A Flexible Security System for Metacomputing Environments*

Adam Ferrari, Frederick Knabe, Marty Humphrey,
Steve Chapin, and Andrew Grimshaw

University of Virginia
Department of Computer Science

Abstract. A metacomputing environment is a collection of geographically distributed resources (people, computers, devices, databases) connected by one or more high-speed networks, and potentially spanning multiple administrative domains. Security is an essential part of metasystem design—high-level resources and services defined by the metacomputer must be protected from one another and from corrupted underlying resources, and underlying resources must minimize their vulnerability to attacks from the metacomputer level. We present the Legion security architecture, a flexible, adaptable framework for solving the metacomputing security problem. We demonstrate that this framework is sufficiently flexible to implement a wide range of security mechanisms and high-level policies.

1 Introduction

Legion [5, 6] is a distributed computing platform for combining very large collections of independently administered machines into single, coherent environments. Like a traditional operating system, Legion provides convenient user abstractions, services, and policy enforcement mechanisms over a diverse set of lower-level resources. The difference is that in Legion, these resources may consist of thousands of heterogeneous processors, storage systems, databases, legacy codes, and user objects, all distributed over wide-area networks spanning multiple administrative domains. Legion provides the means to pull these scattered components together into a single, object-based *metacomputer* that accommodates high degrees of flexibility and site autonomy.

Security is an essential part of the Legion design. In a metacomputing environment, the security problem can be divided into two main concerns: (1) protecting the metacomputer's high-level resources, services, and users from each other and from corrupted underlying resources, and (2) preserving the security policies of the underlying resources that form the foundation of the metacomputer and minimizing their vulnerability to attacks from the metacomputer level. For example, restricting who is able to configure a metacomputer-wide scheduling service would fall in the first category. Its solution requires metacomputer-specific definitions of identity, authorization, and access control. Meanwhile, enforcing a policy that permits only those metacomputer

* This work was funded by DARPA contract N66001-96-C-8527, DOE grant DE-FD02-96ER25290, DOE contract Sandia LD-9391, and DOE D459000-16-3C

users who have local accounts to run jobs on a given host falls in the second category. Its solution might require a map between local identities and verifiable metacomputer identities.

To satisfy users and administrators, a full security solution must address and reconcile both of these security concerns. Users must have confidence that the data and computations they create within the metacomputer are adequately protected. Administrators need assurances that by adding their resources to a metacomputer (and thus making those resources more accessible and valuable to users), they are not also introducing unreasonable security vulnerabilities into their systems.

Attempting to incorporate security as an add-on late in the implementation process has been problematic in a number of first-generation metacomputing systems such as PVM, MPI, and Mentat. To avoid this pitfall, the Legion group has addressed security issues since the earliest design phases [10]. Our metacomputing security model has three interrelated design goals: flexibility, autonomy, and breadth. *Flexibility* demands that the framework be adaptable to many different security policies and allow multiple policies to coexist. *Autonomy* is essential so that organizations and users within a metacomputing environment can select and enforce their desired security policies independently. Finally, *breadth* refers to the ability of the metacomputer's architectural framework to enable a rich set of security policy features.

These goals are strongly driven by our view that a fundamental capability of a metacomputer is its ability to scale over and across multiple trust domains. A Legion "system" is really a federation of meta- and lower-level resources from multiple domains, each with its own separately evaluated and enforced security policies. As such, there is no central kernel or trusted code base that can monitor and control all interactions between users and resources. Nor is there the concept of a superuser—no one person or entity controls all of the resources in a Legion system.

If it is to satisfy a broad range of security needs, our architecture must allow the implementation of a number of different security features. These include

– Isolation
– Access control for resources
– Identity of principals
– Detection and recovery
– Communication privacy and integrity
– Integration with standard mechanisms

The first point, isolation, refers to the ability of components in the metacomputer to insulate themselves from security breaches in other parts of the system. This feature is particularly important in large Legion networks, where we must generally assume that at least some underlying hosts have been compromised or may even be malicious.

In this paper we elaborate a metacomputing architecture based on our design goals that addresses both parts of the metacomputing security problem. In our discussion, we present examples of mechanisms we have designed or implemented within the architecture that enable a number of useful security policies, and provide examples of those policies.

2 Architectural Support for Security

Legion is composed of independent, active objects. All entities of interest within the system—processing resources, storage, users, etc.—are represented by objects [7]. Legion objects communicate via asynchronous method calls supported by an underlying message passing system. Each method call contains actual parameters and an optional set of *implicit parameters,* metadata that is available to called objects. Objects are instances of classes that define their interface, which is required to be a superset of a minimal *object-mandatory* interface. Object-mandatory methods include functions such as an interface query and methods to implement object persistence.

Legion objects are persistent, and are defined to be in one of two states: *active* or *inert*. When an object is active, it is hosted within its own running process and can service method calls. When an object is inert, its state (called its Object Persistent Representation, or OPR) is preserved on a storage device managed within the system. Objects implement internal methods to store and recover their dynamic state.

Legion Runtime Library The implementation of Legion objects is supported by a *Legion Runtime Library* (LRTL) interface. The LRTL defines the interfaces to services such as message passing, object control (e.g., creation, location, deletion), and other basic required mechanisms.

A critical element of the LRTL is its flexible, configurable protocol stack [9]. All of the processing performed in the construction of method calls at the sender and in handling them at the recipient is configured using a flexible, event-based model. This feature makes it especially convenient for tool builders to provide drop-in protocol layers for Legion objects. For example, adding message privacy through a cryptographic protocol is simply a matter of registering the appropriate message processing event handlers into the Legion protocol stack—the added service is transparent to the application developer.

Core Objects Within the Legion object model we define the interfaces to a set of basic classes that are fundamental to the operation of the system and that support the implementation of the object model itself.

Host Objects in Legion represent processing resources. When a Legion object is activated, it is a Host Object that actually creates a process to contain the newly activated object.[1] The Host Object thus controls access to its processing resource and can enforce local policies, e.g., ensuring that a user does not consume more processing time than allotted.

Vault Objects in Legion represent stable storage available within the system for containing OPRs. Just as Host Objects are the managers of active Legion objects, Vault Objects are the managers of inert Legion objects. For example, Vaults are the point of access control to storage resources, and can enforce policies such as file system allocations.

Hosts and Vaults provide the system with interfaces to processing and storage resources. The use of these interfaces is encapsulated by *Class Manager Objects*.[2] Class

[1] In some environments, the Host Object may enter the object as a new job to run in a queue management system, but this difference is transparent to the rest of Legion.

[2] In many of the cited Legion references, Class Manager Objects are referred to simply as "Class Objects."

Managers are responsible for managing the placement, activation, and deactivation of a set of objects, or *instances,* of a given class. They provide a central mechanism for specifying policy for a set of like objects. Policies set by the Class Manager include defining which implementations are valid for instances, which hosts are suitable for execution of instances, which users may create new instances, and so on. In addition to setting policy for instances, Class Managers serve as location authorities for instances, supporting the binding of object ids to low-level object addresses (typically an IP address plus port number).

A critical aspect of the Legion core object classes is that they define interfaces, not implementations. The Legion software distribution provides a number of default reference implementations of each core object type, but the model explicitly enables and encourages the configuration, extension, and even replacement of local core object implementations to suit site- and user-specific requirements. For example, by replacing the implementation of the Host Object, a site can define arbitrary mechanisms and policies for the usage of their computational resources.

3 Security Features in Legion

The Legion architecture is the critical foundation for satisfying the flexibility, autonomy, and breadth goals for our metacomputing security model. We now consider how those goals are met in the current system implementation.

Identity In Legion, every object is identified by a unique, location-independent Legion Object Identifier, or LOID. LOIDs consist of a variable number of binary fields. As a default Legion security practice, we use one of the LOID fields to store an X.509 certificate including (at a minimum) an RSA public key. By including an object's public key in its LOID, we make it easy for other objects to encrypt communications to that object or to verify messages signed by it. Objects can just extract the key from the LOID, rather than looking it up in some separate database, which eliminates some kinds of public key tampering.

Users in Legion also have LOIDs. A user creates his own LOID, which is then registered with the system and entered in appropriate system groups and access control lists by resource providers. When an object makes a call on behalf of the user, the user's LOID and associated credentials provide the basis for authentication and authorization. The ownership of a user's LOID resides in the user's unique knowledge of the private key that is paired with it. The private key is kept encrypted on disk, on a smart card, or in some other safe place.

For a resource, the essential step in deciding whether to grant an access request is to determine the identity of the caller. If a user communicates directly with the target object, he can establish his identity relatively easily with an authentication protocol. In a distributed object system, however, the user typically accesses resources indirectly, and objects need to be able to perform actions on his behalf. To transfer the user's identity in Legion, we issue *credentials* to objects. A credential is a list of rights granted by the credential's maker, normally the user or his proxy. A credential is passed through call chains, and is presented to a resource to gain access. The resource checks the rights in the credential and who the maker is, and uses that information in deciding to grant access.

There are two main types of credentials in Legion: *delegated credentials* and *bearer credentials.* A delegated credential specifies exactly who is granted the listed rights, whereas simple possession of a bearer credential grants the rights listed within it. A credential specifies the period for which it is valid, who is allowed to use the credential, and which method calls it can be used for. The credential also includes the identity and digital signature of its maker.

Tools or commands directly executed by the user create the credentials they need to carry out their actions. The credentials are made as specific as possible to avoid unnecessary dispersion of authority. Short timeouts in credentials, coupled with user-specific *Refresh Objects* that can revalidate expired credentials, permit a variety of recovery tactics if a credential or user key is stolen. Additional details concerning credentials and credential refresh can be found in [2].

Access Control In Legion, access is defined as the ability to call a method on an object. The object may represent a file, a Legion service, a device, or any other resource. Access control is not centralized in any one part of the Legion system. Each object is responsible for enforcing its own access control policy. It *may* collaborate with other objects in making an access decision, and indeed, this allows an administrator to control policy for multiple objects from a single point. The Legion architecture does not require this, however.

The general model for access control is that each method call received at an object passes through a *MayI* layer before being serviced. MayI is defined on a per-object basis, and is specified as an event in the configurable LRTL protocol stack [9]. MayI decides whether to grant access according to whatever policy it implements. If access is denied, the object will respond with an appropriate security exception.

MayI can be implemented in multiple ways. The default LRTL MayI implementation is based on access control lists and credential checking. In this MayI, *allow* and *deny* access control lists containing user and group LOIDs can be specified for each method in an object. When a method call is received, the credentials it carries are checked by MayI and compared against the access control lists. Multiple credentials can be carried in a call; checking continues until one provides access.

The form of access control provided by the default MayI is sufficient for some kinds of objects, such as file objects, but not for others. The LRTL configurable, event-based protocol stack makes it easy to replace or supplement the default MayI with extra functionality. Furthermore, the default MayI itself is relatively simple to modify if, for example, new forms of credentials or different kinds of access control lists must be supported. With the Legion security architecture, these types of changes can be made on a local basis without affecting other parts of a Legion system.

Communication Privacy and Integrity Encryption and integrity services are provided at the level of Legion messages. When a Legion message is prepared for sending, an event handler that implements a message security layer is triggered. This layer inspects the implicit parameters accompanying a message to determine which security functions to apply. In the current LRTL, a message may be sent with no security, in *private mode,* or in *protected mode.* In both private and protected modes, certain key elements of a message (e.g., any contained credentials) are encrypted using the public key of the recipient. The functional difference between the two modes is in how the rest of the message is treated. In private mode it is encrypted, whereas in protected mode only a digest is generated to provide an integrity guarantee. Unless private mode is already on, protected mode is selected automatically if a message contains credentials. This is a failsafe measure to prevent credentials from being transmitted in the clear. Details of the encryption mechanisms can be found in [2].

Because the mode in use is stored in implicit parameters, it propagates through call chains. For example, a user can select private mode when calling an object. All subsequent calls made by objects on behalf of the user will also use private mode. The default security layer does not provide mutual authentication. The sender can be assured of the identity of the recipient, because only the desired recipient can read the encrypted parts of the message. The recipient usually doesn't care who the actual sender is; its decisions are based solely on the credentials that arrived in the message.

Object Management and Isolation The management of active and inert objects by Legion core objects is an important point of local security mechanism and policy in Legion. Fundamentally, Legion software runs on existing operating systems with their own security policies. It is therefore critical that the implementation of the Legion object model ensure that extra-Legion mechanisms cannot be used to subvert higher-level security mechanisms. Similarly, it is important to ensure that Legion does not break local security policies at a site. A local system administrator is generally concerned with who can create processes on his system via Legion, what those processes can do, and who pays for their resource use. On Legion's side, there is a need to prevent user objects from interfering with one another or with core system objects (e.g., Hosts and Vaults), and to maintain the privacy of persistent state (OPRs). The latter is particularly significant because objects store their private keys in their OPRs.

The needs of Legion are common to any multi-user operating system, and our approach to providing them is to leverage off of existing operating system services. Our general strategy for isolating objects from one another in the default Legion implementation is to use separate accounts to execute different user objects. Similarly, we use local accounts and storage system protections to protect OPRs.

Accounts that can be used for these purposes fall into two categories. For those Legion users who happen to have accounts on the local system, processes and storage that represent the user's objects can be owned by the user's local account. For other users, we support the use of a pool of generic accounts that are designated for Legion use. The generic accounts usually have minimal permissions (e.g., no home directory, no group memberships, etc.). The local Host and Vault Objects use their own dedicated local accounts to ensure isolation from other user objects.

We encapsulate the privileged operations necessary for this policy in a *Process Control Daemon* (PCD) that executes on the host, providing services to the Host and Vault in a controlled fashion. The PCD is a small, easily vetted program that runs with root permissions. It is configured only to allow access by the user account on which the Host and Vault Objects are running. Its key functions are recursive change ownership of a directory, process creation under a designated account, and process termination. The PCD limits the user-ids to which these operations can be applied to a set configured by the local system administrator. The set includes the generic Legion accounts and potentially the accounts of local Legion users.

Alternatively, a local site policy may require that Kerberos be used to authenticate access to all local user accounts. Depending on the local Kerberos configuration, the Host Object can use forwarded Kerberos credentials, entries in users' Kerberos authorization files, or callbacks to user credential proxies to start objects on the appropriate accounts. The point is not how this is done, but that it can be done: Legion can adapt to a large range of security standards as necessary.

4 Policy Examples

Although Legion's flexibility allows the implementation of a wide variety of security mechanisms, application developers and site administrators typically have higher-level policy specifications in mind when using software. The particular underlying mechanisms are less important, as long as the user can be assured that high-level policy requirements are being met. In this section, we consider illustrative examples of how the Legion system architecture and existing Legion tools can be organized to meet sample site and application policies.

Site Isolation A Legion system can consist of multiple domains, each possibly in a different organization or trust domain. System administrators contributing resources to a larger metasystem typically require certain site-isolation properties. For example, consider a site that makes resources available to Legion, and is managed by a given local Legion administrator, who we will call *Admin*. A reasonable policy is that no matter how subverted any external sites in the Legion system might be, no intruder can invoke methods on local Legion resources as Admin. Such a policy is clearly desirable since Admin is likely to have administrative control over critical local resources: who can use which machine, and for how long; who can access which locally stored OPRs; etc. The ability to invoke methods as Admin is tantamount to complete control of the local Legion software.

The desired isolation policy can be achieved through a number of straightforward safeguards enabled by the Legion framework. First and foremost, all of the core objects managing the local site should be started and configured by Admin. This isolated domain startup avoids any external trust dependencies on outside systems. However, to achieve the desired functionality of a metacomputer, the local domain will be connected to some set of external Legion domains. After this link to the external (and untrusted) system is made, Admin must ensure that no messages containing his credentials are sent to off-site objects, as a subverted or malicious external site could then use Admin's credentials to break the isolation policy. However, simply stating that Admin should not pass credentials off-site is not good enough—Admin might make a simple mistake that could break the policy, so we would like automated enforcement of this safety measure. Such automated enforcement is easy in Legion: Admin simply uses a version of the LRTL in which the protocol stack is configured with an extra event handler for the message-send event. If a message is inadvertently directed off-site while containing Admin credentials, the message is blocked and the event handler raises an exception. With this simple modification to Admin's Legion environment, he can be assured that his credentials will not be dispersed to untrustworthy off-site objects.

Ensuring that Admin does not communicate with off-site objects has a desirable secondary effect. Since Admin cannot communicate with external, untrustworthy sites, he cannot place critical objects on resources at these sites. This benefit extends to an array of potentially critical, but not necessarily obvious, resources. For example, suppose Admin maintains a local Group Object listing the set of users that are allowed to start objects on local resources. If this object were allowed to execute on an untrustworthy site, its contents could be modified by a malicious resource owner, and local site-resource usage policy could be broken.

The two mechanisms described above, in combination with carefully configured access control for local core objects such as Hosts, Vaults, and critical Class Managers, ensure that the desired isolation policy will be met. Off-site objects will neither be able to generate nor steal local Admin's credentials. External callers will be prevented from invoking unauthorized methods on local critical resources, ensuring that local access control is not tampered with, local resource usage policies are not modified, and that security failures in other domains do not have serious consequences for the local site.

Site-Wide Required Access Control The Legion access control model as presented in Section 2 is based on the assumption that users will configure access control for their own objects. This concept adds a powerful level of flexibility to the system—for example, it makes arbitrary resource access policies possible. However, on first examination it appears to relinquish the ability for a system administrator to set site-wide policies about access control for user objects. For example, the default Legion access control configuration does not grant the administrator for a Legion domain access to other users' objects within the domain—there is no root user who can read any file or use any program in the domain. Such lack of ability to configure global, site-wide, mandatory access control policies may be unacceptable at some sites. However, the flexibility of the Legion architecture allows us to address this issue in a straightforward fashion using existing tools.

As an example of a site-wide access control policy, we consider the problem of prohibiting access to files by outside users. The Legion system defines a basic File Object that can be used to represent a file in the system. Access control for the normal Legion File Object is based on the default Legion MayI mechanism, which places no restrictions on what LOIDs (i.e., what users) may be placed on access control lists. To enforce the policy that files may not be accessed by outside users, we effectively want a way to control which LOIDs may be placed on the ACL for local file objects. We can achieve this policy using the power of local Host Objects to control access to local resources. The Host Objects at the site (which are owned and controlled by the local administrator) are a point of resource access policy—they define which types of objects may run at the site. Using this feature, the site administrator can strictly limit the classes of objects that may run at the site. In particular, the allowable set of classes can be limited to those that are approved by the system administrator. The list of allowable classes can be configured to only include file objects with an alternate MayI layer—an extended version of the default ACL mechanism that also verifies that allowed LOIDs are in a well-known group containing only the local site users. Given this simple configuration, the site administrator can ensure that files are not inadvertently exported to outside users through Legion. Furthermore, this approach generalizes to other site-wide access control restrictions, and other similar site-wide policy enforcement problems.

Firewalls Firewalls are a simple fact of life at many security-conscious institutions. While firewalls are not addressed explicitly in the Legion model, the Legion architecture is sufficiently flexible to accommodate firewalls with ease. As is typical in firewall situations, a proxy on the firewall host is the natural solution. However, the ability to use custom versions of the Legion core objects, and the flexible protocol stack model of the LRTL, allow proxy-based solutions to be employed in Legion in an especially straightforward, user-transparent way.

Objects started on hosts behind a firewall automatically have a Proxy Object on the firewall host assigned to them by their Host Object (in some cases, each user might desire their own proxy object; in other cases, a shared proxy object is acceptable; either model is simple to support). The object address for a newly activated object behind the firewall that is reported to the object's Class Manager is actually the address for the Proxy Object—when callers of the object bind its LOID to an object address, they will be given the address of the Proxy Object. The Proxy Object then acts as a simple reflector, forwarding any received messages to their intended destinations behind the firewall. Use of the Proxy Object to forward outbound messages from callers behind the firewall is automated by a transparent add-in event handler in the LRTL protocol stack.

Resource Selection Policy In principle, a user of a metacomputer shouldn't need to care which resources are used to execute his jobs. In practice, however, the trustworthiness of the resources that are selected for certain applications is of critical interest to the user. Policies regarding which resources may be used to execute objects are logically localized within the Class Managers of a user's object classes. In principle, any site selection policy can be encoded in a user's Class Manager Objects, giving the user total control over the selection and use of trustworthy sites.

Although this problem is solved cleanly at the architectural level in Legion, we deemed this issue of site selection for application users important enough to warrant special features in the default Class Manager Object reference implementations. All default Class Managers in Legion check for certain implicit parameters that can be used to limit resource selection. By setting these implicit parameters in his Legion environment (using a provided tool), the user can configure a resource selection policy that will propagate to all "create instance" methods called on Class Manager objects on behalf of the user. Of course, the architectural principle that users can encode any resource selection policy they wish in their own Class Manager implementations still holds; in fact, a convenient model for such customization is supported by the default Class Manager's ability to be configured to use an external *Scheduler Object* with a well-known interface. However, in the common case, where a user can generate a list of sites that he deems trustworthy and indicate this in his environment, the default implementation provides the mechanism to implement an effective resource selection policy.

5 Related Work

Two projects that incorporate security into large-scale distributed computing platforms are Globus and WebOS. Globus [3] is a "bag of services" model for metacomputing, in contrast to Legion's integrated environment approach. The Globus Security Infrastructure is a single sign-on authentication system that is deployed at each site in a Globus network. Different underlying authentication protocols such as Kerberos and SSL may be plugged into the infrastructure via GSS-API modules. A local Globus site uses the authentication information it receives in a request to make authorization decisions; it can also call back to a user's proxy to confirm the request.

The Globus Security Infrastructure essentially focuses on one component of the overall metacomputing security problem. Legion, with its "network operating system" perspective, addresses broader issues that allow the development of sophisticated security policies to manage metacomputer resources, as described in Section 4. The Legion architecture fundamentally permits greater autonomy and flexibility in the choice of security technologies and approaches. Globus does address an important part of the metacomputing security puzzle, however, and it could be chosen as an alternate mechanism to the current RSA approach for implementing identity and integrity in a Legion system.

CRISIS [1] is the security architecture for WebOS. WebOS is fundamentally different from Legion in terms of the basic services provided. WebOS provides a single, traditional file system and a fixed interface for authenticated remote process creation. CRISIS defines careful, effective security policies for these basic services. However, the CRISIS solution does not provide a means for easily developing security policies for new mechanisms as they are added to WebOS, nor does it provide a means for modifying the security policies supported for the existing services.

Two other projects related to security efforts in Legion, although not with the focus on metasystems, are Java and CORBA. The computational model of Java [4] (JDK 1.2) requires identity and authentication in order to execute digitally signed code downloaded from a remote site. The JDK provides per-class (or per-application) protection domains. However, it differs significantly from Legion in its lack of support for per-site security mechanisms, delegation, and user authentication.

The security model of CORBA [8] encompasses identification and authentication, authorization and access control, auditing, security of communication, non-repudiation, and security information administration. Typically, an ORB vendor implements CORBA security using existing technology such as GSS-API, Kerberos, and SESAME. Many of the goals of the CORBA security model are similar to the goals of the Legion security model, including simplicity, scalability, usability, and flexibility. However, CORBA is not a metacomputing system—it does not construct an operating system-like environment using underlying distributed resources. Given this fundamental difference in target use, CORBA does not address the metacomputing security problem.

6 Conclusions

We have presented the basic security architecture of the Legion system, and we have demonstrated that our design is sufficiently flexible to accommodate a wide variety of security-related mechanisms. This flexibility is critical to the successful deployment and use of metacomputing software. One-size-fits-all software dictated by a single group will never satisfy the requirements of the wide range of users and resource providers in a large-scale, cross-domain environment. We have also demonstrated that flexibility does not come at the price of complete lack of control. Within the flexible Legion framework, we showed how a number of important site-wide and application-wide security policies could be achieved. Naturally, the set of policies presented is only a small fraction of the policies that will be needed across the complete Legion environment.

The Legion system, including the security features described here, is currently publicly available. It is widely deployed on hundreds of machines at dozens of sites spanning multiple trust domains. Key portions of the software, such as the PCD described in Section 2, have been vetted and approved by system administrators at sites such as the San Diego Supercomputing Center and the US Naval Oceanographic Office (NAVO). In the future, we plan to continue deployment of Legion, developing additional mechanism and adapting to new site-local policies as required. We are also in the process of measuring the performance impact of key Legion security mechanisms.

References

1. E. Belani, A. Vahdat, T. Anderson, and M. Dahlin. CRISIS: A wide area security architecture. In *Seventh USENIX Security Symposium*, Jan. 1998.
2. A. Ferrari, F. Knabe, M. Humphrey, S. Chapin, and A. Grimshaw. A flexible security system for metacomputing environments. Technical Report CS-98-36, Department of Computer Science, University of Virginia, Charlottesville, Virginia, Dec. 1998.
3. I. Foster, C. Kesselman, G. Tsudik, and S. Tuecke. A security architecture for computational grids. In *Fifth ACM Conference on Computers and Communications Security*, Nov. 1998.
4. L. Gong, M. Mueller, H. Prafullchandra, and R. Schemers. Going beyond the sandbox: An overview of the new security architecture in the Java development kit 1.2. In *USENIX Symposium on Internet Technologies and Systems*, pages 103–112, Dec. 1997.
5. A. S. Grimshaw and W. A. Wulf. Legion: A view from 50,000 feet. In *Fifth IEEE Symposium on High Performance Distributed Computing*, Aug. 1996.
6. A. S. Grimshaw and W. A. Wulf. The Legion vision of a worldwide virtual computer. *Communications of the ACM*, 40(1):39–45, Jan. 1997.
7. M. Lewis and A. Grimshaw. The core Legion object model. In *Fifth IEEE Symposium on High Performance Distributed Computing*, Aug. 1996.
8. Object Management Group. CORBAservices: Common object services specification, security service specification. Version 97-12-12, 1998.
9. C. Viles, M. Lewis, A. Ferrari, A. Nguyen-Tuong, and A. Grimshaw. Enabling flexibility in the legion run-time library. In *International Conference on Parallel and Distributed Processing Techniques and Applications*, pages 265–274, June 1997.
10. W. A. Wulf, C. Wang, and D. Kienzle. A new model of security for distributed systems. Technical Report CS-95-34, Department of Computer Science, University of Virginia, Charlottesville, Virginia, Aug. 1995.