Lilith: A Software Framework for the Rapid Development of Scalable Tools for Distributed Computing

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

<gentile, evensky, rob>@ca.sandia.gov
Sandia National Laboratories
Livermore CA

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. The scalable performance provided by Lilith is crucial to the development of effective tools for large distributed systems. Furthermore, since Lilith handles the details of code distribution and communication, the user code need focus primarily on the tool functionality, thus, greatly decreasing the time required for tool development. In this paper, we concentrate on the use of the Lilith framework to develop scalable tools. We review the functionality of Lilith and introduce a typical tool capitalizing on the features of the framework. We present new Objects directly involved with tool creation. We explain details of development and illustrate with an example. We present timing results demonstrating scalability.

1. Introduction

Lilith is a general purpose framework that provides a highly scalable, easy distribution of user code across a heterogeneous computing platform. This capability is of value in the

DISTRIBUTION OF THIS DOCUMENT IS UNLIMITED

MASTER
DISCLAIMER

This report was prepared as an account of work sponsored by an agency of the United States Government. Neither the United States Government nor any agency thereof, nor any of their employees, makes any warranty, express or implied, or assumes any legal liability or responsibility for the accuracy, completeness, or usefulness of any information, apparatus, product, or process disclosed, or represents that its use would not infringe privately owned rights. Reference herein to any specific commercial product, process, or service by trade name, trademark, manufacturer, or otherwise does not necessarily constitute or imply its endorsement, recommendation, or favoring by the United States Government or any agency thereof. The views and opinions of authors expressed herein do not necessarily state or reflect those of the United States Government or any agency thereof.
development of tools to be employed in the use and administration of very large
(thousands of nodes) clusters. The scalability provided by Lilith promotes effective
performance times. Furthermore, since Lilith handles the details of code distribution and
communication, the user code need focus primarily on the tool functionality, greatly
decreasing the time required for tool development.

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 and frequently rely on relatively weak security.

Lilith is written in Java, providing platform independence 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.

In a previous work, we presented the core Objects in the Lilith framework
responsible for low level communications internal to the framework. In this paper, we
concentrate on the development of tools using the Lilith framework. We describe Lilith’s
basic functionality in Section 2. In Section 3 we introduce a typical tool capitalizing on
Lilith’s features. In Section 4 we present new Objects in the framework directly involved
with tool creation. In Section 5 we discuss details of tool development, presenting a
simple example illustrating calls in the user code. In Section 6 we discuss the start-up of
the Lilith environment and the distribution of the user code. In Section 7 we present
results demonstrating Lilith’s scalability. We conclude in Section 8.
2. Lilith Functionality

Lilith’s principle task is to span a tree of machines executing user-defined code. 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. The tree structure in Lilith was chosen to provide scalability\(^4\). 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\(^5\), down the tree. Tools are produced by suitable development of Lilim. The Lilim execute, performing user-designated functions on every machine. Finally, Lilith propagates the results of the user code back up the tree. Lilim can do processing on the results as they are passed back up the tree. Overall, the user code undergoes a three-phase process: distribution, execution, and result collection.(Figure 1).

![Diagram](image)

**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.
3. System Status Monitoring Tool

Figure 2. Snapshot of the display for the system status monitoring tool. Values are updated dynamically by the periodic updates from the Lilim.

Features of Lilith are capitalized on in a tool used for monitoring the status of a rely distributed system. Scalability is essential since normal tools like shell scripts using serial
algorithms would not finish in a meaningful time. Furthermore, the Lilith code that handles the distribution and communication is separate from the user’s tool code that actually performs the system monitoring functionality, minimizing the overall changes that would be necessary to adapt to a changing system configuration.

A prototype of such a tool is in use on Sandia’s computational plant. The Lilim code on each node periodically reads several system files in /proc. It then processes these values and merges them with the values returned by other processors as the results are passed back up the tree.

The display is shown in Figure 2. The application provides statistics on CPU utilization, memory utilization, Ethernet packet, and error statistics. For systems that have an ATM interface, it provides values for the AAL5 traffic. Users can select which machines and quantities to view. Results are displayed as color-coded bars sized proportionally to their value, similar to the display provided by gr_osview. Total scales for the display values can be entered by the user and updated dynamically.

4. User-interactive Objects in the Lilith Framework

In this section, we present new Objects in the Lilith framework with which the user interacts in order to build tools. These are: the Lilim which encompasses the user’s tool code, the LilithHost objects which maintain the tree and (indirectly) provide the basic functionality of Lilith, the object by which the Lilim communicates with those host objects, and finally the messaging object used in communications.
Lilim are Objects that carry user code within them. The Lilim run as threads or autonomous processes under the LilithHosts. Lilith-based tools are created by construction of suitable Lilim. The Lilim need primarily 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 framework.

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. LilithHosts maintain the tree and, via lower level objects, communicate with one another.

Lilim interact with the Host Object and each other through the Lilim-Implementation-CommunicationsObject, a.k.a. LICO. Thus each node of the tree supports at least three Objects: 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. By compartmentalizing the interactions of Lilim with the Lilith framework 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 network 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.

Communications are handled through the sending of Message Objects, MOs. The MO is a general-purpose data rack that can hold a list of data objects. It is capable of
marshaling data to and from a byte stream and recreating the data in a new MO. 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(), push/pull/peekString(), as well as push/pull/peekMO(). 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 can be used to provide random access to internal MO data structures. The methods are hashedPut/Get/Remove(); they support the Java wrapped types corresponding the primitive types supported by MO.

MOs also carry with them a tag used for identification of the MO and for matching sends and returns. The tag is in the form of a user-defined String. It can be set in the MO’s constructor or by methods provided by MO.

5. Tool Development: LICO API and Example Lilim

The LICO provides a number of methods for distributing data down the tree and for processing results up the tree. In this section, we discuss these methods, and present an example Lilim exhibiting a basic structure common to many tools.

MO tags are used for matching up messages sent from parent Lilim to child Lilim and vice versa. (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.)

To send messages down the tree, the methods LICO.get() and LICO.scatterToChildren() are used in tandem. These methods are illustrated in
Figure 3. 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 then gets an MO from its LICO via LICO.get() using the appropriate tag as the argument. Messages are thus sent recursively down the tree by each Lilim first calling get(myTAG) and then calling scatterToChildren(myMO) where myMO has been tagged with the same tag, myTAG. Communications both down and up the tree are thus only with the levels in the tree directly above or below the current level.

![Diagram](image)

**Figure 3: Methods involved in sending data down the tree.**

Processing results up the tree also are performed recursively. In this case the methods LICO.put() and LICO.gatherFromChildren() are used in tandem. These methods are illustrated in Figure 4. A Lilim calls LICO.gatherFromChildren()
with a tagged MO as its argument to receive an MO array containing all MO results from its children. This method gathers tagged MOs from each of the children’s LICOs. The call `gatherFromChildren()` blocks execution in the Lilim until returns from all the children have come into that Lilim’s LICO. The 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 `put()` in anticipation of the parent calling `gatherFromChildren()` to collect that MO. Thus the processes of sending messages down the tree and of gathering returns back up the tree are both initiated by the parent.

![Diagram](image)

**Figure 4:** Methods involved in returning data back up the tree.
The recursive up and down calls and tagging are illustrated in an example implementing a distributed sort. The downward processing will consist of: each Lilim getting a list of numbers to sort, subdividing that list into pieces for itself and its children to sort, and then sending those pieces down to its children. After 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 then passing the combined sorted list up to its parent. The pseudo-code is as follows:

```java
public class Lsorter implements Lilim {
    private LICO myLICO; /* field for LICO with which this Lilim communicates. This reference is assigned by the LilithHost */

    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. */
        /* 2) Divides array into subarrays for self and children. */
        /* 3) Packs arrays for children into MO[] kids_piecesMO. */
        /* 4) Sets tags on each member of kids_piecesMO[] to TAG1. */

        myLICO.scatterToChildren(kids_piecesMO); /* Scatters MO's with arrays for children to sort */

        /* Code here which sorts own piece using own sort method */

        kids_piecesMO = myLICO.gatherFromChildren(TAG2); /* gather MO's containing sorted arrays from children */
    }
}
```
.../* Code here which
1) Unpacks sorted arrays from kids_piecesMO.
2) Merge sorts those arrays and own array.
3) Packs final array into tmpMO.
4) Sets MOUUID tag on tmpMO to TAG2.
*/

myLICO.put(tmpMO); /* Put MO containing final sorted array into LICO for parent to gather */
}
}

In the above example, the packing and unpacking of messages are straightforward calls to push/pullInt(). 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.

6. Lilith Client Operation

In this section we discuss the start-up of the LilithHosts, the building of the tree of Hosts, and the distribution of Lilim down the tree. There is an automatic start-up
mechanism in which initially only the root LilithHost is manually started and the other Hosts participating in the tree are started as the tree is built. The user creates a list of hosts, their ports, and protocols (UDP or TCP) that will participate in the tree. The running LilithHost takes the list, starts its immediate children, and passes them the list of their descendants. This process is then repeated, building the entire tree in a scalable fashion. This process is implemented in the root’s LilithHost.buildtree() method.

The user then creates and sends the Lilim down the tree. 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(). Finally, the user starts the Lilim on the hosts by calling the root’s LilithHost.runLilim() method.

Trees are not necessarily intended to be long-lived; users can build a tree for each application run. Alternatively, Objects in the Lilith framework are designed so that nodes in the tree can support multiple Lilim, allowing many users to take advantage of the same, or pre-existing trees.

7. Scaling Behavior

To demonstrate the scaling behavior of Lilith we consider the scaling behavior of Lilith in two cases: constant work per processor (increasing total work) and constant total work. In both these cases, we first establish the tree. Then, the wall clock time is measured in the client, and the Lilim is sent to the tree. After the root node returns to the client, the wall
clock time is again measured and the results are tabulated for varying number of nodes in the tree. At each node the Lilim is first sent to its children, the Lilim is then instantiated and executed locally, and finally the results from the children are collected. Timings were generated on a 32 processor SGI Origin 2000 and only one server was run per processor.

Figure 5: Scaling behavior for constant work per processor, i.e. increasing total work. Larger plot is 100 prime numbers generated per processor; inset is 500. Scaling is overall logarithmic. Steps occur due to increasing tree depth such that additional time is required for communications. Steps are still seen for increasing work/communications time, though curve flattens out.

Figure 5 shows the scaling behavior for constant work per processor, i.e., increasing total work. The outer plot is 100 random prime numbers generated per processor; inset is 500. This calculation was chosen since each node could be assigned the same seed,
guaranteeing the same amount of work per node. We observe the expected overall logarithmic scaling. The step-like behavior occurs due to increasing the tree depth through addition of nodes such that additional time is required for communications. The steps are still seen for the case of greater work/processor, although curve flattens out. As the ratio of work to communications time increases, the work overwhelms the communication time causing all nodes to essentially run in parallel. It is no surprise that the greatest benefit of the logarithmic scaling thus comes in cases where the communications/work ratio is kept small.

![Graph](image)

Figure 6: Scaling behavior for constant total work (a fixed total number of random numbers calculated), i.e., decreasing work per processor. Overall time for the calculation decreases with logarithmic scaling.
Figure 6 shows the scaling behavior for constant total work, i.e., decreasing work per processor. This plot shows the timings for calculation of a fixed total number of random numbers. In this case, the total number of primes generated per Host decreases as the number of participating Hosts increases. As expected, the overall time for the calculation decreases and logarithmic scaling is again observed.

8. Conclusions

Lilith's 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 developed for the use and management of distributed systems.

This scalability provided by Lilith enhances the ability of users and administrators to function when using a very large cluster. Timings obtained by Lilith exhibit logarithmic behavior. Tools can be developed rapidly under Lilith, since Lilith frees the tool writer from the details of code distribution and communication. Scalability and rapid development are essential in the development of tools such as the system status monitoring tool presented in this paper.

New Objects in Lilith provide a simple user interface for tool development. Call were illustrated in an example exhibiting a typical basic structure for tools.

Work supported by the U. S. Department of Energy under contract DE-AC04-94AL85000.
Lilith is the mythological “Mother of D(a)emons.

A comparison of Lilith with NASA’s LAMS package, Platform Computing’s LSF, and the Ptools project can be found in Ref. 3.


Scaling for a binary tree is $\log_2(N)$ where $N$ is the number of nodes.

Lilim (the children of Lilith) will be used for both the plural and singular name of the Objects that are sent down the tree.

This includes download-able kernel modules that expose kernel interfaces via the /proc interface.

g_r-osview is a component of SGI’s IRIX, see http://www.sgi.com.

Multiple independent Lilim can be supported simultaneously on each node.
SAND-98-8874C
CONF. 980719...