Component-Based Cluster Systems Software Architecture: A Case Study

Narayan Desai, Rick Bradshaw, Ewing Lusk
Argonne National Laboratory, Argonne, Illinois 60439

Ralph Butler
Middle Tennesee State University, Murfreesboro, Tennessee 37132

Abstract

We describe the use of component architecture in an area to which this approach has not been classically applied, the area of cluster system software. By "cluster system software," we mean the collection of programs used in configuring and maintaining individual nodes, together with the software involved in submission, scheduling, monitoring, and termination of jobs. We describe how the component approach maps onto the cluster systems software problem, together with our experiences with the approach in implementing an all-new suite of systems software for a medium-sized cluster with unusually complex systems software requirements.

1. Introduction

This paper describes an application of the principles of component architecture to cluster system software. By "cluster system software," we mean the collection of programs that are used in configuring and maintaining individual nodes, together with the software involved in submission, scheduling, monitoring, and termination of jobs. For our definition of component software we adopt the principle presented by Szyperski in [**?**]:

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

Components as a guiding idea for software architecture have been applied in diverse areas, from scientific software for supercomputers [**?**] to business software for Windows [**?**]. Only recently, however, has the idea been applied in a comprehensive way to the system software that manages clusters.

The Scalable Systems Software Project [**?**] is studying the feasibility of using a component approach to structure systems software for highly parallel computers. Some of us are participants in that project, in which specific component implementations are being developed. Others of us are responsible for the day-to-day operation of the Chiba City cluster [**?**] at Argonne National Laboratory, which as a cluster dedicated to scalability research in computer science provides a number of challenges in the area of systems software. In this paper, we describe the merging of these two efforts, as an abstract system architecture with public interfaces among components contributed by various members of the community is deployed on a real cluster and put into production use. This experience provides an opportunity to evaluate the benefits and drawbacks of the component approach applied in the area of cluster systems software.

In Section 2 we expand the definition of components and summarize the claimed benefits of the component approach. In Section 3 we describe the current state of cluster system software, which is distinctly *not* component based. In Section 4 we motivate the design and deployment of an entirely new "system software stack" on Chiba City. In Section 5 we describe the component architecture deployed. In Section 6 we discuss our experiences, and in Section 7 we present our conclusions and outline directions for future work.

2. Expected Benefits of a Component Architecture

As described in [**?**], a component approach to software architecture has several advantages. First, the localization of functionality encourages reuse. That is, having a single implementation of functionality reduces

code volume and leads to more reliable functionality. Second, a particular implementation of a component may evolve or be replaced by a better instantiation without affecting the architecture of the overall system or the implementations of other components. Here, "better" may mean "more efficient," "simpler," "less expensive," or the like. One benefit of this substitutability is that suitability metrics and decisions are in the hands of the component integrator rather than the author of the monolithic system, and thus are closer to the end users. Third, the functions of disparate components can be assembled in ways not envisioned by their implementors. This process is referred to as third party composition. In our case, the components came either from contributors to the SciDAC Scalable Systems Software project or from the system administrators; the component architecture allowed easy combination of components from multiple sources.

3. Monolithic System Software Architecture

Generally, cluster system software has consisted of groups of monolithic applications without abstract, public interfaces. OpenPBS [**?**] exemplifies such a monolithic application suite. Internally, it implements queue management, scheduling, process management, and some monitoring capabilities. Of these, only the scheduling portion of OpenPBS's functionality can be disabled or replaced.

Software providing monolithic suites of functionality poses problems for system managers. Since a single piece of software provides several different types of functionality, advances in any one of these areas cannot be utilized effectively. For example, both the MPICH2 team and the LAM teams have put substantial effort into parallel process management over the last few years. These improved solutions are typically layered inside OpenPBS's process management system. While this approach allows improved functionality in some cases, performance benefits cannot be realized because two process management systems are in use.

Another issue with monolithic software suites is accessibility to internal functionality. As mentioned previously, process management functionality is provided by OpenPBS. Unfortunately, no comprehensive interface is provided for external usage. Hence, multiple process management facilities are used in different cases. Not only is this approach inefficient from an implementation perspective, but it also leads to subtleties where none are required. Multiple implementations of any subsystem will have differences in functionality, interface semantics, and implementation quirks. This is obviously a suboptimal solution.

We do not intend to target OpenPBS specifically; POE from IBM and SGE from Sun have a similar architecture and hence experience similar problems, as do other resource management suites. Furthermore, these systems have varied amounts of integration with other system software components because of the lack of standard component interfaces.

4. Motivations

Chiba City is a medium-sized testbed cluster at Argonne National Laboratory. Unlike many clusters, it is primarily dedicated to computer science research, as opposed to computational science, and hence faces a number of issues not faced by computational clusters [**?**]. The first is user-level configuration management. Chiba City is used for all levels of system software development, ranging from development of kernel and driver software to high-level library and application development. For this reason, configuration management demands on the system are more extensive than in the usual computational cluster model. Users must be able to specify the compute node software configuration in a reasonable fashion. This task is especially difficult because most users are not familiar with system administration issues.

In order to operate a system in production with user-specified configurations, these configurations must be automatically deployable, without system administrator intervention. Hence, the resource management system must be able to reliably initiate and complete compute node reconfiguration operations during individual jobs, based on user input. Moreover, the fact that users often need elevated privileges has a variety of implications for system configuration, both on nodes and system wide. This requirement also affects how the overall system software stack works. Sensitive data cannot be left on nodes when a user is given root access, and nodes must be rebuilt automatically after jobs complete. This again requires interactions between the resource management and configuration management systems. These issues and others are discussed in more detail in [**?**].

4.1. Previous System Software Stack

The initial implementation of Chiba City's system software stack had two operational modes for nodes. In one case, nodes participated in the cluster as standard computational nodes, with complete integration into the cluster and central management. In the other

case, nodes were partially disconnected from the cluster system software stack; although in many ways the nodes looked the same, the resource management stack treated the nodes as down. This mode allowed suitable configurations to be produced manually for users on a case-by-case basis. When ad-hoc usage was finished, nodes were manually rebuilt and reintegrated into the scheduled pool. While these reconfiguration steps were partially automated, they still relied on manual administrator initiation.

4.2. Scalable System Software

One of the primary reasons we chose to use a component architecture when implementing this system software was our involvement with the Scalable System Software (SSS) project [**?**]. Substantial effort had already been put into developing a component architecture and an infrastructure to aid in inter-component communication. Implementing a system software component architecture from scratch is difficult and time-consuming. Had this not already been completed, the decision to use a component architecture would not have been nearly as appealing.

5. Component Design and Implementation

In this section we describe the component-based systems software that has actually been deployed on our cluster. It consists of an infrastructure for component interaction, some components that we have contributed to the SSS project (see [**?**], for example), and some instances of components that are part of the SSS architecture but have implementations by us that are specialized to the Chiba City environment.

5.1. Scalable Systems Software SciDAC Design

The Scalable Systems Software project has undertaken to design a component-based architecture for cluster system software and to provide a prototype implementation. The set of components and their relationships are shown in Figure 1. The Scalable Systems Software project early on decided that components would communicate over sockets using XML messages, and much of the group's effort so far has gone into defining the exact format and content of such messages. The current status of this standardization effort can be found at the Web site [**?**], where proposed standards for each component's API are presented. The following is a list of components defined by the Scalable System Software APIs.

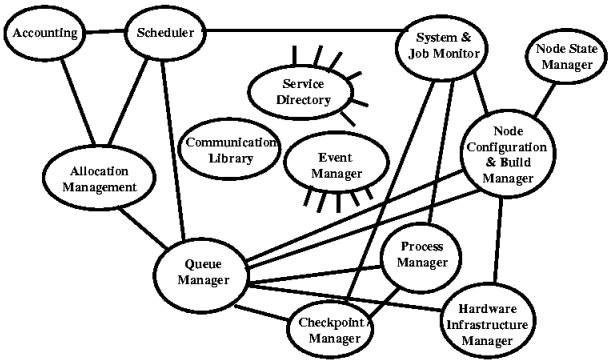

Figure 1: Scalable System Software Components

- Service Directory: stores location information for all active components.
- Event Manager: delivers asynchronous events based on client-defined subscriptions.
- Node State Manager: tracks node operational and administrative states and provides access to system diagnostic capabilities.
- Node Configuration and Build Manager: provides configuration management and node build functionality.
- Hardware Infrastructure Manager: provides access to power controllers and node consoles.
- Process Manager: provides all functions needed for parallel process group management.
- Queue Manager: queues and executes user jobs.
- Scheduler: allocates resources based on system policy.
- Allocation Management: associates users with projects and resource allocations for resource usage tracking.
- System and Job Monitor: aggregates monitoring data.
- Accounting: provides summaries of resource usage.
- Checkpoint Manager: provides job checkpoint control and associated data movement functions. (This is the only component that does not yet have an instantiation on Chiba City.)

5.2. Infrastructure

We have implemented, both for the SSS project in general and for Chiba City in particular, an infrastructure that makes it convenient for components to register with a global service directory, find other components, agree on line protocols, and then exchange messages with other components. Most components use widely available XML parsers, and the communication library has bindings for components written in Python, C, C++, Perl, and Java. This infrastructure supports the convenient integration of large components (such as a version of the Maui scheduler, or the MPD process management system) with PBS-compatible queue managers, node monitors, configuration managers and with accounting systems.

5.3. Implementation

A large portion of the Chiba City system software stack consists of component implementations from the Scalable System Software project. However, local requirements necessitated the implementation of a site-specific queue manager, a scheduler, and a file staging component.

The queue manager implemented in the Chiba City testbed provides native support for jobs that change node configurations on the fly. This model differs substantially from the usual job model implemented by queue managers; in most cases, nodes are assumed to remain up available to a process management system throughout the execution of a job. In our case, the queue manager needed to have an internal representation of different user-specified configuration management operations and an associated set of observable characteristics of nodes in the process of reconfiguration. This is a substantial difference from the typical model.

The second component we implemented locally was a scheduler. In this case, we needed a simple scheduler that accommodated the testbed operational mode. That is, the scheduler needed to allow down (or rebuilding or rebooting) nodes to participate in active jobs or reservations. Our scheduler provides basic scheduling functionality, including backfill, and uses some node state management functionality to track nodes through reconfiguration operation.

The last locally implemented component provides file staging functionality. Chiba City doesn't have a global shared file system, and the SSS architecture doesn't specifically deal with this issue. When specifying job parameters, users can describe the data needed for their job, and it will be propagated to nodes executing the job. This component doesn't exist in the Scalable System Software design; however, the inherent extensibility of the component architecture easily allowed its addition.

6. Experiences

Overall, the component deployment and administration process has been very positive. Through the last nine months, we have realized many of the benefits and the two main shortcomings predicted by component architecture proponents. This overall conclusion suggests that our implementation of a component architecture is comparable to those described in component architecture literature [**?**]. In this section we focus first on issues involving component development. We then discuss our experiences as they relate to system administration.

6.1. Development

Since the component architecture approach was developed in the software development community, the assessments offered by its proponents directly apply to systems software development. Particularly, we benefitted from component substitutability and localization of functionality. Development using component architectures was not a uniformly positive experience. Component architecture programming is similar to multithreaded programming, hence, many subtle issues can cause problems. Also, changing component APIs is a time consuming and difficult process when required.

Component substitutability allows development to become separable on a component by component basis. When bugs or functionality shortcomings become apparent in the systems software, development can focus on the single component requiring work. Work on other component implementations can continue independently, since the interfaces are standardized. This separation requires developers to maintain familiarity with smaller volumes of code; only API knowledge is required for other components.

Localization of functionality reduces overall code volume, improves reliability, and eases component reimplementation. All of these benefits are exhibited by the contrast between process manager implementations used on Chiba City. In the original Chiba City software stack, several independent implementations of process management functionality were used simultaneously. OpenPBS used its own daemons to start job-related processes. User job processes were started internally with `mpirun`, which used `rsh` internally. Configuration management and ad hoc pro-

cesses were started by using `pdsh`, which also used `rsh`. Each of these tools contained independent implementations of process management functionality, with different feature sets and idiosyncrasies. The OpenPBS process management mechanism wasn't accessible outside the context of jobs. The `mpirun` command provided parallel process group bootstrapping required by MPI processes but didn't provide exit codes from individual processes. The version of `mpirun` also varied depending on the implementation of the MPI library; other versions had different tradeoffs. The parallel command `pdsh` didn't provide parallel startup bootstrapping. Without a feature-based motivation, using multiple implementations of the same functionality is rarely a good idea. Moreover, each of these implementations will suffer from bugs in different ways, will perform at different levels, and will require independent implementation of new features.

The structure of the current process management system used on Chiba City differs substantially. All process management functions are provided by a centralized process manager, based on MPD [**?**]. A `pdsh` analogue, called `dsh` has been implemented with a similar feature set. As all complicated functionality is implemented in the process manager, `dsh` consists of simple code performing argument handling. Similarly, user-job processes and MPI processes are also managed using the process manager. This approach allows performance and feature enhancements to seamlessly propagate to all users of the process manager. Bug fixes need only be implemented once. This approach has yielded a large reduction in code volume. Client code, for example `dsh`, is simple and small. Finally, consolidation of process management functionality has improved its reliability and usability.

Component architecture benefits are appealing to developers of system software; however, they come at a price. We experienced several problems during development. The first of these is the need for proper synchronization and coherence assumptions between components. Programming for component architectures is similar to multithreaded programming. If implementors aren't careful, race conditions can easily appear. Problems caused by implementation errors in one component can propagate to other components in unexpected ways, causing serious, larger system software failures. Overall synchronization must be maintained.

A second, and related issue is congestive failure. Since a component's underlying behavior isn't specified outside of the API and high-level functionality, components may block during API operations. If this occurs, the client may also block, as its implementation is also not specified. If this client is itself a component implementation, then the client's clients may block. This pattern can easily cause deadlocks. If one were to construct a directed graph of all components active in the system, with directed edges between a component and other components whose interfaces it consumes, deadlocks could occur whenever the graph contained cycles. Since component implementors determine which external interfaces are consumed, deadlocks can be detected only when considering a complete set of system software component implementations.

Deadlocks can be avoided by careful implementation design and are mitigated by putting timeouts in all stages of communication operations provided by the latest version of the infrastructure library. A particularly bad case can occur if a component's API operation blocks while the component executes an operation in another component. In this case, blocking behavior can propagate through the system much more easily.

We encountered a congestive failure case during the development of the service directory. Originally, component registration sent an event via the event manager in a synchronous fashion. This approach worked well, except when the event manager registered. As the event manager was already blocking in registration, the service directory blocked while attempting to send the event. The whole system came to a halt, as the service directory and event manager are at the center of all operations in the component infrastructure. At this point, any component needing to communicate with the service directory would block. As more components are stuck in this state, other components can be drawn in. With the addition of timeouts in communication operations, the system could eventually right itself, but this could potentially take the timeout value for each component locked up. In addition, it is appealing to implementors to retry failed operations, so it is possible for the system to plunge back into the locked state after emerging.

A third issue arises from the fact that, while implementations' interfaces are fixed, the underlying behavior of the components is not fixed in a rigorous way. The high-level behavior of the component is specified and defined in terms of the component API. The underlying implementation of this functionality is unspecified, in order to restrict future implementors as little as possible. This means that the specific behavior of a component may vary from one implementation to the next. Implementations of two components may interact in unexpected ways if the public interfaces do not capture all relevant aspects of a component's semantics.

The solution to this third issue is to design components that do not block during operations. This can be partially achieved at the communication protocol level; however, components also need to be carefully designed. Moreover, some amount of interoperability testing is required for a complete suite of system software. Contractually defined interfaces merely reduce this requirement; they don't remove it entirely.

Finally, component interface changes are required in certain cases. Interface changes are extremely problematic, as they generally require a code changes to all clients and a synchronized upgrade. This process can be hindered by the distributed nature of component development; all consumers of a component's interface may not be known. However, in practise this issue doesn't present itself. Fortunately, we have not needed to perform component interface changes often; over time they have stabilized to the point that we are able to treat them as static.

6.2. System Administration

The operational characteristics of a cluster operated with component systems software are considerably different from those of a cluster operated with monolithic software. Both system administrators and users are affected, since both regularly interact with the system software stack. At a high level, both benefit from increased system software simplicity, transparency, and agility.

System administrators are primarily concerned with the system software operating correctly and reliably. This process consists of finding a suitable software stack, configuring it properly, debugging problems, and helping users get work done. In many cases, finding a suitable software stack can be quite difficult. Many monolithic suites comprise complex, highly configurable software. These developed because it seemed easier to incrementally add support for all users than to have multiple instances of components. In large part this attitude can be traced to the complexity in communicating with multiple other software entities, without common interfaces. With the addition of a set of common interfaces, this argument disappears, and small, tuned components become much more viable.

The assumption with monolithic software is that it can be reconfigured for any scenario. If this isn't the case, problems can develop. Even if the software can be reconfigured appropriately, it is often difficult to do so, as the configuration is so complex. When using component systems software, a set of appropriate components can be chosen based on their feature sets and operational model. If an appropriate instance of a component doesn't exist, then well-formed interfaces allow for easy reimplementation. For example, the resource management suite for Chiba City, tuned for the testbed environment, consists of less than 1,000 lines of code. We decided that having a simple scheduler, implementing FIFO with reservations and backfill, would be adequate for our needs, and we were able to implement a queue manager that properly interacts with the configuration management system. We were also able to easily add support for a file staging system, since Chiba City doesn't have a global shared file system.

Second, system administrators benefit from increased component transparency. In order for componentized systems to function at the same level as their monolithic equivalents, component interfaces must include exposure of all useful data so that external components can work properly. For example, the process management interface must provide all of the information necessary for its consumers to function properly. In this case, a variety of information about running and finished processes is exposed. Clients can query process groups based on criteria including effective user ID, process ID, execution host, and session ID. Information provided by legacy process management environments are not nearly as rich. This richness allows for the formulation of complex queries, impossible to predict at the time of component implementation. Moreover, when considering the associated possibilities for third part composition, administrators can associate information from multiple components. During normal operations, this means that administrators have complex tools at their disposal for both one-time problem determination and automation of complex tasks. One example is the location of all of a single job's processes across the entire system for the purpose of monitoring. Information from the scheduler, queue manager, and process manager can be used to locate all processes. The scheduler can provide all active resource allocations for the job. The queue manager can provide a list of all process groups active for that job. The process manager provides a list of hostname and process ID tuples for the process group. This sort of query is a simple example of the functionality that component architectures expose for all users. The exposure of such information allows sites to compose real-time information in ways systems software implementors were completely unable to predict.

Third, component systems software benefits from a substantial improvement in agility. Localization of functionality and a shared component infrastructure dramatically reduce the size of components. Since the effort required to maintain and modify code increases

greatly with the size of code, components are easier to develop or modify locally than their monolithic equivalents. Because of this, system administrators are empowered in a way that is impossible with monolithic systems. Since component and client code is simpler, it can be more easily modified based on local requirements or problems experienced. This simple difference causes an important change in attitude. With monolithic suites, administrators accept bugs and problems because the cost of creating and maintaining fixes is just too high. In contrast, the cost of modifying components is lower and component reimplementation is enabled. Lowering the cost of localized system software development can substantially alter the experience of system management.

7. Conclusions and Further Work

Component architectures have a variety of useful properties for implementors and users of system software. Overall, our experience with component architectures on Chiba City has been positive. The complexity of the system software has decreased while agility and reliability have increased. Development of system software has been eased by compartmentalization of function and independence of deployment. Moreover, cluster operations have been improved by increased transparency and adaptability.

We believe that the use of component architectures in system software development is a promising methodology for both development and usage. As use of this approach increases a larger variety of component implementations will become available. The current Scalable System Software reference implementation is available packaged as an OSCAR distribution, and public release of our testbed components is forthcoming. Administrators will be able to choose an appropriate set of system software for their environment, without the need for local development. In the case where local requirements are unique, development is aided by a complete set of documented, public APIs.

Acknowledgments

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