A Runtime Environment for Supporting Research in Resilient HPC System Software & Tools

Geoffroy Vallée, Thomas Naughton, Swen Böhm, and Christian Engelmann
Oak Ridge National Laboratory
Computer Science and Mathematics Division
Oak Ridge, TN 37831, USA
Email: {valleegr, naughtont, bohms, engelmannc}@ornl.gov

***Abstract*—The high-performance computing (HPC) community continues to increase the size and complexity of hardware platforms that support advanced scientific workloads. The runtime environment (RTE) is a crucial layer in the software stack for these large-scale systems. The RTE manages the interface between the operating system and the application running in parallel on the machine. The deployment of applications and tools on large-scale HPC computing systems requires the RTE to manage process creation in a scalable manner, support sparse connectivity, and provide fault tolerance. We have developed a new RTE that provides a basis for building distributed execution environments and developing tools for HPC to aid research in system software and resilience. This paper describes the software architecture of the Scalable runTime Component Infrastructure (STCI), which is intended to provide a complete infrastructure for scalable start-up and management of many processes in large-scale HPC systems. We highlight features of the current implementation, which is provided as a system library that allows developers to easily use and integrate STCI in their tools and/or applications. The motivation for this work has been to support ongoing research activities in fault-tolerance for large-scale systems. We discuss the advantages of the modular framework employed and describe two use cases that demonstrate its capabilities: (i) an alternate runtime for a Message Passing Interface (MPI) stack, and (ii) a distributed control and communication substrate for a fault-injection tool.**

I. Introduction

The runtime environment (RTE) is an important layer in the high-performance computing (HPC) software stack. The RTE sits between the operating system and the application, interfacing with both. The runtime, in coordination with the job manager and the resource manager of the platform, is traditionally responsible for the creation, monitoring and termination of processes on compute nodes in the context of a given run of an application. Therefore this layer must be able to accommodate a variety of platform specific interfaces, e.g., to job and resource managers. The RTE also must provide appropriate support for applications and higher-level interfaces, e.g., the Message Passing Interface (MPI).

As part of our research in system software and resilience for HPC we have developed a runtime called, Scalable runTime Component Infrastructure (STCI). This RTE provides a basis for building distributed execution environments and developing tools for HPC. Many of the challenges that are emerging in HPC require enhancements and customizations at the RTE level. For example, the ever-increasing size of large-scale systems raises concerns regarding decreasing system mean time to failure (MTTF) rates. The HPC community, both application scientists and system developers, has to review the software stack to identify proper mitigation strategies. This point was highlighted in recent reports that indicated an increased need by runtimes to support HPC application resilience [1], [2].

Changes to improve scalability and resilience will result in new interfaces and the RTE must be adapted to support such enhancements. However, existing/legacy interfaces must be maintained as well. Therefore a modular and extensible approach is needed. The expertise required to develop and the overhead to maintain runtime software also mandates such a design in order to assemble the RTE building blocks to tailor the system for different use cases. As such, the STCI system is component based and leverages the Modular Component Architecture (MCA) infrastructure to allow different services to be constructed by combining individual components. This MCA functionality is the same component infrastructure that was developed for flexibility in the Open MPI project [3].

In this paper we provide an updated summary of the software building blocks offered by STCI. We review the current design and highlight features of the implementation. We discuss the advantages of the modular framework employed and describe two use cases that demonstrate these modular building blocks: (i) as an alternate runtime for an MPI stack, and (ii) as a distributed control and communication substrate for a fault-injection tool. The main contributions are:

- An updated description of the STCI runtime;
- Details on capabilities offered by the RTE; and
- Two use-cases showing support for HPC resilience research.

The remainder of this paper is organized as follows, in Section II we discuss the software architecture and in Section III provide details on the current implementation. We describe the two use-cases and initial evaluation results in Section IV. In Section V we highlight related work on HPC runtime environments. Finally, in Section VI we provide concluding remarks and plans for future work.

II. Software Architecture

HPC systems can vary deeply in terms of configuration: some are based on a job scheduler for the assignment of an "allocation" to a user for the execution of a run of their application (users never have direct access to the compute nodes; all tasks are performed via the job manager), while some other platforms provide direct access to compute nodes (for instance via SSH). Because STCI aims at being scalable,

the creation of execution contexts (typically processes) across compute nodes is particularly important. For instance, if the execution contexts are created in a linear manner, the startup time will become prohibitive on very large scale systems, e.g., with 100,000 compute nodes.

The second main challenge for the implementation of a scalable process management infrastructure is fault tolerance: because of the scale of the target systems, many distributed hardware components are involved in the execution of a job and the probability of a failure increases with the size of the job. In the context of this study, we do not aim at providing fault tolerant mechanisms for the job itself, but we intend to detect failures, report them, maintain communication channels between the different execution contexts of the job, and give the opportunity to users and/or applications to decide the best policy to apply (from clean termination of remaining job's processes to automatic recovery). To achieve this, the STCI architecture is based on the notion of topologies that abstract the management of communication channels inside the deployed infrastructure, as well as, mechanisms for fault detection, process restart and fine-grain management of communication channels.

Furthermore, since HPC platforms can be very different in nature (both in terms of hardware and software configuration), it is important to be able to customize and optimize the software environment to a given target HPC platform. To provide this feature, STCI is based on a modular architecture in which "frameworks" implement certain software capabilities and are composed of "plug-ins". As a result, by selecting different plug-ins, it is possible to adapt both the system mechanisms and policies that are internally used in STCI. Section III-A gives more details about the design and implementation of the modular architecture.

A. Agents

In order to separate the system aspects (such as resource allocation) from the job management, the STCI architecture is based on the concept of agents. Three different types of agents have been defined: root agents (typically system agents), session agents (specific to a job), and tool agents (specific to a "tool", a tool being a self-contained part of a job, e.g., one of the binaries of a job when the parallel application is composed of different sub-applications). For simplification, we refer to agent instantiations as processes but STCI has been designed to support different instantiation methods, for instance via threads (see Section III-C).

Root agents are in charge of resource allocation and release. Thus, these agents are privileged agents. Only one root agent is on each compute node and is used to deploy other agents (both session and tool agents). Root agents may be shared between jobs.

Session agents are in charge of instantiating a job on a given compute node. This is not a privileged agent and it acts on behalf of the user. A single session agent is deployed on compute nodes of a given job allocation. In some configurations, the session agent may be used to deploy tool agents.

Tool agents are instantiating the job itself, multiple tool agents can be deployed on compute nodes of a job allocation, and all tool agents act on behalf of the users. These are generally the end-user's executable, e.g., a MPI application binary.

In addition to these agents, a *Controller agent* is running on the HPC system. The Controller is in charge of creating an internal representation of a job. The Controller is also responsible for coordinating the deployment of the different agents and the creation of communication channels between the agents. The communication channels are organized based on *topologies* (e.g., trees and meshes). These topologies describe how all the different agents, i.e., the Controller, the Root Agents, the Session Agents, and the Tool Agents can communicate with each other. Figure 1 presents an example tree-based topology. Topologies are also used to set routing tables (which are then used to send messages from one agent to another).

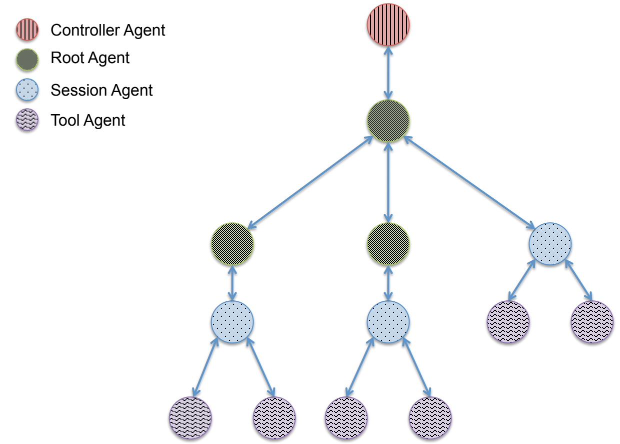

Fig. 1. Example STCI Topology that connects agents in a tree-based fashion.

Finally, a *Front-end agent* runs on the user's machine or on the HPC system login node, and interacts with the Controller. This separation enables the implementation of advanced mechanisms, so that the Front-end can connect/disconnect to/from the HPC platform without compromising the execution of the user's job.

B. Topologies

The concept of topology is key in the STCI architecture: it describes how the different agents are connected to each other and how messages should be routed during communications between agents.

STCI implements a few key topologies: trees, meshes, and binomial graphs (BMGs) [4]. For instance a k-ary tree is used to deploy agents across the compute nodes, providing a scalable startup. For fault detection, a mesh topology is used for the detection of root agent failures. Finally, a BMG topology is instantiated for the deployment of a fault tolerant communication infrastructure by providing redundant communication channels between agents. All the topologies are using the communication substrate to instantiate the links between the agents (see Section II-E).

Finally, STCI provides a boot topology which describes how the agents of a given job are connected to each other. This allows us to have sparse connectivity, and therefore ease

the implementation of scalable capabilities. Currently, the boot topology is a k-ary tree.

C. Launcher

For the deployment of a job, STCI is implementing a *launcher*. If the system is configured based on a job manager, the launcher typically interacts with it for the deployment of the agents. For example, on Cray supercomputer systems, Torque and Application Level Placement Scheduler (ALPS) are respectively in charge of allocating resource to a job and deploying processes on compute nodes. In such a context, STCI first queries Torque to get a list of nodes and then interact with ALPS for the creation of the root agents on compute nodes. Once the root agents are in place, communication channels are setup between some root agents and the controller based on the boot topology. Then, it is possible to send information to the different root agents so the session and tool agents can be created.

Two challenges are raised for the development of any launch policy: what is the best method for the deployment of all the agents on a target platform to guarantee scalability (e.g., start root agents first using a tree and then ask the root agents on each node to start local session and tool agents)? And, how will the agents actually be created (e.g., using SSH or ALPS)? To address this challenge the launcher capability is separated into two different entities: a low-level launcher (called *launcherproc*) that is in charge of deploying a process based on a known method (e.g., SSH), and a "driver" that orchestrates the deployment of agents using one or more launcherproc plug-ins and implementing a policy that is scalable in a given context (called *launcher*). By separating these two aspects, it is possible to easily support new mechanisms for the creation of processes (e.g., ALPS) and implement new policies (it is possible to easily develop a new launch policy optimized for a target platform).

D. Event System

The runtime includes an event system to support the operations that are performed while managing the execution environment. This event driven model provides a good basis for providing fairness when servicing requests and supporting an asynchronous execution model. An event is an abstraction that ties some occurrence in the system to an action. The action is defined in an event handler. A dispatcher is responsible for running these event handlers when the event is triggered. When an event gets triggered an event activation object is assigned to the dispatch queue for that particular event's domain. An event domain is a container that keeps track of all events and associated observers that registered as a recipient for these events.

These elements are combined to create the event loop that drives the system forward. A STCI agent may choose different modes for entering this event loop based on how work should be performed. How the agent chooses to enter the event loop defines the progress model for that agent. The agent may choose an *implicit progress* or *explicit progress* model. The explicit approach makes progress via direct invocation of a progression function at appropriate locations in the implementation, whereas the implicit approach defers progress to the event system and all progress is based entirely on handling events. The implicit model is often an effective mode for daemon or system processes that should only perform work in the presence of events and otherwise idle in the background.

E. Communication Substrate

The STCI communication substrate is composed of two different communication sub-systems: a bootstrapping communication substrate named *bootcom* and an active message communication substrate.

Note that we do not provide any high performance communication substrate, which is a completely separate challenge in the context of large-scale HPC systems. However, it is possible for application developers to use STCI for the deployment of tools and applications, and then extend the tool agents to initialize a third-party high-performance communication subsystem (STCI has been designed to not generate any system noise outside of the process management tasks in order to avoid degrading the performance of any high-performance networking solution).

1) Bootstrapping Communication Substrate: The STCI bootstrapping communication substrate is based on the following requirements: (1) it must be self-bootstrapping; (2) communications are reliable and ordered; (3) it must support sparse connectivity; (4) it must implement fine-grained state monitoring of all communication links between agents to avoid sending messages to an agent that already terminated or was previously reported as failed; and (5) it must implement asynchronous communications.

To initialize the communication substrate in the context of an agent, the launcher passes "parent" information along with bootstrapping data. The parent is defined as the parent agent in the tree based boot topology. Based on this information, during the agent bootstrapping, a communication link is opened with the parent, and then the two agents perform a handshake. Also, all agents maintain a routing table for the communication substrate. When a message is received from an agent and if this is not the final destination, we lookup the next hop and forward the message. During the bootstrapping phase of an agent, the parent is added to the routing table as the default route.

2) Active Message Communication Substrate: STCI provides different active message interfaces, which are specialized for communication at different levels of the STCI architecture. They all have the following requirements: (1) communications are reliable; (2) blocking/non-blocking send; (3) avoid copies when the user is not ready for the data, that is, memory needs to be left in the library buffers, with users releasing it, if this is not done within the call to the AM function. (4) it must support sparse connectivity; and (5) it must implement asynchronous communications.

The three different APIs are: (1) *point-to-point, non-routed fragment-based communications*; this layer is used to support the upper-level AM layers; (2) *point-to-point, routed messages-based communications*; this API can be for routed communications (message granularity instead of fragment granularity); (3) *stream based, routed message-based communications*; a stream being defined as an overlay network connecting different agents. A stream typically relies on a topology to describe

how agents are connected. This topology will also ultimately drive the routing of messages between the peers. Currently for simplification, it is assumed that a stream is based on a tree topology and the communications over a stream are based on an up/down premise.

a) Low-level Active Message Layer: As stated earlier, the granularity of the communication is a fragment. No routing is performed, instead, fragments are sent on a direct connection between two peers. At the moment, it is assumed that this layer will perform fragmentation.

b) Mid-level Active Message Layer: In contrast to the low-level active message layer, the communication granularity is a message that is routed between peers. The main responsibility is this layer is to provide message routing.

c) High-level Active Message Layer: This layer provides the support for stream-based communications using the up/down primitives. The routing component is in charge of defining the route over a user-defined topology. Note that STCI provides a default stream where the root of the stream is the controller and the leaves are all tool agents of the job. This stream is used, for instance, for I/O forwarding. Users can also create additional streams using a dedicated stream API.

F. Fault Tolerance

For fault tolerance purposes, we provide two capabilities: failure detection and a fault tolerant topology. These two capabilities ensure that even if a node fails or if an agent fails, it is still possible to send/receive messages to/from agents that are still alive. This allows the user to decide the best policy to apply in the context of a failure, for instance, triggering the clean termination of remaining agents.

1) Fault Detection: A key point to tolerate failures is first of all to detect failures. STCI provides support for both, inter- and intra-node fault detection. By detecting failures, it is possible to update routing tables and eventually re-establish communication channels between agents to ensure that we can still control alive agents. In that context, we propose a set of *detectors*.

a) Detectors for Inter-node Fault Detection: A mesh-based detector that establishes connections between root agents and reports an error if a connection is closed. A separate detector establishes connections between root agents based on a mesh topology and performs a periodic ping-pong. If the ping-pong fails, a failure is reported.

b) Detectors for Intra-node Fault Detection: A wide range of detectors could be implemented for the detection of process or thread failure. STCI currently provides a detector for the failure of processes, which is a signal-based detector that can be used to detect the local failure of any session or tool agent (when they die, the generated SIGCHLD signal is caught).

2) Communication Fault Tolerance: Since our boot topology is tree-based, the failure of any agent will prevent communication between different parts of the tree, leading to unreachable agents. To address this issue, we setup a BMG topology [4] that provides redundant communication links between agents. As a result, even if communication channels are closed because of a failure, it is possible to find another route to reach the destination, wherever the destination is.

3) Failure Notification: The failure notification sub-system is designed to propagate any local notification from detectors. This component aims at abstracting the way the propagation is made. For instance, a type of detectors may rely on a broadcast notification method, whereas others may rely on point-to-point communications or a tree-based fan-in/fan-out method. Each detector should register a notification method. For simplification, we assume at the moment that a detector is using one and only one notification mechanism. Since multiple types of detectors can be active at the same time, multiple notification mechanisms can also be active at the same time.

4) Error Manager: The *error manager* is in charge of implementing the consensus policy for failure recovery. This covers simple policies such as cleanly terminate the rest of the job to more advanced policies that restart and/or reconfigure the RTE infrastructure to cope with a detected failure. As previously stated, the goal is not to recover the application but the RTE infrastructure in order to guarantee correctness of all system-level services that the application may require. Like any policy in a distributed or parallel environment, it can be decomposed into two parts: (i) the part that receives the data based on which actions are decided on (failure notifications) and decides a set of actions (decision making) and (ii) the part that implements such actions on a given node. The decision making software component is called the Global Recovery-Manager, while the component performing the actions is called Local Recovery-Manager. Even if the logical separation can always be made, a simple policy may implement the two together. In case the global and local components are implemented separately, the communication requirements between the two is based on the design of the policy. Therefore, we do not provide any generic interfaces; we assume existing communication mechanisms are used.

III. Implementation

This section presents details about the current STCI implementation. We highlight the aspects that are relevant in the context of scalable and fault tolerant process management.

A. Modular Architecture

The STCI runtime is based on a library initially implemented in the context of the Open MPI project [5]. The Open Portable Access Layer (OPAL) library allows to abstract some of the low-level system APIs and provides some generic classes such as graphs and lists. We also reuse the Modular Component Architecture (MCA) from OPAL, which provides basic capabilities for the implementation of modular architectures based on plug-ins that can be selected, loaded, and parameterized. Since the normal MCA framework performs a selection of applicable plug-ins at load time, we implemented an extension for a dynamic selection of plug-ins based on parameters. For instance, topologies are implemented via a framework where each type of topology (at the moment, trees, meshes and BMG) are implemented through a plug-in. On top of this framework, we implemented an object class with which the underlying plug-in is dynamically selected during the creation of a new object. As a result, the framework is

loaded at initialization time, only once, but it is still possible to perform the selection of the best underlying plug-ins based on run-time parameters (such as the topology type).

B. Fault Tolerant Bootstrapping

As mentioned previously, the goal of the STCI project is to provide a scalable and fault tolerant process management infrastructure. An issue that is often overlooked, but that will become critical at exascale when deploying applications and tools composed of millions of processes, is the failure of processes during application startup. For instance, most users assume that when submitting a job, the application will come up at some point. In the worst case, the application startup fails and users try again by submitting a new job. At extreme scale, this approach is not sustainable: the probability to have a failure during the application startup is actually very high.

The same problem applies in the context of STCI: how can we ensure that the infrastructure bootstraps correctly despite the occurrence of agent failures? For that, we separated the initialization of the STCI infrastructure into two distinct phases: a bootstrapping phase where the basic infrastructure and the basic STCI services (e.g., communications between agents) are not yet fully setup, and an initialization phase where higher-level services are setup using the basic STCI capabilities. We think that it is important to separate the two phases because during bootstrapping, the possible actions to detect and tolerate failures are very limited, whereas during the initialization phase, it is possible to use the basic STCI infrastructure to detect and recover failures.

For root, session or tool agents, the bootstrapping phase is typically the creation of the process in the context of the agent and then the creation of communication links that reflect the boot topology. Thus, at the end of the bootstrapping phase, agents are reachable by their "parent" in the boot topology (the boot topology is currently a tree) and the new agent assumes that the parent is the default route for any received message for which the route is not explicitly available. To detect failures during the bootstrapping phase, we combine timers with the handshake mechanism at the communication substrate level: when we create an agent, if the handshake does not occur within a window of time, the agent is assumed failed during bootstrapping. In such a situation, the current implementation terminates all remaining running agents; a future implementation will try to restart the failed agent.

Once a given agent has finished the bootstrapping phase, it starts the initialization of the remaining capabilities. These capabilities are typically the deployment of the BMG topology, the initialization of the failure detection mechanisms, and the initialization of the consensus policy for failure recovery. When these services are initialized, the agent is assumed running.

C. Agents Implementation

Currently, agents are implemented via heavy-weight processes. However, root, session and tool agents have been designed so they can be implemented differently, for instance using threads. For this, it will only be required to implement a new launcher that creates threads instead of processes, and to update the communication substrates to support communications between threads. Current APIs implemented by STCI should not need to be modified. In other terms, only new plug-ins for the launcher, bootcom and the active message subsystem have to be developed for the usage of execution entities different than processes.

IV. Use Cases

In this section we discuss two relevant use-cases for HPC that demonstrate the applicability of the STCI runtime. The first instance involves the use of STCI as an alternate runtime for an existing MPI implementation. The second instance describes the use of STCI to support building a fault-injection tool for resilience experimentation.

A. Alternate Runtime for MPI

To demonstrate and validate the usability of STCI as an alternate runtime for an MPI implementation, we selected the Open MPI prototype because of the technical similarities between the two projects (we use OPAL and the MCA framework support initially developed in the context of the Open MPI project).

The Open MPI prototype is composed of three different software projects: OPAL, which abstracts the underlying operating system and, to some extend, the underlying hardware; Open RunTime Environment (ORTE), the default runtime system for Open MPI; and the implementation of MPI itself, named OMPI. Fortunately, the Open MPI community lately included an abstraction layer for the underlying runtime. As a result, Open MPI easily supports the ORTE, as well as, PMI (PMI — process manager interface — support is required for systems that provide only a micro-kernel, for which ORTE is not an option). This abstraction provides a generic API for out-of-band communications and for a naming service to identify, select and manipulate processes that implement MPI ranks. Based on this we developed a plugin specific to STCI for this abstraction layer, mapping each function to a corresponding STCI set of APIs. With the resulting prototype, STCI can be used to deploy, monitor and terminate an MPI job. The MPI ranks are instantiated via tool agents and the STCI communication substrate is therefore used during the deployment and termination of the job, as well as, for I/O forwarding, monitoring and failure management (potentially from detection to recovery of the RTE infrastructure). The HPC communication substrate of Open MPI is still used for all performance-critical MPI communications, including collective operations.

Based on this prototype, we also initiated the implementation of a fault tolerant MPI prototype. That work is based on the ongoing MPI 3.x standardization effort in the MPI Forum, which is focusing on User-Level Failure Mitigation (ULFM) [6] capabilities for MPI applications.

B. Runtime Support for Fault Injection

Fault injection, i.e., the purposeful triggering of faults during parallel application execution, is becoming an invaluable tool to study the impact of faults and to validate mitigation mechanisms, both at runtime. We leveraged the modular STCI runtime to develop a tool to help with fault-injection experiments for HPC resilience. The basic pieces of the *finject*

prototype are: (i) front-end and distributed control, (ii) experiment setup/management, (iii) monitoring and event logging, and (iv) fault injection mechanisms. The *finject* prototype is implemented as an optional STCI framework that can be enabled at compile time. This "fi" (fault-injection) framework provides components that implement different fault-injection mechanisms, e.g., process kill ("pkill") for process fail-stop tests. Additionally, a customized Front-end is provided that offers a simple user-interface for starting resilience experiments relying on these fault injection capabilities. The *target* for the experiments is the end-user application that will be subjected to the injection of intentional errors, which can be used to study their effects on the given application. A customized STCI agent is launched to manage the experiment called the *target manager*. We leverage the communication substrate offered by the runtime for triggering events, logging their occurrence and forwarding the output of the target application.

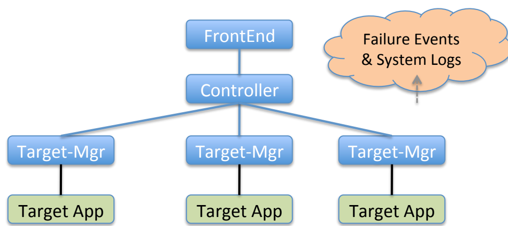

Fig. 2. Basic structure for the fault injection tool.

The general structure of the framework to support building fault injection tools using the STCI runtime is shown in Figure 2. The custom *finject* Front-end is the agent that actually starts the other agents and is simply a modified version of the standard STCI Front-end ("stcirun"). The Controller is slightly modified to include the additional "fi" framework that relays user-triggered events from the Front-end to the job using a specific "fi:ctrl" component. The Root agents are not shown in the diagram but are used in their normal fashion as described in Section II-A. The Session agents and the Tool agents are tailored to provide the Target Manager and the Target entities shown in Figure 2. The agent identifiers, communication substrate, and I/O forwarding services provided by the runtime are useful for creating failure event logs and capturing the state of the system during the tests.

The current *finject* prototype is still very basic, but the STCI building blocks allow for embedding this tool into the infrastructure, so that it can be enabled at runtime as needed. These types of on-demand testing and diagnostic capabilities will become more important as developers continue to work on the challenges of resilience for HPC tools and applications.

V. Related Work

The Multipurpose Daemon (MPD) [7] is designed to be a generic and scalable processes management system that was used in the context of MPICH. MPD sets up a set of system daemons (privileged daemons) on each compute node of the HPC platform and set up a communication ring between them. When a user starts a job, the system daemons start on each node a management daemon, called *managers*, that runs on behalf of the user and spawn processes that ultimately instantiate the job (e.g., MPI tasks), which are called *clients*. A communication ring is set up between the managers. Also, managers redirect I/O, i.e., stdin, stdout, and stderr, from/to the clients; and a binomial tree is set up between the managers for the communications related to I/O redirection. MPD was also designed to provide some fault tolerance capabilities in the context of the system daemons and the managers. Unfortunately, the usage of rings limit the scalability of the system, and the use of a binomial tree may not be the more efficient topology on all HPC systems, especially those based on a 3D-torus network.

The Process Management Interface (PMI) [8] has been defined to be a generic interface for process management of parallel systems, that could be used for the implementation of various parallel programming library such as MPI. The key concept of PMI is the separation of the resource management capabilities (e.g., start/stop processes) from the creation of communication channels between the different execution contexts of a given job (e.g., communications between MPI tasks). To the best of our knowledge, PMI does not specify any fault tolerance capability to deal with failures during the startup of parallel jobs, and PMI does not provide any particular capabilities for the specification and instantiation of various topologies for communications in the context of a job. Several implementation, such as Hydra [8] in the context of MPICH [9], implements the PMI interface. On Cray platforms, an extended version of PMI is provided that does include some additional features to support system-level checkpoint/restart.

As mentioned in Section IV, Open MPI uses the Open RunTime Environment (ORTE) [10] as its default runtime system. ORTE offers several useful features for debugging MPI applications and has a mature set of features. The STCI project was started by several members of the Open MPI project to explore alternate approaches for HPC resilience. As such, the OPAL and MCA systems used by Open MPI and ORTE are also used in STCI. An early design for the project was presented in Buntinas et al. [11] and detailed the initial plans and motivation behind a Scalable Tools and Communication Infrastructure (STCI).

The Simple Linux Utility for Resource Management (SLURM) [12] is a cluster job scheduling and resource management solution designed to provide high scalability (to tens or hundreds of thousands of compute nodes), fault tolerance, and high job throughput. Architecturally, SLURM shares several characteristics with STCI. A SLURM cluster contains at least one control daemon (slurmctld) which is responsible for keeping track of the state of the entire cluster, scheduling batch jobs, and launching and monitoring processes within the cluster. For fault tolerance, the control daemon may be supplemented by a backup control daemon which can take over for the primary daemon in the event of a failure. The control daemon communicates with daemon processes (slurmd) that run on each compute node and are responsible for launching and monitoring the user processes that make up a batch job. For scalability, the slurmd processes arrange themselves into a tree topology (by default, this is a 50-ary tree). Functionally, the slurmctld is similar to the STCI controller agent while the slurmd is similar to the STCI root agent. Further, the tree topology used by the slurmd processes is similar to the current default bootstrap topology of the STCI root agents (as well as session agents and tool agents), although agents in STCI

would generally add additional arbitrary topologies to allow more efficient communication patterns after bootstrapping. Similarities between SLURM and STCI are also reflected in the fact that both pieces of software are readily extensible via software plug-ins. Perhaps the largest difference between SLURM and STCI is that SLURM is intended to manage resources within a cluster on behalf of a batch job while at the same time remaining orthogonal to the program (or programs) making up the job. STCI, on the other hand, is usually intended to, at some level, link directly to a user's program in order that it can act as a run-time software layer to control process launching and monitoring as well as inter-process communication at a finer level of granularity.

VI. Conclusion

We have described the software architecture for the STCI process management infrastructure for high-performance computing. The runtime is being used to support ongoing research activities in fault-tolerance. The modular runtime includes support for monitoring and process failure detection. A *topology* abstraction is used throughout the process management, monitoring and communication regions of the runtime to provide a consistent means of describing configurations for performance and/or resilience purposes.

The goal of the STCI architecture is not necessarily to provide fault tolerance capabilities for applications and tools, but instead to tolerate failure at the infrastructure level so that users have the opportunity to be notified of failures and decide the appropriate actions (including clean termination of remaining processes, or continuation with surviving processes). The architecture separates initialization of the infrastructure into two phases: (i) a bootstrapping phase where available capabilities are limited, constraining the options for fault tolerance, and (ii) an initialization phase during which more capabilities are available, which enables the implementation of more advanced fault tolerant techniques. As a result, the window of time during which the overall infrastructure is more likely to not tolerate a failure is decreased. After the initialization is complete, STCI provides fault tolerance capabilities such as failure detection, redundant communication links, which guarantee that even in the context of failure, the infrastructure remains functional based on the alive processes.

We have used the STCI prototype to develop an alternate runtime for use with the Open MPI implementation of MPI. This is being used to support resilience research into the current draft from the MPI Fault Tolerance Working Group (MPI-FTWG). The MPI-FTWG has a reference implementation of their proposal that is based on Open MPI. As part of our future work we plan to support the MPI-FTWG's extensions using our STCI runtime backed version of Open MPI. We are also using the STCI prototype to develop resilience tools, e.g., fault-injection tools, to support HPC resilience experiments.

VII. Acknowledgments

We would like to acknowledge the individuals that have contributed to this project in the past, including Richard Graham, Wesley Bland, Joshua Hursey, Christos Kartsaklis, Rainer Keller, Gregory Koenig, Pavel Shamis and Chao Wang. We also would like to thank the Oak Ridge Leadership Computing Facility for supporting this work.

This manuscript has been authored by UT-Battelle, LLC, under Contract No. DE-AC05-00OR22725 with the U.S. Department of Energy. The United States Government retains and the publisher, by accepting the article for publication, acknowledges that the United States Government retains a non-exclusive, paid-up, irrevocable, world-wide license to publish or reproduce the published form of this manuscript, or allow others to do so, for United States Government purposes.

References

[1] J. Daly, B. Harrod, T. Hoang, L. Nowell, B. Adolf, S. Borkar, N. DeBardeleben, M. Elnozahy, M. Heroux, D. Rogers, R. Ross, V. Sarkar, M. Schulz, M. Snir, P. Woodward, R. Aulwes, M. Bancroft, G. Bronevetsky, B. Carlson, A. Geist, M. Hall, J. Hollingsworth, B. Lucas, A. Lumsdaine, T. Macaluso, D. Quinlan, S. Sachs, J. Shalf, T. Smith, J. Stearley, B. Still, and J. Wu, "Inter-agency workshop on hpc resilience at extreme scale," Feb. 2012.

[2] N. DeBardeleben, J. Laros, J. T. Daly, S. L. Scott, C. Engelmann, and B. Harrod, "High-end computing resilience: Analysis of issues facing the HEC community and path-forward for research and development," Whitepaper, Dec. 2009.

[3] B. Barrett, J. M. Squyres, A. Lumsdaine, R. L. Graham, and G. Bosilca, "Analysis of the component architecture overhead in Open MPI," in *Proceedings, 12th European PVM/MPI Users' Group Meeting*, Sorrento, Italy, September 2005.

[4] T. Angskun, G. Bosilca, and J. Dongarra, "Binomial graph: A scalable and fault-tolerant logical network topology," in *International Symposium on Parallel and Distributed Processing and Applications*, pp. 471–482.

[5] E. Gabriel, G. E. Fagg, G. Bosilca, T. Angskun, J. J. Dongarra, J. M. Squyres, V. Sahay, P. Kambadur, B. Barrett, A. Lumsdaine, R. H. Castain, D. J. Daniel, R. L. Graham, and T. S. Woodall, "Open MPI: Goals, concept, and design of a next generation MPI implementation," in *Lecture Notes in Computer Science: Proceedings of the 11th European PVM/MPI Users' Group Meeting (EuroPVM/MPI) 2004*, vol. 3241, Budapest, Hungary, Sep. 19-22, 2004, pp. 97–104.

[6] W. Bland, A. Bouteiller, T. Herault, J. Hursey, G. Bosilca, and J. J. Dongarra, "An evaluation of user-level failure mitigation support in MPI," in *Proceedings of the 19th European conference on Recent Advances in the Message Passing Interface*, ser. EuroMPI'12. Berlin, Heidelberg: Springer-Verlag, 2012, pp. 193–203.

[7] R. Butler, W. Gropp, and E. Lusk, "A scalable process-management environment for parallel programs," in *In Euro PVM/MPI*. Springer-Verlag, 2000, pp. 168–175.

[8] P. Balaji, D. Buntinas, D. Goodell, W. Gropp, J. Krishna, E. Lusk, and R. Thakur, "PMI: A scalable parallel process-management interface for extreme-scale systems," in *Proceedings of the 17th European MPI users' group meeting conference on Recent advances in the message passing interface*, ser. EuroMPI'10. Berlin, Heidelberg: Springer-Verlag, 2010, pp. 31–41.

[9] W. Gropp and E. Lusk, "Sowing MPICH: A case study in the dissemination of a portable environment for parallel scientific computing," *The International Journal of Supercomputer Applications and High Performance Computing*, vol. 11, no. 2, pp. 103–114, Summer 1997.

[10] R. H. Castain, T. S. Woodall, D. J. Daniel, J. M. Squyres, B. Barrett, and G. E. Fagg, "The Open Run-Time Environment (OpenRTE): A transparent multi-cluster environment for high-performance computing," in *Proceedings, 12th European PVM/MPI Users' Group Meeting*, Sorrento, Italy, September 2005.

[11] D. Buntinas, G. Bosilca, R. L. Graham, G. Vallée, and G. R. Watson, "A scalable tools communications infrastructure," in *Proceedings of the 2008 22nd International Symposium on High Performance Computing Systems and Applications*, ser. HPCS'08. IEEE Computer Society, 2008, pp. 33–39.

[12] M. A. Jette, A. B. Yoo, and M. Grondona, "SLURM: Simple Linux Utility for Resource Management," in *Job Scheduling Strategies for Parallel Processing in Lecture Notes in Computer Science*, vol. 2862. Springer-Verlag, 200, pp. 44–60.