Lilith: A Java Framework for the Development of Scalable Tools for High Performance Distributed Computing Platforms

D.A. Evensky, A. C. Gentile, and R.C. Armstrong

< evensky, gentile, rob>@ca.sandia.gov

Sandia National Laboratories

Livermore CA

Abstract

Increasingly, high performance computing constitutes the use of very large heterogeneous clusters of machines. The use and maintenance of such clusters are subject to complexities of communication between the machines in a time efficient and secure manner. Lilith is a general purpose tool that provides a highly scalable, secure, and easy distribution of user code across a heterogeneous computing platform. By handling the details of code distribution and communication, such a framework allows for the rapid development of tools for the use and management of large distributed systems. Lilith is written in Java, taking advantage of Java's unique features of loading and distributing code dynamically, its platform independence, its thread support, and its provision of graphical components to facilitate easy-to-use resultant tools. We describe the use of Lilith in a tool developed for the maintenance of the large distributed cluster at our institution and present details of the Lilith architecture and user API for the general user development of scalable tools.

1. INTRODUCTION

Lilith is a general purpose tool that provides a highly scalable, 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 clusters increasingly used in high performance computing today. Because users are only minimally responsible for writing their user tool code, with Lilith handling the details of propagation and distribution and ensuring scalability, Lilith is easy to use and Lilith-based tools can be rapidly developed.

Lilith can be used for the creation of tools employed for both the control of user processes on the distributed system as well as for general administrative tasks on the system itself. Although there exist tools to accomplish some of these tasks, they are not scalable or rely on relatively weak security.

Lilith is written in Java, taking advantage of Java's unique features of loading and distributing code dynamically, providing platform independence and support for threads, and allowing easy creation of graphical user interfaces with browser front ends. Java is object-oriented, further facilitating rapid Lilith-based tool development through object reuse.

This body of this paper is divided, conceptually, into two parts: the first part (Section 2) discusses the advantages of the Lilith framework to the use and maintenance of high performance computing platforms; the second part (Section 3) discusses details of the use of Lilith for building scalable tools.

2. APPLICATIONS FOR LILITH IN HIGH PERFORMANCE CLUSTER COMPUTING

In this section, we discuss the advantages of the Lilith framework for the use and maintenance of high performance computing platforms. In Section 2.1 we discuss Lilith's applications in the use and maintenance of a large cluster, discussing the advantages of scalability in a large cluster. In Section 2.2, we present the particular of Lilith as a framework upon which we have built a tool for obtaining information about the status of Sandia's Computational Plant.

2.1 Applications and Scalability

Lilith's[1] principle task is to span a tree of machines executing user-defined code. The tree structure is chosen to provide logarithmic scaling[2]. Beginning from a single object, Lilith recursively links host objects, LilithHosts, on adjacent machines until the entire tree is occupied. The LilithHosts propagate code objects, called Lilim[3], down the tree. The Lilim then perform user-designated functions on every machine. Finally, through calls to the LilithHosts, the results of the user code are propagated back up the tree. Thus, the user code undergoes a three-phase process: distribution, execution, and result collection. (Figure 1).

This scalable distribution of code makes Lilith ideal for the basis for the development of tools for controlling user processes and general system administrative tasks to be used in the management of large distributed systems. For example, such tools could allow a user to first query the system to find the status of the nodes in a system, use that information to determine which subset of nodes on which to run his code, and finally handle the distribution of his code to those nodes. Likewise, a system administrator can use such tools to monitor system status or create user accounts in a scalable fashion. Although there exist tools to accomplish some of these tasks, they are not scalable or rely on relatively weak security. (A comparison with such tools as NASA's LAMS package[4], Platform Computing's LSF[5], and the Ptools[6]project has been discussed in detail in Refs. 7 and 8). The scalable nature of Lilith-based tools in these tasks vastly improves the time serial queries and code distribution would take: a one second operation performed on a thousand nodes would take 16 minutes under a serial distribution, as opposed to 10 seconds via a binary tree distribution.

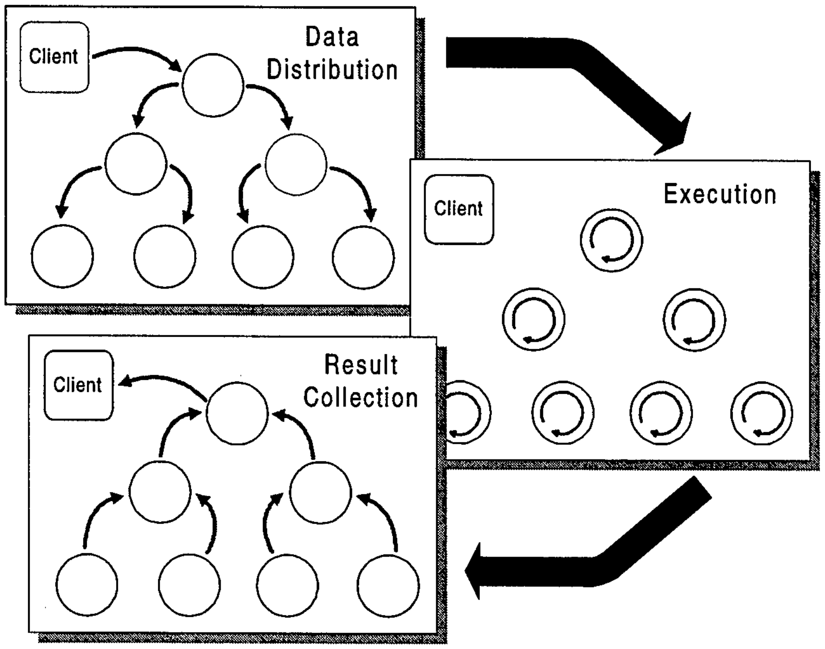

Figure 1. Three phase process for user code, Lilim, under Lilith: distribution, execution, and result collection. Lilith handles the details of Lilim distribution and communication amongst LilithHosts. The tree structure is used for scalability

2.2 Lilith-based System Status Tool Example Applied to MP Distributed Computing

In the past computers were expensive, hard to build, and rare. Today standard hardware specifications allow computers to be built with interchangeable parts. Only the components that really make a difference in computational simulations need to be one-of-a-kind and non-commodity. The effect of these developments is that massively parallel computers can be assembled out of commodity parts to perform the largest simulations. Special-purpose hardware for high-performance computing is necessarily more expensive and takes longer to implement. Conversely, the standards-based commodity computers are well suited for rapid deployment of new technologies. If, for example, a faster processor is developed, it can be quickly integrated into the existing commodity architecture reducing costs and time-to-market. In addition, interchangeable parts provide the potential for reuse of components from previous generations of hardware.

At SC97[9], Sandia National Laboratories and Lawrence Livermore National Laboratory staged a distributed cluster parallel computer using high-speed networks to connect machines at the SC97 show floor, Sandia California, and Sandia New Mexico. Sandia's Computational Plant distributed cluster (*CPlant* for short) was sited at these three geographical locations linked with an ATM network. That network used the LLNL-operated National Transparent Optical Network (NTON), capable of 10+ giga-bit/second capacity. (Figure 2). The aggregate compute power of this system was more than 100 Gigaflops comprised of DEC Alpha systems running Linux. This was done to demonstrate both the viability of commodity components for supercomputing applications and WAN connectivity uniting distributed-parallel resources to run a problem of immense proportions.

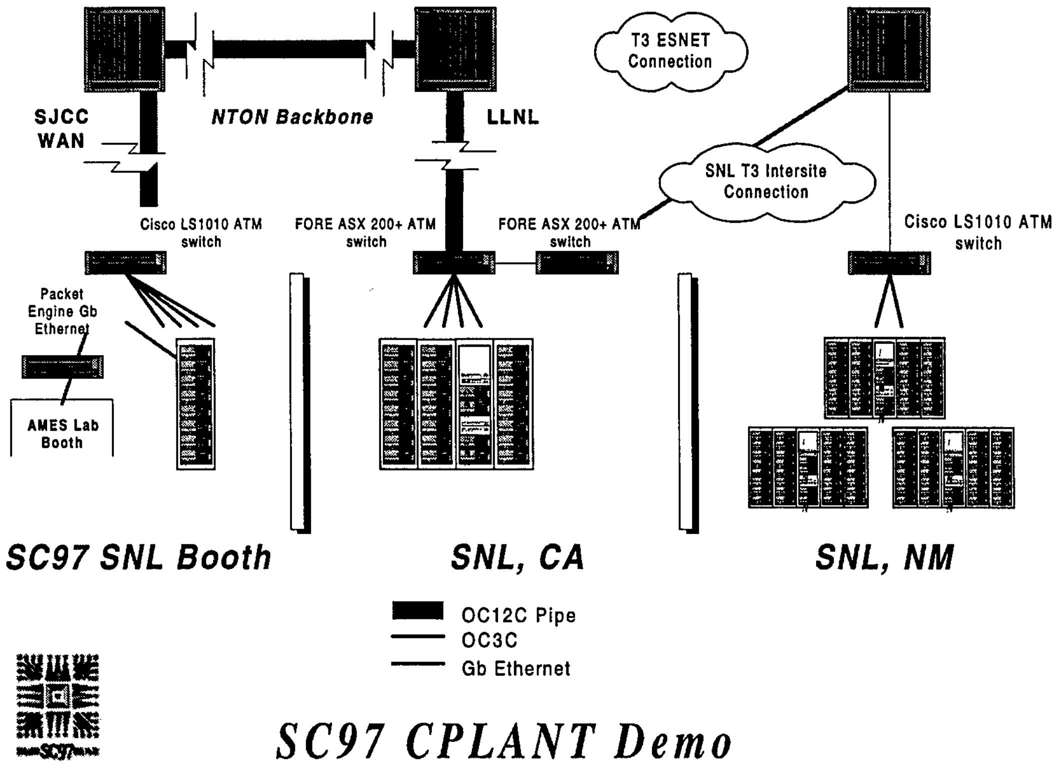

Figure 2. SC97 CPlant Demonstration Distributed Cluster. The three sites shown are the San Jose Convention Center (SJCC) in San Jose, CA., SNL/CA in Livermore, CA., and SNL/NM in Albuquerque, NM.

While such systems can be built, monitoring and maintaining such a system has difficulties which do not encumber more conventional systems. Even the simple task of obtaining system load, memory usage, etc. for each component of the machine can require some thought. This sort of system status monitoring is required by parallel applications and essential for a remotely hosted system because normal status indicators are absent. This information is necessary to system administrators to monitor system performance, and is needed by users to check the status of their jobs and to select the placement of those jobs in a clustered environment. Here Lilith was not just a convenience, but vital. Normal tools like shell scripts that rely on serial algorithms will not finish in a meaningful period of time. In addition, system architects need to alter the system monitoring kernel of the code frequently because of a rapidly evolving system configuration. Lilith is ideal for this situation because the infrastructure needed to scalably and securely span the machine is written and debugged once. The much smaller code, that actually performs the system monitoring function, can be altered independently. Lilith takes care of getting the code and data to the machine in a scalable way, executing it on every configured node, and scalably returning data to the originator.

The application has been tailored to perform queries only on demand, thus avoiding adverse effects on system load. For this application we take advantage of Java's unique features of loading and sending code dynamically, its platform independence, and its graphical components to produce an informative, dynamically updated display.

The user code consists of a thread that waits for a flag from its parent describing the type of data to be returned (a delta or absolute display). The flag is passed to the children, and several system files in /proc are read (this includes downloaded load-able kernel modules that expose kernel internals via the /proc interface). The values are read, processed, and merged with values returned by other processors. All the user code has to do is specify the code for processing the values; Lilith handles the details of code distribution and result collection.

The application provides statistics on CPU utilization, memory utilization, Ethernet packet, and error statistics. For the systems that have an ATM interface, we provide values for the AAL5 traffic. These files are read in such a way to minimize the system overhead (file open/close on an NFS mounted root file system).

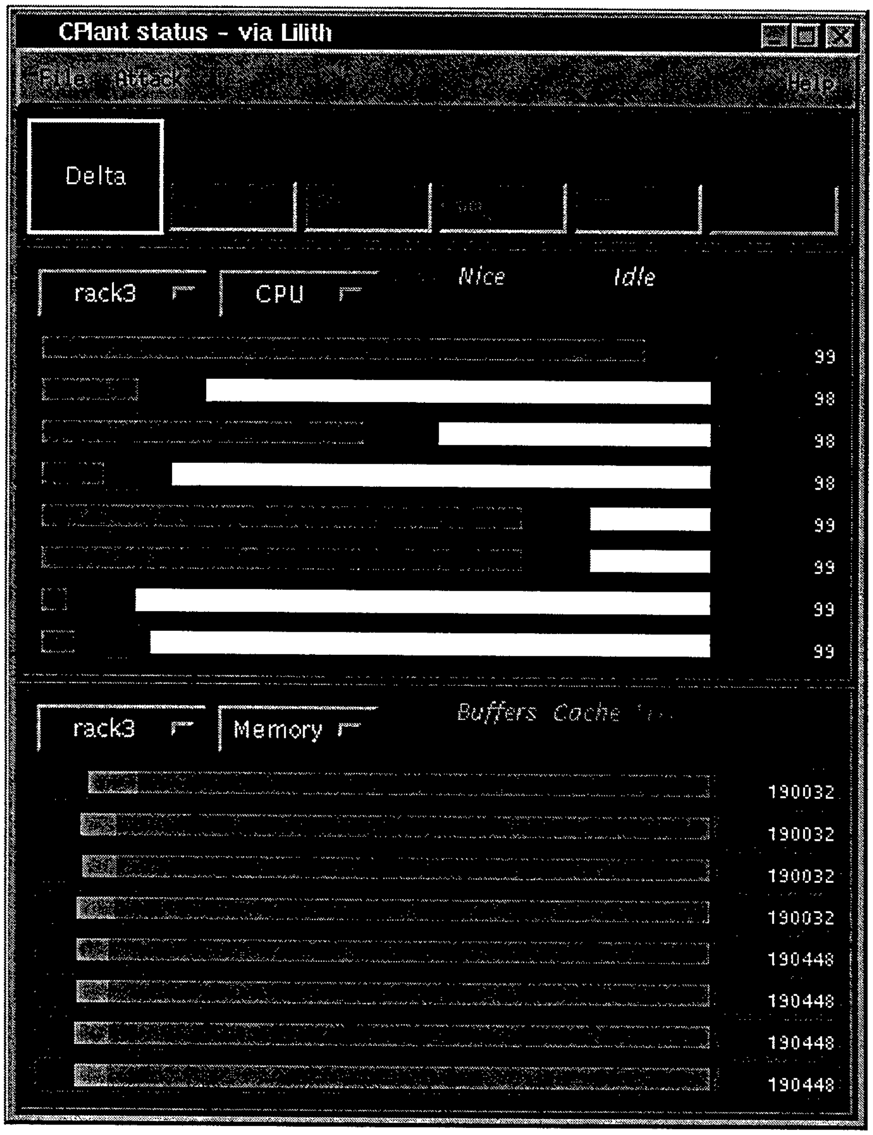

Figure 3. Snapshot of the display for the Cplant status information tool. Values are updated dynamically by the periodic updates from the Lilim.

A snapshot of the graphical display appears in Figure 3. Users can select which rack of machines and which quantities to view. The set-up in Figure 3 is limited to two quantities displayed simultaneously; this number is a user-specified parameter at start-up and, in practice, is set higher. Results are displayed as color-coded bars sized proportionally to their value, similar to the display provided by gr_osview[10]. Total scales for the display values can be entered by the user and updated dynamically.

3. BUILDING SCALABLE TOOLS WITH LILITH

In this section, we discuss details of the use of Lilith for scalable tool building, focusing on those elements that the user needs to interact with to construct his own tools. In Section 3.1 we describe those objects in the Lilith Distributed Object System with which the user interacts. In Section 3.2 we review the Message Object by which communications are handled. Section 3.3 covers the methods LICO provides by which the Lilim communicate with other Lilim, illustrating calls in the user code in a simple example. Then Section 3.4 covers Lilim distribution down the tree.

3.1 User-interactive Objects in the Lilith Distributed Object System

The purpose of Lilith is to span user code to large numbers of machines scaleable, not to invent a new distributed object system. Currently, we implement a simple distributed object system, based on the Legion[11] object system, written in Java. The Lilith Distributed Object System was presented in detail in Ref. 7. Here, then, we present only those objects with which the user has to be directly concerned.

Lilim are the Lilith objects that carry user code within them. Lilim, as well as the core Lilith objects, export well known interfaces which allow the user code to interact with the Lilith environment. The Lilim run as threads under the LilithHosts. Lilith-based tools are created by construction of suitable Lilim. The Lilim need only be written with the goal of the tool itself in mind; details of code distribution and communication between nodes in the tree are handled by objects in the Lilith environment.

The LilithHost object is responsible for protecting the computing resource on which it is running from other Lilith objects and for instantiating Lilith objects on that host. For purposes here, we define a host to be a system running under a single OS. If Lilith is being used simply as a platform for remote objects there may be more than one LilithHost per host. LilithHosts maintain the tree and, via lower level objects, communicate with one another. Details of the host to host communications and the building of the tree are not germane to this discussion and are discussed in Refs. 7 and 8 respectively; details of instantiating the Lilim are given in Section 3.4.

Lilim interact with the Host Object and each other through the Lilim-Implementation-CommunicationsObject, a.k.a. LICO. Thus each node of the tree consists of a LilithHost with a Lilim running on it, and a LICO used for communications between the two. The LICO provide a well defined set of methods by which Lilim can send data up and down to other Lilim in the tree. LICO pass arguments to the Lilim and gather up their return messages for passage back up the tree. These methods are discussed in detail in Section 3.3. By compartmentalizing the interactions of Lilim with Lilith in this way, not only is the entry-level knowledge needed to use Lilith small but also access to Lilith Objects by a potentially malicious user is contained. For instance, rogue Lilim cannot by *direct* call get illegal control of the lower level objects in the system which provide socket access. This restriction is enforced though Java Package assignments and checks on the sequence of classes in the execution stack whenever the SecurityManager is invoked.

3.2 Message Objects

Communications are handled through the sending of Message Objects, MOs. The MO is a general-purpose data rack that can hold a queue of data objects and provide for marshaling and unmarshaling the data to/from a byte array. Data is placed into and removed from the MO through a well-defined set of calls pertaining to the primitive data types such as push/pull/peekInt()[12], push/pull/peekString. Push places an object into the MO, pull removes it from the MO, and peek returns the value without removing it from the MO. There is also hash table interface which allows the users to assign a label to each item. This interface can be used to provide random access to internal MO data structures. The methods are hashedPut(), hashedGet(), and hashedRemove() and they support the Java wrapped types corresponding the primitive types supported by MO. The total set of user-related calls for assembling/disassembling the MO is covered in detail in Ref. 8.

MOs also carry with them an id field, MOUUID, used for identification of the MO and for matching sends and returns. The MOUUID is in the form of a user-defined String. MOUUIDs are set/returned using set/getMOUUID or can be set by specifying the MOUUID in the MO's constructor. Use of the MOUUIDs is described in more detail in the next sub-section.

3.3 LICO API and Example Lilim

The LICO provides a simple set of methods through which Lilim communicate with each other up and down the tree. The LICO API is given in Table 1. In this sub-section, we discuss these methods, presenting their use in a structurally typical example.

LICO METHOD	ACTION
MO get(String)	returns an MO from LICO with MOUUID equal to the String value
void put(MO)	puts MO into LICO
void scatterToChildren(MO[])	scatters MO[] to children
MO[] gatherFromChildren(String)	returns MO's from children corresponding to MOUUID equal to the String value
MO getArg(String)	returns initial MO sent down with Lilim with MOUUID equal to the String value

Table 1: LICO User API. Methods in the LICO called by the Lilim.

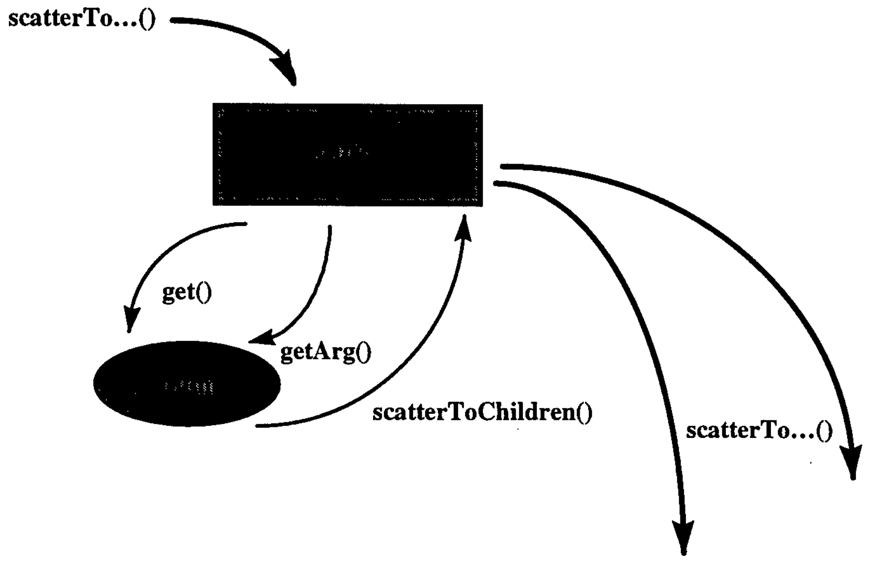

Figure 4: Methods involved in sending data down the tree.

The methods get() and scatterToChildren() are used as a pair to get messages from the parent Lilim and send messages down to the child Lilim[13]. (These methods are illustrated in Figure 4.) A Lilim sends a message to its children via LICO.scatterToChildren() using a tagged MO as its argument. This method puts each MO into the LICO corresponding to each child. A Lilim gets an MO from its LICO via LICO.get()[14] using the appropriate MOUUID tag as the argument. Messages can then be sent *recursively* down the tree by each Lilim first calling LICO.get(myTAG) and then calling scatterToChildren(myMO) where myMO has been tagged with the same tag. (This will be illustrated in the example later in this sub-section.)

Processing results up the tree is also performed recursively. In this case the methods put() and gatherFromChildren() are used. These methods are illustrated in Figure 5. A Lilim calls gatherFromChildren() with an MO tag as its argument to receive an MO array containing all such tagged results from its children's LICOs. This call blocks execution in the Lilim until returns from all the children have come into the parent Lilim's LICO. That same Lilim then makes its own results available to its own parent by calling LICO.put() with an argument of its own identically tagged MO.

Note that a child Lilim calls LICO.put() in anticipation of the parent calling gatherFromChildren() to collect that MO. Thus both the processes of sending messages down the tree and of gathering returns back up the tree are initiated by the parent. Communications both down and up the tree are only with the levels in the tree directly above or below the current level.

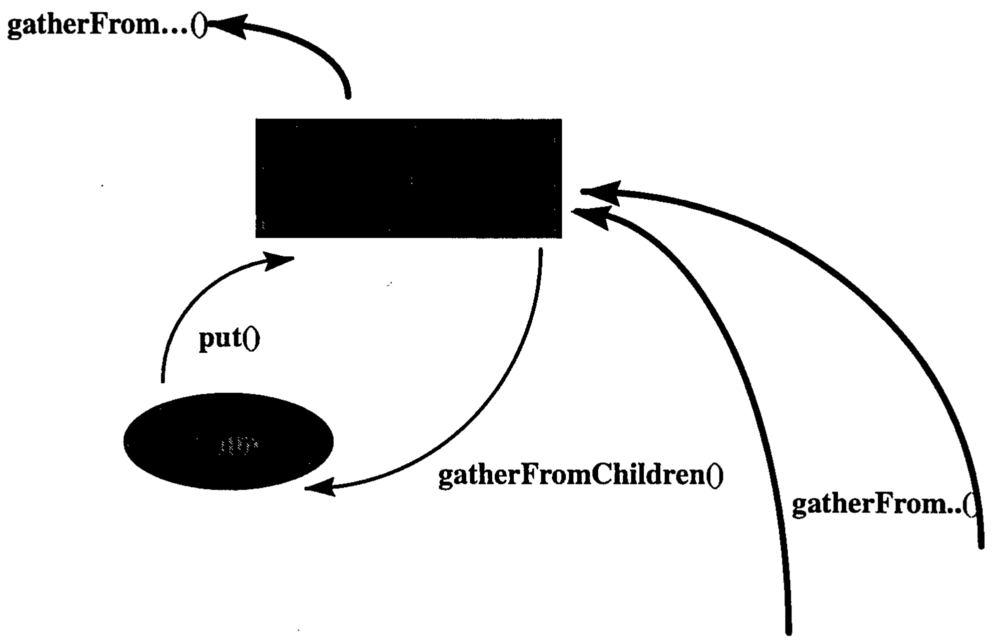

Figure 5: Methods involved in returning data back up the tree.

The recursive up and down calls and tagging are illustrated in an example using a distributed sort. This usage capitalizes on the fact that it is faster to sort a set of presorted sets than to sort an entire unsorted list from scratch. Downward processing consists of each Lilim getting a list of numbers to sort, subdividing that list into pieces for itself and its children to sort, and sending those resultant pieces down to its children. Once a Lilim has sorted its own piece, it then processes the results back up the tree by gathering the children's sorted list, combining their results with its own via a merge sort, and finally passing the combined sorted list up to its parent. The relevant pseudo-code is as follows:

```
public Class Lsorter implements Lilim{
  private LICO myLICO;   // field for LICO with which this Lilim
                                 communicates
      ...
  public void run(){
     MO tmpMO = myLICO.get(TAG1); /* gets an MO from the LICO
                                containing the list of numbers to be sorted */

     ... /* Code here which:
                  1) Unpacks MO to get array of numbers to sort via calls to
                     tmpMO.pullInt();
                  2) Divides array into subarrays for self and children.
                  3) Packs arrays for children into MO[]kids_piecesMO via calls
                     to kids_piecesMO[i].pushInt(int)
                  4) Set MOUUIDs on each MO to TAG1 via
                     kids_piecesMO[i].setMOUUID(TAG1)
        */

     myLICO.scatterTOChildren(kids_piecesMO); /* Scatters MO's
                                        with arrays for children to sort */
     sort(myArray); /* sort own piece using own sort method */
     kids_piecesMO = myLICO.gatherFromChildren(TAG2); /*
                            gather MO's containing sorted arrays from children */

     ... /* Code goes here which unpacks sorted arrays from kids_piecesMO
         and places them into kidsArrays*/

     finalArray = mergeSort(myarray,kidsArrays); /* merge sort
                                                all sorted arrays */

     .../* Code here which:
                     1) Packs final array into tmpMO
                     2) Sets MOUUID tag on tmpMO to TAG2
        */
     myLICO.put(tmpMO); /* Put MO containing final sorted array into LICO
                                for parent to gather */
  }
}
```

In this example, the packing and unpacking of messages is not explicitly shown - these are straightforward calls to push/pullInt() and setMOUUID(). The key thing to note is the usage of the tags in the operation of the recursive calls. TAG1 is used to obtain the correct

MO from the parent via get(), and is therefore also used to make the MO sent from the parent in scatterToChildren(); thus `TAG1` is used for the signaling in sending the messages down the tree. Similarly, `TAG2` is used in put() and gatherFromChildren() on the sending returns back up the tree.

Many tools can be written using this basic structure. In the most general case, the code section handling the sort can be replaced with code to execute a shell script that performs some action on each node. The sequence of calls to scatter information down the tree and gather it back up, as well as the tagging, can be reused unchanged. Only the packing and unpacking of the messages will have to be tailored to reflect the specific types involved.

3.4 Lilim Distribution

Before the Lilim can be distributed down the tree of machines, the LilithHosts must be started and the tree of Hosts must be established. However, the tree, once built, can support multiple sequential or concurrent Lilim; in many cases, then, users will not be responsible for startup, but will merely be taking advantage of preexisting trees. Therefore, we discuss here sending and starting Lilim on a preexisting tree. Details for complete startup can be found in Ref. 8.

The Lilim is a Java class file with a class that implements a Lilim interface in the package lilim. These Lilim objects are loaded using a Lilim specific ClassLoader object. The client reads this class into a byte array when it is executed, and sends it as an argument of LilithHost.sendLilim(). The client can also send initial data that the Lilim can read with getArg() by specifying a tagged MO as the second argument to the call:

```
    byte[] mylilim = new byte[LILIM_LENGTH];
```

```
// read class file, named  "dostuff.class" into mylilim
MO initial_data = new MO("initdata4567");
initial_data.pushInt(30);
root.sendLilim("dostuff",mylilim,initial_data);
```

The user then starts the Lilim on the hosts by:

```
root.runLilim("dostuff");
```

The client can send down additional data to the Lilim and get results from the Lilim using LilithHost.scatterToChild(MO), and MO LilithHost.gatherFromChild(String). On the server side, these methods simply call LICO.put(MO) and LICO.get(String) respectively on the LICO corresponding to that host. It is no coincidence that these methods are reminiscent of the communications between parent and child Lilim discussed in the previous sub-section: LICO.scatterToChildren() and LICO.gatherFromChildren() call these Host methods as part of their functionality. From the child Lilim's perspective, calls initiated by the client or by the parent Lilim appear the same - the Lilim has only to interact with the LICO via LICO.get(String) and LICO.put(MO) to get the scattered information or to return information irrespective of the initiator of the caller. Tagging the MO's for LilithHost's scatter and gather calls thus proceeds identically to the tagging for the LICO's scatter and gather calls. For example:

```
MO additional_data = new MO("more data 042376");
additional_data.hashedPut("first int",
     new Integer(1021));
root.scatterToChild(additional_data);
```

This will then be picked up by the child calling:

```
MO data2 = myLICO.get("more data 042376");
int ifirst = ((Integer)(data2.hashedGet(
     "first int"))).intValue();
```

4. CONCLUSIONS

Lilith is a general purpose framework that provides a highly scalable, easy distribution of user code across large, heterogeneous, computing platforms. Lilith can be used for the creation of tools employed for both the control of user processes on the distributed system as well as for general administrative tasks on the system itself.

Lilith is written in Java, taking advantage of Java's unique features of loading and distributing code dynamically, its platform independence, its thread support, and its provision of graphical components to facilitate easy-to-use resultant tools.

We have shown the use of Lilith in a tool designed for monitoring status information for Sandia's Computational Plant distributed cluster. Without the scalability inherent in Lilith, querying and obtaining status information on such a system would be time prohibitive; with Lilith, dynamic updates can be done in a timely manner.

We have presented the user API, by which users can create their own Lilith-based tools. Because users are only minimally responsible for writing their user tool code, with Lilith handling the details of propagation and distribution and ensuring scalability, Lilith is easy to use and Lilith-based tools can be rapidly developed.

The use of Lilith as a framework for tool building enhances the ability of users and administrators to function when using large clusters, as is increasingly common in high performance computing today.

Work supported by the U. S. Department of Energy under contract DE-AC04-94AL85000.

[1] Lilith is the mythological "Mother of D(a)emons".

[2] Scaling for a binary tree is $\log_2(N)$ where N is the number of nodes. Timing results demonstrating Lilith's logarithmic scaling can be found in Refs. 7 and 8.

[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] Login Account Management System (LAMS), Final Report on Subcontract CSC/ATD-WR-MV-NAS2-9005, Matt Bishop, National Aeronautics and Space Administration, Ames Research Center, Moffett Field, CA 94035-1000.

[5] Load Sharing Facility (LSF), see http://www.platform.com

[6] The Parallel Tools Consortium (Ptools), see http://www.ptools.org.

[7] D. A. Evensky, A. C. Gentile, L. J. Camp, and R. A. Armstrong, "Lilith: Scalable Execution of User Code for Distributed Computing", Proceedings of the 6th IEEE International Symposium on High Performance Distributed Computing, Portland, OR, August 1997.

[8] A. C. Gentile, D. A. Evensky, and R. C. Armstrong, "Lilith: A Software Framework for the Rapid Development of Scalable Tools for Distributed Computing", 18th International Conference on Distributed Computing Systems, Amsterdam, The Netherlands, 1998, submitted.

[9] SC97: High Performance Networking and Computing, San Jose, CA, November 1997.

[10] gr-osview is a component of SGI's IRIX, see http://www.sgi.com.

[11] 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.

[12] Method calls will be given as "functionname()" with the arguments only specified when germane to the discussion.

[13] Although the terms "parent" and "child" are more accurately used in terms of the LilithHosts which maintain the tree, the extension of this terminology to the Lilim residing on those hosts is straightforward and unambiguous.

[14] The call getArg() is similar to that of get() and is used by a Lilim to get data, in the form of an MO, initially sent down with the Lilim bytecode. Sending this information down with the Lilim reduces the number of messages required.

M98052521

Report Number (14)	SAND--98-8448C
	CONF-980223--
Publ. Date (11)	19980319
Sponsor Code (18)	DOE/ER, XF
UC Category (19)	UC-405, DOE/ER

DOE