THE LILITH FRAMEWORK FOR THE RAPID DEVELOPMENT OF SECURE SCALABLE TOOLS FOR DISTRIBUTED COMPUTING

David A. Evensky, Ann C. Gentile,
Pete Wyckoff and Robert C. Armstrong

Sandia National Laboratories
P.O.Box 969 MS 9011
Livermore, CA, 94551, USA
{ evensky | gentile | pw | rob } @ca.sandia.gov

Abstract: Lilith is a general purpose framework written in Java, that provides a highly scalable distribution of user code across a heterogeneous computing platform. By creation of suitable user code, the Lilith framework can be used for tool development. Lilith promotes rapid development since it handles all the details of code distribution and communication, while the user code need only define the tool functionality on a single node, in accordance with a simple API. The scalable performance provided by Lilith is crucial to the production of time-effective tools for large distributed systems. This time-efficiency, however, allows both constructive and destructive tasks to be accomplished quickly; therefore, security is a concern. We present the Lilith API and a prototype example of the usage of Lilith in distributed computing and we discuss the security model, which is currently under design.

Keywords: Scalable tools, secure tools, frameworks, distributed computing

1 INTRODUCTION

The future of high-performance computing lies in massively parallel simulations, which traditionally have been performed on monolithic MPPs. These machines are becoming too expensive on a per-flop ratio when compared to their smaller commodity

L. Kutvonen et al. (eds.), *Distributed Applications and Interoperable Systems II*
© Springer Science+Business Media New York 1999

brethren. Cheap, widely available PCs can be purchased whole or piece-wise from a variety of vendors and, to a large extent, conform to the same basic set of hardware standards *e.g.*, having a PCI bus. Networking components required to assemble many PCs into a single system are likewise cheap and plentiful.

Several projects[1, 2, 3] exist to build supercomputers out of commodity parts. A major stumbling block for cluster builders has been the failure to ensure that the system would scale. This is distinct from the usual requirement that applications must scale well to be efficient, and speaks to the reliability and usability of the system itself. Sandia's Computational Plant cluster, CPlant, is designed as an arbitrarily connected aggregation of individual quantized pieces (scalable units) each of which is entirely self-contained and self-managing. This allows pieces to be easily swapped in and out as the state of technology evolves. The ideal tools for managing and monitoring such systems would be scalable—taking advantage of both the short execution times provided by parallelism and the scalable design of the cluster itself.

Lilith[4] is a software framework, written in Java, that provides for a highly scalable and easy distribution of user code across a heterogeneous computing platform. This capability is of value in the development of tools to be employed in the use and administration of very large (thousands of processors) clusters. Since Lilith, rather than the user, handles the details of code distribution and communication, the time required for tool development is greatly decreased. The user only defines the tool functionality required on a single node, in accordance with a simple API.

Scalability, because of its inherent power, is a double-edged sword. It allows both constructive and destructive tasks to be performed efficiently. Security is therefore a concern. For large clusters handling many users, security must be flexible, supporting a variety of needs. The design and implementation of flexible, scalable security in Lilith is currently in progress. Secure start-up is in place through the use of ssh[5]. A secret key is distributed which will be used for security on a per-method invocation basis. The Legion model for security[6] is the basis for our model.

Lilith can be used for the creation of tools employed for the control of user processes on the distributed system as well as for general administrative tasks. Although there exist tools[7] to accomplish some of these tasks, they are either not scalable or rely on relatively weak security. We describe Lilith's functionality and API, present an example tool that capitalizes on Lilith's features, and discuss the security model under design.

2 LILITH FUNCTIONALITY AND API

The principle task of Lilith is to span a group of machines with user-defined code. Lilith uses a tree structure for its internal communications; this is distinct from the physical connectivity of the machines. The Lilith framework consists of a number of objects necessary for handling the details of maintaining the tree and communicating amongst nodes in the tree.

Beginning from a single object, Lilith recursively links host objects, LilithHosts, on adjacent machines until the entire tree is occupied. The LilithHosts distribute user code objects, called Lilim (both singular and plural), down the tree. The Lilim

execute, performing user-designated functions on every machine. Tools are produced by suitable development of Lilim.

Lilith provides the mechanisms by which data can be passed down to the Lilim and results collected from them. The messages are in the form of a MessageObject, MO, that supports both a queue and a hash table interface. The Hosts communicate with one another *via* a Remote Proceedure Call (RPC) mechanism. For now, we implement our own RPC because this allows us to insert the security hooks at the lowest functional levels. Support for the Java Remote Method Invocation (RMI) and other message passing communications are being considered as well. Interactions within Lilith involving the actual implementations of the objects handling the communications have been designed to simplify making such a change.

In many cases, a tool requires user code to undergo a (possibly repeating) three-phase process: data distribution, code execution, and result collection. Lilith provides the user with a special Lilim with a simple interface to handle this standard case. This Lilim defines default functionality for each of the three phases and ensures that these functions are called at the proper times and that messages are passed correctly. The user's code then inherits from this Lilim, overriding the methods for the three phases to suit his own tool's needs. The special Lilim methods are:

MO[] distributeOnTree(MO m, int[] numDesc) defines the actions of the Lilim as information is passed down the tree. It receives as arguments a message from its parent, and an integer array specifying the total number of descendents of each child. It processes the message and returns an array of MOs, one for itself, and one that will be given to each child as an argument to their distributeOnTree() methods. By default it returns an array of copies of the argument MO.

MO onTree(MO m) defines the action of the Lilim on that node. It receives as an argument the MO for itself that it had returned from distributeOnTree(). It returns an MO containing the result of the action. By default it returns the MO it received as an argument.

MO collateOnTree(MO[] m) defines the actions of the Lilim results are returned up the tree. This could involve further processing or condensing of results. It receives as an argument an array of the result messages, one returned from itself from onTree(), and one from each of its direct children returned from their collateOnTree() methods. It returns a combined message to be sent to its parent. By default it returns an MO containing all the MO's it received as arguments.

A simple example would be a distributed sort. An MO containing the entire list of numbers to sort is sent as the argument to distributeOnTree(). This method then divides the list into subpieces, one for itself and for each of its direct descendents, and returns MOs containing these pieces. The method onTree() receives the sublist to be handled on that node, sorts it, and returns it. Finally, collateOnTree() takes the sorted

lists produced by itself and its children, performs a merge sort, and returns an MO containing the combined sorted list.

Note that MOs for the node itself to handle are returned from distributeOnTree() and onTree() and passed as parameters to onTree() and collateOnTree(), respectively. This avoids requiring the the user code to override methods simply in order to save messages from one phase to the next. For instance, if one doesn't want to alter the message on the way down the tree, *i.e.,* one wants the originally distributed message value to processed on *all* nodes by onTree(), then it is not necessary to override distributeOnTree() simply to save the message since it will be received again as the argument to onTree(). The default behavior of distributeOnTree() provides onTree() with the unadulterated message as its argument. Relatedly, even if all methods are overridden, the user still need not explicitly save state information since these values can be placed in the MO that will be returned to itself.

Lilith uses a tree structure[8, 9, 10, for discussion of tree architectures and their cost-performance tradeoffs] for its communications in order to provide scalability. In trees in general, there can be congestion problems at the root, especially if many nodes attempt to talk to distant nodes at the same time. In Lilith, however, communications follow a full distribution down the tree or a full collection up the tree, rather than a random pattern. This suits tools that match the divide-and-conquer strategy. Messages from the children are combined and can be condensed as they are returned up the tree. Therefore such congestion is unlikely to occur. No application that sends information of limited compressibility from each node can be truly scalable as the number of nodes goes to infinity. We present in the next section, an example tool which uses a fixed data size for all communications, in order to preserve scalability.

3 EXAMPLE TOOL

Lilith was instrumental in developing a tool for network traffic visualization called Lilith Lights[11]. This tool provides graphical information about the CPU usage of nodes and information about communication among nodes. It can be used for debugging parallel and distributed codes and for resource management decisions.

Network and CPU usage data are captured using small hooks placed in the lowest level device driver to record packet transmission and arrival. A message is distributed down the tree to the Lilim requesting traffic information. The Lilim then collect this information from the kernel code. Each Lilim returns to its parent accumulated state information, consisting of its own information merged in with that of its children. The combined return information is handled in a format that maintains a fixed message size at each stage of the return.

Unlike typical software and hardware implemenations for this sort of status gathering, Lilith Lights is a relatively non-invasive, portable tool that can display information on distributed clusters *concurrently* with the application under study. Since the information is gathered from the system by the Lilim, the tool does not require relinking of the application, which would change the conditions under which the code was being run. Because Lilith maintains its own communications structure, the Lilith traffic can be sent across a secondary network to avoid interference with the application's traffic.

The support the of Lilith infrastructure for the data passing greatly reduced the time that would have been necessary to have written this tool from scratch.

4 SECURITY

The scalable nature of Lilith allows fast access to a large number of computing resources. If not protected against, widespread damage could be done quite quickly. Futhermore, the large clusters for which Lilith was designed typically support many users from differing organizations, possibly with varying security policies. For these reasons, we are currently designing a security scheme based on the Legion security model[6] allowing a rich set of access control checks per remote method invocation.

LilithHosts are started on the remote machines through the use of ssh [5]. A secret key is generated and distributed at the time of tree building which will be used for access control. All objects are required to define or inherit a MayI() function, possibly a trivial one; whenever a method is invoked remotely, it will trigger a call to the remote object's MayI() function, which may reject the remote invocation. Lilith will enforce that each invocation to instantiate and run a user's Lilim will check for access permission through the invocation of MayI() in the LilithHost on that node. While the MayI() provides discretionary access control, an additional function method that the Lilim writer can provide, CanI(), allows the Lilith framework to enforce mandatory access control. This is currently being designed using a combination of the Java SecurityManager and checks in the RPC stubs. The complexity of the checks done in the MayI() and CanI() can be arbitrarily costly depending on the needs of the Lilim writer and can be tuned for different classes of Lilith's or the tool's users.

In the current version of Lilith we chose to implement our own RPC to have complete freedom and knowledge on how the security hooks are put into the call chain. We are investigating using other mechanisms as well.

Lilith has also been designed to reduce the opportunities for rogue Lilim to gain illegal control of Lilith objects. The Lilim code interacts with Lilith through only one object and only has references to this one object. This makes it more difficult for Lilim to get control of other Lilith objects. Additionally, when calls trigger the SecurityManager the call stack is checked at that point. The SecurityManager checks if the classes involved in the call were invoked in the proper order; if a Lilim appears in an inappropriate place in the stack, the call is rejected. Information on such illegal attempts could be used to further restrict a malicious user's privileges.

5 CONCLUSIONS

Lilith is a Java framework whose purpose is to provide a highly scalable, easy distribution of user code across a heterogeneous computing platform. By suitable development of user code, Lilith can be used as the basis of tools for the use and management of distributed systems. Lilith promotes rapid development by handling all the details of code distribution and communication, while the user code only needs to define the tool functionality on a single node, in accordance with a simple API. Lilith's features are capitalized on in tools such as the Lilith Lights network traffic visualization tool.

Flexible security in Lilith is currently under development. Such security is necessary for large clusters which support a number of users with a variety of security needs. This also safeguards against widespread damage possible by malicious use of the quick access that Lilith's scalability provides.

References

[1] Computational Plant see http://z.ca.sandia.gov/cplant,

[2] http://www.beowulf.org.

[3] J. LAROCO, R. ARMSTRONG, AND R. CARTER, Commodity Clusters: Performance Comparisons Between PC's and Workstations", *Proceedings of the 5th International Symposium on High Performance Distributed Computing*, IEEE, 1996.

[4] http://dancer.ca.sandia.gov/Lilith.

[5] http://www.ssh.fi/.

[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.

[7] D. A. EVENSKY, A. C. GENTILE, L. J. CAMP, R. ARMSTRONG, Lilith: Scalable Execution of User Code for Distributed Computing, *Proceedings of the 6th International Symposium on High Performance Distributed Computing*, IEEE, 1997.

[8] G. ALMASI, AND A. GOTTLIEB, *Highly Parallel Computing*, Benjamin/Cummings, 1994.

[9] F. T. LEIGHTON, *Introduction to parallel Algorithms and Architectures: Arrays, Trees, Hypercubes*, Morgan Kaufmann, 1992.

[10] K. HWANG, *Advanced Computer Architecture, Parallelism, Scalability, Programmability*, McGraw-Hill, 1993.

[11] D. A. EVENSKY, A. C. GENTILE, AND P. WYCKOFF, Lilith Lights: A Network Traffic Visualization Tool for High Performance Clusters, *Proceedings of the 7th International Conference on High Performance Computing and Networking - Europe*, Springer-Verlag, 1999, to be published.

Biographies

David Evensky, **Ann Gentile**, **Pete Wyckoff**, and **Rob Armstrong** have been involved with distributed computing and distributed frameworks for many years. This has included the DAISy, and CPlant clusters of commodity computers and the frameworks for high performance scientific computing, POET and Indeps. Sandia National Laboratories have had a long-standing interest and activity in distibuted and parallel computing as the future of scientific computing.