A Framework for Bootstrapping Extreme Scale Software Systems

Joshua D. Goehner, Dorian C. Arnold
University of New Mexico

Dong H. Ahn, Gregory L. Lee, Bronis R. de Supinski, Matthew P. LeGendre, Martin Schulz
Lawrence Livermore National Lab

Barton P. Miller
University of Wisconsin

ABSTRACT
As the sizes of high-end computing systems continue to grow to massive scales, efficient boostrapping for distributed software infrastructures is becoming a greater challenge. Distributed infrastructure bootstrapping is the procedure of instantiating all processes of the distributed system on the appropriate hardware nodes and disseminating to these processes the information that they need to complete the infrastructure's start-up phase. In this paper, we describe the lightweight infrastructure-bootstrapping infrastructure (LIBI). LIBI is both a bootstrapping API specification and a reference implementation. We describe the LIBI architecture, abstractions and implementation and present performance results from our first prototype.

Categories and Subject Descriptors
D.4.9 [**Operating Systems**]: Systems Programs and Utilities

Keywords
Infrastructure bootstrapping, System Software, Scalability

1. INTRODUCTION
In high-performance computing (HPC) environments, we continue to observe dramatic growth in system sizes. On the most recent Top 500 list [13], 149 (or 29.8%) of the 500 entries have greater than 8,192 cores, compared to 12 (or 2.4%) just 5 years ago. On this most recent list, three are larger than 200K cores; an additional 6 are larger than 128K cores, and an additional 7 are larger than 64K cores. The Lawrence Livermore National Laboratory is scheduled to receive its 1.6 million core system, Sequoia [3], this year. Further, exascale systems are projected to have on the order of tens to hundreds of millions of cores within the current decade [6]. HPC system software, capability class applications, which are large, groundbreaking applications that can utilize the entire system, and HPC tools for these applications must scale to these massive sizes.

To appear in the First International Workshop on High-performance Infrastructure for Scalable Tools (WHIST 2011) held as part of 25th International Conference on Supercomputing (ICS 2011), Tucson, Arizona, May 31-June 4, 2011.
Contact author: jgoehner@cs.unm.edu

All distributed software systems must undergo some bootstrapping phase in which their processes are started and some rudimentary information is exchanged. As depicted in Figure 1, we can define distributed software infrastructure bootstrapping as follows: given an allocation of physical computational nodes, bootstrapping is the procedure of instantiating the infrastructure's composite processes appropriately on the computational nodes and exchanging the information necessary for these processes to complete their setup and enter their primary operational phases. Technically, bootstrapping is not complete until the processes actually act upon exchanged information; this final activity is indeed infrastructure dependent.

In the basic approach for distributed infrastructure bootstrapping, a master process uses a remote process creation mechanism, like `rsh` or `ssh`, to sequentially instantiate the other processes. The master process then uses direct point-to-point communication to deliver requisite initialization information to the sub-processes. Figure 2 shows the time it takes for sequentially instantiating a set of processes using `rsh` on Lawrence Livermore National Laboratory's Atlas cluster. As one might expect, this approach does not scale! Extrapolating the dashed trend line, which has a r-squared value of .99, suggests it would one minute to instantiate just 8,192 processes.

For scalability, many infrastructures implement custom mechanisms to launch their application processes. For example, as we describe in Section 3, the MRNet [11] infrastructure uses a hierarchical approach where the master process starts a small subset of the processes that, in turn, start other processes, that in turn start other processes until all the processes of the system are instantiated. While this approach is more scalable than sequential start-up, it is still limited by the mechanism used to start each process (again usually `rsh` or `ssh`) and the number of processes started by each individual process. Furthermore, these mechanisms are embedded into the system itself and cannot be leveraged by other distributed infrastructures.

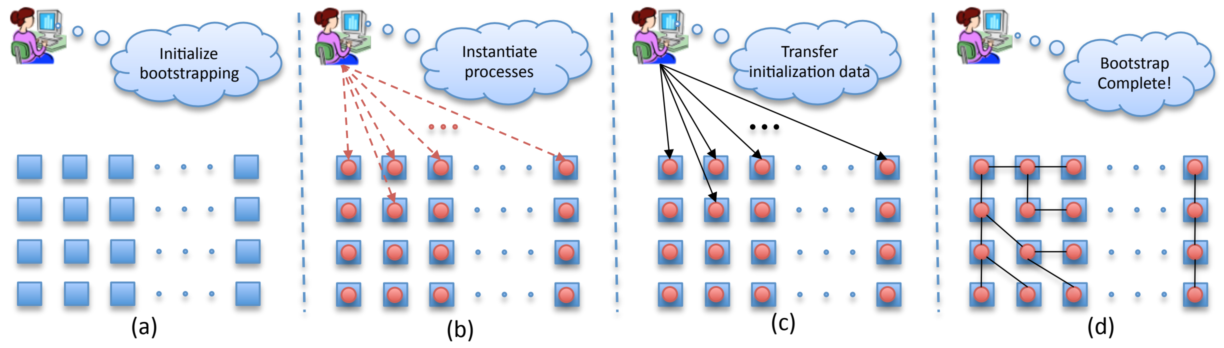

Figure 1: Distributed Application Bootstrapping: instantiating the distributed application's processes and propagating to these processes their requisite initialization information.

Applications and tools also can leverage resource managers (RMs) like LoadLeveler [4], LSF [9], PBS [8] and SLURM [5] for process creation. Typically in such cases, an RM daemon persists on each computer node in the system, and these RMs are used to start processes very efficiently on their resident nodes. Generally, RMs do not provide facilities for disseminating application data to complete the infrastructure-bootstrapping process. Additionally, while the myriad RMs indeed provide a similar service for process creation, they have different, incompatible interfaces, requiring that job launching scripts be ported from platform to platform.

In earlier work, we began to address these issues by developing the LaunchMON infrastructure [1]. With LaunchMON, we aimed to provide an abstraction layer to wrap **existing** resource management services. An application or tool that used LaunchMON can indirectly leverage any of the set of RM services that LaunchMON supports. LaunchMON's abstractions explicitly targeted HPC tools and were shaped primarily by existing RM mechanisms. These design decisions meant that LaunchMON is not always flexible and ideal for more general use cases. For example, LaunchMON supports the capability to deploy tool processes on the same nodes of already running MPI applications. The mechanisms to support such capabilities can encumber using LaunchMON in cases where the capabilities are not needed.

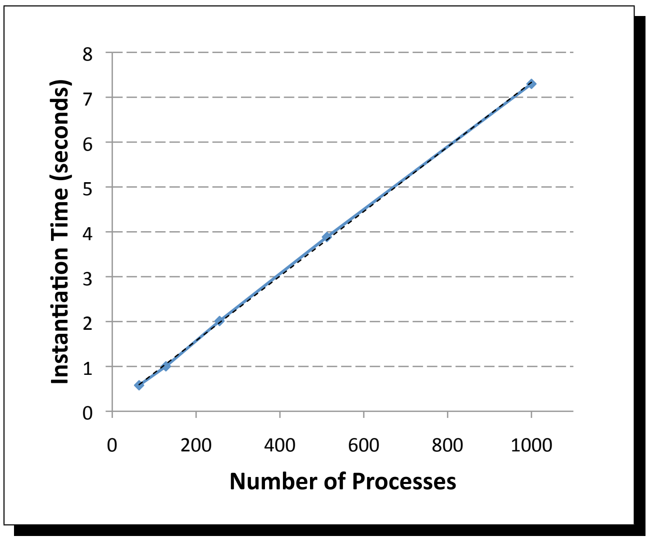

Figure 2: Sequential Process Instantiation

In this work, we focus on identifying the proper set of abstractions and mechanisms for scalable infrastructure bootstrapping of general software distributed systems. We present the lightweight infrastructure-bootstrapping infrastructure (LIBI), a reference implementation of our system for launching distributed applications. LIBI is not intended to replace existing RMs. LIBI is an extension to our earlier LaunchMON work intended to provide a more intuitive and flexible system bootstrapping interface and mechanisms for leveraging RMs in a portable manner. Indeed, we describe how this work will culminate eventually in a refactorization of LaunchMON's services. Additionally, LIBI should provide efficient, alternative bootstrapping mechanisms for environments where scalable native services are unavailable. The work in this paper makes several contributions:

- a set of abstractions and mechanisms for the bootstrapping of applications, tools and system software on extreme scale distributed systems;
- an early prototype of these abstractions and mechanisms; and
- a preliminary evaluation of this prototype to demonstrate both the necessity and feasibility of such infrastructures.

In the following section, we describe several examples of bootstrapping that motivate LIBI's design. In Section 4, we describe the architecture, interface and implementation of our proposed LIBI infrastructure. In Section 5, we present our current LIBI performance results. We conclude this paper with a discussion of outstanding issues and future work.

2. MOTIVATING EXAMPLES

We have identified three general modes of infrastructure bootstrapping based on the number of different images used for launching the processes and whether some processes cannot be created by our infrastructure, for example, because they are already running or must be created by some specific third party service like `mpirun`. In the basic case, we can create all processes from a single image. This case supports distributed systems based on the single program multiple data (SPMD) model that do not rely on special environmental features (like those provided by MPI).

In the second case, we can create all processes, but from multiple images. We give a concrete example of this in Section 3, in which a subset of the processes are started using one image and the remainder are started using another image. Currently, we focus on these two cases, but we are working to support the third case in which a subset of the processes are started by a separate service. In this situation, we need "out-of-band" mechanisms for communicating with those processes to give them the information needed to properly join into a single distributed system session.

3. A CONCRETE EXAMPLE

In this section, we describe a real distributed boostrapping scenario. Specifically, we describe the MRNet instantiation process. MRNet, the multicast-reduction network, is a prototypical tree-based overlay network, a network of hierarchically organized processes that leverages the scaling properties of the tree organization to provide scalable data multicast, data gather, and in-network aggregation. The root of the MRNet tree is called the *front-end*; the leaves are called *back-ends*, and the intermediate processes are the *internal processes*. The MRNet start-up process entails instantiating the MRNet internal and back-end processes and propagating information such that each process can establish a connection with its parent in the tree. After MRNet instantiation is complete, MRNet-based applications can use MRNet's communication and aggregation services.

MRNet's primary bootstrap mode implements a *parent-creates-children* scheme in which process instantiation and information dissemination are integrated. The front-end uses a remote shell mechanism, like `rsh` or `ssh`, to create its children processes for the first level of the tree. As command line arguments during this creation process, each parent propagates the information necessary for their children to connect back to it, for example, parent IP address and port number. Each newly created child process establishes a connection back to its parent process and receives the portion of the topology configuration relevant to that child. Each child then uses this information to instantiate its immediate children. This procedure is repeated until the entire tree of communication and application processes is created. As a final step in this start-up procedure, MRNet propagates the complete topology information to all processes (for fault-tolerance purposes).

By concurrently instantiating processes in disjoint branches, the MRNet start-up procedure is much more efficient than the baseline sequential approach. However, this procedure still suffers from serialization bottlenecks due to each parent's responsibility to instantiate its children. For example, with a fan-out of 64, we frequently experience start-up times on the order of tens of seconds for our STAT tool [2, 7].

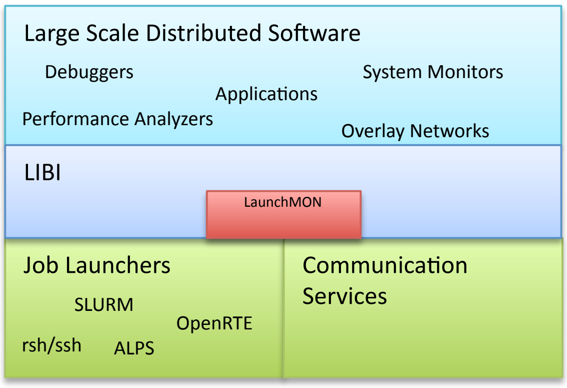

Figure 3: The LIBI Architecture:

4. LIBI OVERVIEW

The lightweight infrastructure-bootstrapping infrastructure (LIBI) is aimed at providing the set of abstractions needed by distributed infrastructures to initialize themselves and a scalable set of mechanisms that implement these primitive operations. Motivated by the various *start-up* modes, described in the previous section, we now describe LIBI's architecture and API the current status of our prototype implementation.

As described in Section 1, bootstrapping a distributed infrastructure entails instantiating the infrastructure's processes and disseminating to those processes the information they need to complete the system's start-up procedure. LIBI is designed and implemented to allow infrastructures to bootstrap themselves in a portable and scalable way by leveraging native resource managers or job launchers and other scalable services.

Figure 3 shows how LIBI is intended to be interpositioned in an HPC system software stack. Software infrastructures can use LIBI services for boostrapping themselves. LIBI leverages the best resource management and communication services available to deliver an efficient bootstrapping service. LIBI is designed to access these lower-level services via LaunchMON or directly when LaunchMON is unavailable or not suitable for the application at hand.

Since LIBI is intended to provide an efficient and scalable service, LIBI is designed to use scalable resource management (RM) services when available or an efficient baseline approach when such RM services are not available. For process launching, we prefer to use scalable RMs that can bulk launch all necessary processes. We also support a second launching strategy partially motivated by the approach taken to port MRNet to the Cray XT Series. In this approach, MRNet uses Cray's Application Level Placement Scheduler (ALPS), which is limited to launching only a single process on each node on which an MRNet process will be placed. These initial processes act as agents to start other processes on their respective nodes as well as to communicate necessary start-up information. LIBI takes a similar approach: LIBI uses process agents on each node to execute node-local process instantiation and communication services on its behalf. Finally, LIBI is designed to support a fallback strategy in which `rsh`-based mechanisms can be used to instantiate LIBI agents on the relevant nodes. These agents are organized into an efficient communication tree and used to launch the requisite application processes.

Similarly for communication services, LIBI is designed to leverage available scalable services or fall back to efficient strategies. For example, when a scalable communication service is not natively available, LIBI can use the communication tree formed by the agent processes described above for efficient group communication.

4.1 LIBI API

LIBI provides services for both process instantiation and rudimentary data communication. We present the core LIBI API functions, simplified for presentation.

LIBI's primary abstraction is a *session*, which encapsulates the set of processes to be created. There are several abstractions related to the session: the *session member*, a process within the session, a *session master*, one session member designated to manage the others, and the *session front-end*, the process that launches the session. The session front-end communicates with the session master, who then in turn communicates with the other session members. Since LIBI targets only a scalable lightweight bootstrapping service, LIBI only supports mechanisms for communication between the session master and the other session members. In other words, non-masters cannot communicate directly with each other via LIBI. For bootstrapping, the goal is typically to disseminate some configuration information from the session front-end, to the the session members.

LIBI uses a *process distribution* to specify how and where to create the requested processes. A process distribution is a 5-tuple, <sid, exe, args, hd, env>, where sid is a session handle, exe is the path to an executable file, args are the arguments to pass to the executable during process creation, hd is a *host distribution*, and env is the environment to use for the created processes. A host distribution is a 2-tuple <hostname, num-procs>, which defines how many processes to create on the named host.

Given these abstractions, the core LIBI API functions are defined as follows:

launch(process-distribution-array) instantiates the appropriate sets of processes according to the input process distributions.

[send|recv]UsrData(session, msg) transfers data between the session master and the session front-end.

broadcast(sndbuf, nbytes) transfers `nbytes` bytes of data from `sndbuf` from the session master to all other session members.

[scatter|gather](sndbuf, nbytes, rcvbuf) transfers `nbytes` bytes of data from/to the session master to/from all other session memers.

barrier( ) blocks until all session member call this routine.

```
front-end() {
   LIBI_fe_init();

   LIBI_fe_createSession(sess1);

   proc_dist_req_t pd;
   pd.sessionHandle = sess1;
   pd.proc_path = get_ExePath();
   pd.proc_argv = get_ProgArgs();
   pd.hd = get_HostDistribution();

   LIBI_fe_launch(pd);

   //test broadcast and barrier
   LIBI_fe_sendUsrData(sess1, msg, len );
   //see session member pseudo code
   LIBI_fe_recvUsrData(sess1, msg, len);

   //test scatter and gather
   LIBI_fe_sendUsrData(sess1, msg, len );
   //see session member pseudo code
   LIBI_fe_recvUsrData(sess1, msg, len);

   return 0;
}
```

Figure 4: LIBI Front-end Code

4.2 A LIBI Example

In the following example, the two parts of the LIBI micro benchmark are outlined. The first listing outlines the pseudo-code for the session front-end. The second listing outlines the pseudo-code for the session members.

The code snippet in Figure 4 shows the creation and launch of a process distribution. Once the process distribution has been launched, this front-end code sends a message to the session master and receives a message in return

Figure 5 shows code snippet from a LIBI-based application that launched by a LIBI front-end. The LIBI_recvUsrData() and LIBI_sendUsrData() are acting as start and stop signals to and from the front-end. This allows the front-end to time

```
session_member() {
  LIBI_init();

  //test broadcast and barrier
  LIBI_recvUsrData(msg, msg_length);
  LIBI_broadcast(msg, msg_length);
  LIBI_barrier();
  LIBI_sendUsrData(msg, msg_length)

  //test scatter and gather
  LIBI_recvUsrData(msg, msg_length);
  LIBI_scatter(msg, sizeof(rcvmsg), rcvmsg);
  LIBI_gather(sndmsg, sizeof(sndmsg), msg);
  LIBI_sendUsrData(msg, msg_length);

  LIBI_finalize();
}
```

Figure 5: Code for LIBI-launched Application

the associated group communications.

4.3 Implementation Status

Our current LIBI prototype implements the interface described in the Section 4.1. In this section, we list the notable features (and shortcomings) of this first prototype.

Two launch mechanisms are available in our prototype of LIBI, srun and rsh. Both of these mechanisms are accessed via LaunchMon. While LaunchMon supports a wide variety of other launch mechanisms we have only tested with these two. Similarly, LIBI uses LaunchMON's communication interface; we have tested LIBI using LaunchMON's COBO PMGR service [10].

5. PRELIMINARY LIBI EVALUATION

The primary goal of our evaluation of an early LIBI prototype is to demonstrate the ease with which LIBI can be integrated into existing software infrastructures and the performance boost that can be attained by this integration. We first evaluate LIBI for doing some basic process launching and information dissemination. Then we describe the relative ease with which we were able to integrate LIBI into MRNet for bootstrapping that infrastructure.

We ran all experiments on Lawrence Livermore National Laboratory's Atlas system, a 1,152 cluster of AMD Opteron nodes. Each node has 8 2.4 GHz CPUs, and the nodes are interconnected via a double data rate InfiniBand network. To increase test scales, we placed up to 8 processes on a single node in our testing configurations.

5.1 Microbenchmark Experiments

Our microbenchmark experiments are based on the code snippets from Section 4.2. Summarily, we use our LaunchMON-based LIBI implementation to instantiate a session of processes on remote nodes and disseminate some information to all session members. We measure the (1) time to start the processes, (2) the time to broadcast 128 bytes of data (followed by a barrier), and (3) the time to scatter 128 bytes of data to each process and gather 128 bytes from each process. Figures 6 and 7 compare these times for increasing numbers of processes with the same operations executed using non-scalable operations, sequential `rsh`-based process creation and point-to-point communication operations. As expected, the LIBI-based, scalable versions of these operations scale very well up to our largest experiments, which entailed 2,800 processes. There are two note-worthy observations to be made from the sequential performance results: (1) as we already know, the performance complexity of the sequential operations is linear and, therefore, not scalable. (2) Barring performance issues, using centralized topologies are still infeasible at large scales due to resource constraints: the reason our sequential results for the communication-based experiments stop at a node count of about 1,000 is because of resource constraints, like the number of open socket connections, that restrict us from deploying larger test jobs[1]. While, clearly the hierarchical communication service upon which LIBI is based, demonstrates more scalable performance than a centralized one, it is worth noting that for our latency-bound communication experiments, at small scales a centralized topology will outperform a hierarchical one which may impose extra communication hops that are unnecessary at smaller scales.

[1]Our sequential launching experiments also stop 1,000 because we used the same experimental framework to test both launch and communication times.

5.2 MRNet/LIBI Integration

MRNet [11] is a software overlay network that provides efficient data multicast and reduction communications for distributed software systems. MRNet uses a tree of processes between the application's front-end and back-ends to improve group communication performance. The tree also is used to distribute important activities, like data reductions and data analyses, keeping front-end loads manageable.

In Section 3, we described MRNet's traditional bootstrap mechanism in which parent processes create their children processes using `rsh` in as concurrent a fashion as possible. We modified MRNet to use LIBI for creating the tree processes and for disseminating the topology information needed for children processes to form the tree-based overlay network by establishing connections with their parents. Previously, MRNet's start-up process integrated process launch and information dissemination: when a parent created its children, it passed on the command line the necessary port information the children needed to establish a connection with the parent. LIBI completely separates the process launch and information dissemination interactions. In the new LIBI-based MRNet the session master gathers the relevant information and scatters it to the other session members.

6. RELATED WORK

We acknowledge that the developers of many distributed infrastructures have implemented infrastructure-specific mechanisms for scalable boostrapping. Since this work targets a generic solution that can be leveraged by any distributed infrastructure, we do not further discuss these one-off solutions. We now describe how LIBI compares to other generic solutions.

The scalable and extensible launching architecture for clusters (ScELA) [12] was designed for scalable, extensible high-performance job launching. ScELA deploys a *node launch agent* (NLA) on each target node. The NLA is then responsible for starting each process for that target node. In contrast, LIBI aims to use the best launching service that may be available on a target platform. For example, if the target platform has a service based on persistent daemons that can be used for launching, LIBI can forego launching its own NLA-like processes and leverage that service.

Resource managers, like SLURM [5], PBS [8], LSF [9] and LoadLeveler [4], use persistent system daemons to provided scalable job creation and management services. For example, when SLURM receives a request to start a set of processes on a set of nodes, SLURM dynamically forms a tree to communicate the request and other control and response information amongst the relevant nodes. Generally, these services only provide job launch capabilities and not the services for scalable information dissemination. Additionally, the many different resource management services have

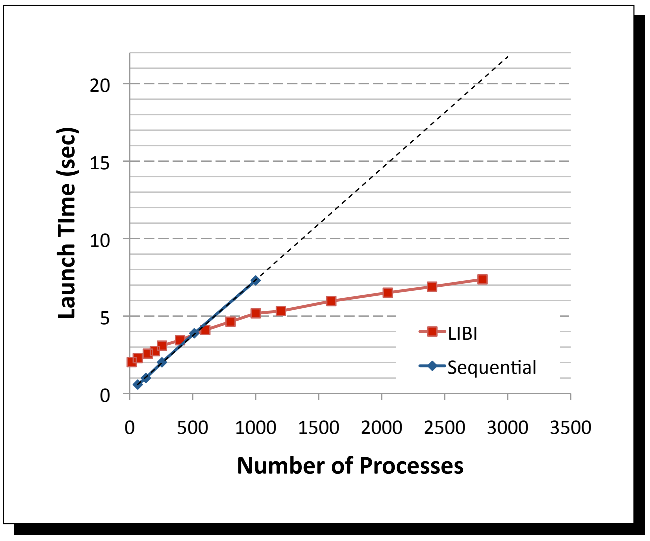

Figure 6: Microbenchmark Launching Results: Comparing the time it takes to launch jobs via LIBI v.s. a centralized (sequential) scheme.

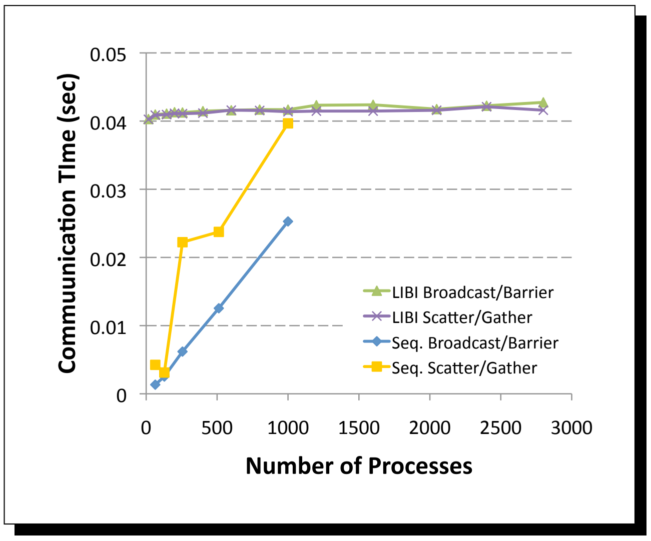

Figure 7: Microbenchmark Communication Results: Comparing group communication performance via LIBI v.s. a centralized (sequential) scheme.

different and incompatible interfaces. An infrastructure like LIBI is meant to rectify these differences making distributed infrastructures that want use these services more portable. LIBI would also provide an efficient bootstrapping mechanism for platforms where these services were not available.

7. DISCUSSION AND FUTURE WORK

In this paper, we presented the description and an evaluation of an early LIBI prototype. There are a number of key research and development issues that we plan to address in the future. We conclude this paper by highlighting some of these key issues.

Our current LIBI prototype built on top of LaunchMON supports only a subset of our planned launching and communication mechanisms. Next, we plan to implement an efficient `rsh`-based mechanism for environments in which LaunchMON is not available. This mechanism will use `rsh` to instantiate LIBI agents on the relevant nodes, form an efficient communication tree amongst these agents, and use these agents to launch the requisite application processes and communicate information amongst them. Additionally, we plan to implement our hybrid launch mechanism that will use these agents in environments, like those of the Cray XT series, where the ALPS resource manager only launches a single process per node.

Filesystem contention is a common problem in high-end computing systems. We plan to leverage mechanisms like the Scalable Binary Relocation Service (SBRS) [1] that we previously developed. When a common file (like an executable image) is needed on many processes on different nodes, SBRS alleviates filesystem contention by having a single process access the filesystem. The process then transmits the file or relevant portions of the file to the other processes in a scalable fashion. We will explore improving LIBI's launch efficiency with such a mechanism.

Currently, LIBI requires that applications input a list of previously allocated nodes in the form of the host distributions described in Section 4.1. We will design more flexible mechanisms that do not require a previous allocation. In this mode, LIBI will efficiently combine the acquisition of the necessary nodes and the instantiation of the specified processes.

A major goal of this project was to use our experiences with the LaunchMON experience to shape how distributed system bootstrapping should be interfaced and implemented. As its name suggests, LaunchMON provides facilities for both application/tool launching and application monitoring. One of the lessons learned is that while many tools need both these services, many tools and applications do not the need monitoring capabilities. We are in the process of revisiting LaunchMON's design to refactor and separate these two functionalities, with a possible outcome of LaunchMON's launching support mechanisms becoming a part of LIBI proper. One current challenge we face in this process is the fact that certain architectures, like BlueGene-based systems, make it difficult to separate launching from monitoring capabilities for the tools that need both.

8. ACKNOWLEDGMENTS

This work was supported in part by Lawrence Livermore National Security, LLC subcontract B590510. A part of this work was performed under the auspices of the U.S. Department of Energy by Lawrence Livermore National Laboratory under contract DE-AC52-07NA27344.

9. REFERENCES

[1] D. H. Ahn, D. C. Arnold, B. R. d. Supinski, G. L. Lee, B. P. Miller, and M. Schulz. Overcoming Scalability Challenges for Tool Daemon Launching. In *37th International Conference on Parallel Processing*, ICPP '08, pages 578–585, Washington, DC, USA, 2008. IEEE Computer Society.

[2] D. C. Arnold, D. H. Ahn, B. R. de Supinski, G. Lee, B. P. Miller, and M. Schulz. Stack Trace Analysis for Large Scale Applications. In *21st IEEE International Parallel & Distributed Processing Symposium (IPDPS '07)*, Long Beach, CA, March 2007.

[3] ASC Sequoia. https://asc.llnl.gov/computing_resources/sequoia/ (visited May 2011).

[4] IBM Tivoli Workload Scheduler LoadLeveler. http://www-03.ibm.com/systems/software/loadleveler (visited May 2011).

[5] M. A. Jette and M. Grondona. SLURM: Simple Linux Utility for Resource Management. In *ClusterWorld Conference and Expo*, San Jose, California, June 2003.

[6] P. Kogge. ExaScale Computing Study: Technology Challenges in Achieving Exascale Systems. Technical report, Defense Advanced Research Projects Agency Information Processing Techniques Office (DARPA IPTO), September 2008.

[7] G. L. Lee, D. H. Ahn, D. C. Arnold, B. R. de Supinski, M. Legendre, B. P. Miller, M. Schulz, and B. Liblit. Lessons Learned at 208K: Towards Debugging Millions of Cores. In *Supercomputing 2008 (SC2008)*, Austin, TX, November 2008. To Appear.

[8] PBS Works. http://www.pbsworks.com/ProductPBSWorks.aspx (visited May 2011).

[9] Platform LSF. http://www.platform.com/workload-management/high-performance-computing (visited May 2011).

[10] PMGR_COLLECTIVE. http://sourceforge.net/projects/pmgrcollective/ (visited May 2011).

[11] P. C. Roth, D. C. Arnold, and B. P. Miller. MRNet: A Software-Based Multicast/Reduction Network for Scalable Tools. In *2003 ACM/IEEE conference on Supercomputing (SC '03)*, page 21, Phoenix, AZ, November 2003. IEEE Computer Society.

[12] J. K. Sridhar, M. J. Koop, J. L. Perkins, and D. K. Panda. ScELA: Scalable and Extensible Launching Architecture for Clusters. In *15th International Conference on High performance Computing*, HiPC'08, pages 323–335, Berlin, Heidelberg, 2008. Springer-Verlag.

[13] Top 500 Supercomputer Sites. http://www.top500.org/ (visited May 2011).