A Component Architecture for LAM/MPI*

Jeffrey M. Squyres and Andrew Lumsdaine

Open Systems Lab, Indiana University
<jsquyres,lums@lam-mpi.org>

Abstract. To better manage the ever increasing complexity of LAM/MPI, we have created a lightweight component architecture for it that is specifically designed for high-performance message passing. This paper describes the basic design of the component architecture, as well as some of the particular component instances that constitute the latest release of LAM/MPI. Performance comparisons against the previous, monolithic, version of LAM/MPI show no performance impact due to the new architecture—in fact, the newest version is slightly faster. The modular and extensible nature of this implementation is intended to make it significantly easier to add new functionality and to conduct new research using LAM/MPI as a development platform.

1 Introduction

The Message Passing Interface (MPI) is the *de facto* standard for message passing parallel programming on large-scale distributed systems [1–3]. Implementations of MPI comprise the middleware layer for many large-scale, high-performance applications [4, 5]. LAM/MPI is an open source, freely available implementation of the MPI standard. Since its inception in 1989, the LAM project has grown into a mature code base that is both rich with features and efficient in its implementation, delivering both high performance and convenience to MPI users and developers.

LAM/MPI is a large software package, consisting of over 150 directories and nearly 1,400 files of source code. Research, development, and maintenance of this code base—even for the LAM/MPI developers—is a complex task. Even though the LAM/MPI source code is fairly well structured (in terms of file and directory organization), contains many well-abstracted and logically separated functional code designs, and includes several highly flexible internal APIs, new LAM/MPI developers are inevitably overwhelmed when learning to work in the code base. Third party developers and researchers attempting to extend the LAM/MPI code base—or even to *understand* the code base—are frequently stymied because of the intrinsic complexities of such a large software system. Hence, not only does it take a long time to train new LAM developers, external contributions to LAM/MPI are fairly rare and typically only small, specific patches.

* Supported by a grant from the Lilly Endowment and by National Science Foundation grant 0116050.

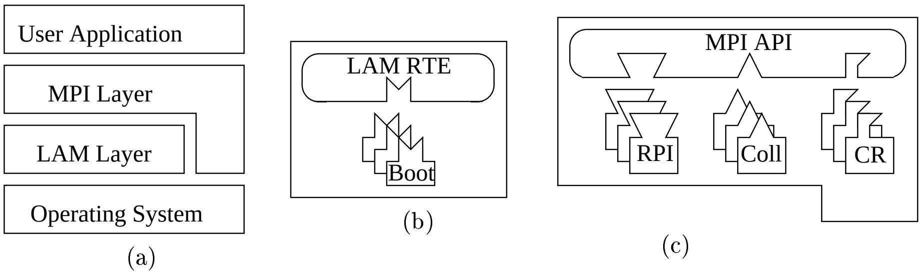

Fig. 1. (a) High-level architecture showing the user's MPI application, the MPI layer, the LAM layer, and the underlying operating system. The LAM layer (b) and MPI layer (c) each have public interfaces that are implemented with back-end components

For these reasons, a natural evolutionary step for LAM is a modular, component-based architecture. In this paper, we present the component architecture of LAM/MPI: the *System Services Interface* (SSI), first available in LAM version 7.0. To create this architecture, existing abstractions within the LAM/MPI code base have been identified and re-factored, and their concepts and interfaces formalized. Components for new functionality were also added. Hence, it is now possible to write (and maintain) small, independent, component modules that "plug in" to LAM's overall framework. It was a specific design goal to allow multiple modules of the same component type to co-exist in a process, one (or more) of which and be selected for use at run-time. Hence, different implementations of the same component can be created to experiment with new research directions, provide new functionality, etc. The architecture was designed to be lightweight, high-performance, and domain-specific (in contrast to general purpose component architectures such as CORBA [6] or CCA [7]).

The result is not simply an MPI implementation, but rather a framework that effects an MPI implementation from specified components. The rest of this paper is organized as follows. Section 2 provides a brief overview of LAM/MPI. LAM's component architecture and component implementations are described in Section 3. Future work and conclusions are given in Sections 5 and 6.

2 Overview of LAM/MPI

LAM/MPI is an open source implementation of the MPI standard developed and maintained at Indiana University. It implements the complete MPI-1 standard and much of the MPI-2 standard. LAM/MPI is made up of the LAM run-time environment (RTE) and the MPI communications layer (see Fig. 1). For performance reasons, both layers interact directly with the operating system; requiring the MPI layer to utilize the LAM layer for all MPI communications would impose a significant overhead.

2.1 LAM Layer

The LAM layer includes both the LAM RTE and a companion library providing C API functions to interact with the RTE. The RTE is based on user-level daemons (the "`lamds`"), and provides services such as message passing, process

Table 1. Modules that are available for each component type.

Component type	Modules available
RPI	`gm`, `lamd`, `tcp`, `sysv`, `usysv`
Coll	`lam_basic`, `smp`
CR	`blcr`
Boot	`bproc`, `globus`, `rsh`, `tm`

control, remote file access, and I/O forwarding. A user starts the LAM RTE with the `lamboot` command which, in turn, launches a `lamd` on each node (from a user-specified list) using a back-end component to start processes on a remote node (see Fig. 1). After the RTE is up, MPI programs can be run. When the user is finished with the RTE, the `lamhalt` command is used to terminate the RTE by killing the `lamd` on every node.

2.2 MPI Communications Layer

The MPI layer itself consists of multiple layers. The upper layer includes the standardized MPI API function calls and associated bookkeeping logic. The lower layer has recently been re-architected into a component framework. As such, the upper layer acts as the public interface to multiple types of back-end components that implement MPI functionality (see Fig. 1).

3 Component Architecture

The LAM/MPI component architecture supports four major types of components [8]. "RPI" (Request Progression Interface) components provide the back-end implementation of MPI point-to-point communication [9]. "Coll" components provide the back-end implementations of MPI collective algorithms [10]. "CR" components provide interfaces to checkpoint-restart systems to allow parallel MPI jobs to be checkpointed [11]. Finally, "Boot" components provide the capability for launching the LAM RTE in different execution environments [12]. Table 1 lists the modules that are available in LAM for each component type. RPI, Coll, and Boot were selected to be converted to components because they already had well-defined abstractions within the LAM/MPI code base. The CR type represented new functionality to LAM/MPI; it only made sense to design it as a component from its initial implementation.

3.1 Supporting Framework

Before transforming LAM's code base to utilize components, it was necessary to create a supporting framework within LAM/MPI that could handle component configuration, compilation, installation, and arbitrary parameters, both at compile-time and run-time. The following were design goals of the supporting framework:

- Design the component types to be implemented as plug-in modules, allowing both static and dynamic linkage.

- Simplify the configuration and building of plug-in modules; explicitly do not require the module to be aware of the larger LAM/MPI configuration and build system.
- Enable multiple modules of the same component type to exist in the same MPI process, and allow the selection of which module(s) to use to be a run-time decision.
- Allow a flexible system of passing arbitrary compile-time defaults and user-specified run-time parameters to each module.

This supporting infrastructure is responsible for the configuration and compilation of modules, and initializes relevant modules at run-time. It is used to provide a small number of run-time utility services to modules for cross-component functionality. Each of the component types has its own "base" that offers common utility functionality to all of its modules. This allows modules to share some functionality for tasks that are likely to be common across modules of the same component type.

3.2 MPI Point-to-Point Communication Components

The Request Progression Interface (RPI) is responsible for the movement of messages from one process to another. The RPI revolved around the concept of an MPI request; its API functions essentially follow the life of the request (creating, starting, advancing, finishing). The actual message passing is almost a side-effect of the life cycle of the request; implementation of how the bytes actually move from one process to another is not directly addressed in the API.

LAM/MPI has long supported a formal API and set of abstractions for the RPI. However, prior to converting the RPI to be component-based, selection of which RPI to use was a compile-time decision; only one RPI could be compiled into a LAM installation; LAM had to be installed multiple times to support different underlying communication mechanisms. As such, the RPI was a natural choice to be LAM's first component design. Converting the existing RPI code to be component-based provided not only a sound basis for testing the base SSI framework, but also added the capability to support different underlying message passing transports in a single installation of LAM (by allowing multiple RPI modules to co-exist in the same MPI process).

In addition to the previously existing RPIs for TCP and shared memory, two new RPI modules have been added: native support for Myrinet networks using the `gm` message passing library [13], and a modified version of the TCP RPI that supports checkpointing (named "CRTCP").

3.3 MPI Collective Communication Components

The design and implementation of the majority of the collective communication component was straightforward: the main public functions are identical to the MPI collective functions themselves. Hence, LAM's top-level MPI API functions check arguments for errors and perform minimal bookkeeping, and then simply invoke the corresponding underlying component function with the same arguments that it received.

The collective components provide two versions of each collective operation—one for intracommunicators and one for intercommunicators. If a module does not support intercommunicator algorithms, for example, it can provide NULL for its function pointer. This design provides a clear method for modules to indicate whether they support intercommunicator collectives. Hence, LAM/MPI fully supports the capability to have intercommunicator collectives (although no collective modules implement them yet).

The collective component was designed such that each communicator may use a different module. For example, collectives invoked on MPI_COMM_WORLD may use one set of algorithms, while collectives invoked on my_comm may use an entirely different set of algorithms (i.e., a different module). As such, significant effort was put into the design of the query and initialization of the collective component functions. Modules can be manually selected by the user, or automatically nominate which module is "best" for a given communicator.

Although all current collective modules are implemented as "layered" implementations of the MPI collective algorithms (i.e., they use MPI point-to-point functions for message passing), this is not required. For example, it is possible that future collective modules may provide their own communication channels.

In addition, to support complex communication patterns, collective modules can utilize other collective modules in a hierarchical pattern. That is, a collective module can create sub-communicators to simplify the execution of its algorithms. A new module was written that utilized this concept: collective algorithms tuned for SMPs on a LAN [14]. For example, the SMP-based algorithm for an MPI barrier is to perform a fan-in to a local manager, followed by a barrier among the set of managers, and then a local fan-out from the managers. The SMP module implements this algorithm by creating two sub-communicators: one that group processes on the local node, and another for the set of managers. For example, the local fan-in and fan-out is implemented as a zero-byte MPI gather and MPI broadcast (respectively) on the local communicator. Invoking collective operations on the sub-communicators simply invokes LAM's basic collective algorithms.

3.4 Checkpoint/Restart Components

The checkpoint/restart (CR) component represents new functionality in LAM/MPI. Parallel MPI jobs can be involuntarily checkpointed and restarted essentially any time between MPI_INIT and MPI_FINALIZE using a coordinated approach. The role of the CR component is twofold: interface with a back-end checkpointing system and coordinate other MPI SSI modules to checkpoint and restart themselves.

Three abstract actions are defined for CR components: **checkpoint**, **continue**, and **restart**. The **checkpoint** action is invoked when a checkpoint is initiated. It is intended to be a bounded-time action that allows an MPI process to prepare itself to be checkpointed. The **continue** action is activated after a successful checkpoint. This can be an empty action, but if migration is supported, communication channels may need to be re-established if processes have moved. The **restart** action is invoked after an MPI process has been restored from a prior checkpoint. This action will almost always need to re-discover its MPI process peers and re-establish communication channels to them.

The CR functionality requires that all MPI SSI modules that are selected at run-time support the ability to checkpoint and restart themselves. Specifically, API functions are included in both the MPI collective and point-to-point components that are invoked at checkpoint, continue, and restart time. Each SSI module can do whatever it needs for these three actions. For example, RPI modules may need to consume all "in-flight" messages and ensure that the communications network is fully quiesced before allowing the checkpoint to continue (this is exactly what the CRTCP RPI does to support checkpointing).

Since the checkpoint/restart capability is new to LAM/MPI, there is only one module available: an interface to the BLCR single-node checkpointer [15]. The CR functionality in LAM/MPI is discussed in further detail in [11].

3.5 Boot Components

Boot components are used as the back-end implementations for starting and expanding the LAM RTE. Previously, LAM only had the capability to launch `lamds` on remote nodes via `rsh` or `ssh`. The creation of the boot component formalized the abstractions required to identify remote hosts, start a process on a remote node, and communicate initialization protocol information with the newly-started process(es). Inspiration for the design of this interface was strongly influenced by the existing `rsh/ssh` boot code, as well as expectations of what would be necessary to utilize the native remote execution facilities under the PBS batch queue system [16].

Once these actions were identified and prototyped into functions, the `rsh/ssh` code was converted into a module. A second boot module was also written to verify the design that uses the Task Management (TM) interface to PBS to launch arbitrary processes on allocated nodes. After these two modules were completed, support was also added for BProc systems [17] and Globus-enabled systems [18]. With each of these boot modules in place, LAM can now be "natively" booted in a wide variety of execution environments.

4 Performance

To ensure that the new architecture of LAM/MPI does not negatively impact performance, the new component architecture (version 7.0) was compared against the last stable monolithic version of LAM (version 6.5.9) using the NetPIPE MPI benchmarks [19]. NetPIPE measures both the latency and the bandwidth for MPI message passing over a range of message sizes. Fig. 2 shows the percentage difference of TCP bandwidth between versions 7.0 and 6.5.9 using NetPIPE 2.4 under Linux 2.4.18-smp (RedHat 7.3) on a pair of 1.5GHz Pentium IV nodes, each with 768MB of RAM, connected via fast Ethernet (on the same switch). The figure shows that the performance difference between the two architectures is literally in the noise (less than $+/-1\%$).

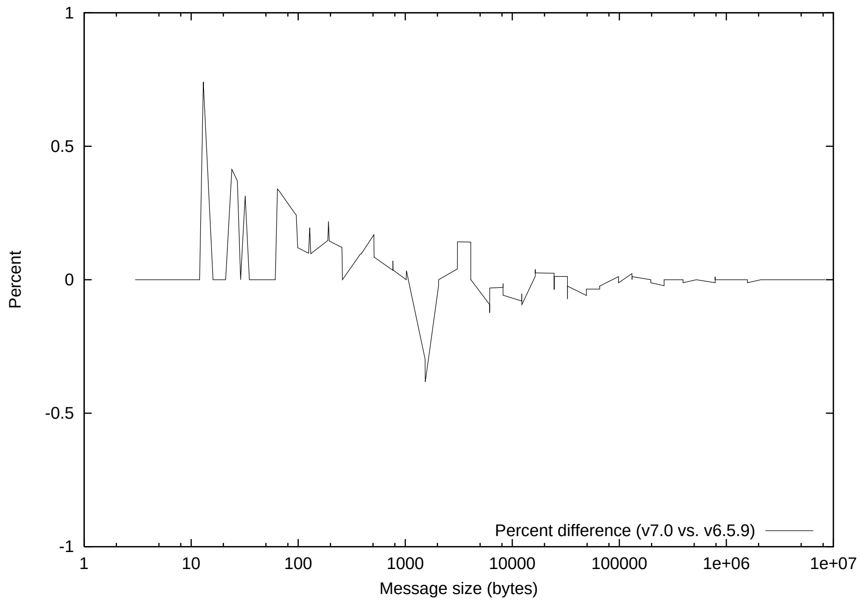

Fig. 2. TCP bandwidth percentage difference between LAM/MPI versions 7.0 (component architecture) and 6.5.9 (monolithic architecture)

5 Future Work

Our main focus for future work will be equally divided among three directions:

1. Continued refinement and evolution of component interfaces. Expansion to allow concurrent multithreaded message passing progress, for example, will necessitate some changes in the current component designs.
2. Creating more component types for both the MPI and LAM layers. Possible candidates for new component types include: MPI topology functions, MPI one-sided functions, MPI I/O functions, operating system thread support, and out-of-band messaging.
3. Creating more modules for the component types that already exist. Possible candidates include: native MPI point-to-point modules for other high-speed networks, and further refinement of SMP and wide-area MPI collectives.

6 Conclusions

The component architecture of LAM/MPI allows for much more fine-grained research and development than has previously been possible with MPI implementations. The current set of components not only allows independent research in MPI functional areas, it also fosters the spirit of open source development and collaboration. For example, researchers studying new MPI collective implementations can simply write a small run-time module and focus on their algorithms without having to deal with the intrinsic complexities of LAM/MPI itself. This freedom has been previously unavailable in widely-used MPI implementations and is a significant reason why researchers have been forced to either create small research-quality MPI subsets that focus on their specific work, or create entire derivative MPI implementations to show their methodologies.

The overall size of the LAM/MPI code base grew with the new component architecture; new code had to be written to selectively configure, compile, and select modules at run-time. However, this new code was essentially a one-time cost since it will change much less frequently than the code of the modules themselves. Additionally, with the segregation of code into orthogonal, functionally distinct modules, the impact on maintenance and new functionality has been dramatic. In short: the use of a component architecture has resulted in a significantly simpler code base to develop and maintain.

LAM/MPI, including the source code, technical specifications, and documentation of all the component architectures described in this paper [8–12], is freely available from `http://www.lam-mpi.org/`.

7 Acknowledgments

This work was performed using computational facilities at Indiana University and the College of William and Mary. These resources were enabled by grants from Sun Microsystems, the National Science Foundation, and Virginia's Commonwealth Technology Research Fund.

References

[1] Gropp, W., Huss-Lederman, S., Lumsdaine, A., Lusk, E., Nitzberg, B., Saphir, W., Snir, M.: MPI — The Complete Reference: Volume 2, the MPI-2 Extensions. MIT Press (1998)

[2] Message Passing Interface Forum: MPI: A Message Passing Interface. In: Proc. of Supercomputing '93, IEEE Computer Society Press (1993) 878–883

[3] Snir, M., Otto, S.W., Huss-Lederman, S., Walker, D.W., Dongarra, J.: MPI: The Complete Reference. MIT Press, Cambridge, MA (1996)

[4] Burns, G., Daoud, R., Vaigl, J.: LAM: An Open Cluster Environment for MPI. In Ross, J.W., ed.: Proceedings of Supercomputing Symposium '94, University of Toronto (1994) 379–386

[5] Gropp, W., Lusk, E., Doss, N., Skjellum, A.: A high-performance, portable implementation of the MPI message passing interface standard. Parallel Computing **22** (1996) 789–828

[6] Object Management Group: The common object request broker: Architecture and specification (1999) Revision2.3.1.

[7] Armstrong, R., Gannon, D., Geist, A., Keahey, K., Kohn, S.R., McInnes, L., Parker, S.R., Smolinski, B.A.: Toward a common component architecture for high-performance scientific computing. In: HPDC. (1999)

[8] Squyres, J.M., Barrett, B., Lumsdaine, A.: The system services interface (SSI) to LAM/MPI. Technical Report TR575, Indiana University, Computer Science Department (2003)

[9] Squyres, J.M., Barrett, B., Lumsdaine, A.: Request progression interface (RPI) system services interface (SSI) modules for LAM/MPI. Technical Report TR579, Indiana University, Computer Science Department (2003)

[10] Squyres, J.M., Barrett, B., Lumsdaine, A.: MPI collective operations system services interface (SSI) modules for LAM/MPI. Technical Report TR577, Indiana University, Computer Science Department (2003)
[11] Sankaran, S., Squyres, J.M., Barrett, B., Lumsdaine, A.: Checkpoint-restart support system services interface (SSI) modules for LAM/MPI. Technical Report TR578, Indiana University, Computer Science Department (2003)
[12] Squyres, J.M., Barrett, B., Lumsdaine, A.: Boot system services interface (SSI) modules for LAM/MPI. Technical Report TR576, Indiana University, Computer Science Department (2003)
[13] Myricom `http://www.myri.com/scs/GM/doc/html/`: GM: A message passing system for Myrinet networks. (2003)
[14] Kielmann, T., Bal, H.E., Gorlatch, S.: Bandwidth-efficient Collective Communication for Clustered Wide Area Systems. In: International Parallel and Distributed Processing Symposium (IPDPS 2000), Cancun, Mexico, IEEE (2000) 492–499
[15] Duell, J., Hargrove, P., Roman, E.: The design and implementation of Berkeley Lab's linux checkpoint/restart (2002) http://www.nersc.gov/research/FTG/checkpoint/reports.html.
[16] Veridian Systems: Portable Batch System / OpenPBS Release 2.3, Administrator Guide. (2000)
[17] Hendriks, E.: BProc Manual, `http://bproc.sourceforge.net/`. (2001)
[18] Foster, I.: The anatomy of the Grid: Enabling scalable virtual organizations. International Journal of Supercomputer Applications **15** (2001)
[19] Snell, Q., Mikler, A., Gustafson, J.: Netpipe: A network protocol independent performace evaluator. In: IASTED Internation Conference on Intelligent Information Management and Systems. (1996)