FlexiNet — A flexible component oriented middleware system. In *Proceedings of SIGOPS'98* (1998).

[18] Kihlstrom, K. P., Moser, L. E., and Melliar-Smith, P. M. The SecureRing protocols for securing group communication. In *Proceedings of the IEEE 31st Hawaii International Conference on System Sciences* (Kona, Hawaii, January 1998), vol. 3, pp. 317–326.

[19] Kohl, J., and Neuman, B. C. The Kerberos Network Authentication Service (Version 5). RFC-1510, September 1993.

[20] McDaniel, P., Prakash, A., and Honeyman, P. Antigone: A flexible framework for secure group communication. In *Proceedings of the 8th USENIX Security Symposium* (August 1999), pp. 99–114.

[21] Medvinsky, G., and Neuman, B. C. Netcash: A design for practical electronic currency on the internet. In *Proceedings of First ACM Conference on Computer and Communications Security* (November 1993).

[22] Moser, L. E., Amir, Y., Melliar-Smith, P. M., and Agarwal, D. A. Extended virtual synchrony. In *Proceedings of the IEEE 14th International Conference on Distributed Computing Systems* (June 1994), IEEE Computer Society Press, Los Alamitos, CA, pp. 56–65.

[23] Nystrom, M. The SecurID SASL mechanism. RFC-2808, April 2000.

[24] Reiter, M. K. Secure agreement protocols: reliable and atomic group multicast in RAMPART. In *Proceedings of the 2nd ACM Conference on Computer and Communications Security* (November 1994), ACM, pp. 68–80.

[25] Renesse, R. V., K.Birman, and Maffeis, S. Horus: A flexible group communication system. *Communications of the ACM 39* (April 1996), 76–83.

[26] Rodeh, O., Birman, K., and Dolev, D. The architecture and performance of security protocols in the Ensemble group communication system. *ACM Transactions on Information and System Security* (To appear).

[27] Samar, V., and Schemers, R. Unified login with Pluggable Authentication Modules (PAM). OSF-RFC 86.0, October 1995.

[28] Thompson, M., Johnston, W., Mudumbai, S., Hoo, G., Jackson, K., and Essiari, A. Certificate-based access control for widely distributed resources. In *Proceedings of the Eighth Usenix Security Symposium* (August 1999), pp. 215–227.

[29] Whetten, B., Montgomery, T., and Kaplan, S. A high performance totally ordered multicast protocol. In *Theory and Practice in Distributed Systems, International Workshop* (September 1994), Lecture Notes in Computer Science, p. 938.