SAND--97-8479C
CONF-970729--2

Lilith: A scalable secure tool for massively parallel distributed computing

R.C. Armstrong, L.J. Camp, D.A. Evensky, and A.C. Gentile
Sandia National Laboratories
Livermore CA, USA

1. Introduction

Changes in high performance computing have necessitated the ability to utilize and interrogate potentially many thousands of processors. The ASCI (Advanced Strategic Computing Initiative) program conducted by the United States Department of Energy, for example, envisions thousands of distinct operating systems connected by low-latency gigabit-per-second networks. In addition multiple systems of this kind will be linked *via* high-capacity networks with latencies as low as the speed of light will allow. Code which spans systems of this sort must be scalable[1]; yet constructing such code whether for applications, debugging, or maintenance is an unsolved problem. Lilith is a research software platform that attempts to answer these questions with an end toward meeting these needs. Presently, Lilith exists as a test-bed, written in Java, for various spanning algorithms and security schemes. The test-bed software has, and enforces, hooks allowing implementation and testing of various security schemes.

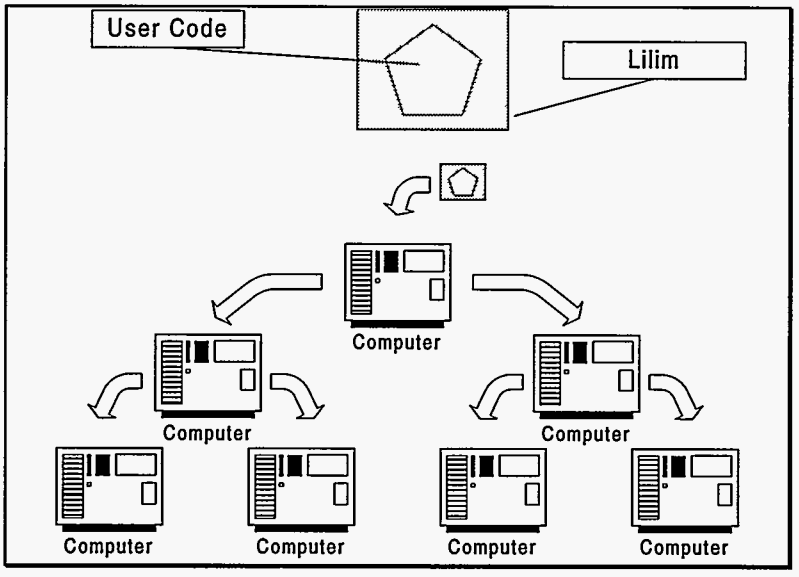

Figure 1. User code contained in Lilim to be propagated down tree of LilithHosts.

Lilith's principle task is to span a tree of machines executing user-defined code in a scalable and secure fashion. Beginning from a single object, Lilith recursively links child host objects, LilithHosts, on adjacent machines until the entire tree is occupied[2].

[1] We us scalable in the sense that a binary tree (N-cube) fan-out is scalable within a factor of $\log_2$(No. of Processors).

[2] Lilith is the "Mother of D(a)emons" in Middle Eastern mythology.

DISCLAIMER

The child host objects propagate down the tree code, Lilim[3], that performs user-designated functions on every machine. Although other uses for the system are envisioned, this is the mode of operation that is currently of most interest. Because of its platform independence and wide-spread use, Java is the language in which Lilith is written. Java also allows easy creation of graphical user interfaces with browser front ends. To adhere to existing standards, Lillith is targeted to be compatible with the Legion[4] object system. This standard has been selected because of the need for high-performance asynchronous operations Legion has been chosen over more established object systems, such as CORBA, primarily because of the need for security designed in from the ground up, supporting fine security gradations and degrees of control.

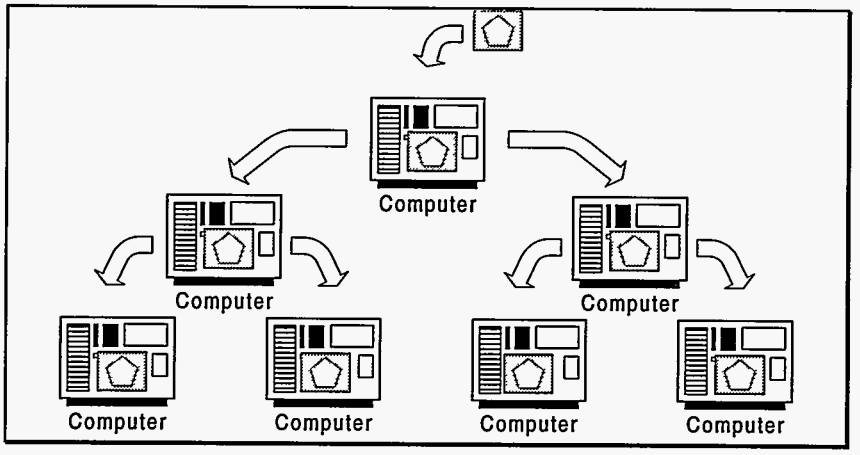

Figure 2. After propagation, Lilum are started as threads under the LilithHost objects.

Lilith is easy to implement in Java, which is designed to move across networks and execute its own code. An essential aspect of the Lilith effort is securing against unauthorized and uncontrolled access. Without a credible and reliable system for protecting resources from rogue Lilith implementations and protecting Lilith from rogue resource managers , there is little likelihood of Lilith being used. A related concern is that of unintentional but damaging user mistakes: The scalable nature of Lilith could magnify the consequences to disastrous proportions. We are concentrating on preventing unauthorized or destructive actions, however, we cannot prevent users from making authorized, ill-considered actions which affect only themselves.

Though Lilith may seem exotic, its intended use is quite pragmatic. Lilith will be employed for controlling user processes as well as general system administrative tasks

[3] Lilim (the children of Lilith) will be used for both the plural and singular name of the first class objects that are sent down the tree.

[4] A.S. Grimshaw and W.A. Wulf, "Legion -- A View From 50,000 Feet", Proceedings of the Fifth IEEE International Symposium on High Performance Distributed Computing, IEEE Computer Society Press, Los Alamitos, California, August 1996, and references therein.

(e.g., creation of user accounts, monitoring of system status.). Although there exist tools[5] to accomplish some of these tasks, they are not scalable (or rely on relatively weak security).

For example, a user's parallel code, consisting of thousands of workers scattered across the network, stalls because of an unknown problem with a single worker. Parallel computations are commonly structured such that, if one worker is held up, it will eventually stall the entire application. There is no *a priori* way of determining which worker is the culprit. Each machine has a debugger capable of determining which worker processes are not in a blocked in a receive state (a likely sign that *it* is the culprit). Lilith can carry the necessary code to all of the worker machines and perform such a query with the local machine's debugger simultaneously and scalably. If a likely culprit is found, a debugger window can be attached to the user's X-server. If the script implementing this query is run on each machine from a single host in a loop, the time to completion may be prohibitive. The most obvious reason for this is that a loop lacks scalability; a less obvious reason is that the host may have a fast link to some of the machines and slower to others.

Clearly Lilith meets the need for scalability, Since each machine has its connections preconfigured to its "nearest" neighbors, Lilith can always use the fastest route. It is expected that the greatest benefit will come from scripts and code that configure and interrogate the parallel program. This includes placing data files where worker processes expect to find them; steering and debugging the running parallel code, retrieving output after the code completes.

2. The Lilith Object Model

The Lilith object model inherits from several sources, chief among them is Legion. The design and organization of Lilith's class structure and basic security mechanisms are guided by compatibility with Legion. Lilith builds upon this framework by implementing a general interface query method, using Java to gain first class objects, and adopting a tree structure for scalablilty. Lilith consists of a subset of the interfaces necessary to implement the goals described above. As Lilith matures it will encompass a full Java implementation of Legion including specialized interfaces targeted towards the needs of a tool-oriented system. We will discuss here the basic structure of a LilithObject and LilithClass. We will also present two particular LilithObject's: LilithHost and Lilim. There are many classes that are used by the above (e.g. messaging classes) that will be presented only as needed. The classes that deal exclusively with the Lilith security mechanisms, and the interaction of that mechanism with the Java sandbox will be addressed in the following section.

[5] NASA's NAS LAMS package for secure remote user account management and Platform Computing's LSF (http://www.platform.com) for parallel job control are representative examples.

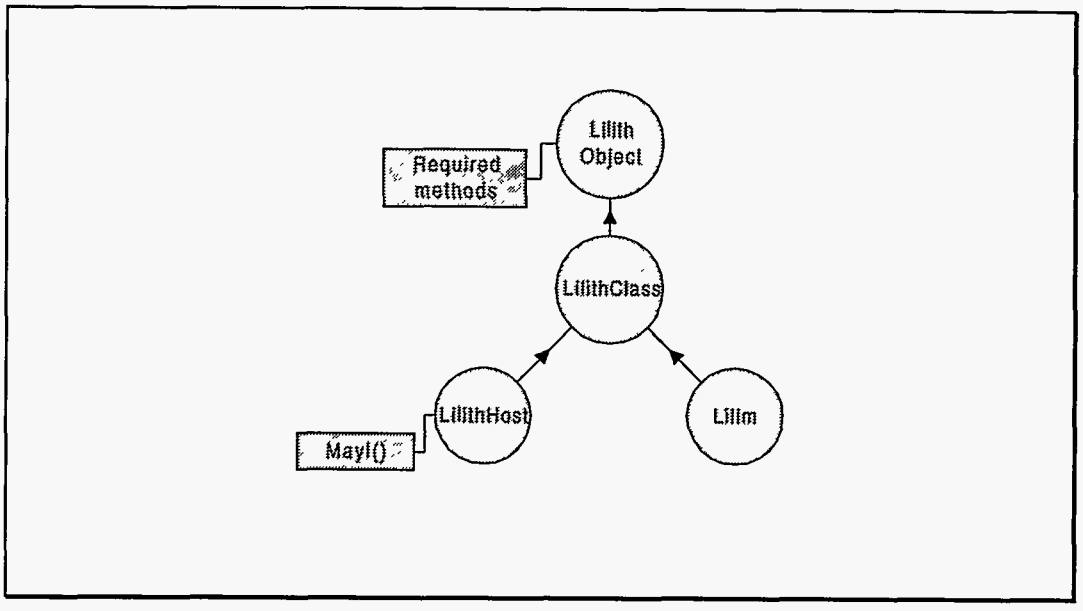

Figure 3. Lilith object hierarchy

The LilithObject class is the root class of all objects participating within Lilith (excluding, of course, utility classes). It defines well-known interfaces that all LilithObjects must implement and its methods are the mechanism by which one LilithObject communicates with another. It also maintains state information pertaining to the security environment.

The LilithClass class is used primarily for name resolution and control over the instantiation of objects and the maintenance of the objects while in a persistent state. In the initial implementation, it is primarily a place holder; it will prove most useful when Lilith becomes a Java implementation of Legion.

The LilithHost object is responsible for protecting the computing resource in which it is running from other Lilith objects (in particular, Lilim, described below) and for instantiating Lilith objects on that host. For the purposes here, we define a host to be a system running under a single OS.

Lilim are the Lilith objects that carry within them user code. They include well known interfaces that allow the user code to interact with the environment. The Lilim run principallyas threads under the LilithHost.

There are also several utility classes that will be used in implementation of the above. Within the object-oriented RPC mechanism, there is a messaging class that will be responsible for marshalling data to and from the stub code and the implementation code on the server side. It is expected that this will eventually be generated by an IDL

compiler. In our initial implementation, these are generated by hand. The message class is fully multithreaded. There will also be classes to implement digital signatures and encryption. These will be described below.

3. Lilith security model

There are two fundamental research areas in the security arena in Lilith: the relationship between performance and security, and distributed trust. In addition, the issue of transactional reliability is critical. At this point the security protocol is in the preliminary design stage.

That security is a critical element of this project is obviously illustrated by simply imagining the project without security. Security is integrity, authentication, and confidentiality. Lilith requires authentication to assure that the initiator is a trusted LilithObject, and that no object is gaining inappropriate access. Lilith requires integrity to assure that the program a LilithObject accepts is the one sent by trusted LilithObject and has not been altered.

Lilith may or may not require confidentiality. This may be an option included for the user, or it may be included depending on both the algorithm chosen and the distribution of message sizes. If DSS is the selected signature algorithm it will be because it is determined that signature generation is more centralized than signature verification. In this case confidentiality would require session key generation or additional public key operations. This would imply that confidentiality presents a performance/security trade-off best made by each user. Should signature verification dominate generation (a likely case) then the algorithm used for authentication, RSA, could seamlessly provide confidentiality for small messages.

The final determination of algorithms will not take place until the spanning tree algorithm and trust relationships have been determined for any particular case. Trust relationships depend upon both access control and authentication. Once access has been extended, authentication becomes a critical issue.

The security model for Lilith will draw from the Legion[6], Java, electronic commerce, and electronic privacy paradigms.

Legion has solved the problem of mapping object names and keys by making the public key be the object's name. In effect, the name of a LilithObject will provide its own authentication through the use of Legion key and name mapping. Legion allows users to choose their own optimization of the security/performance trade-off. MayI() is the Legion access control mechanism, which will be adopted in a modified form in Lilith. MayI() is a required method that Lilith objects must (at least trivially) define. When a method is invoked remotely, it will trigger a call to the object's MayI(), which may reject

[6] W.A. Wulf, C. Wang, D. Kienzle, "A new model of security for distributed systems", UVa CS Technical Report CS-95-34, August 1995

the remote invocation. Lilith requires each Lilim to check for access permission through invocation of MayI() in the LilithHost. LilithHost's MayI() can then alter the behavior of the Java sandbox. This forces compliance with mandatory security practices rather than implementing optional access control. This modification of the sandbox prevents Lilim from calling any method without passing first asking, MayI(). When MayI() is called authentication and integrity have to be verified.

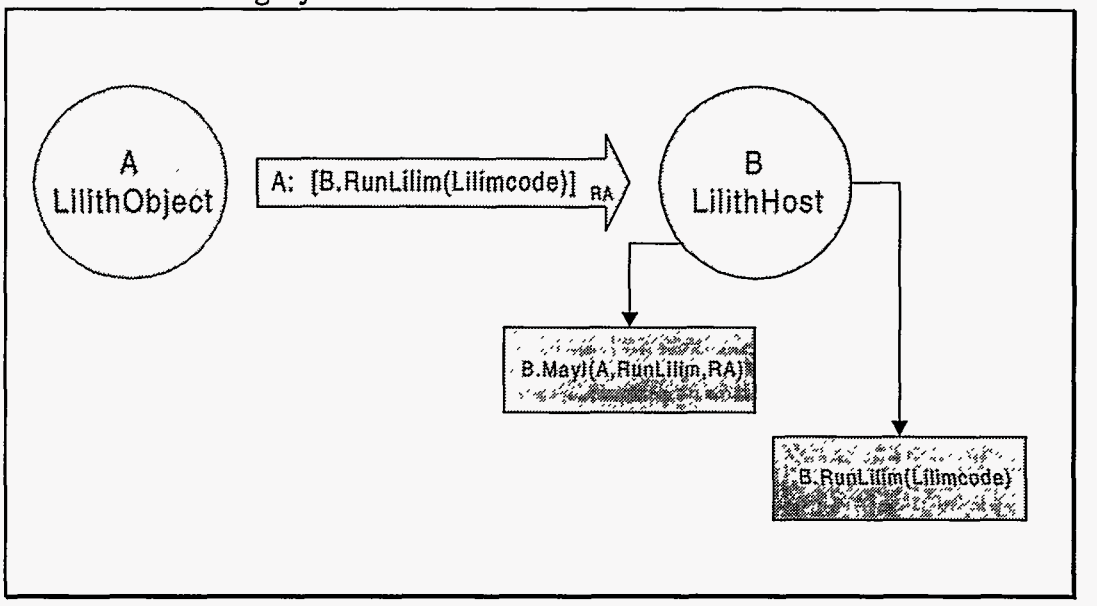

Figure 4. An object of type LilithObject (possibly a LilithHost) invokes the RunLilim method on the LilithHost, B. The message is signed and perhaps encrypted with the key RA (Responsible Agent). When B receives the message, it invokes it's MayI, which checks to see if the call will be allowed. If it is, then B.RunLilim will be invoked, otherwise the call will fail.

Authentication requires three things:

- the requesting LilithObject claims an identity and a public key;
- the mapping between public key pair and identity is verified;
- the requester proves knowledge of the corresponding secret key.

The requesting LilithHost claims ownership of a public key by presenting its own name. LilithHost use their own public keys to sign their names. A requester proves knowledge of the corresponding secret key by this self-signature, as well as the integrity-assuring signature on a Lilim.

The mapping between the public key pair and the name is verified by a combination of the Web of Trust[7] and public key certificates. Public key certificates provide third party verification of key/identity mappings using a central server and key hierarchies. Lilith needs verification of mappings, but a central server is an unacceptable bottleneck. The

[7] Simson Grarfinkel, "PGP: Pretty Good Privacy", O'Reilly & Associates, Inc., Sebastopol CA, pp 235-236.

Web of Trust provides key/identity mappings without a key hierarchy, using a central server to maintain endorsements[89].

Lilith uses the conceptual model of the Web of Trust with the practical application of public key certificates. The Web of Trust consists of various users who are all partially trusted. In order to provide widespread trust the LilithObject generates a self-signed certificate. The LilithObject then sends this certificate to some subset of other objects for signatures. However, unlike the Web of Trust, there will exist in any domain more established or trusted objects. LilithObjects will select some k of these hosts and obtain verification. Each Lilim will be verified by each LilithHost according to that LilithHost's sensitivity to security. Rather than verifying a chain leading to a single trusted root; each LilithHost will verify some subset of partially trusted signatures; verification resembles a cut and choose technique as much as a key hierarchy.

Key revocation is also at issue. There are two possibilities for key revocation. First a key was not valid when assigned. In this case the key is obtained by an attacker who joins Lilith with intent to subvert the system. Second, a key is valid but then it is subverted. A primary issue with key revocation is assuring that the key revocation technique itself does not create an opportunity for a trivially easy denial of service attack.

4. Conclusions

Lilith's purpose is to carry user code to span a large number of independent operating systems where scalability of operations is important. Principle to its practicality and wide-spread acceptance is security. Without security the Lilim are little more than viruses and LilithHost is little more than an open door. This security must itself be scaleable, however. Traditional means of authentication would introduce bottlenecks. The Lilith scheme presented in Section 3 accomplishes this by employing a mechanism that allows a subset of a small number of trusted objects (k of n) to vouch for the authenticity of the Lilim and its signature. The envisioned uses for Lilith are controlling, steering and debugging parallel-distributed programs over thousands of operating systems. It is likely that parallel programs themselves could take advantage of Lilith to initialize and run, but currently the focus is on getting small pieces of code executed on large numbers of processors quickly.

The scope of this project is confined to this purpose. It does not seek to be a general distributed-object model, but rather draws upon the work already in the distributed-object field. The Java-based test-bed for these security schemes, draws upon the Legion security model, admitting a wide variety authentication mechanisms while conforming to an established standard for networked objects. Lilith does not seek to soften, or make transparent, the differences in operating system architecture, file systems, and installed

---

[8] C. Wang and W.A. Wulf, "A distributed key generation technique", http://www.cs.virginia.edu/~cw2e/kgen.ps, 1995.

[9] The only endorsement made in a key signature in the Web of Trust is the endorsement that a key/identity pair are valid. There is no endorsement of the trustworthiness of the holder of the key.

software capabilities, *etc.*. The Java Virtual Machine provides a considerable amount of insulation by itself and is the principle reason for its use here. However it maynot be possible to fork-off a debugger window to an Xserver (to continue the example from Page 3) without a knowledge of the local machine architecture. If the distributed machines in use are heterogeneous, the user must test for the type of environment and make appropriate provision for it. It is not that general networked object models and transparency across heterogeneous architectures aren't important, those concerns are just outside of the scope of this investigation.

Lilith, and its security scheme, is intended for scientific computing, requiring access and "need to know" controls only within a small community of users. It is clear, however that if the scope were broadened, there are many commercial applications of a capability of this kind. For example, we might wish Microsoft to initiate an upgrade for their Windows 95 operating system for users who are unable to do so on their own (an increasing and probably dominant number of users on the Internet today). We might wish to institute a security scheme in which a Lilim, authenticated as Microsoft, has access only to the operating system and not, say, to files with our credit card number in them. This would require a much finer degree of security control than exists today. Today, the current practice is: attach to something the user thinks is ftp.microsoft.com; download and execute a file with unrestricted access; hope for the best. Compared to accepted standards a system like Lilith does not seem too dangerous.

Prompted by the US Dept. of Energy Accelerated Strategic Computing Initiative, the forseeable use for Lilith is for massively parallel distributed computing for scientific and engineering applications alone. To enable the use of on-the-order-of a thousand machines, with a thousand separate operating systems, one has to have the ability to address them with one voice. Lilith provides that voice and this work is the beginnings of its practical reality.