SCOM: A Security and Privacy Layer for Horus

David A. Cooper*

Dept. of Computer Science
Cornell University
Ithaca, NY 14853–7501

Abstract

A large number of distributed applications will soon be implemented for which the inclusion of some form of security will be important. While the programmers who design these applications may have expertise in the application area, they most likely will not be experts in the areas of distributed computing or security. In both of these areas, a firm knowledge of the subject is necessary in order to avoid subtle, hard to detect flaws in design.

The Horus system was created to enable the design and implementation of distributed applications. The ultimate goal of this work is to allow for the implementation of efficient distributed applications while hiding many of the problems associated with the design of such applications from the applications programmer. In this paper, we describe work we have done to facilitate the implementation of secure distributed applications on top of the Horus system.

1 Introduction

The design of secure distributed applications can be an error prone and time consuming process. Many of the problems associated with implementing such applications can be avoided by using a well designed distributed computing toolkit. One complication in the design of such toolkits is that different applications vary widely in the type of support that they need. As a result of the varying needs of applications, many toolkits that work well for one type of application are unsuitable for many other types of applications. This may be the result of the toolkit being too heavyweight or too lightweight. A toolkit that is too heavyweight may include unnecessary features which would make it impossible or inefficient to implement many applications using the toolkit. A toolkit that is too lightweight, on the other hand, may not provide enough support for the application.

The Horus system [16] was designed to address this problem by providing a highly flexible interface to the applications programmer which allows the programmer to choose those features of the system which are useful and leave out those which are not needed or which would adversely affect the application. The system includes support for many of the properties that are useful in the design of distributed applications (consistent group membership, reliable message delivery, total and causal ordering, etc.) but allows the application to specify which properties are needed. Since support for many of these properties impacts system performance, the ability to exclude unnecessary properties from the system can improve performance.

The majority of the work done in Horus so far has involved providing support for a large number of features in environments in which only benign failures occur. Recently, we began work to extend the Horus architecture to support applications which must be able to tolerate arbitrary or malicious failures. The overall plan for our work is to begin by providing low level support for such applications and then, over time, to add higher levels of support by building on top of the lower levels.

In this paper, we describe the first stage of our effort. We begin by providing a brief overview of the Horus system and by describing the model we are using for our security work. After this, we will describe a server we have implemented which is responsible for disseminating information vital for secure communication (such as authentication information). We will then describe the design of a layer we have added to the Horus system, the SCOM layer, which implements secure communications channels and which facilitates the implementation of access control mechanisms. We conclude by presenting an overview of our plans for providing higher level support for the implementation of secure applications.

*The work reported was supported by ARPA/ONR grant N00014-92-J-1866. The views expressed herein are those of the author and do not represent the opinions of ARPA/ONR.

2 Related Work

Many systems have been designed to enable the implementation of secure distributed applications. Most of these systems can be classified as being based on either group communication or objects. In object based systems (such as Theta [14] and Legion [6, 17]), clients invoke the objects' methods which are instantiated on a set of servers. Usually, the servers are trusted and the primary security concerns are authentication and access control.

Horus was designed around the group communication model as were Rampart [10], Totem [7], and Transis [2]. In these systems, processes organize themselves into groups for communications purposes. Within these groups, processes can communicate with each other using either point-to-point or multicast communication. Generally, these systems provide support to the application programmer by providing features such as consistent failure detection, all-or-nothing message delivery, and message ordering (e.g. FIFO, causal, or total).

Many of the ideas for the current security architecture for Horus came from an earlier effort to create a security architecture for a previous version of Horus. In [11], Reiter, Birman and van Renesse describe a security architecture which allows groups to protect themselves from outside attack. In this system, processes wishing to join a group must identify themselves to a group coordinator which controls access to the group. If the new process is granted permission to join the group, then it is provided with copies of the keys used by the group members to authenticate each other and to encrypt messages sent among each other. The system makes use of time and authentication servers whose design were used as the basis for the trusted server in our system. The main drawback of this system for use in the current version of Horus is its inability to tolerate corrupt group members.

The Rampart system presents one possible solution to this problem. In Rampart, the basic group communication primitives such as group membership [9] and totally ordered multicast [8] were designed to handle the arbitrary failure of a limited number of group members. As was described in [10], a system with these primitives can be used in combination with the state machine approach to replication [13] to easily create services which can tolerate the malicious failure of a limited number of servers. Unfortunately, the protocols necessary to provide such a high level of support are very expensive. While it may be necessary to pay such a cost in some circumstances, in many applications the failure model will be simpler or such properties as consistent group membership and totally ordered multicast will not be needed. It is our belief that it is important to make less costly mechanisms available for these applications.

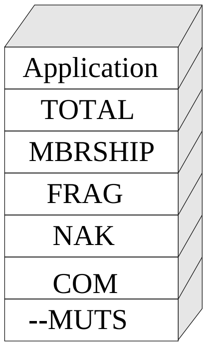

Figure 1: A typical protocol stack

3 Horus

Horus is a system which provides a group communication abstraction to applications programmers. It consists of a small, machine dependent kernel and a set of machine independent protocol layers. The kernel, −−MUTS, provides a machine independent interface to basic primitives such as memory allocation, message construction, and message sending and receiving. For each group within the system, the programmer chooses a stack of protocol layers to place on top of −−MUTS over which the application's code for the group will execute.

In order to make the layers stackable, each layer imports and exports the same interface, the Horus Common Protocol Interface (HCPI), of upcalls and downcalls. The interface includes downcalls and corresponding upcalls for sending, receiving, and acknowledging messages; joining, leaving, and merging groups; and installing group views. Figure 1 shows a typical protocol stack. The first layer on top of −−MUTS is the COM (communication) layer. The COM layer is different from the other layers in that its main purpose is to anchor the rest of the protocol stack to −−MUTS. It does this by using HCPI to communicate with layers above it while using a simple send and receive interface to communicate with −−MUTS. Currently, the COM layer is the only anchor layer available for applications which do not need the security features of the SCOM layer. Since the COM layer acts as an all-purpose anchor layer, it was designed to do as little as possible, leaving the work to the higher level, optional layers.

The remaining layers in the stack are responsible for providing the "nice" semantics which ease the work of the applications programmer. The NAK (negative acknowledgement) layer uses negative acknowledgements to provide FIFO, reliable communication. It also notifies the higher layers when other processes in the group view are unresponsive. The FRAG (fragmentation) layer breaks up large messages into pieces small enough for the underlying network. The MBRSHIP (membership) layer ensures that group members maintain consistent views of the group membership and that message delivery obeys virtual synchrony. The MBRSHIP layer relies on a lower layer (such as NAK) to provide information about failed members. Finally, the TOTAL layer enforces a total ordering on the delivery of multicast messages.

In addition to the layers used in the example above, there are currently more than 20 other layers which provide causally ordered message delivery, message stability information, state transfer, hierarchical group support, and many other properties.

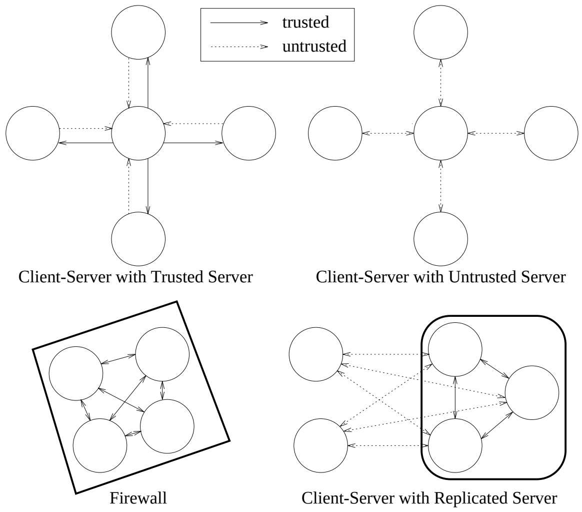

Figure 2: Trust Models

4 Trust Model

One of the main issues that must be considered in designing a secure system is the trust model. The trust model describes which processes trust which other processes in the system and to what degree the processes are trusted. In figure 2, we show four possible trust models. The first model (client-server with trusted server) represents the trust model of our trusted server (described below). All of the clients trust the server and assume that any information that is verified to have come from the server is correct. The server, on the other hand, does not trust any of the clients. This model can be used in cases in which the server carefully scans all incoming messages before acting on those messages and in which any "state changes" resulting from client messages are carefully considered.

Sometimes, the lack of trust between clients and servers is mutual. One example of this involves World Wide Web clients and servers. Clients send requests for information to Web servers. Since the servers do not trust the clients, the requests are carefully scanned. Similarly, responses from the servers are also treated as untrustworthy [1].

In some cases, however, all of the processes within a group fully trust each other. This is the firewall model shown in figure 2 which was the model used in the earlier version of Horus. A group using this trust model only needs a way to prevent untrusted members from joining the group and to prevent processes outside of the group from impersonating group members (encryption may also be necessary if group members send confidential messages among themselves). This model may be combined with the other models in situations in which replicated servers are used.

While not shown in figure 2, the degree to which processes trust each other may vary. In most protocols which are tolerant to arbitrary failures, it is assumed that some of the processes may fail, but that there can be no a priori knowledge of which processes will fail. In this case, processes neither fully trust nor distrust each other, but instead use voting schemes in order to gain confidence in information acquired from others.

While the trust models that we described above are representative of some of the most common models used by applications, there are many other models which programmers may wish to use. Since it would be impossible for us to predict all of the possible trust models which programmers may wish to use, one of our main goals in designing the SCOM layer was to avoid placing any restrictions on the trust model used by applications. Higher level layers which provide more sophisticated semantics to the application (such as group membership, all-or-nothing message delivery, total ordering, etc.) may, by necessity, place various restrictions on the trust model. The design of the SCOM layer, however, with its simple semantics, did not require us to make many assumptions in order to ensure correct behavior. Since the SCOM layer was designed to be an all-purpose replacement for the COM layer in secure systems, the flexibility of this layer was most important.

[1] For example, Java code [4] is sent through a bytecode verifier before it is executed and the Java language is designed to limit the actions that a Java program may take.

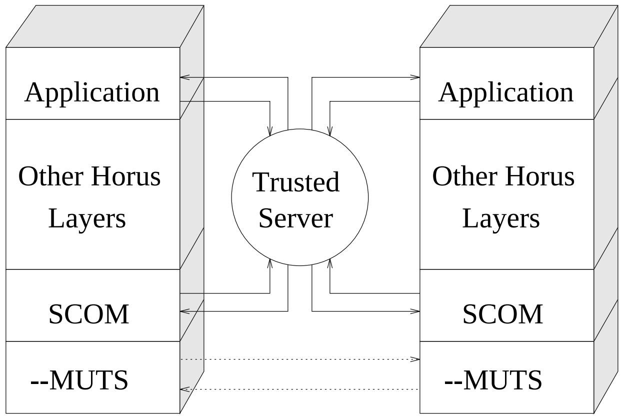

Figure 3: A Horus system using the SCOM layer

5 The Trusted Server

One of the fundamental requirements of a secure system is the ability to perform authentication. In order to perform authentication, processes must either have or have access to information about other processes in the system. One way to accomplish this would be to distribute authentication information off-line and to assume that the information will not change over time (i.e. assume that keys will never be stolen and that access rights will never be revoked). However, in most cases this would be an unworkable solution. In our system, we use a server to distribute information necessary for authentication. In order for this solution to be effective, all of the processes in the system must trust the server. While this solution imposes a trust requirement on the trust model used by any application built on top of the SCOM layer, we believe that the benefits of an on-line authentication server justify our decision to use one [2].

In figure 3, we show a typical setup for a two process system which uses the SCOM layer. As shown, both the SCOM layer and the application interact with the trusted server. In addition to providing information vital to authentication, there is other useful (but less important) information which may be distributed by a trusted server. Our trusted server is also used in order to enable processes to synchronize their clocks as well as to securely distribute directory information. In this section, we will describe each of the functions of the trusted server in detail.

[2] As was described in [11], an authentication service can be built using a set of servers in such a way that the service as a whole will continue to behave correctly even if a limited number of the servers crash or behave maliciously. In the future, we may make such a replicated version of the authentication service available in our system.

5.1 Time

The SCOM layer makes use of time in two places. In order to limit the ability of an attacker to make use of a stolen key, the trusted server distributes authentication certificates with timestamps. The processes use the timestamps in order to determine the freshness of certificates. Time is also used in the key exchange and replay detection protocols. Since these protocols are essential to the security of the system, a failure in the clock synchronization protocol could lead to serious security problems (see [3]).

In our system, we use the clock synchronization protocol proposed by Cristian in [1] which works (approximately) as described below. The local clock of the trusted server is treated as a source of "real" time and each process in the system periodically polls the trusted server for the time. If the server responds that the current time is t_s and the round-trip time needed to make the request was δ then the process knows that the current time when it receives the response is between t_s and $t_s + \delta$. As time passes on its local clock, the processes' certainty about the current time will diminish as a result of clock drift. However, if the maximum clock drift can be determined, then the process can continue to maintain an estimate of the current time. If the maximum clock drift is ρ, then after Δ_t seconds have passed on the local clock since receiving the response from the trusted server, the process will know that the actual time is between $t_s + \Delta_t/(1+\rho)$ and $t_s + \delta + \Delta_t/(1-\rho)$. When an estimate of the time is needed, either the lower bound, L_t, or the upper bound, H_t, is used, depending on which is the conservative estimate.

In our system, we use a value for ρ of 2×10^{-5}. Using this value, we can expect H_t and L_t to remain within a few seconds of each other even if several hours pass between clock synchronizations. As a result, it is reasonable to use an unreplicated time server even though this will result in the time server being unavailable periodically (as a result of power failures or server crashes). Since a single server is easier to protect than multiple servers, this is one way to help ensure that the time service will not be compromised.

5.2 Authentication Certificates

One of the most effective ways for an attacker to compromise a system is to find some way to impersonate a legitimate user of the system. An attacker could attempt to accomplish this by either stealing the keys of one of the users or by tricking processes into accepting phony authentication information. Therefore, in order to prevent this type of attack, it is essential that any source of authentication information be carefully protected. In our system, users authenticate themselves to others by demonstrating that they possess the private half of a public key pair. It is the responsibility of the authentication server to associate public keys with users and to securely distribute this information to the processes.

The first line of defense against impersonation attacks must be handled by the administrator of the authentication service. The administrator must carefully identify each of the users (off-line) and collect their public keys. These (user, public key) pairs are placed in a file which is loaded by the trusted server at start-up. The administrator must also be available to remove (user, public key) pairs from the system whenever a user's key is stolen.

The trusted server distributes (user, public key) pairs by issuing authentication certificates. An authentication certificate contains a user's name along with a copy of the user's public key and is signed with the private key of the trusted server. The result of using a private key signature is that the certificate can be forwarded by one process to another process allowing processes to authenticate each other without directly involving the trusted server.

Even though the trusted server does not need to be directly involved when processes authenticate each other, the server does need to be contacted periodically in order to ensure that (user, public key) pairs are still valid. In order to enable this, the trusted server includes a timestamp [3] in every authentication certificate it creates. Each certificate is valid for a specific length of time, KEY_TIMEOUT, and will not be used by a process after its local estimate of the time, H_t, exceeds TS(certificate) + KEY_TIMEOUT.

5.2.1 Authenticating Addresses

In the Horus system, addresses (entity identifiers) are used to specify communications endpoints. In many cases, the access control mechanisms of a process group rely on the ability to use entity identifiers (eid's) as a means of uniquely identifying an endpoint. A failure of the system to support this assumption could lead to unexpected results.

Figure 4 shows an example of a possible attack against a system which does not protect against address spoofing. In the figure, coordinator, member

[3] Since the local clock of the trusted server is defined to be "real" time, authentication certificates are timestamped with the "real" time. If the authentication service was being run on a different computer than the time service, then the authentication service would act as a client of the time service and would use its lower bound on the time, L_t, to timestamp certificates.

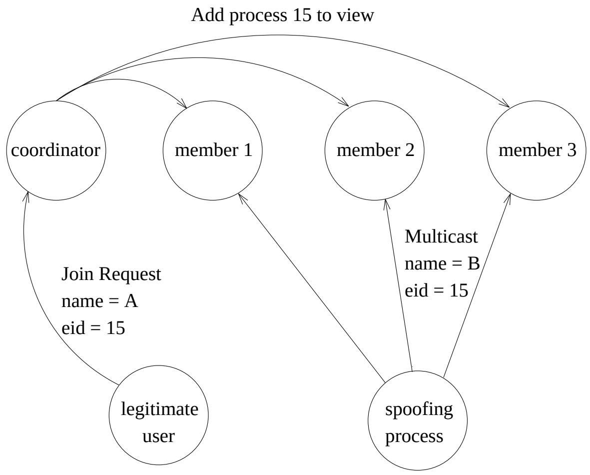

Figure 4: Entity identifier spoofing

1, member 2, and member 3 form a group which is running on top of the stack depicted in figure 1 and legitimate user is a process which is attempting to join the group. All of the group members trust coordinator and, in order to maintain consistent group views, allow coordinator to make all access control decisions. Legitimate user requests permission to join the group by sending a JOIN_REQUEST message to coordinator containing its eid and enough information for coordinator to identify it as user A. The coordinator grants the request and sends a message to the other group members informing them of the new member by giving them the eid of the new member. Later, another process, spoofing process, sends a message to member 1, member 2, and member 3. Since spoofing process claims to be at eid 15 and the coordinator told the group members to accept messages from the process at eid 15, the members accept the message. The result is that spoofing process has circumvented the access control mechanisms of the group even though it is unable to lie about its identity (only its eid).

In order to protect against this type of attack, we use the authentication certificates of the trusted server in order to prevent more than one user from claiming "ownership" of the same eid. Since it would be infeasible to have an administrator manually assign ownership to eid's, we allow processes to register eid's with the trusted server on-line. A process wishing to establish a communications endpoint uses the `mt_neweid` function to acquire a fresh eid and then attempts to register the eid under its owner's name with the trusted server [4]. Under normal circumstances, the eid

[4] A single process may take on multiple identities as long as a different eid is used for each identity.

will be successfully registered. However, the trusted server assigns ownership to eid's on a first-come first-served basis making it possible that any given eid may have been previously registered by an adversary. In this case, the process must continue to call `mt_neweid` until it receives an eid which it can successfully register. Once the eid has been registered, the process may request a certificate for that eid from the trusted server. The certificate will contain the eid, the eid's owner, and the public key of eid's owner. The certificate will also be signed and will contain a timestamp. The certificate will then be used by the process when authenticating itself to other processes in order to prove its ownership of the eid that it is using.

5.3 Directory Information

Even though the SCOM layer does not directly make use of a directory service, processes need to be able to locate each other in order to be able to communicate. In order to prevent an attacker from being able to disrupt the ability of processes to acquire location information, the system must have some secure way of disseminating this information. Since users must already trust the trusted server as a source of time and authentication certificates, it makes sense to use the trusted server to disseminate directory information as well.

Directory information differs from the other types of information that the trusted server distributes in that the trusted server makes no attempt to verify the correctness of this information. Instead, the server authenticates the source of the information and allows processes to accept or reject the information based on whether or not they trust the source. In order to prevent one user from destroying the directory information stored by another user, the trusted server maintains a separate database for each user. The database of any user can be read, but not modified, by any other user. It is our assumption that any process requesting directory information will know the user (or set of users) from which it is willing to accept such information.

6 The SCOM Layer

The SCOM (secure communication) layer is an anchor layer which was designed to be an alternative to the COM layer. We designed SCOM as an anchor layer since anchor layers are the only layers which have access to the entire contents (including headers) of every message. Any security layer placed above the anchor layer would not have access to the message headers attached by layers stacked below it and therefore would be unable to sign those headers, leaving them vulnerable to modification.

As with the COM layer, the semantics provided by the SCOM layer are simple. One of the goals in the design of the SCOM layer was to mimic the non-security related semantics of the COM layer as closely as possible. In addition, since the SCOM layer was designed to be a general purpose replacement for the COM layer, we attempted to minimize any restrictions that the SCOM layer might place on the trust model (see section 4).

The main contributions of the SCOM layer over the COM layer are transparent message authentication and the support for message encryption. The SCOM layer acquires certificates from the trusted server, handles key exchange between communicating processes, signs all outgoing messages, and verifies all incoming messages. In addition, the SCOM layer will encrypt any messages for which an encrypt option is specified in the message send downcall and will transparently decrypt any encrypted incoming messages. The result is that higher level layers are protected from eavesdropping, message modification, message replay, and address spoofing attacks. Since message origin and freshness are guaranteed by the SCOM layer, simple access control policies can be implemented very easily.

6.1 Message Authentication

Message authentication can be divided into two separate parts: verification of message origin and verification of freshness. Message origin is verified by securely exchanging signature keys and using those keys to sign and verify messages. In our system, all messages are signed using a secret key signature scheme. Since one of the objectives of the SCOM layer was to prevent one process from impersonating another process, even if both processes belong to the same group, the use of secret key signatures requires that all message authentication be performed on a point-to-point basis. Since exchange of the point-to-point keys is expensive, we made the key exchange process separate from the group membership mechanism (i.e. a pair of communicating processes share exactly one secret signature key between them independent of the number of groups to which they belong in common).

6.1.1 Replay Detection

One well known attack against distributed system security is to collect and replay messages that have been previously sent by valid participants in a protocol. Since the messages were generated by valid participants, they will be properly signed and so, without protection, may be accepted by an unlucky recipient. We prevent this type of attack in our system using two separate mechanisms. First, all messages are timestamped and will only be accepted as "valid" for a limited period of time. Second, all messages received by a process are cached until they are no longer valid in order to detect duplicates.

When a message is sent, it is timestamped with the sender's current estimate of the time, L_t. Upon delivery, the recipient checks the timestamp, t, and only accepts the message if $H_t \leq t +$ MSG_TIMEOUT. The use of L_t by the sender and H_t by the recipient ensures that the valid period of the message will not exceed MSG_TIMEOUT no matter how far L_t and H_t drift from "real" time. One consequence of using these conservative estimates is that as the local time estimates of the processes deteriorate, the amount of time in which messages will be valid will decrease. This may be particularly troublesome if the time server is unavailable for an extended period since the result may eventually be that messages will expire even before they are sent. In order to avoid this, the value of MSG_TIMEOUT must be set in order to tolerate an extended period of unavailability of the time server. In our system, we have set MSG_TIMEOUT to 30 seconds. In general, this value should be sufficient to allow to system to continue to operate even if the time server is unavailable for a few days. However, the exact value of MSG_TIMEOUT should take into account such things as the expected latencies for messages and the load in the system (which may affect the amount of time necessary to receive responses from the trusted server and thus the quality of the initial estimate of the time).

The use of timestamps and expiration times ensures that only messages that were recently sent will be delivered, however, it does not guarantee that a message will not be delivered more than once during its valid lifetime. Duplicate messages are detected by having each process maintain a set of caches which together store every message the process has received which is still valid [5]. Each cache is implemented as a hash table of message entries. In order to save memory, instead of storing the entire message, an entry in a table consists of the message's timestamp and 4 bytes of the message's signature.

Caution must be used in determining when an entry can be removed from a cache. A process will reject a message as invalid once its high estimate of the time, H_t, exceeds the expiration time of the message. However, due to the nature of the clock synchronization protocol, H_t does not increase monotonically since the result of a synchronization may be for H_t to jump backwards (closer to "real" time). The result of such a backwards jump could be that a message that at one point would be rejected as expired may at some point in the future be considered valid. Therefore, an entry should not be removed from a cache until it can be assured that H_t will exceed the expiration time of the message at all points in the future. One property of the clock synchronization protocol is that the inequality $L_t \leq$ "real" time $\leq H_t$ always holds. Since "real" time is monotonic, we know that for any "real" time, t, the value of H_t will never drop below t (or L_t) at any point in the future. From this we can conclude that an entry can safely be removed from the cache once L_t has exceeded the expiration time of the message (timestamp + MSG_TIMEOUT).

6.1.2 Key Exchange

In designing the SCOM layer, we imposed several restrictions on the design of the key exchange protocol. First, other than the message exchanges necessary to acquire certificates and time information from the trusted server, we wanted to avoid having the SCOM layer send any extra messages (i.e. any information sent by the SCOM layer had to be piggybacked on messages being sent by other layers). Second, the protocol had to be resistant to message loss, reordering, duplication, modification, and forgery. Also, since key exchange is expensive, we wanted to minimize the total number of key exchanges that would occur in the system. Finally, the protocol had to contain the effect that a malicious process could have on processes with which it was communicating.

There were several factors which complicated the design of the key exchange protocol for the SCOM layer. First, since authentication certificates have limited lifetimes, the key exchange protocol must be repeated periodically in order for the signature keys to remain valid. Given the possibility of message loss and our requirement that no extra messages be sent, it is possible that one key exchange may not complete before the next one begins.

A second complication results from the desire to minimize the number of key exchange operations that must be performed. Ideally, for any pair of communicating processes, one process will initiate the key exchange and the other process will use the key generated by the initiating process. However, the communications patterns of some Horus layers (such as the NAK layer) make it likely that processes will initiate communication, and thus key exchange, in par-

[5] Since each message contains the address of its source in the message header and the SCOM layer does not allow a process to send a message to itself, there is no need to cache outgoing messages.

allel. This was, again, complicated by the possibility of message loss.

Figures 6 and 7 show a simplified version of the protocol used in the SCOM layer for the exchange of signature keys and point-to-point encryption keys. When a process wishes to send a message, it first determines whether its own keys or the keys created by the intended recipient are fresher [6]. The fresher keys are used for three reasons. First, if the sending process has already received a set of keys from the intended recipient, but has not yet generated any keys of its own, it can avoid the cost of generating and sending a set. Second, if the two processes send their keys in parallel, the keys with the longest remaining lifetime will be used. Finally, since the processes use the same set of keys, acknowledgement of the receipt of the keys is implicit.

After choosing the fresher of the two keys, the sending process checks the age of the chosen keys to see if they can still be used. In order to avoid having a message be dropped as a side-effect of the key exchange protocol, the sender must ensure that the recipient will still be able to use the signature key to verify the message up until the time when the message expires. If the sender is using a key that it generated, then the only requirement is that the key remain valid for at least as long as the message. This means that the key expiration time, TS(send keys) + KEY_TIMEOUT, must be at least as late as the message expiration time, L_t + MSG_TIMEOUT.

A similar analysis is used if the keys generated by the intended recipient are to be used. However, in this case, the sender must also ensure that the recipient will not have replaced its keys before the message expires. The recipient may replace its keys as soon as its local clock, L^r_t, reaches TS(receive keys) + KEY_TIMEOUT − MSG_TIMEOUT. Since a message with a timestamp of t will only be accepted by the recipient until H^r_t exceeds t + MSG_TIMEOUT and $H^r_t \geq L^r_t$, the recipient's keys can be safely used until L_t + MSG_TIMEOUT > TS(receive keys) + KEY_TIMEOUT − MSG_TIMEOUT.

If the fresher key can no longer be used, then a new pair of secret signature and encryption keys are generated. The sender first attempts to acquire an authentication certificate for the intended recipient in order to verify that its public key is still valid. If it successful, then the sender acquires a certificate for itself to be included in the message. This certificate

header -	group address message type message flags group encryption key acknowledgement	source address [destination address] message timestamp

sender's certificate -	user name process eid timestamp	public key trusted server's signature
point-to-point keys encrypted with recipient's public key -		secret signature key secret encryption key
signature of point-to-point keys using private key of sender		

secret key signature of -	header [sender's group encryption key] [group encryption key sequence num.]	hash of message

group encryption key -	secret encryption key sequence number

message (possibly encrypted with point-to-point or group encryption key)

Figure 5: Message format

is used by the recipient in order to verify the validity of the sender's public key. The certificate and the signed, encrypted keys are then piggybacked onto the outgoing message (using the format shown in figure 5). The sender continues to attach this information to all subsequent messages until it knows that the keys have been received or until it stops using the keys.

When receiving a message, the recipient begins by checking the ages of its keys and those of the sender and discarding any expired keys. Next, it checks to see if the message contains a set of point-to-point keys. If so, it first verifies that the included certificate is valid, that the eid of the certificate matches that of the sender, and that the certificate has not yet expired. If the timestamp of the certificate matches that of the most recently received certificate from the sender, then it is assumed that the included keys are the same as those already in use. In this case, the recipient merely compares the encrypted keys to the previously received encrypted keys and avoids the cost of decrypting them a second time.

After extracting the keys, the recipient decrypts the message (if necessary) and verifies the signature. If the signatures on the certificate, keys, and message are all valid then the recipient accepts the message and the included keys as valid.

If the message from the sender does not include any point-to-point keys, then the recipient checks the flags in the message header to see which keys the sender

[6] If the timestamps on the send keys and the receive keys in the first line of figure 6 are the same, then an ordering on the eid's of the processes involved is used to determine which key is "fresher".

```
/* If the other processes' keys are fresher, then start using those keys */
if not keys_same and TS(receive keys) > TS(send keys) then
        using_receive_keys = 1;
        keys_same = 1
fi;

/* Check to see if we need to generate new keys */
if (not using_receive_keys and L_t > TS(send keys) + KEY_TIMEOUT − MSG_TIMEOUT) or
    (using_receive_keys and L_t > TS(receive keys) + KEY_TIMEOUT − 2 * MSG_TIMEOUT) then
        acquire fresh certificates for myself and message recipient;
        generate secret signature and encryption send keys;
        using_receive_keys = 0;
        keys_same = 0;
        TS(send keys) = TS(my certificate)
fi;

/* Sign and Encrypt keys, if necessary, and send message */
if not keys_same then
        send Certificate_s, K_s^-1(keys), K_r(keys), <signed message>
else
        send using_receive_keys, <signed message>
fi;
```

Figure 6: Point-to-Point Key Exchange: Sending a Message

```
/* Delete expired keys */
if TS(send keys) < H_t− KEY_TIMEOUT then delete send keys fi;
if TS(receive keys) < H_t− KEY_TIMEOUT then delete receive keys fi;

if message contains a certificate then
        cert_s, key_sig, enc_keys, signed_msg = message;
        receive keys = K_r^-1(enc_keys);
        TS(receive keys) = TS(cert_s);
        using_receive_keys = 0;
        keys_same = 0;
        sender_using_my_keys = 0
else
        sender_using_my_keys, signed_msg = message
fi;

if sender_using_my_keys then
        keys_same = 1;
        verify signed_msg using send keys
else
        verify signed_msg using receive keys
fi;
```

Figure 7: Point-to-Point Key Exchange: Receiving a Message

used. If the sender used the keys generated by the recipient then the recipient knows that the sender has received the keys and does not have to include them in any subsequent outgoing messages.

6.1.3 Signing Messages

As was mentioned above, message authentication, for both point-to-point and multicast messages, is handled on a point-to-point basis. The result of this is that, in general, several secret key signatures will need to be generated for each multicast message sent by a process. Generally a signature is created for a message by first computing a one-way hash of the message and then signing the hash. This is done for efficiency purposes since the length of the hash of the message is independent of the length of the message and is short (typically 16 bytes). In addition, hash functions can usually be computed very quickly.

Ideally, using this scheme, a single hash could be computed for a multicast message which would be used to generate each of the signatures. Unfortunately, the information attached to the message by the SCOM layer differs slightly for each destination as a result of the information added by the key exchange protocols. In the SCOM layer, we get around this problem by first computing a destination independent hash of the message and then computing a set of destination dependent hashes of the destination independent hash, the message header, and the group encryption key information (if included). Since the message header added by the SCOM layer is relatively small, the result of this approach is that the most expensive part of the message signature operation is computed only once per multicast.

6.2 Message Encryption

Unlike message authentication, message encryption is handled differently for point-to-point and multicast messages. Point-to-point and multicast messages are both encrypted using the same encryption scheme but use a different set of keys and a different key exchange protocol. In our implementation, encryption is performed using DES. While DES can be replaced with another encryption scheme, stream ciphers must be avoided since messages may be lost or reordered by the network.

As was mentioned above, the encryption keys for point-to-point messages are distributed using the same protocol which is used to exchange the signature keys. Unlike signatures, however, the sender can decide on a per message basis whether or not to encrypt. In addition to allowing the application programmer to decide which messages need to be encrypted, this also allows for other Horus layers to send messages (such as acknowledgement messages) without paying the cost of encryption.

Since encryption keys are used only for privacy and not authentication, a single encryption key may be held by a large number of processes as long as any message sent using the key is intended to be read by all of the key holders. In many cases, this would mean that all of the members of a group could share a single group encryption key to use for multicasting messages within the group. In our system, however, at the SCOM layer, there is no requirement that group members agree on the group membership. In order to maintain privacy, a message multicast by a process should only be readable by the processes in its view of the group. In order to guarantee this, the SCOM layer uses a separate group encryption key for each group member. The key for each group member is generated by that group member and is shared only with the processes in that group member's view of the group. In order to maintain this policy, a process changes its group encryption key whenever it removes a process from its view.

The result of the key sharing policy is that the key exchange protocol used to exchange group encryption keys is much simpler than that used for point-to-point keys. Each process in a group creates a group encryption key for its own multicast messages whenever necessary. In each message that the process sends within the group (point-to-point or multicast), the sending process includes an encrypted copy of the group encryption key along with the key's sequence number unless it has received an acknowledgement from the recipient for that key. In order to acknowledge group encryption keys, processes include the sequence number of the highest numbered key received so far from the the recipient of the message for the group in which the message is being sent. Since the group encryption key can be encrypted using the point-to-point secret encryption key instead of the public key, key exchange for group encryption keys is substantially less expensive than for point-to-point keys.

In our current implementation, we chose to include message encryption in the SCOM layer in order to share the cost of point-to-point key exchange with the message authentication scheme. However, unlike message authentication, encryption could be performed in a separate layer which could be placed higher on the protocol stack. The primary advantage of placing encryption in a separate layer would come if the encryption layer were placed on top of a layer which guarantees FIFO, reliable delivery. Currently, reliable message delivery is handled by a layer which sits on top of the SCOM layer which retransmits messages which may have been lost. Since encryption is handled in the SCOM layer, the message must be re-encrypted each time it is sent. While under most circumstances, message loss is infrequent, this could be expensive in a high loss network and could be avoided by moving encryption up in the protocol stack.

A second side-effect of the lack of a FIFO, reliable delivery abstraction at the SCOM layer is that stream ciphers can not be used. Stream ciphers are useful in many cases since the computation of the pseudo-random bit stream can be moved off the critical path.

7 Performance

As was described in section 6, the only extra communication performed by an application running over the SCOM layer compared to an application running over the COM layer is the interaction with the trusted server. The interaction with the trusted server involves little work by the application and is performed off the critical path of the computation. As a result, interaction with the trusted server has little impact on the overall performance of the system and the only significant performance costs involved with using the SCOM layer come from the internal computation necessary for key exchange, signing and verifying messages, and encrypting and decrypting messages.

The public key signature and decryption steps used in the point-to-point key exchange protocol are the most expensive operations performed by the system. Each key exchange involves the generation of one public key signature by the sender and one private key decryption by the recipient. The costs of these operations can vary widely depending on the software package (or hardware chip) used. In our system, we use 512-bit RSA [12]. Using the RSAREF(TM) cryptographic toolkit from RSA Laboratories on a SUN-4/LX running SunOS 4.1.3, each signature generation and decryption took 0.75 seconds. However, much faster implementations of RSA are available. CryptoLib [5] on a SPARCstation 10 can perform 512-bit RSA in about 50 milliseconds [9]. There is also cryptographic hardware available which can perform 512-bit RSA in under 1 millisecond [15].

Compared to costs of the public key signature and decryption steps (except when using cryptographic hardware), the costs of the other steps in the key exchange protocol are insignificant. Fortunately, point-to-point key exchange is performed infrequently. In our system, we have set KEY_TIMEOUT to 10 minutes and MSG_TIMEOUT to 30 seconds. This leads each pair of processes to perform key exchange once every 9 minutes (see figure 6). So, if the cost of point-to-point key exchange is amortized across all of the messages sent in the system, it becomes insignificant compared to other costs.

The frequency in which the key exchange protocol for group encryption keys must be executed depends on the behavior of the application since the a group encryption key is changed whenever the key's owner removes a process from its view of the group. In most applications, however, group membership changes are very infrequent. As a result, very few messages will need to have group encryption keys piggybacked onto them. Unlike point-to-point key exchange, group encryption keys are signed and encrypted using secret key schemes. As a result, the key exchange process is inexpensive and should not significantly impact performance even if a group's membership changes frequently.

It is the signing and verifying (and encrypting and decrypting) of each of the messages that constitutes the bulk of the cost overhead. The messages are signed (and encrypted) using secret key signature (and encryption) schemes which can be computed significantly faster than public key schemes, however, since these operations are applied to every message, their costs overwhelm the costs involved with the infrequent use of public key operations. Just as was the case with the public key operations, the actual costs involved in signing and encrypting messages can vary widely depending on the software package (or hardware chip) used to perform the operations.

In figure 8, we show the results of a series of tests used to compare the time needed send a message over various platforms. In each of the tests, two processes bounced a single message back and forth as quickly as possible. The overall run time was divided by twice the number of round trips in order to compute the time needed to send a message. In order to accurately measure the time needed to send an average message, we excluded the initial (key exchange) round trip from the timing for all tests run over Horus.

Each of the tests was conducted using one process on a SUN-4/670 running SunOS 4.1.2 communicating over 10Mbit/s Ethernet with another process on a SUN-4/LX running SunOS 4.1.3. Since each of the Horus tests used UDP for the underlying communication, we used the cost of sending a message over raw UDP as the baseline for comparison.

The second line in figure 8 shows the added cost of using Horus. Most of the extra cost of using Horus with the COM layer is borne by the recipient of the message. Once a message has been received, its header

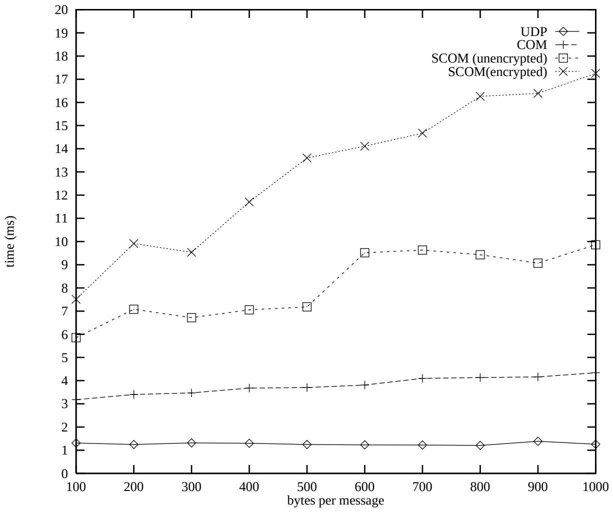

Figure 8: End-to-End Communication Times

must be parsed, the group to which the message is being addressed must be determined, and an upcall thread must be created to allow the application to process the message.

Using the SCOM layer instead of the COM layer adds the overhead needed for signing and verifying messages. Currently, we sign messages using a keyed one-way hash function (MD5 from the RSAREF(TM) toolkit). For large messages, the time needed to compute the one-way hash of the message is responsible for the most of the added cost compared to the COM layer. However, since processes running over the SCOM layer may communicate with processes which they do not trust, the SCOM layer must also spend time to perform sanity checks on the headers of all incoming messages.

The SCOM layer also, optionally, encrypts messages. In our tests, we used the DESX enhancement of DES from RSAREF(TM) to perform encryption. Using this package in our system, messages could be encrypted and decrypted at a rate of about 300 Kbytes/s.

8 Future Work

As was mentioned earlier, the design and implementation of the SCOM layer represents only the beginning of our effort to add support for secure applications to Horus. The main focus of our future work will be the design of higher level layers which provide more sophisticated semantics to applications. At some point, however, we hope to look into designing a set of anchor layers which can be used as alternatives to the SCOM layer.

The next layers that we build will be secure alternatives to the NAK and FRAG layers. The secure versions of these layers should be relatively simple to design. The main difference between the secure versions of these layers and the current versions will be the need run sanity checks on the incoming messages.

Once these layers have been implemented, we will begin to build the infrastructure necessary to create a secure version of the MBRSHIP layer. The secure version of the MBRSHIP layer will be designed to operate in an environment in which group members can fail arbitrarily. This will involve a redesign of the protocols for failure detection, view change coordination, distribution of message stability information, and message flushing [7]. Unlike the protocols for the lower level layers, the design of these layers will involve making assumptions about the trust model used by the application. In particular, each of these protocols will need to assume that more than two-thirds of the processes in any group view are correct. Fortunately, as a result of the design of the Horus system, this restriction on the trust model will only apply to applications which actually use the membership layer.

While one of the goals of the design of the SCOM layer was to make it an all-purpose replacement for the COM layer for applications which require security, we were forced to make several decisions in the design of the layer which may be inappropriate for some applications. First, in order to make message authentication as fast as possible, we chose to use a secret key signature algorithm to sign all messages. One side-effect of this decision is that the SCOM layer provides no support for non-repudiation of messages. While non-repudiation is not always necessary, it can be useful in applications which involve the forwarding of messages (such as the authentication certificates distributed by the trusted server). The problem of message repudiation could be solved either by creating an alternative SCOM layer or by creating a non-repudiation layer which sits on top of the SCOM layer.

The SCOM layer also has built-in assumptions about the source of time and authentication information. The SCOM layer assumes that the authentication service will provide forwardable certificates which are timestamped and which guarantee the uniqueness of the entity identifiers used by the processes. In the future, we plan to create a modified version of the SCOM layer which makes use of Fortezza cards instead of a software encryption library. Since the Fortezza cards generate authentication certificates of their own

[7] The design of our secure membership layer will be based on the work done by Reiter in [9] for the Rampart project.

which will not include entity identifier information, we will need to change the SCOM protocols in order to accommodate these certificates.

One consequence of the decision to minimize the restrictions that the SCOM layer places on the trust model was the need to perform a relatively large number of key exchange operations. One problem with this is that key exchange involves the use of public key encryption and signature schemes. As was described in section 7, public key signature and decryption operations are very expensive, especially in systems which do not have hardware encryption chips. For applications which make use of certain restrictive trust models, such as the firewall model (see figure 2), different key exchange protocols could be used which would be less costly. While implementing anchor layers which use these more restrictive models would not increase the functionality of Horus, it would make use of Horus more efficient for the applications which use these trust models.

9 Conclusion

Different applications have different requirements with respect to support for distributed computing and security. The Horus system attempts to provide as many applications as possible with the amount and type of support they need. This is accomplished through the use of a set of protocol layers which can be stacked together by an applications programmer in order to create the environment which is most appropriate for any given application.

Recently, we began the process of supplying support for applications which will run in insecure environments. Just as most of the layers currently available in the Horus system have been designed to handle crash failures, the new layers we will add will be tolerant to malicious failures. In this paper, we have described the details of our initial security layer. As was shown in section 7, the use of this layer will not significantly affect the overall performance of a system unless it is message passing intensive. Even for these types of systems, the impact of using the SCOM layer can be minimized through the use of fast software or hardware cryptography.

Acknowledgements

We would like to thank Robbert van Renesse, Mike Reiter, and others who have previously worked on the Horus project as well as Uwe Wilhelm who provided valuable comments during the early stages of this work.

References

[1] F. Cristian. Probabilistic clock synchronization. *Distributed Computing*, 3(3):146–158, 1989.

[2] Danny Dolev and Dalia Malki. The Transis approach to high availability cluster communication. *Communications of the ACM*, 39(4):64–70, April 1996.

[3] Li Gong. A security risk of depending on synchronized clocks. *Operating Systems Review*, 26(1):49–53, January 1992.

[4] J. Gosling. Java intermediate bytecodes. *SIGPLAN Notices*, 30(3):111–118, March 1995.

[5] J.B. Lacy, D.P. Mitchell, and W.M. Schnell. CryptoLib: Cryptography in software. In *Proceedings of the Fourth USENIX Security Workshop*, pages 1–17, October 1993.

[6] Mike Lewis and Andrew Grimshaw. The core Legion object model. Technical Report CS-95-35, Department of Computer Science, University of Virginia, August 1995.

[7] L.E. Moser, P.M. Melliar-Smith, D.A. Agarwal, R.K. Budhia, and C.A. Lingley-Papadopoulos. Totem: A fault-tolerant multicast group communications system. *Communications of the ACM*, 39(4):54–63, April 1996.

[8] Michael K. Reiter. Secure agreement protocols: Reliable and atomic group multicast in Rampart. In *Proceedings of the 2nd ACM Conference on Computer and Communications Security*, November 1994.

[9] Michael K. Reiter. A secure group membership protocol. In *Proceedings of the 1994 IEEE Symposium on Research in Security and Privacy*, pages 176–189, May 1994.

[10] Michael K. Reiter. Distributing trust with the Rampart toolkit. *Communications of the ACM*, 39(4):71–74, April 1996.

[11] Michael K. Reiter, Kenneth P. Birman, and Robbert van Renesse. A security architecture for fault-tolerant systems. *ACM Transactions on Computer Systems*, 12(4):340–371, November 1994.

[12] R.L. Rivest, A. Shamir, and L. Adleman. A method for obtaining digital signatures and public-key cryptosystems. *Communications of the ACM*, 21(2):120–126, February 1978.

[13] F.B. Schneider. Implementing fault-tolerant services using the state machine approach: A tutorial. *ACM Computing Surveys*, 22(4):299–319, December 1990.

[14] Mike Seager, David Guaspari, Matt Stillerman, and Carla Marceau. Formal methods in the Theta kernel. In *Proceedings of the 1995 IEEE Symposium on Security and Privacy*, pages 89–100, May 1995.

[15] M. Shand and J. Vuillemin. Fast implementations of RSA cryptography. In *1993 IEEE 11th Symposium on Computer Architecture*, pages 252–259, 1993.

[16] Robbert van Renesse, Kenneth P. Birman, and Silvano Maffeis. Horus: A flexible group communications system. *Communications of the ACM*, 39(4):76–83, April 1996.

[17] Wm. A. Wulf, Chenxi Wang, and Darrell Kienzle. A new model of security for distributed systems. Technical Report CS-95-34, Department of Computer Science, University of Virginia, August 1995.