Managing Security in High-Performance Distributed Computations

Ian Foster[*], Nicholas T. Karonis[†], Carl Kesselman[‡], Steven Tuecke[*]
http://www.globus.org/

Abstract

We describe a software infrastructure designed to support the development of applications that use high-speed networks to connect geographically distributed supercomputers, databases, and scientific instruments. Such applications may need to operate over open networks and access valuable resources, and hence can require mechanisms for ensuring integrity and confidentiality of communications and for authenticating both users and resources. Yet security solutions developed for traditional client-server applications do not provide direct support for the distinctive program structures, programming tools, and performance requirements encountered in these applications. To address these requirements, we are developing a security-enhanced version of a communication library called Nexus, which is then used to provide secure versions of various parallel libraries and languages, including the popular Message Passing Interface. These tools support the wide range of process creation mechanisms and communication structures used in high-performance computing. They also provide a fine degree of control over what, where, and when security mechanisms are applied. In particular, a single application can mix secure and nonsecure communication, allowing the programmer to make fine-grained security/performance tradeoffs. We present performance results that enable us to quantify the performance of our infrastructure.

1 Introduction

The use of high-performance networks to couple geographically distributed supercomputers, database systems, specialized scientific instruments, etc., is enabling novel applications in areas such as collaborative engineering, computer-enhanced instrumentation, and ultra-large-scale scientific simulation [2, 3]. However, widespread use of such applications depends crucially on the availability of appropriate security mechanisms. Owners of resources require authentication mechanisms to protect themselves against malicious users. Users of resources may also demand authentication of resources, in order to protect themselves against spoofing by malicious resource providers. Users will often need to ensure that the integrity and confidentiality of

[*]Mathematics and Computer Science Division, Argonne National Laboratory, Argonne, IL 60439, U.S.A.
[†]Department of Computer Science, Northern Illinois University, DeKalb, IL 60115, U.S.A.
[‡]Information Sciences Institute, University of Southern California, CA, U.S.A.

data communicated between resources are not compromised, particularly when communication occurs over public networks. Other forms of attack can also be of concern, such as denial of service attacks against applications that use supercomputers to control remote devices.

The task of meeting these security requirements is complicated by the distinctive program structures, computing environments, and performance requirements encountered in high-performance systems. Traditional distributed systems often have a client-server structure, with limited mutual trust between client and server. In contrast, parallel programs may comprise hundreds or thousands of tightly coupled, fully trusting processes. Distributed systems employ remote procedure call (RPC) or TCP/IP as their primary communication mechanism. In contrast, the applications that we consider here may communicate by using two-sided message passing, streaming protocols, multicast, and/or single-sided get/put operations, as well as RPC; furthermore, they are typically programmed by using message-passing libraries such as the standard Message Passing Interface (MPI [13]) or with specialized parallel languages (e.g., HPF [17] or HPC++). Programs must run on parallel computers, which typically provide specialized mechanisms for process creation, communication, and so forth, and which may even run specialized operating systems. At the same time, programs often must achieve a substantial fraction of peak computer and network performance.

Historically, we find that security technologies are used only if they are incorporated into common tools in a seamless and painless fashion. In the case of high-performance computing, this suggests a need for secure versions of parallel programming tools such as MPI. These security enhanced tools must support the diverse process creation and authentication mechanisms encountered in high-performance systems, and must address scalability issues that arise when dealing with hundreds or thousands of processes. In addition, the demanding performance requirements of high-performance applications introduces a need for mechanisms that provide programmers with fine-grain control over what, when, and where security mechanisms are used in programs.

We are developing a secure communications infrastructure that addresses these various concerns. This infrastructure builds on existing components and standards whenever possible (e.g., SSL [14], Kerberos [28], GSS-API [19]), while also extending the state of the art to provide four new capabilities:

- A secure process creation interface that supports the wide range of process creation mechanisms encountered in high-performance computing systems, and that addresses scalability issues that arise in programs that may need to create hundreds or thousands of processes.
- Techniques for managing the use of multiple security mechanisms within a single application, in a way that provides a uniform high-level programming model while allowing the choice of low-level security mechanism to vary according to what is communicated, where it is communicated, and when it is communicated.
- Techniques for managing the transfer of secure logical communication links among processes in large-scale distributed computations.
- Security-enhanced implementations of multiple parallel libraries (MPI, CAVEcomm, etc.) and languages (HPF, HPC++, etc.), that enable programmers to use our secure process creation and communication mechanisms while using familiar tools.

These new capabilities have been implemented and evaluated in the context of Nexus [12], a low-level multithreaded communication library designed to support high-performance communication in heterogeneous environments. (The security-enhanced libraries and languages referred to above are all layered on top of Nexus.) While these capabilities are not in themselves a complete solution to the problem of providing security in high-performance distributed applications, we do believe that they represent useful steps towards that goal.

The rest of this paper, an expanded and revised version of [10], is as follows. In Section 2, we introduce the problems that we seek to address in our work. In Section 3, we provide an overview of our approach and review the Nexus communication infrastructure. In Sections 5 and 6, we describe our secure communications infrastructure, and in Section 8, we present some experiments that allow us to evaluate its effectiveness. Finally, in Sections 9 and 10 we discuss related work and present our conclusions, respectively.

2 Requirements

We are interested in applications that integrate geographically distributed computing, network, information, and other systems to form "virtual" networked computational resources. For example, global climate scientists often employ large coupled simulation models, constructed by linking models of atmospheric and ocean behaviors. Such coupled models may use multiple supercomputers to exploit large aggregate memory or to run different components more quickly on different architectures [21, 25]. High-end collaborative engineering environments connect supercomputers, databases, and advanced display devices to provide remote access to shared state, which may include simulated entities as well as people [4, 5]. "Smart instruments" connect scientific instruments or other data sources to remote computing capabilities [18]. In each case, computations span heterogeneous collections of resources, often located in multiple administrative domains. They may involve hundreds or even thousands of processes. Communication costs are frequently critical to achieved performance, and programs often use complex computation/communication structures to reduce these costs.

The development of a comprehensive solution to the problem of ensuring "security" in such applications is clearly a complex and multi-faceted problem. In this article, we focus our attention on two significant subproblems, namely the authentication of users and resources when creating computational entities ("processes") on local and/or remote computer systems (the *process creation* problem), and the assurance of integrity and confidentiality when exchanging data between these processes (the *communication* problem).

2.1 Process Creation

We use the term process creation to refer to the mechanism by which computational resources are integrated into computations. These resources may all be acquired before the computation starts (i.e., static allocation) or may be acquired and released during the course of the computation (dynamic allocation). Computational resources of interest include both single-processor and multiprocessor systems, and the low-level mechanisms used to initiate computation may be quite different in each case. For example, on a workstation we might use secure or unsecure "remote shell" (`rsh`) mechanisms or hand-crafted process creation servers; in contrast, parallel computers typically provide specialized mechanisms that start a user-supplied executable on multiple processors and may require interfacing with local resource management systems such as a partition manager or scheduler [15]. (A program may also need to attach to other running processes during its execution; we view this as a generalization of the usual startup problem, and do not discuss it explicitly here.)

A secure process creation facility for high-performance programs must support a heterogeneous mix of process creation mechanisms. It should support authentication of the user of remote resources and/or of the resources themselves. The process creation mechanism must provide for the establishment of *security contexts* that hold security state and configuration information needed for subsequent secure communication within the program. Because a computation may comprise hundreds or thousands of processes, which typically are mutually trusting once created, it is both impractical and unnecessary to perform a formal authentication process between every pair of processes. Instead, we need scalable mechanisms for process creation that allow a process to transfer to its offspring the right to communicate with other processes in a computation.

2.2 Communication

Once processes have been created, they need to be able to exchange data and synchronize their execution. As noted above, the applications in which we are interested communicate by using a variety of interaction mechanisms. Communication performance is often critical, but as messages are often small, latency can be as important as bandwidth. Collective communication operations across multiple processes can exacerbate the impact of latency on performance. Furthermore, performance and functionality requirements frequently motivate the use of multiple low-level communication methods within a single application. For example, coupled models often need to use machine-specific communication methods within computers and optimized wide area protocols between computers [21, 25]. Collaborative environments require a mixture of protocols providing different combinations of high throughput, multicast, and high reliability [4, 5]. Smart instrument applications may need to be able to switch among alternative communication substrates in the event of error or high load [18]. In general, the method used for a communication can vary according to where communication is being performed, what is being communicated, or when communication is performed [8].

These considerations place demanding requirements on a secure communications infrastructure. It is clearly critical to be able to specify the security mechanism used for a particular communication independently of the low-level method used to achieve that communication. More challenging perhaps is that programmers must be able to write programs that mix secure and unsecure communication. For example, let us consider a coupled climate model as a prototypical scientific simulation for which security mechanisms may be required. (While this example may appear contrived, the controversy that surrounds global change studies suggests that security could well be a concern, if computing in an open environment.) Assume that the model runs the ocean and atmosphere model components on two separate IBM SP2 parallel computers, connected by an open high-speed network. The programmer writes the coupled model so that all communication is expressed using MPI; the MPI implementation selects communication methods for each message, according to message destination [9]. Communication between two nodes in the same SP2 takes place over a dedicated, high-speed switch using IBM-specific protocols, and as this environment is tightly controlled, we might reasonably de-

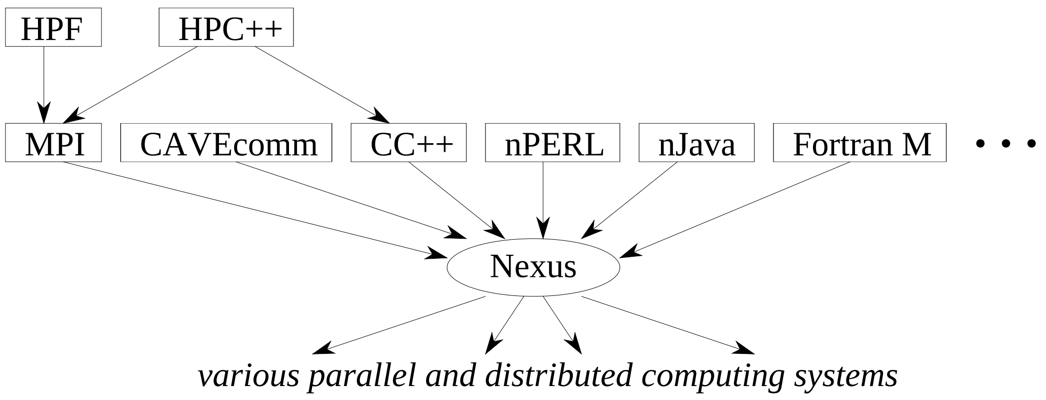

Figure 1: The Nexus communication infrastructure

cide that security measures such as encryption are not required. In contrast, communication between two nodes in different SP2s occurs over a general purpose computer network using TCP/IP, and may well require security measures. In Section 8, we present performance results that demonstrate the advantages of applying security mechanisms only between models.

In this example, it is sufficient to select security mechanisms according to where communication is directed: that is, according to the underlying physical communication structure. In other situations, we believe that it is important that programmers be able to vary security mechanisms according to the logical communication structure of a program. For example, we may want to use different security mechanisms for communications representing "control" and "data."

3 Our Approach

We seek to address the requirements outlined in the preceding section by constructing a secure communications infrastructure based on a portable communications library called Nexus [12]. We chose to work with Nexus for two reasons. First, it supports many of the tools that are commonly used for application development in parallel and distributed systems, such as the Message Passing Interface (MPI) [13], High Performance Fortran (HPF) [17], and CAVEcomm [5] (a specialized library for collaborative environment applications). Second, its architecture has been designed to support the coexistence and concurrent use of different process creation and communication methods [8]. The latter feature simplifies the integration and management of different security methods.

Figure 1 shows some of the parallel tools that have been constructed with Nexus mechanisms. Each of these libraries or languages use Nexus facilities to create processes and to exchange data between processes; Nexus handles automatically the various low-level issues relating to the process creation and communication methods to be used in different situations.

3.1 Nexus Structure

The Nexus communication library is structured in terms of five basic abstractions: nodes, contexts, threads, communication links, and remote service requests. A computation executes on a set of *nodes* and consists of a set of *threads*, each executing in an address space called a *context*. (For the purposes of this paper, it suffices to assume that a context is equivalent to a process.) An individual thread executes a sequential program, which may read and write data shared with other threads executing in the same context. Inter-context references called *communication links* provide a global name space for objects, while the *remote service request* (RSR) is used to initiate communication and invoke remote computation. Nexus support for threads is relevant to this paper to the extent that threads can be an important latency hiding device, and multithreading can have implications for how we maintain and use security information.

In the following, we expand upon two aspects of the Nexus system: communication links and management of multiple communication methods.

3.2 Communication Links

As illustrated in Figure 2, communication links connect data structures called startpoints and endpoints. (Prior papers on Nexus [12] referred to communication links as global pointers; we adopt the alternative terminology to emphasize that we are not assuming a global address space.) A communication link is formed by binding a startpoint to an endpoint. Many startpoints can be bound to a single endpoint and there can be many startpoints and endpoints within a process.

Nexus supports a single communication operation: the remote service request, or RSR. An RSR is directed from a startpoint to an endpoint, causing the transfer of data from the startpoint process to the endpoint process and the remote execution of a function specified to be an endpoint *handler*. An advantage of the startpoint construct in a distributed computing environment is that the startpoint can be used to encapsulate not only information about *where* a remote object is located, but also *how* to communicate with that remote object. This feature has been exploited to manage the use of multiple communication methods [8].

The endpoint construct allows us to associate local state with the remote location referenced by a startpoint. This state can be used to maintain security information, and hence is valuable when implementing stream-oriented communication routines, such as encryption based on stream ciphers. As illustrated in Figure 2, multiple versions of this local state can be maintained, one for each startpoint in the case where multiple startpoints are associated with a single endpoint.

A startpoint/endpoint pair represents a simplex communication channel: that is, it specifies a remote destination to which a communication operation can be directed by an RSR. These channels can be created dynamically; once created, a startpoint (but not an endpoint) can be communicated between nodes by including it in an RSR message buffer. Hence, a startpoint can be thought of as a capability granting rights to operate on the associated endpoint. The RSR mechanism allows point-to-point communication, remote memory access, streaming protocols, and multicast to be supported within a single framework.

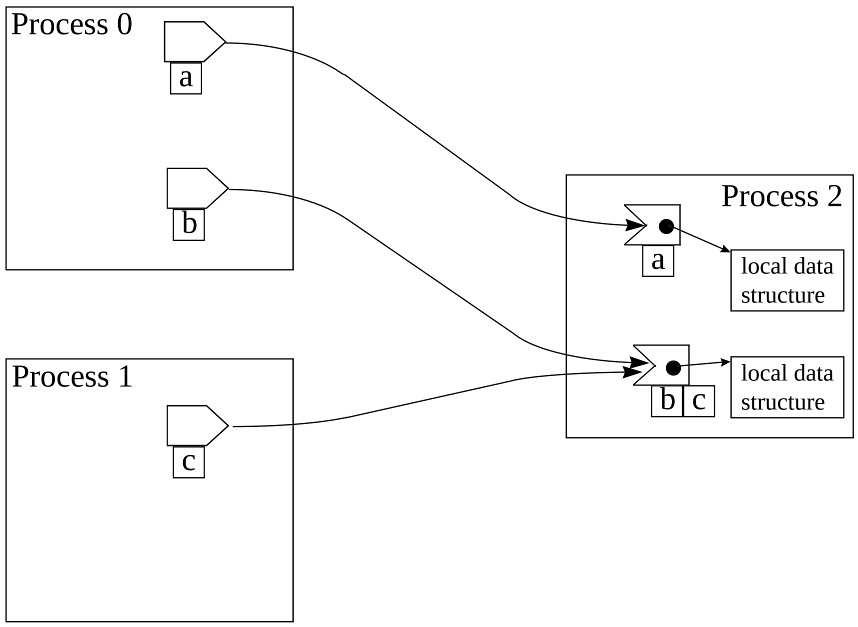

Figure 2: The Nexus secure communications infrastructure. The figure shows three startpoints (in the two processes on the left) referencing two endpoints (on the right). The boxes labeled “a,” “b,” and “c” are security contexts; these are discussed below.

3.3 Communication Method Selection

As noted earlier, high-performance applications can require the use of different communication mechanisms in different situations. Nexus incorporates automatic configuration mechanisms that allow it to use configuration information provided by a Metacomputing Directory Service (MDS) [7] to determine which startup mechanisms, network interfaces, and communication methods to use in different situations [8]. These mechanisms allow Nexus programs to execute unchanged in different environments, with communication methods selected according to default rules, depending on the source and destination of the message being sent. For example, automatic selection allows Nexus RSRs to use IBM's Message Passing Library (MPL) within an IBM SP2 and TCP/IP between computers. Manual selection is also supported, for example allowing selection of specialized ATM protocols when appropriate. In each case, selection mechanisms are employed whenever a startpoint is received from another process, and hence apply both during initial process creation and subsequently as additional communication links are established.

3.4 Per-link Transformations

Nexus provides a general mechanism for performing per-link mapping on the data carried by a RSR. For a given communication link, a set of *transform* functions, called a *transform module*, can be associated with the link startpoint and endpoint. Each transform module provide functions that are called when a startpoint or endpoint is initialized, when a startpoint is copied, and when data is sent from a startpoint or received at an endpoint.

The startpoint transform function is called after an RSR is assembled, but prior to placing the RSR onto the network. It can alter the contents of a message, changing the values being communicated or adding additional data to a message or its header. The startpoint transfer function can access state information stored in the startpoint. Conversely, the endpoint transform (or untransform function) is called after the RSR is received, but prior to invoking the remote function. It too can modify head contents, message value, or size. Transform state can be associated with the endpoint as well. The key to transform modules is that the existence of a transform module does not effect the way in which startpoints and endpoints are used. After the user specifies a particular security configuration, the existence of the transform is transparent to the user.

The data-security methods discussed in this paper are implemented by Nexus transform modules. However, the transformation mechanism is very general: for example, it has also been used to implement per-link compression and application-level framing [27].

4 Security Contexts

As we describe in the next two sections, we extend Nexus in two main ways to develop our secure communications infrastructure. First, we define a secure process creation interface, which we integrate with Nexus process creation mechanisms. Second, we extend Nexus communication mechanisms to use and manage security information. Both extensions make extensive use of a *security context* similar to that used in GSS-API [19].

The Nexus security context is a data structure used to encapsulate security information. Security contexts are associated with communication links and always exist in pairs: one context is stored in a startpoint and the other in the associated endpoint. Each security context is composed of two parts: the security configuration and the security state. The *security configuration* describes what type of security should be used for the communication link as well as the manner in which those security measures should be applied. For example, we might configure one security context so that each message is encrypted using DES/ECB, while another security context is configured to encrypt with RC4 and also to perform authentication. This design not only allows us to configure security characteristics on a per-link basis, but also provides a framework in which we may exploit different implementations of security algorithms within a single application, e.g., exploiting high-performance encryption hardware [29] that only exists on some machines.

The *security state* houses the values needed to enforce the security specified by the configuration, such as keys and initialization vectors. Some encryption algorithms change the values of these keys and/or initialization vectors as a function of the plaintext they encrypt and ciphertext they decrypt. Hence, security state can change over time.

5 Process Creation

Parallel programs use process creation mechanisms to initiate computation on other computers. In Nexus, process creation involves a call to a "create process" function, which invokes machine-specific mechanism to create the new process and instantiates a startpoint referencing an endpoint in the newly created context. Subsequent communication with that context occurs over the new communication link. The same interface is used to create multiple contexts (for example, when initiating computation on a parallel computer), except that the call returns a vector of startpoints, one per new process.

Typically, process creation involves interaction with some remote service, whether this be an `rsh` daemon, a scheduler on a supercomputer, or some other specialized server. Authentication of the requester and/or the remote server may be required, and an initial security context must be established for subsequent communication between requester and newly created process. As noted previously, we need to deal with a wide variety of process creation and authentication mechanisms, and must address scalability issues that arise when creating large number of processes.

5.1 Interface

We address the need to deal with a wide variety of process creation and authentication mechanisms by defining a standard interface. Two of the functions in this interface are `start_process` and `init_process` (Figure 3). Process creation is initiated by a call to `start_process`. Using the Metacomputing Directory Service, `start_process` can determine the authentication protocols that are acceptable to the specified host. Based on this information, the initiating process selects the appropriate authentication service and contacts this service to initiate process creation. Depending on the value of the supplied authentication flag and the requirements of the host being contacted, authentication may be required just for the client, or for both the client and the server. An initial security context can be provided to the

```
int start_process(char *hostname,
                  char *directory,
                  char *executable,
                  char **argv,
                  char **environment,
                  int authenticate_flag,
                  security_context_t *sec_context)

int init_process(int *argc,
                 char ***argv,
                 security_context_t *sec_context)
```

Figure 3: Functions used to add a new process to a Nexus computation.

`start_process` call; this is encoded as a byte array, passed over a secure channel, and made available to the newly created process by placing it in an environment variable.

Successful authentication results in the creation of a new process on the specified host, with directory, executable, arguments, and environment as specified in the `start_process` call. The newly created process must call the `init_process` function before performing other computation. The call returns the process arguments (`argc`, `argv`) and populates a user-supplied security context with the one provided by the process that called `start_process`.

The two functions just described allow us to create a set of processes and an initial set of shared security contexts. The Nexus implementation then completes the negotiation process by using these shared security contexts to establish an initial communication link (and associated security context) from the requesting process to the created process. Note that subsequent communication with the newly created process can occur with any communication mechanism supported by Nexus (TCP, vendor-specific libraries, etc.). The interface also includes a split-phase version of the `start_process` function, so that multiple process creation requests can proceed concurrently.

Once established, these initial security contexts can be used in a variety of ways. In Nexus computations, we use them to create communication links, implemented (as described above) as startpoint/endpoint pairs. We may also want to create specialized communication structures designed to allow the rapid execution of secure versions of various collective operations, such as reduction, broadcast, or multicast. These mechanisms can, in many situations, be implemented with specialized communication structures (e.g., spanning tree) or low-level protocols or hardware. Care must be taken that security mechanisms do not prevent the use of the specialized communication mechanisms.

5.2 Implementation Examples

Implementations of the process creation interface require mechanisms for authenticating the user and/or the process creation servers, and for establishing a secure channel for the exchange of the initial security context. We have developed a variety of such implementations. As an example, we consider a Secure Socket Library (SSL)-based process creation server. This acts as an SSL server, while the process calling `start_process` (the creating process) acts as an SSL client. The client connects to the server using normal SSL mechanisms, thus performing authentication and establishing a secure channel between the client and server. The client then uses this channel to pass the various process creation arguments to the server, which creates the new process. When the new process calls `init_process`, it configures itself using the passed command line arguments, and initializes its security context argument using the information passed to in by the server in environment variables. This negotiation process completes with a communication link (and associated security context) being created from the requesting process to the created process.

As a second example, we consider what happens when we need to create many processes at a remote location. One approach would be to make multiple `start_process` requests to the appropriate remote server. However, this approach has significant scalability problems. Hence, we instead use a single request to ask that multiple requests be created. The process creation server then creates the processes independently, accumulating the startpoints as they become available; when it is done, it returns the vector of startpoints to the requesting process. Note that no additional authentication is required when transferring the startpoints (and associated security contexts) from the "proxy" node to the requesting process, because we assume that processes in a parallel program are mutually trusting. These mechanisms allow a program to create large numbers of processes quickly, by using a hierarchical process structure.

We are currently engaged in recasting these and other implementations in terms of the functions provided by GSS-API [19], with the goal of simplifying code, supporting a wider range of security mechanisms and promoting reuse.

6 Communication

As described above, Nexus allows security contexts to be associated with communication links. This structure gives the tool developer (or application programmer) a fine degree of control over how security mechanisms are applied during communication. Different contexts can be associated with different links; in particular, some links may not have any security context at all. Critical to the success of this strategy is that links that do not require security do not have to pay a performance penalty.

Figure 2 shows how startpoints and endpoints are extended with security contexts. In this figure, the boxes labeled "a," "b," and "c" represent security contexts. Notice that the lower endpoint (on the right) has two security contexts associated with it, one for each associated startpoint. This ability to associate multiple security contexts with an endpoint is important for several reasons. First, different startpoints might communicate by using different security mechanisms; second, even if they use the same security mechanism, multiple security contexts are required when using encryption mechanisms (e.g., DES stream ciphers) that update the security state as a function of the previously encrypted plaintext.

Nexus mechanisms that manipulate startpoints and endpoints are extended to deal with security contexts. Whenever a startpoint is copied or sent to another process as part of an RSR (hence establishing a new communication link), a new pair of security contexts is created. Depending on the type of security context being created, the copy operation may require communication with the endpoint, requiring a round trip communication.

The application of security mechanisms when initiating or receiving an RSR is triggered by an "escape" tag associated with a Nexus startpoint and endpoint. If this escape tag is set, a specified security transformation is applied to communicated data. At the endpoint, we must identify the correct security context for the incoming communication. To facilitate this, we must place a context identifier in the message header. Exchanging the context identifier is one reason why copying a security context may require communication with the endpoint.

The mechanisms just described have the desirable property of introducing little unnecessary overhead, particularly in the case when they are not used. When they are used, costs associated with this mechanism (relative to a communication method that always performs encryption, for example) are a test on the "escape" flag followed by a lookup of a small table to see what transformation should be applied. If a startpoint is replicated, a small security context index must be included in each RSR. Space overhead comprises the encoding of the security context. When not in use, the only time overhead is the test on the escape flag; there is no space overhead. See Section 8 for additional discussion of performance.

Nexus constructs a remote service request by a series of "put" calls (used to designate the data to be transferred) followed by a "send" (which completes the transfer). Our current security-enhanced Nexus copies data into a contiguous buffer, to which a single encryption call is applied. An alternative approach is to incorporate encryption operations in the "put" calls, hence reducing the number of times that data is copied. We have experimented with both approaches, and find that for DES/ECB the latter approach is typically 5–7 percent faster. The difference would be larger for lower-cost encryption techniques.

6.1 Logical Connections

Because security mechanisms are integrated into Nexus at a low level, they need not be visible to the programmer. That is, it is straightforward to configure a Nexus application (and hence an application code using any of the various libraries or languages layered on Nexus) so that *all* communications are secured using the same standard mechanism. Furthermore, this security need not interfere with the various communication optimizations incorporated in Nexus. For example, in a heterogeneous environment, Nexus can, as usual, use TCP/IP between parallel computers and vendor-supplied communication libraries or shared memory within parallel computers.

Nevertheless, the full power of our architecture becomes apparent when the programmer (or tool developer) wants to implement more sophisticated communication structures. Because security contexts are associated with startpoints and endpoints, rather than processes, we can maintain multiple logical connections between a pair of processes, and associate different security mechanisms with different connections. This capability allows the programmer to apply security mechanisms selectively, depending on what is being communicated, where it is being communicated, and even on when communication is performed. For example, we may protect the integrity of control messages at all times, but encrypt data messages only when these are passed over open networks; or we can use specialized encryption techniques for particular types of data [20, 1]. Note that because security context information is associated with communication links, not communication calls, the code that actually performs communication does not need to be aware of whether security mechanisms are being applied.

The ability to associate security contexts with logical connections is particularly useful in multithreaded environments, where communications over different logical connections can be interleaved at the physical level. The Nexus architecture avoids the need for an additional layer of multiplexing/demultiplexing, as would be required, for example, if all communications between two processes had to occur within a single stream cipher-based security context.

A number of approaches can be taken to specifying the security contexts that are to be used for specific communications. As noted above, Nexus mechanisms provide a degree of automatic management. Once a startpoint/endpoint pair has been created, the startpoint can be communicated to other processes, and any process receiving the startpoint can then communicate securely with the original process, by using the startpoint and its associated security mechanism. For more fine-grain control, Nexus provides functions for setting the security attributes of a startpoint and endpoint. Libraries layered on top of Nexus can use other, higher-level mechanisms. For example, an MPI implementation can associate security attributes with a communication structuring mechanism called communicators [13].

7 Implementation of Communication Security

Communication security in Nexus can be completely implemented by the Nexus transformation mechanism introduced in Section 3.4. Obviously, encryption can be viewed as a mapping of the contents of a Nexus message. In the case of stream ciphers, this mapping is state dependent. However, the link-state facilities provided by the Nexus transform modules address this requirement. If only data integrity is required, the transform can compute a message authentication code (MAC) and insert it into the message header, without changing the data at all. Untransforming the data on the receiving side involves calculating a new MAC based on the received data and comparing it to the MAC in the message header.

A user can specify their security requirements by creating a transform module that implements the needed security mechanisms and then associating the desired module with startpoints and endpoints when they are created. In general, Nexus allows a user to control aspects of startpoint and endpoint behavior by providing a startpoint or endpoint attribute data structure when the startpoint of endpoint is created. A transform module is one of the characteristics that is controlled by the endpoint attribute data structure. In the following example, we set the transform module for two endpoint attributes, `epa1` and `epa2`. In the first case, we use a set of transform functions that perform Electronic Codebook (ECB) DES encryption only, in the second, the transform module implements Cipher Block Chaining (CBC) DES encryption with MD5 authentication.

```
nexus_endpoint_attr_t epa1, epa2;

nexus_endpointattr_set_transform(&epa1,
                                 NEXUS_TRANSFORM_ECB,
                                 (void *) NULL);

nexus_endpointattr_set_transform(&epa2,
                                 NEXUS_TRANSFORM_CBC_AUTH,
                                 (void *) NULL);
```

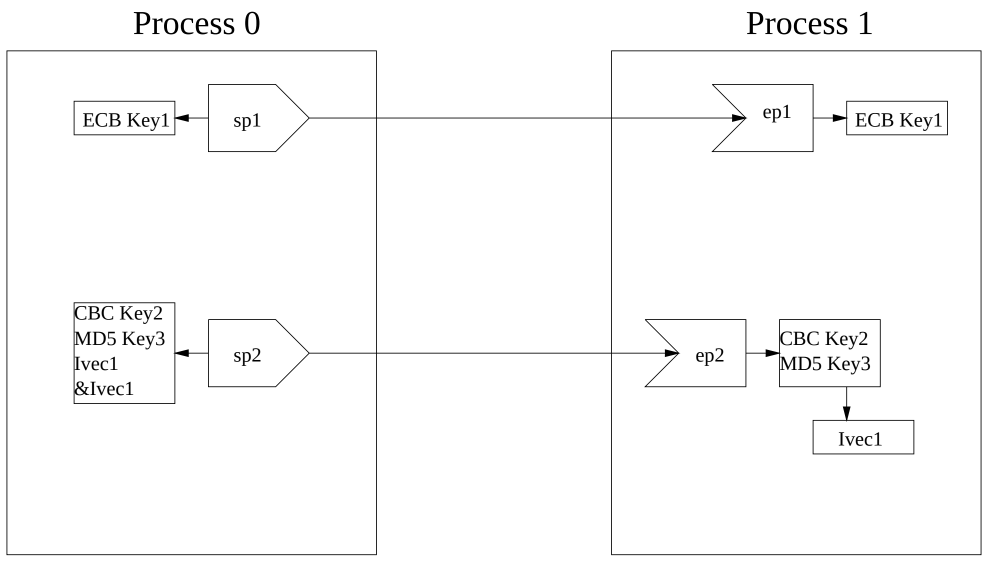

Figure 4: Two pairs of startpoints and endpoints, each with their security contexts.

Once created, an endpoint attribute can be used to initialize any number of endpoints, each inheriting the endpoint attribute's security specification. Each endpoint is used, in turn, to bind a number of startpoints to it. Each startpoint inherits the endpoint's transform functions and consequently its security specification. Thus in the following, we create two endpoints. The first uses ECB DES encryption for any RSRs sent to it, while the second will use CBC with MD5 authentication for all incoming communication.

```
nexus_endpoint_t ep1, ep2;

nexus_endpoint_init(&ep1, &epa1);
nexus_endpoint_init(&ep2, &epa2);
```

The endpoint initialization function in the transform module is used to initialize the endpoint security context where the security context is stored in the endpoint as the transform module state. The structure of each security context is different for each transform module and its contents are different for each endpoint. Some security contexts remain constant over time (e.g., ECB encryption) while others change (e.g., CBC encryption).

The endpoint initialization function for ECB encryption acquires a key dynamically and stores its value as the transform module endpoint state. For CBC encryption with MD5 authentication, endpoint initialization is more complicated as the context consists of separate keys for CBC and MD5 along with a initialization vector (ivec) that is used to implement the CBC encryption algorithm. Because CBC is state dependent, we will need a unique ivec for each communication link terminating at an endpoint. Consequently, the endpoint security context for CBC and MD5 must actually contain two keys and a set of ivecs. The structure of the two types of security contexts is shown in Figure 4.

When a startpoint is bound to an endpoint, the startpoint inherits the transform modules, and hence the security specification, of the endpoint. The startpoint initialization function in the transform module is called, and this function is used to create and initialize the sender's security context, which is stored as the startpoint transform module state. For ECB, the startpoint security context is identical to the endpoint context: the endpoint key is copied into the startpoint context. For CBC with authentication, the CBC and the MD5 keys are also copied into the startpoint security context. However, a new ivec must also be created for the startpoint and this ivec added to the endpoint security context. A copy of the ivec is also placed in the startpoint security context. Finally, in order to identify which ivec to use to decode an incoming message, an ivec identifier is placed in the startpoint security context as well. Thus, with CBC encryption, not only does the startpoint's security context differ from its endpoint's, but the process of binding the startpoint to the endpoint modifies the endpoint's security context also.

7.1 Copying Startpoints

The startpoint copying function in the security transform modules work much in the same way as the startpoint initialization functions. In the case of ECB, the context for the new startpoint can be copied from initial startpoint. The endpoint's security context remains unchanged. For CBC encryption the process is more complicated as the ivec in the endpoint context is modified as a function of the data being encrypted. In order for CBC encryption to work, not only do both sides (encrypting and decrypting) require the exact same key, but they also must both must start with identical ivecs. The ivec values are kept in sync only by encrypting and decrypting the same data stream. Consequently, each communication link (startpoint/endpoint pair) must have a distinct ivec, and copying a startpoint must result in the creation of a new ivec entry in the endpoint's security context. For this reason, the implementation of the `nexus_startpoint_copy` operation will arrange for the startpoint copy function to have access to the potentially remote endpoint to which the source startpoint is bound, allowing the creation of an additional endpoint ivec structure for the new startpoint.

7.2 Sending and Receiving Secure Messages

When an RSR is issued over a communication link with a transform module, the transform and untransform functions are applied to the data. For ECB transforming and untransforming the data is straightforward. Transforming the message ECB encrypts it using the startpoint's key and untransforming it ECB decrypts it using the endpoint's key.

For CBC and MD5 authentication the process of transforming and untransforming is more complicated. Transforming the message starts by CBC encrypting the message using the startpoint's key and ivec. The CBC encryption algorithm modifies the ivec as a function of the ivec's initial value and the data being encrypted. Then, after CBC encrypting, a MAC is calculated using MD5 authentication with the startpoint's MD5 key and the *encrypted* message. In addition to sending the encrypted message, the MAC and the address of the ivec node on the endpoint side are both placed into the message header.

On the receiving side, the untransform operation starts by attempting to authenticate the message using MD5 authentication. A MAC is calculated using the endpoint's MD5 key and the *encrypted* message. This MAC is compared to the MAC in the message header. If the two are equal, the message is authenticated, and only then is the message decrypted. Decrypting

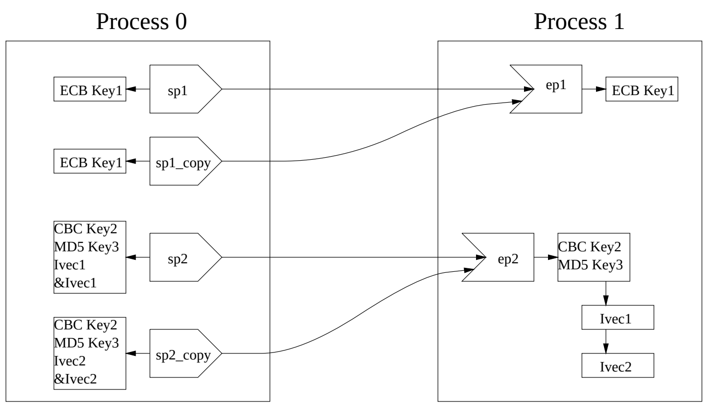

Figure 5: Two pairs of startpoints and endpoints with copies.

the message requires extracting (from the message header) the address of the appropriate ivec node in the endpoint's list of ivec nodes. Then, using the endpoint's CBC key and the ivec node, the message is CBC decrypted. The CBC decryption algorithm changes the value of the ivec as a function of the ivec's original value and the decrypted message. In this example we illustrate that security contexts may or may not change with use.

8 Experimental Results

We report on a number of experiments that we have conducted to study the performance of our techniques. These comprise a simple microbenchmark, designed to yield insights into the costs associated with basic communication operations, and a large-scale application study. We emphasize that these experiments have all been performed in the context of a large-scale working system.

All experiments are performed on the Argonne IBM SP2, which connects 80 Power 2 processors with an SP2 high-speed switch. The SP2 supports both a fast, machine-specific communication library (MPL) and TCP/IP. MPL has performance characteristics typical of high-speed parallel computer communication libraries (55 MB/sec bandwidth, small-message latencies of around 50 μsec). TCP over the SP2 switch runs at about 22 MB/sec and incurs small-message latencies of around 320 μsec; hence, it has performance characteristics similar to a tuned OC3 or faster ATM network in a metropolitan area network. TCP communication on the SP2 that does not used the high-speed switch has performance characteristics similar to Ethernet.

8.1 Microbenchmark Results

We use a microbenchmark to compare the performance of secure and unsecure versions of our basic communication mechanisms. This Nexus program bounces a vector of fixed size back and forth between two processors a large number of times. Each communication is achieved by an RSR to the remote node, with the RSR handler that executes on the remote node invoking an RSR back on the originating node. The experiment is repeated for different vector sizes. Figures 6 and 7 show results obtained in four different configurations: Nexus when using IBM's low-level MPL communication library, with and without DES encryption and MD5 authentication (MPL Secure, MPL Unsecure); and Nexus when using TCP/IP communication, with and without DES encryption and MD5 authentication (TCP Secure, TCP Unsecure). In those experiments that used TCP/IP communication (TCP Secure and TCP Unsecure) we did not utilize the SP2 high-speed switch. Note that identical source code, DES encryption libraries, and MD5 authentication libraries were used for all experiments.

In all the microbenchmark experiments encryption was performed using a DES library in cipher block chaining (CBC) mode. The library used is `libdes` version 3.00 written by Eric Young. Authentication in all the microbenchmark experiments used the MD5 message digest algorithm. The library used is RSAREF version 2.0 from RSA Laboratories.

The results reveal a number of interesting attributes of our Nexus secure communication infrastructure. Looking first at Figure 6, we note that for small messages, the underlying communication protocol (TCP vs. MPL) makes a bigger difference to performance than whether or not security is enabled. For a 10-byte message, unsecure MPL communication takes 78 μsec, while secure MPL takes 128 μsec: 64 percent slower than unsecure MPL, but still a lot faster than both secure and unsecure TCP, which take 394 and 472 μsec, respectively. These results emphasize the importance of using optimized low-level communication mechanisms when these are available.

For larger messages, encryption costs dominate communication time. Beyond 300 bytes, secure MPL is slower than unsecure TCP (but still considerably faster than secure TCP). Looking at Figure 7, we see that for messages larger than a few thousand bytes, secure MPL and TCP have essentially the same cost. This is because the communication costs for large messages are dominated by the limited performance of the DES encryption library; the Power 2 processor can encrypt and then decrypt data at only 1 MB/sec, far slower than the SP2 network.

8.2 Application Results

Our application study uses the FOAM fast ocean-atmosphere model, designed to run at relatively low resolutions for multicentury simulations [30]. This model uses MPI for communication and combines a large atmosphere model (the Parallel Community Climate Model [6]) with an ocean model (from U. Wisconsin). The two models execute concurrently and perform considerable internal communication. Periodically, the models exchange information such as sea surface temperature and various fluxes.

To provide a controlled environment for our experiments, we run the two model components not on two different computers but instead on distinct sets of nodes (partitions) of the Argonne SP2 (Figure 8). Communication between partitions is always performed by using TCP, this time utilizing the high-speed switch hence approximating a situation in which we have two computers connected by an ATM metropolitan area network. Communication within a partition may be performed by using either MPL or TCP; we present results for both cases. In all cases, user-level communication is achieved by using the MPI implementation that we have constructed by layering on top of Nexus [9]. (This layering adds an execution time overhead of about 6 percent when compared with a "native" MPI.) No changes to the application program were required to run the different scenarios considered below.

Table 1 gives our results. We present results for three different scenarios: no encryption

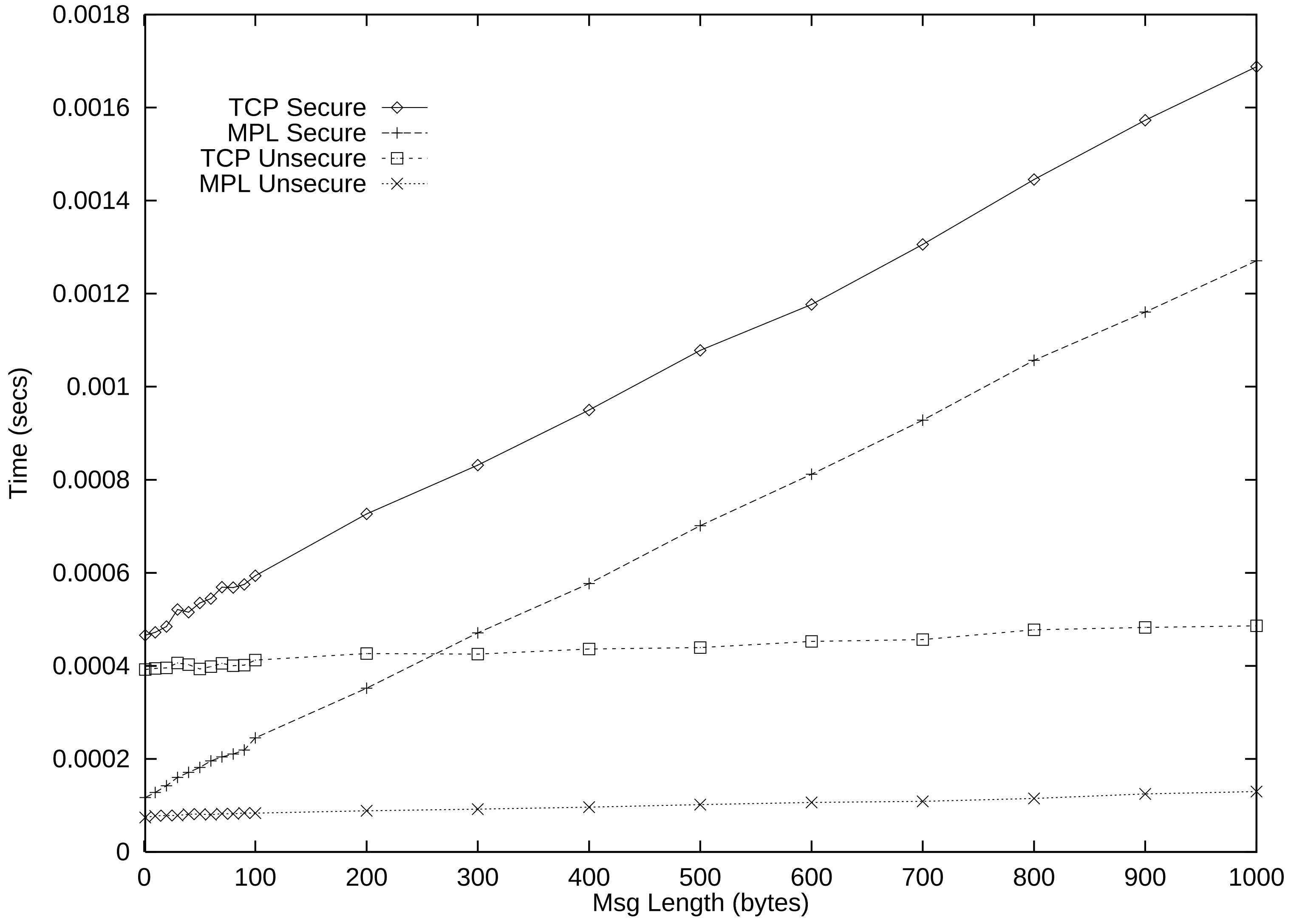

Figure 6: Microbenchmark results: See text for details

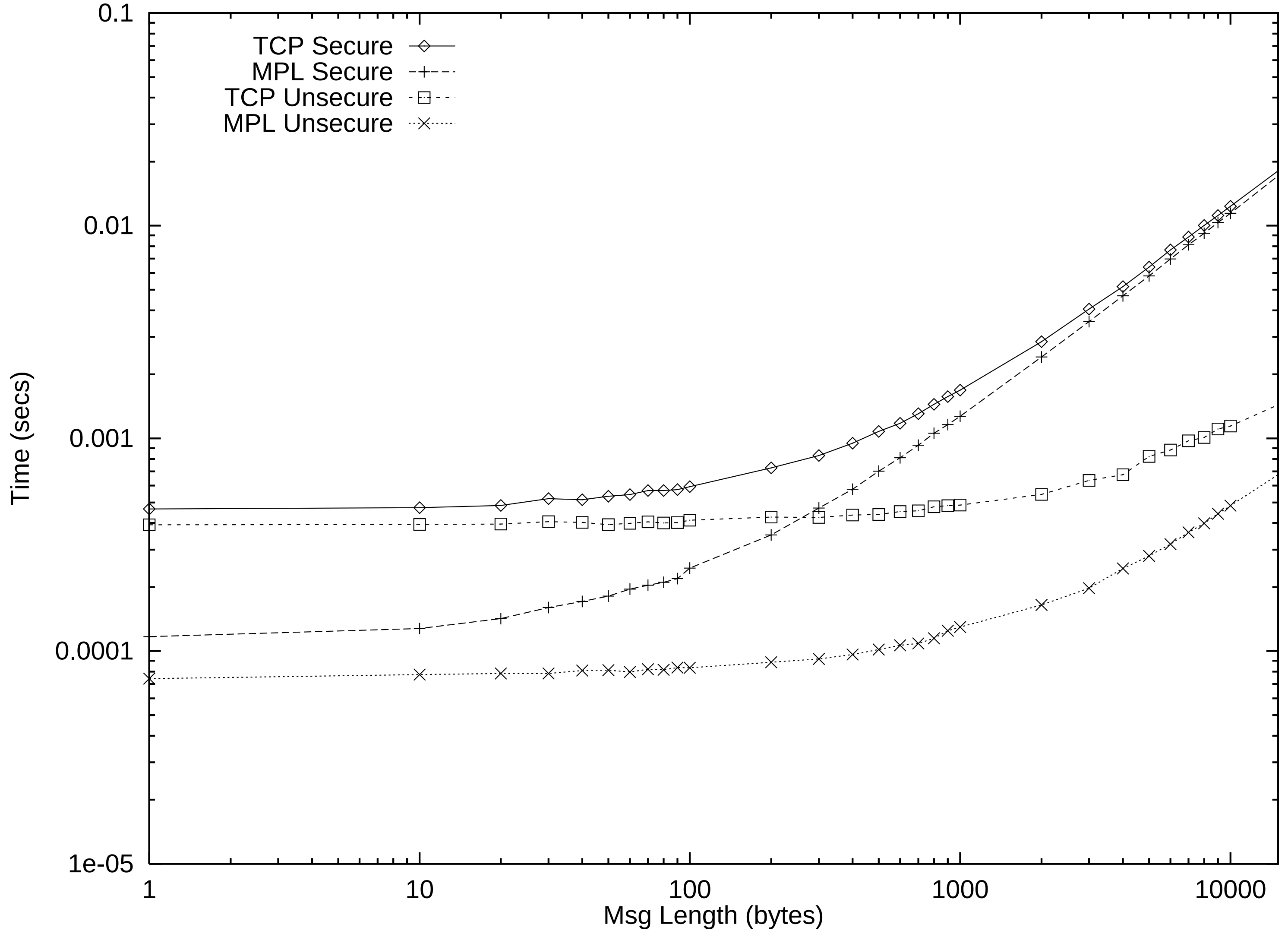

Figure 7: Microbenchmark results; note the use of log scales. See text for details

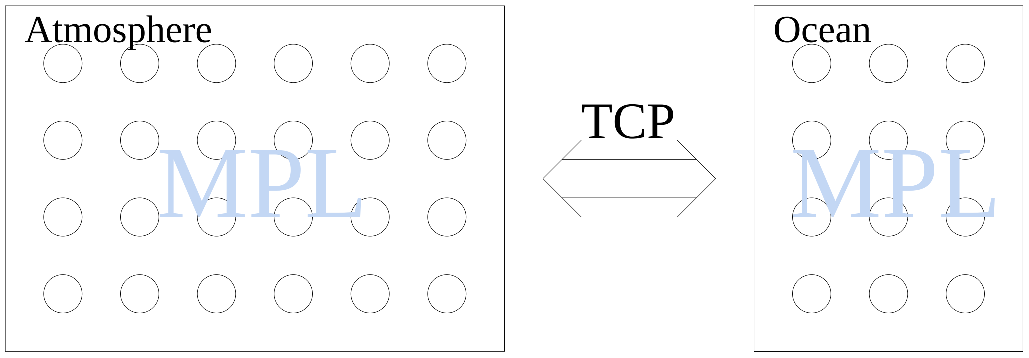

Figure 8: The Argonne/Wisconsin coupled ocean/atmosphere model in the configuration used for our multimethod communication experiments, showing the two IBM SP partitions.

Table 1: Time per simulated day for the coupled ocean/atmosphere model, with different security modes and communication protocols on an IBM SP2

Mode	TCP time (secs/day)	MPL time (secs/day)
No secure	854	574
Coupler secure	897	590
All secure	1459	1187

("No secure"), encryption (ECB mode) only on communications between models ("Coupler secure"), and encryption (ECB mode) on all communications ("All secure"). In each case, we consider configurations in which either TCP or MPL are used within a partition.

Our results demonstrate the importance of a communications infrastructure that can both support the use of multiple low-level communication methods (MPL as well as TCP) and permit selective application of encryption. When using encrypted TCP for all communication, total time is 1459 seconds per simulated day. Allowing the use of MPL within a partition reduces execution time by 19 percent, to 1187 seconds/day. Turning off encryption within partitions reduces execution time by a further 50 percent, to 590 seconds/day. The latter time is only 3 percent slower than when using no encryption at all.

9 Related Work

While there has been considerable earlier work on portable security mechanisms for distributed computing, issues relating to high-performance computing have received less attention.

The Secure Socket Library [14] (SSL) allows the programmer to associate different security mechanisms with different physical connections (sockets), but does not permit the use of specialized communication methods. In contrast, Nexus allows different security mechanisms to be associated with different logical connections, which furthermore can communicate with different low-level protocols.

Jaspan [16] describes the use of GSS-API to implement secure remote procedure calls. He reports an overhead of over 11 milliseconds for a secured RPC with no arguments; clearly, this work does not emphasize performance.

Venugopal [32] describes a secure implementation of Parallel Virtual Machine, a popular message passing library. He uses a secure `rsh` for remote process creation and Diffie-Hellman key exchange to communicate a secret session key from the initial user process to all other processes. Encryption is enabled on a per-session basis, at the command line, although the programmer also has the option of specifying that a specific message should be secured using a particular technique. There is no support for associating a security mechanism with a particular logical connection.

The Prospero Resource Manager (PRM) [24] uses Kerberos mechanisms to provide secure process creation mechanisms for PVM. Depending on the level of security required, PRM can be configured to execute (a) only those programs whose executables reside in the PRM binaries directory (b) executables residing on the filesystem local to the site (c) local executables as well as those downloaded from remote sites from which jobs are submitted.

The x-kernel [26] and Horus [31] use protocol composition techniques to construct security enhanced versions of communication methods without the specialized "escape" used in Nexus. This approach introduces certain overheads but has high flexibility. We hope to explore its use in future work.

10 Conclusions

We have described the design and implementation of a secure communications infrastructure for high-performance distributed computing applications. This infrastructure integrates authentication, encryption, and data integrity mechanisms into the tools typically used to develop high-performance applications. These security-enhanced tools make it possible to run large-scale distributed applications in a secure manner, without any changes to the applications themselves. In addition, the tools provide hooks that programmers can use to manage explicitly the security mechanisms used for different communications. Experimental results demonstrate that in heterogeneous environments we can obtain significant performance advantages by employing multiple transport mechanisms and by enabling security mechanisms only when communicating selectively.

In more recent work than is reported here, Nexus has been integrated into Globus [11], a toolkit for constructing high-performance distributed computing, or metacomputing systems. As part of this integration, the process-creation mechanisms have been replaced by a more flexible service called the Globus Resource Allocation Manager, or GRAM. GRAM is responsible for implementing a well-defined Globus security policy that provides a single sign on capability and mutual process to resource and process to process authentication. This security policy has been implemented using the GSS-API [19], eliminated the need to code authentication or transform modules to a specific security API. The transform based encryption methods discussed in this paper are used to provide data privacy in Globus.

In future work, we propose to deploy these security-enhanced communication tools in a wide-area Globus testbed that we are constructing, called GUSTO. This deployment will allow large-scale application experiments and hence provide feedback on how our security mechanisms work in practical situations. It seems certain that encryption performance will be a bottleneck in many situations. Hence, we will experiment with various performance enhancement techniques, including specialized protocols [1], parallel encryption algorithms [22, 23], and use of dedicated encryption processors. Another interesting direction for further work will be to investigate the feasibility of using the Metacomputing Directory Service to determine when secure communication mechanisms must be employed, for example because communication occurs over insecure network connections. Clearly one issue that will be important to address in this context is the authenticity of resource database entries.

Acknowledgments

We are grateful to Greg Koenig and Jonathan Geisler their his considerable assistance with the experiments reported in this paper, and to John Anderson, Robert Jacob, and Chad Schafer for making the coupled model available to us. This work was supported by the Mathematical, Information, and Computational Sciences Division subprogram of the Office of Computational and Technology Research, U.S. Department of Energy, under Contract W-31-109-Eng-38.

References

[1] I. Agi and L. Gong. An empirical study of secure MPEG video transmissions. In *Proc. Symp. on Network and Distributed System Security*, pages 137–144. IEEE Computer Society Press, 1996.

[2] C. Catlett and L. Smarr. Metacomputing. *Communications of the ACM*, 35(6):44–52, 1992.

[3] T. DeFanti, I. Foster, M. Papka, R. Stevens, and T. Kuhfuss. Overview of the I-WAY: Wide area visual supercomputing. *International Journal of Supercomputer Applications*, 10(2):123–130, 1996.

[4] D. Diachin, L. Freitag, D. Heath, J. Herzog, W. Michels, and P. Plassmann. Remote engineering tools for the design of pollution control systems for commercial boilers. *International Journal of Supercomputer Applications*, 10(2):208–218, 1996.

[5] T. L. Disz, M. E. Papka, M. Pellegrino, and R. Stevens. Sharing visualization experiences among remote virtual environments. In *International Workshop on High Performance Computing for Computer Graphics and Visualization*, pages 217–237. Springer-Verlag, 1995.

[6] J. Drake, I. Foster, J. Michalakes, B. Toonen, and P. Worley. Design and performance of a scalable parallel Community Climate Model. *Parallel Computing*, 21(10):1571–1591, 1995.

[7] S. Fitzgerald, I. Foster, C. Kesselman, G. von Laszewski, W. Smith, and S. Tuecke. A directory service for configuring high-performance distributed computations. In *Proc. 6th*

IEEE Symp. on High Performance Distributed Computing, pages 365–375. IEEE Computer Society Press, 1997.

[8] I. Foster, J. Geisler, C. Kesselman, and S. Tuecke. Managing multiple communication methods in high-performance networked computing systems. *Journal of Parallel and Distributed Computing*, 40:35–48, 1997.

[9] I. Foster, J. Geisler, and S. Tuecke. MPI on the I-WAY: A wide-area, multimethod implementation of the Message Passing Interface. In *Proceedings of the 1996 MPI Developers Conference*, pages 10–17. IEEE Computer Society Press, 1996.

[10] I. Foster, N.T. Karonis, C. Kesselman, G. Koenig, and S. Tuecke. A secure communications infrastructure for high-performance distributed computing. In *Proc. 6th IEEE Symp. on High Performance Distributed Computing*, pages 125–136. IEEE Computer Society Press, 1997.

[11] I. Foster and C. Kesselman. Globus: A metacomputing infrastructure toolkit. *International Journal of Supercomputer Applications*, 11(2):115–128, 1997.

[12] I. Foster, C. Kesselman, and S. Tuecke. The Nexus approach to integrating multithreading and communication. *Journal of Parallel and Distributed Computing*, 37:70–82, 1996.

[13] W. Gropp, E. Lusk, and A. Skjellum. *Using MPI: Portable Parallel Programming with the Message Passing Interface.* The MIT Press, 1994.

[14] K. Hickman. The SSL protocol. *Internet Draft RFC*, 1995.

[15] International Business Machines Corporation, Kingston, NY. *IBM Load Leveler: User's Guide*, September 1993.

[16] B. Jaspan. GSS-API security for ONC RPC. In *Proc. Symp. Network and Distributed Systems Security*, pages 144–151. IEEE Computer Society Press, 1993.

[17] C. Koelbel, D. Loveman, R. Schreiber, G. Steele Jr., and M. Zosel. *The High Performance Fortran Handbook.* The MIT Press, 1994.

[18] C. Lee, C. Kesselman, and S. Schwab. Near-realtime satellite image processing: Metacomputing in CC++. *Computer Graphics and Applications*, 16(4):79–84, 1996.

[19] J. Linn. Generic security service application program interface. *Internet RFC 1508*, 1993.

[20] T. Maples and G. Spanos. Performance study of a selection encrytion scheme for the security of networked, real-time video. In *Proc. 4th Intl. Conf. on Computer Communications and Networks*, 1995.

[21] C. Mechoso et al. Distribution of a Coupled-ocean General Circulation Model across high-speed networks. In *Proceedings of the 4th International Symposium on Computational Fluid Dynamics*, 1991.

[22] E. Nahum, S. O'Malley, H. Orman, and R. Schroeppel. Towards high performance cryptographic software. In *3rd IEEE Workshop on the Architecture and Implementation of High Performance Communication Subsystems*, 1995.

[23] E. Nahum, D. Yates, S. O'Malley, H. Orman, and R. Schroeppel. Parallelized network security protocols. In *Proc. Symp. on Network and Distributed System Security*, pages 145–154. IEEE Computer Society Press, 1996.

[24] B. Clifford Neuman and Santosh Rao. The Prospero resource manager: A scalable framework for processor allocation in distributed systems. *Concurrency: Practice & Experience*, 6(4):339–355, 1994.

[25] M. Norman, P. Beckman, G. Bryan, J. Dubinski, D. Gannon, L. Hernquist, K. Keahey, J. Ostriker, J. Shalf, J. Welling, and S. Yang. Galaxies collide on the I-WAY: An example of heterogeneous wide-area collaborative supercomputing. *International Journal of Supercomputer Applications*, 10(2):131–140, 1996.

[26] S. O'Malley and L. Peterson. A dynamic network architecture. *ACM Transactions on Computing Systems*, 10(2):110–143, 1992.

[27] H. Schulzrinne, S. Casner, R. Frederick, and V. Jacobson. RTP: A transport protocol for real-time applications. *Internet RFC 1889*.

[28] J. Steiner, B. Neuman, and J. Schiller. Kerberos: An authentication system for open network systems. In *Usenix Conference Proceedings*, pages 191–202. 1988.

[29] D. Stevenson, N. Hillary, G. Byrd, F. Gong, and D. Winklestein. Design of a key agile cryptographic system for OC-12c rate ATM. In *Proc. Symp. Network and Distributed Systems Security*, pages 17–30. IEEE Computer Society Press, 1993.

[30] M. Tobis, I. T. Foster, C. M. Shafer, R. L. Jacob, and J. R Anderson. FOAM: Expanding the horizons of climate modelling. In *Supercomputing '97*, San Jose, California, 1997. ACM.

[31] R. van Renesse, K. Birman, R. Friedman, M. Hayden, and D. Karr. A framework for protocol composition in Horus. In *Proc. Principles of Distributed Computing Conf.*, 1995. http://www.cs.cornell.edu/Info/People/rvr/papers/podc/podc.html.

[32] N. Venugopal. The design, implementation, and evaluation of cryptographic distributed applications: Secure PVM. Technical report, University of Tennessee, Knoxville, Tenn., 1996.