An Interoperability Approach to System Software, Tools, and Libraries for Clusters

Ewing Lusk, Narayan Desai, Rick Bradshaw, Andrew Lusk, and Ralph Butler
Mathematics and Computer Science Division
Argonne National Laboratory

Abstract

Systems software for clusters typically derives from a multiplicity of sources: the kernel itself, software associated with a particular distribution, site-specific purchased or open-source software, and assorted home-grown tools and procedures that attempt glue everything together to meet the needs of a particular cluster. Whether a cluster is a general-purpose resource serving multiple users or dedicated to a single application, getting everything to work together is a challenge. The challenge is partially met by special software distributions for clusters such as OSCAR or ROCKS. Here we discuss another approach (although it is not inconsistent with existing distributions), in which a small number of concepts are deployed to facilitate the customized integration of various software tools for cluster management, operation, and user jobs. The concepts include (1) a component approach to basic system software such as schedulers, queue managers, process managers, and monitors; (2) a software development kit for constructing networks of system software components, either from scratch or by wrapping "foreign" software, and (3) the use of explicit parallelism in building system tools for high performance. We illustrate this approach with a description of a mid-sized general-purpose cluster operated entirely by software built this way.

1 Introduction

This paper describes an experimental approach to systems software. The approach is that of software *components*, an approach used often in application development but not heretofore in the area of systems software. Our experience has been in architecting, implementing, and deploying such software on clusterlike parallel computing systems, but the approach appears to be generally applicable, and our plans include using it on very large scale systems such as IBM's Blue Gene systems. We describe the basic ideas, a general framework for systems software development in this environment, and our experience with the approach.

In Section 2 we define what we mean by systems software and characterize the current common system software environment, particularly for clusters. In Section 3 we discuss the component approach in general and a multi-institution project whose goal is to apply this approach to systems software. Section 4 describes a layered architecture for systems software that fosters component development. Section 5 covers other components and particularly the advantages of using explicit, scalable parallelism in systems software. Section 6 describes our experiences deploying this framework on a mid-sized cluster used for both production and research, and Section 7 outlines the potential for future development of these ideas.

2 Systems Software for Clusters

By "cluster systems software" we mean the collection of programs, invoked by both systems administrators and users, for configuring and maintaining individual nodes, together with software involved in submission, scheduling, management, monitoring, and termination of parallel user jobs. We don't include single-node, cluster-agnostic software such as compilers or single-node operating systems, although exceptions occur in the case of kernel modules related to parallelism, such as those necessary for the implementation of parallel files systems or collective system calls. Cluster systems software can include 'grid" middleware, but we do not discuss that here. We consider it to include parallel user tools, such as parallel versions of common shell commands and an MPI implementation (which must necessarily interact with scheduling and process management).

In general, cluster systems software is what makes a cluster a *cluster* instead of a collection of individual computers. In some cases, a key ingredient is the *scalability* of system management functions. Operations that would be straightforward on a single machine need to be done differently if they are to be repeated consistently on a large number of nodes.

The traditional approach to cluster system software is `ad hoc`. That is, system administrators assemble a workable collection of tools that are developed *without* an overall system architecture that involves parallelism. Low-level tasks are often accomplished by shell scripts that "parallelize" sequential utilities by looping over `rsh` or `ssh` to a set of nodes, which is inherently nonscalable. Some general-purpose tools, like `pdsh` [16], have emerged to improve the scalability of this approach.

A number of monolithic "resource management systems" are in use, such as PBS [15] and LSF [10], which handle a subset of systems tasks, particularly management of user parallel jobs. A small amount of component-like functionality is available, such as the Maui scheduler, which provides a sophisticated parallel job scheduling capability without being necessarily integrated with other functions such as process initiation or monitoring. A number of "packages of packages" are available, such as OSCAR [14] and ROCKS [11], which attempt to provide a consistent collection of cluster software, a difficult goal in its own right. Finally, some nontraditional approaches, such as `bproc` [8], provide an unusual abstraction for cluster software execution. In all these approaches, software integration occurs "manually"; that is, the global collection of system software depends on individual implementations rather than on a well-defined set of interfaces.

The work described here was motivated by the operation of two clusters at Argonne National Laboratory, one for research and one for production use. A year ago we were using OpenPBS and the Maui scheduler on our research cluster, called Chiba City, along with other aspects of the traditional paradigm. On the production cluster, Jazz, we use PBSPro, the supported version of OpenPBS. These tools have been satisfactory for traditional users, but we became increasingly dissatisfied with this monolithic, inflexible approach, especially on the research cluster. There the normal administrative overhead was augmented by unusual requirements, such as the need to build special kernel versions for the duration of a scheduled job or to provide root access for the nodes scheduled for a job. These requirements were in addition to the needs common to all clusters: elimination of manual sequential tasks on a several-hundred-node cluster, support for thinking "collectively" in order to have a scalable conceptual approach to systems management, and the need for

simplicity and "elasticity" (the opposite of brittleness).

Although most of this work was carried out in the context of finding a replacement architecture for cluster systems software, the ideas apply to any computer system where parallelism, flexibility, and scalability are required.

3 The Component Approach

Clemens Szyperski [20] has defined software components as follows:

> A software component is a unit of composition with contractually specified interfaces and explicit context dependencies only. It can be deployed independently and is subject to composition by third parties.

The expected benefits of components include

1. localization of functionality to encourage reuse,
2. the ability of components to evolve or be replaced without affecting the overall system, and
3. the possibility of assembling disparate components in ways not envisioned by the individual component implementers.

The Scalable Systems Software SciDAC Project [19] has as its goal the exploration of a component approach to systems software with the objective of achieving precisely these benefits. The work described here is the result of our participation in that project and the process of applying its approach to our own systems. The result has been a new paradigm for the development and deployment of systems software.

The central idea is to replace monolithic resource management systems with a collection of simple, single-function components with published interfaces to other components. These components can be wrappers for existing programs, all-new software designed to deliver new functionality, or even throwaway, lightweight components designed for temporary use. This third type of component motivates our desire to make it particularly easy to implement a component.

Essential for this approach is a uniform but flexible communication infrastructure that makes component interconnection simple, reliable, and secure. Furthermore, the process management component needs to provide scalable support for parallel jobs, whether initiated by a user, a system administrator, or another component.

We make component creation convenient by supplying a software development kit (SDK), described in Section 4.4. Certain parallel tools and components are written in MPI for scalability and performance (See Section 5.4). The net result is that systems management becomes easier. Since one can easily substitute individual components into the system to adapt it to varying requirements, intercomponent communication code needs to be done only once, and individual components can be adapted to local requirements.

4 The "Stack"

In this section we describe the environment in which components operate and how the are written. We present this environment from the bottom up: wire protocols, the communication library, an XML syntax style, the SDK for creating components, and the components themselves. We provide some detail on the process manager because it is a critical component.

4.1 Wire Protocol

The SSS Project decided at an early stage that intercomponent communication would be over sockets and would consist of character data in XML format. This decision reflected the ubiquity of sockets on most machines and operating systems, together with the availability of XML parsing libraries in most languages. Thus, components would not be constrained by platform or language, and tedious input editing could be eliminated by choosing an XML style that allows validating parsers to do most of the work, operating from published XML schemas.

This approach to communication still leaves other decisions to be made (or avoided). The exact XML schemas that define the interfaces to each component must be agreed to; defining these APIs has been a primary activity of the SSS project. The XML "style" can vary greatly: which data items are entities and attributes, the types of values, and so on. In Section 4.3 we discuss some of the issues involved in choosing an XML style. The *framing* of messages needs to be defined: what constitutes a complete XML message to be parsed. Moreover, security issues need to be addressed, including the possible use of passwords and encryption.

The approach we have taken is allow multiple wire protocols in the communication library and make it easy to add more to meet local requirements. The communication library (Section 4.2) hides the choice of wire protocol from the component implementor. Example protocols currently included are "basic" (with a challenge/response mechanism for security), HTTP, and SSSRMAP (used by a subcollection of the SSS project components.) Others can be added by contributing a simple module to the communication library. (In each case, interaction between components consists of a message and its response, after which the socket connection is broken.)

4.2 Communication Library

The purpose of the communication library is to hide the details for intercomponent communication from the component implementor. The send/receive *library* handles synchronous communication. It currently has bindings for Python, Perl, C, C++, and Java, permitting components to be written in any of these languages. In addition to the library itself, two specific components support it:

- The *service directory* maintains contact information for each running component. When components begin execution, they register themselves with the service directory. Any component can contact the service directory (at a "well-known" host/port

address) and ask by name for communication information on other components. The information consists of the state of that component, a host/port on which it can be contacted, and the wire protocol used by that component. Once this information is stored in the component, under ordinary circumstances the service directory does not have to be contacted again.

- The *event manager* handles asynchronous intercomponent communication. As part of its interface, each component documents what events it generates. For example, the process manager generates an event (notifies the event manager) when a job completes. Other components can register with the event manager to receive notification of certain events they are interested in that are generated by other components, and then act on them accordingly.

4.3 XML Style

The messages that flow among components accomplish three things:

1. Match a set of objects in the target component's data store, either identifying them or constructing them.
2. Apply a function (with arguments) to that set of objects.
3. Specify the content of a return message.

The XML content of these messages can vary from quite short and simple to long and highly structured (such as when an object-creation message or the response to a query is large.) For such cases scalability is an important aspect of the XML schema design.

The choice of XML style is governed by a number of considerations, not all of them consistent with one another.

Completeness. It is important to be able to express in syntax all of the semantics that the components need or can supply.

High Value of Validation. XML validation is a powerful tool but can be weakened by a poor choice of XML style. A schema that is too abstract can reduce the value of validation by considering too many messages to be valid, so that significant input editing must be done by the component itself. We prefer a concrete style that permits function signature checking as part of XML validation. This simplifies component code and allows a significant amount of the interface definition to be expressed in the formal language of an XML schema.

Readability. It is not essential, but certainly desirable, that a human be able to read and understand the XML text.

Conciseness. XML is wordy enough as it is. One wants messages to be no longer than necessary, although this goal can conflict with that of readability. Here one has choices that affect scalability, and a good XML style will allow one to refer to many objects collectively.

Atomicity. It is necessary to avoid the race conditions that arise when one first queries a component and then sends a separate command based on the response. Rather, one wants the syntax to support race-free semantics, in which objects are identified and acted upon as the result of a single message, as described above in the three functions of a message.

An example of the XML syntax that we use is the following:

```
<SignalProcessGroup signal='SIGINT'>
  <ProcessGroup>
     <PGID>72</PGID>
  </ProcessGroup>
</SignalProcessGroup>
```

This example sends the signal "SIGINT" to the process group with PGID 72.

4.4 Software Development Kit

The idea of a software development kit is to accelerate application development by providing prepackaged versions of functions that any application will need. The goal is to allow the application writer to focus solely on the application logic that cannot be anticipated by the developer of the infrastructure. We consider the lower level of the SDK to be the communication infrastructure, consisting of the service directory component, the event manager component, and the communication library. There remain, however, a set of necessary component services that are independent of any particular component. These include the following:

- Registration/deregistration with the service directory
- Initiating/terminating logging
- Setting up error reporting
- The basic `select` loop, with connection error handling
- Socket setup/cleanup
- XML parsing
- XML validation
- Message parsing

Our approach has been to encapsulate these services in two Python classes that are available for subclassing:

- Server
- Event Receiver – a special case of the Server class

Thus the SDK currently supports writing components in Python, although the same approach could be taken for other languages.

We give an example of a minimal component. This is an "echo server" that simply sends back whatever text message is sent to it. We illustrate its use by also writing an "echo client" to try out the echo server. Here is the server:

```
#!/usr/bin/env python

from sss.server import Server
class Echo(Server):
    __implementation__ = 'echo'          # set log name
    __component__ = 'echo'               # component answers to 'echo'
    __dispatch__ = {'echo':'HandleEcho'} # call HandleEcho method
                                         #    for echo messages
    __validate__ = 0                     # no schema for this component

    def HandleEcho(self, xml, (peer,port)):
          return xml

if __name__ == '__main__':
    e = Echo()
    e.ServeForever()
```

This example server makes extensive use of the Server class from the SDK. It provides all steps common to component setup and operation. Instances of the Server class use a dispatch table to find handlers for messages. In this case, the function "HandleEcho" is used to process messages with the top-level tag "echo."

Here is the client:

```
#!/usr/bin/python

from os import getpid
from sss.ssslib import comm_lib

c = comm_lib()

h = c.ClientInit('echo')
c.SendMessage(h, "<echo><pid id='%s'/></echo>" % (getpid()))
response = c.RecvMessage(h)
c.ClientClose(h)

print response
```

This example client connects to the component "echo" and sends a varying message to it. It reads the response, completes communication with the "echo" component, and prints the response it received.

5 Parallel Systems Software for Clusters

While a component architecture provides a number of useful software-engineering properties, none of them are inherently parallel (or scalable) in nature. This component architecture mainly provides scalability benefits through the process management system. In traditional Unix-based systems, many process management systems are typically used in the course of a single user session. Users may use `ssh` to initially log in, while their jobs may use a spawning mechanism integrated into the queuing system. Once their jobs are executing, they may use yet another process management system such as `rsh` to start their parallel processes.

In component-based systems, there is a single, unified implementation of any given function. In this particular system, the process management system provides all processes with the bootstrapping information needed for their initialization as MPI processes. Hence, any process can easily become an MPI process. MPI processes can use a variety of libraries for tasks ranging for parallel algorithms to scalable I/O. This further eases the development of highly scalable programs for all sectors of system programming. Systems software has typically lagged behind applications in terms of sophistication; this approach allows system software to take advantage of the same libraries as applications.

A number of systems are crucial to the overall environment on our test system yet do not fit the component model we have focused on so far. We include them to give a complete picture of the system environment seen by users.

5.1 MPI Library

Both user applications and the system tools rely on a robust MPI implementation. We use MPICH2 [12] to provide a complete MPI-2 implementation. Startup of MPI applications in batch mode occurs through the queue manager, scheduler, and process manager components described above. Interactive MPI jobs can be started with the standard `mpiexec`. One benefit of considering a process management component abstractly (rather than as part of an MPI implementation, for example) has been that a large number of features were defined, most of which have made their way into MPICH2's `mpiexec`, rendering it more powerful and flexible than it would have been otherwise.

5.2 Parallel I/O

For scalability and high performance, we use the parallel file system PVFS2 [17], the second-generation of the Parallel Virtual File System. [2]. Users access parallel I/O either directly through the parallel I/O functions in the MPI-2 Standard [6] or through application-oriented libraries such as HDF5 [7] or PnetCDF [9].

5.3 MPI-Based User Tools

The Scalalbie Unix Tools exploit fast MPI startup [1] to provide parallel versions of familar user commands such as `ps`, `ls`, `cp`, or `find`. For details, see [13].

5.4 MPI-Based System Tools

The ease of MPI program integration has improved the adoption of MPI for system tasks. On our development system, we have replaced all tools formerly using *ad hoc* parallelism with MPI analogues.

Our development system has no global file system. To support user jobs, we have implemented a file staging system that handles input staging and output return. We have also implemented a parallel file tree synchronization tool, similar to `rsync` [18]. Moreover, we have written a parallel shell that implements the PMI interface, which MPICH2 processes use to bootstrap. This allows the shell, `MPISH`, to start MPI processes in an environment similar to serial Unix shells.

In each case, we found that the MPI versions of these tools performed substantially better than the pseudo-parallel versions. Code volume was smaller and error handling better in the MPI versions as well. These experiences and performance numbers are detailed in [4]. Details about `MPISH` and its execution model are available in [5].

6 Experiences with the New Paradigm

We have installed our collection of components on our 200-node research cluster as a complete set of system software, replacing our PBS/Maui/homebrew collection of system software. A list of components we are running includes the following:

- Service Directory – a component location component
- Event Manager – asynchronous messaging
- Node State Manager – tracks node states and runs diagnostics
- Build and Configuration Manager – manages node configurations
- Process Manager – starts parallel jobs, supports MPI
- Scheduler – simple scheduler with first-come, first-serve, backfill, and preemption.
- Queue manager – coordinates job execution with the scheduler, running multiple job steps
- Account Manager – matches users with their projects

Overall, we characterize this experience as a success. The machine has been running with our software, and users, for eighteen months. The code volume actually running the system has been dramatically reduced; we estimate that our development system is running with 18,000 lines of systems software code. For this reason, the systems software has become a modifiable entity for systems managers. It is now far easier for them to tailor system software to a particular system, while retaining compatibility through the use of the component interfaces. This experience has been described in detail elsewhere [3].

7 Conclusion

We have described a component-based approach to the design and implementation of system software. We find this a promising architecture for building scalable computing environments, especially when combined with other approaches to scalable parallelism. Considerable work remains to be done in this area. Systems software in general, particularly for large-scale systems, needs more attention paid to fault tolerance, security, and scalability. We are confident that the component approach we have outlined here, by providing a way for different parts of the system to evolve separately, can accelerate these developments. In addition, components make the entire system management enterprise more "programmable," while an MPI approach to systems operation can allow one to think and operate more collectively and thus more scalably. One can envision a sequence of operations like

```
collective operation on nodes
allreduce on the local return codes
split nodes into "communicators"
collectively handle those where the operation succeeded
collectively handle those where the operation failed
```

This approach may even lead to high-level parallel language in which to write a program for running a cluster autonomically. The implementation of this language is likely to build on systems software components of the type we have described here.

8 Acknowlegment

This work was supported by the Mathematical, Information, and Computational Sciences Division subprogram of the Office of Advanced Scientific Computing Research, Office of Science, U.S. Department of Energy, under Contract W-31-109-ENG-38.

References

[1] R. Butler, N. Desai, A. Lusk, and E. Lusk. The process management component of a scalable system software environment. In *Proceedings of IEEE International Conference on Cluster Computing (CLUSTER03)*, pages 190–198, 2003.

[2] Philip H. Carns, Walter B. Ligon III, Robert B. Ross, and Rajeev Thakur. PVFS: A parallel file system for linux clusters. In *Proceedings of the 4th Annual Linux Showcase and Conference*, pages 317–327, Atlanta, GA, October 2000. USENIX Association.

[3] N. Desai, R. Bradshaw, A. Lusk, E. Lusk, and R. Butler. Component-based cluster systems software architecture: A case study. In *Proceedings of IEEE International Conference on Cluster Computing (CLUSTER04)*, 2004.

[4] Narayan Desai, Rick Bradshaw, Andrew Lusk, and Ewing Lusk. MPI cluster system software. In Dieter Kranzlmuller, Peter Kacsuk, and Jack Dongarra, editors, *Recent Advances in Parallel Virutal Machine and Message Passing Interface*, number 3241

in Springer Lecture Notes in Computer Science, pages 277–286. Springer, 2004. 11th European PVM/MPI Users' Group Meeting.

[5] Narayan Desai, Andrew Lusk, Rick Bradshaw, and Ewing Lusk. MPISH: A parallel shell for MPI programs. In *Proceedings of the 1st Workshop on System Management Tools for Large-Scale Parallel Systems*, 2005.

[6] William Gropp, Steven Huss-Lederman, Andrew Lumsdaine, Ewing Lusk, Bill Nitzberg, William Saphir, and Marc Snir. *MPI—The Complete Reference: Volume 2, The MPI-2 Extensions*. MIT Press, Cambridge, MA, 1998.

[7] HDF5. `http://hdf.ncsa.uiuc.edu/HDF5/`.

[8] Erik Hendriks. BProc: The Beowulf distributed process space. In *Proceedings of SC 2002*, 2002.

[9] J. Li, W.-K. Liao, A. Choudhary, R. Ross, R. Thakur, W. Gropp, and R. Latham. Parallel netCDF: A scientific high-performance I/O interface. Technical Report ANL/MCS-P1048-0503, Mathematics and Computer Science Division, Argonne National Laboratory, May 2003.

[10] Load Sharing Facility (LSF). http://www.platform.com.

[11] Greg Bruno Mason J. Katz, Philip M. Papadopoulos. Leveraging standard core technologies to programmatically build Linux cluster appliances. In *Proceedings of IEEE International Conference on Cluster Computing (CLUSTER02)*, pages 47–53, 2002.

[12] MPICH2. `http://www.mcs.anl.gov/mpi/mpich2`.

[13] Emil Ong, Ewing Lusk, and William Gropp. Scalable Unix commands for parallel processors: A high-performance implementation. In Y. Cotronis and J. Dongarra, editors, *Recent Advances in Parallel Virtual Machine and Message Passing Interface*, volume 2131 of *Lecture Notes in Computer Science*, pages 410–418. Springer-Verlag, September 2001. 8th European PVM/MPI Users' Group Meeting.

[14] OSCAR: Open source cluster application resource. `http://oscar.sourceforge.net`.

[15] `http://www.openpbs.org/`.

[16] PDSH:parallel distributed shell. `http://www.llnl.gov/linux/pdsh/pdsh.html`.

[17] The Parallel Virtual File System, version 2. `http://www.pvfs.org/pvfs2`.

[18] `http://rsync.samba.org/`.

[19] `http://www.scidac.org/ScalableSystems`.

[20] Clemens Szyperski. *Component Software: Beyond Object-Oriented Programming*. Addison-Wesley Longman Publishing Co., Inc., 2nd edition, 2002.