The Process Management Component of a Scalable Systems Software Environment*

Ralph Butler[1], Narayan Desai[2], Andrew Lusk[2], and Ewing Lusk[2]

[1] Department of Computer Science
Middle Tennessee State University
Murfreesboro, TN
[2] Mathematics and Computer Science Division
Argonne National Laboratory
Argonne, IL 60439, USA
rbutler@mtsu.edu
{desai,alusk,lusk}@mcs.anl.gov

Abstract. The systems software necessary to operate large-scale parallel computers presents a variety of research and development issues. One approach is to consider systems software as a collection of interacting *components*, with well-defined published interfaces. The Scalable Systems software SciDAC project is currently exploring the feasibility of architecting systems software this way. In this paper we present a prototype *process manager* component for such a system. We describe the component abstractly in terms of its functionality and the interface by which its functionality may be invoked. We propose a precise syntax for this interface and describe one implementation of the process manager component, based on an existing scalable process management system called MPD. We conclude with some experiences using this process manager component in conjunction with other systems software components on a medium-sized Linux cluster.
Keywords: Systems software, process management, parallel programming, scalability, XML.
Corresponding Author: Ewing Lusk, lusk@mcs.anl.gov

1 Introduction

Large-scale parallel computers provide new challenges in the area of *systems software*: that collection of programs that manage the system from configuration and boot-up to the scheduling and running of parallel jobs. One approach to the design of such software is to treat it as a collection of separate interacting peer *components* with well-defined published interfaces. In this approach, there is no single monolithic system software design, but rather a collection of component specifications, from which a specific systems management solution can be

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

assembled by choosing from among multiple implementations of the specifications. The Scalable Systems Software Center [6] has been established to explore the feasibility of this approach.

In this paper we present a proposed specification for the process management component of such a system. We provide some background and motivation for this component in Section 2. We define exactly what me mean by process management and outline the functionality of the process management component at an abstract level in Section 3, and make the definition concrete with detailed XML interface examples in Section 4 (The full XML schemas are in the Appendix). We describe a prototype process manager implementation in Section 5, which illustrates how one might fit an existing process management mechanism into this component framework. We are using this process management component in conjunction with other components [4] being proposed as part of the Scalable Systems Software Center on a medium sized (256 nodes) Linux cluster [3], and we report on our experiences in Section 6.

2 Background

The work described here is motivated by the confluence of two research and development directions. The first has arisen from the MPICH project [5], which has had as its primary goal the development of a portable, open source, efficient implementation of the MPI standard. That work has led to the development of a standalone process management system called MPD [1, 2] for rapid and scalable startup of parallel jobs such as MPI implementations, in particular MPICH.

The second thrust has been in the area of scalable systems software in general. The Scalable Systems Software SciDAC project [6] is a collaboration among U. S. national laboratories, universities, and computer vendors to develop a standardized component-based architecture for open source software for managing and operating scalable parallel computers such as the large (greater than 1000 nodes) Linux clusters being installed at a number of institutions in the collaboration.

These two thrusts come together in the definition and implementation of a scalable process manager component. The definition consists concretely of the specification of an interface to other components being defined and developed as part of the Scalable Systems Software Project. Then multiple instantiations of this interface can evolve over time, along with multiple instantiations of other components, as long as the interfaces are adhered to. At the same time one wants to present an implementation of the interface, both to test its suitability and to actually provide part of a usable suite of software for managing clusters. This paper presents an interface that has been proposed to the Scalable Systems Software Project for adoption together with an implementation of it that is in actual use on some medium-sized clusters. Figure 1 shows some of the principal components of the project.

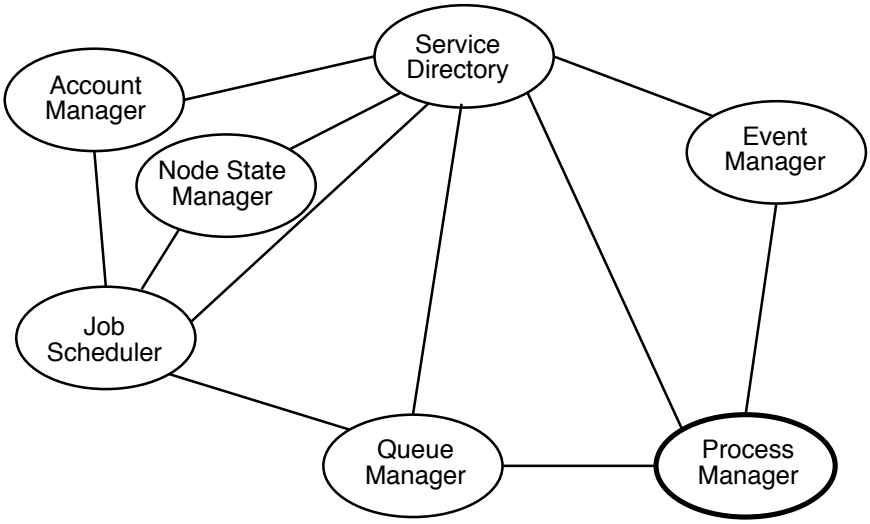

Fig. 1. Components of the Scalable Systems Software Project

3 Defining Process Management

In this section we define a process management component, describing its functionality at an abstract level. The details of precisely how other components interact with this component are given in Section 4.

3.1 Goals and Assumptions

We assume that the component belongs to a family of system software components, with which it communicates using well-defined interfaces. We give one example of what such an interface might look like in Section 4. Therefore the process management component need not concern itself with monitoring hardware or with assigning jobs to processors, since those tasks will be carried out by other components. We assume further that security concerns other than those internal to a specific instantiation of the process manager are handled by other components. Thus we take a minimalist position on what a process manager does. It should do a thorough and complete job of managing processes and leave other tasks to other components.

It is useful to introduce the concept of *rank* as a way of distinguishing and identifying the processes of a parallel job (at least the initial ones). We assume that an n-process job initially has processes with ranks $0,\ldots,n-1$. These need not necessarily coincide with MPI ranks, since there is nothing MPI-specific about job management, but the concept is the same. We will also need the concept of *pid*, which stands for process identifier, an integer assigned to a process that is unique on a particular host computer. Thus a single process in a job may be identified by a (host, pid) pair, or more abstractly by a (job, rank) pair.

3.2 Not Included

The following functions are not included in the functionality of the process manager.

Scheduling. We assume that another component is responsible for making scheduling decisions. It will either specify which hosts various processes of the parallel job will run on, or specifically leave the choice up to the process manager, which is then free to make any decision it prefers.

Node monitoring. The state of a particular host is of interest to the scheduler, which should be responsible for deciding whether a node is available for starting a job. The scheduler can interact directory with a node monitor component to determine the information it needs.

Process monitoring. CPU usage, memory footprint, etc., are characteristics of the individual processes, and can be monitored by a separate component. The process manager can aid monitoring processes by either starting them (see *co-processes* below) or by providing information on the process identifiers and hosts where particular processes of a job are running, in order to assist monitoring components.

Checkpointing. The process manager can help with checkpointing by delivering signals to the parallel job, but checkpointing itself is a separate function and should be carried out by a separate component.

3.3 Included

We are thus left with the following functions. We compensate for the limited number of functions described here by attempting to specify very flexible and complete versions of the functions that are included.

Starting a parallel job. The process manager is responsible for starting a parallel job, without restrictions. That is, the processes should be allowed to have separate executables, separate command-line arguments, and separate environments. Processes may even be run under different user names on different machines. Jobs will be started with appropriate user id's, group id's, and group memberships. It should be possible for the job submitter to assign a job identifier by which the job can be referred to later. A job-start request will be answered by a message confirming successful start or else a failure message with an error code. We allow multiple options for how standard I/O is handled.

Starting coprocesses for a parallel job. An advanced functionality we intend to explore is that of *coprocesses*, which we define to be separate processes started at the same time as the application process for scalability's sake, and passed the process identifiers of the application processes started on that host, together with other arguments. Our motivation is scalable startup of daemons for debugging or monitoring a particular parallel job.

Signaling a parallel job. It is possible to deliver a signal to all processes of a parallel job. Signals may be specified by either name ("`STOP`", "`CONT`", etc.) or by number ("`43`", "`55`", etc.) for signals that have no name on a particular operating system. The signals are delivered to all the processes of the job.

Killing a parallel job. Not only are all application processes killed, but also any "helper" processes that may have been started along with the job, such as the coprocesses. The idea is to clean the system of all processes associated with the job.

Reporting details of a parallel job. In response to a query, the process manager will report the hosts and process identifiers for each process rank.

Reporting events. If there is an event manager, the process manager will report to it both job start and job termination events.

Handling `stdio` A number of different options for handling `stdio` are available. These include

- collecting `stdout` and `stderr` into a file with rank labels,
- writing `stdout` and `stderr` locally,
- ignoring `stdout` and `stderr`,
- delivering `stdin` to process 0 (a common default),
- delivering `stdin` to another specific process,
- delivering `stdin` to all processes.

Servicing the parallel job. The parallel job is likely to require certain services. We have begun exploring such services, especially in the MPI context, in [1]. Existing process managers tend to be part of resource management systems that use sophisticated schedulers to allocate nodes to a job, but then only execute a user script on one of the nodes, leaving to the user program the task of starting up the other processes and setting up communication among them. If they allow simultaneous startup of multiple processes, then all those processes must have the same executable file and command-line arguments. The process manager implementation we describe here provides services to a parallel job not normally provided by other process managers, which allow it to start much faster. In the long run, we expect to exploit this capability in writing parallel system utilities in MPI.

Registration with the service directory. If there is a service directory component, the process manager will register itself, and deregister before exiting.

3.4 Summary of Process Manager Interactions with Other Components

The Process Manager typically runs as root and interacts with other components bing defined and implemented as part of the Scalable Systems Software Project.

These interactions are of two types: message exchanges initiated by the Process Manger and message exchanges initiated by other components and responded to by the Process Manager.

- Messages initiated by the Process Manager. These use the interfaces defined and published by other components.

  Registration/Deregistration The Process Manager registers itself with the Service Directory so that other components in the system can connect to it. Essentially, it registers the host it is running on, the port where it is listening for connections, and the protocol that it uses for framing messages.

  Events The Process Manager communicates asynchronous events to other components by sending them to the Event Manager. Other components that have registered with the Event Manager for receipt of those events will be notified. by the Event Manager. Two such events that the Process Manager sends to the Event Manager are job start and job completion events.

- Messages responded to by the Process Manager. These can come from any authorized component. In the current suite of SSS components, the principal originators of such messages are the Queue Manager and a number of small utility components.

  Start job This is the principal command that the Process Manager is responsible for. The command contains the complete specification of job as described in Section 3.3.

  Jobinfo This request returns details of a running job. It uses the flexible XML query syntax that is used by a number of components in the project.

  Signal job The Process Manager can deliver any signal to all the processes of a job.

  Kill job The Process Manager can be asked to kill and clean up after a given job.

4 A Process Manager Interface

The Scalable Systems Software project early on settled on XML, communicated over sockets, as the underlying inter-component communication mechanism. A variety of wire protocols are supported, and a project-wide library (`SSSlib`) makes it relatively easy for components to register their own protocols, learn the protocols of other components from the service directory, and exchange messages with other components. Each component defines its own XML syntax for the messages it responds to and publishes the schema. Currently these schemas, initially defined independently by various groups as a way of getting started quickly, have begun evolving toward a common style.

In this section we give some examples of the XML syntax used by the process manager component. The precise schemas are given in the Appendix. Notable aspects of the syntax we use are:

- There is sufficient power to express the requirements of Section 3.3. In particular, separate executables, command-line arguments, and environment variables can be specified for each process. This is not the case for most process managers that are integrated into existing resource management systems.
- It contains a flexible yet compact query capability for requesting information on a restricted set of objects and returning only the information requested. The query language itself will be described elsewhere, although we use it in an example below.
- The syntax is "scalable" in the sense that it is not necessary to provide a separate XML entity for each process. Instead, the "range" attribute, an attribute of most XML entities, can be used to describe an aspect of many processes at once.

In the following subsections we present some examples of the messages that are responded to by the Process Manager. The complete schemas are given in the Appendix.

4.1 Creating a Parallel Job

This XML message requests that a set of processes, called a "process group" in our syntax, be created on a specific set of hosts. Here we have asked that the TotalView debug server be started on the same hosts and passed the process id of the application process. Note that we pas different arguments to the master and slave processes.

```
<create-process-group
        pgid='job23'
        submitter='lusk'
        totalprocs='10'
        output='discard'
        >
  <process-spec
          range='1'
          exec='cpi_master'
          user='ell'
          cwd='/home/ell/rundir'
          path='/home/ell/progs'
          coprocess='tvdebuggersrv'
          >
    <arg idx='1' val='-loops' />
    <arg idx='2' val='1000'   />
    <env name='TV_LICENSE' val='23416784' />
  </process-spec>
  <process-spec
          range='2-10'
          exec='cpi_slave'
```

```
          user='ell'
          cwd='/home/ell/rundir'
          path='/home/ell/progs'
          coprocess='tvdebuggersrv'
          >
    <env name='TV_LICENSE' val='23416784' />
  </process-spec>
  <host-spec>
    ccn-64
    ccn-65
    ccn-66
    ccn-67
    ccn-68
    ccn-69
    ccn-70
    ccn-71
    ccn-73
  </host-spec>
<create-process-group\>
```

For multiple processes on one host, one can repeat the host name. An alternative is to use the "squash" format in host names, e.g.

```
<hostspec name='ccn-%d:64-73' />
```

The response to this message will be something like

```
<process-group pgid='1'/>
```

indicating the process manager's identifier for the job. The originator can also supply its own identifier.

4.2 Enquiring about a job

Any component can ask for details about a process group. The following example retrieves the pgid's of processes that were submitted by lusk or desai, and in lusk's case, only returns the process groups that have processes running on two specific hosts. The restrictions are on the process groups; we always return all the processes in a process group.

```
<get-process-groups>
  <process-group submitter='lusk' pgid='*' totalprocs='*' >
    <process-group-restriction pid='*' exec='*' host='ccn-70' \>
    <process-group-restriction pid='*' exec='*' host='ccn-230' \>
  </process-group>
  <process-group submitter='desai' pgid='*' >
  </process-group>
</get-process-groups>
```

The message returned by such a query is a set of process groups, with details on their processes filled in as requested by the query.

```
<process-groups>
  <process-group submitter='lusk' pgid='4521' totalprocs='10'>
    <process pid='3456' exec='cpi_master' host='ccn-64' />
    <process pid='1324' exec='cpi_slave' host='ccn-65' />
    <process pid='7654' exec='cpi_slave' host='ccn-66' />
    <process pid='6758' exec='cpi_slave' host='ccn-67' />
    <process pid='9601' exec='cpi_slave' host='ccn-68' />
    <process pid='5634' exec='cpi_slave' host='ccn-69' />
    <process pid='7865' exec='cpi_slave' host='ccn-70' />
    <process pid='9876' exec='cpi_slave' host='ccn-71' />
    <process pid='6524' exec='cpi_slave' host='ccn-72' />
    <process pid='3452' exec='cpi_slave' host='ccn-73' />
  </process-group>
  <process-group submitter='lusk' pgid='23' totalprocs='1'>
    <process pid='5554' exec='mpd' host='230' />
  </process-group>
  <process-group submitter='desai' pgid='244' >
  </process-group>
</process-groups>
```

4.3 Signaling or Killing a Job

The signal-process-group (deliver a specified signal to a process group) and kill-process-group (completely clean up a process group) are extended to allow one to describe the process groups being signalled or killed to be specified with the same syntax as get-process-group, and they return the same format (¡process-groups¿, defined above, to indicate which processes they acted on. Signals can be specified by either name or number.

The following command sends a signal 3 to all the processes of all jobs submitted by lusk, and returns the details of which processes groups they were.

```
<signal-process-group  signal='3'>
  <process-group submitter='lusk' pgid='*'
</signal-process-group>
```

The following command kills all process groups with processes running on ccn-56, and returns their submitters, so that they can be told the sad news.

```
<kill-process-group>
  <process-group submitter='*'>
    <process-group-restriction host='ccn-56' >
  </process-group>
</kill-process-group>
```

5 Implementing a Process Manager Component

In order to provide a prototype implementation of the process manager component, we began by re-implementing the MPD system described in [2]. A new implementation was necessary because the original MPD system could not support the requirements imposed by the Scalable Systems Software Project, in particular the ability to provide separate specifications for each process. The relationship between the MPD system and the Scalable System Software process manager component is shown in Figure 2.

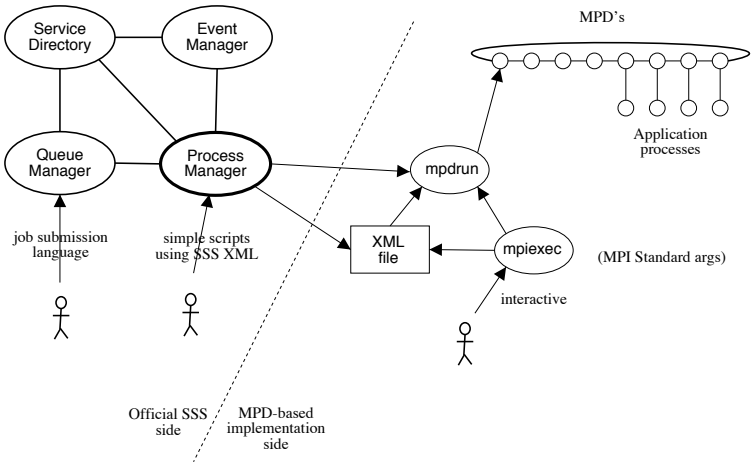

Fig. 2. Relationship of MPD to SSS components.

The process management component itself is written in Python, and uses Python's XML module for parsing. It invokes the `mpdrun` process startup command of the new MPD system. MPD is a ring of pre-existing and pre-connected daemons, running as root or as a user, which implements all the necessary requirements. The most notable one is that it provides an implementation of the process manager interface (PMI) used by MPICH to provide rapid startup of MPI programs and support for MPI-2 dynamic process functionality.

6 Experiments and Experiences

We have demonstrated the use of this process manager component in conjunction with other components of the Scalable Systems Software component collection. In this configuration, jobs are submitted to a Queue Manager, which forwards the request to a scheduler. The scheduler assigns nodes to the ;job based on the results of node-monitoring components and returns the job to the Queue Manager, which sends it to the Process Manager to start. At any time the Queue

Manager (or any other authorized component) can enquire about the details of the job. If the job has not finished within its alloted time, the Queue Manager sends a kill-job request to the Process Manager.

Of course, once the interface has been published, the Process Manager can be used in other ways as well. Since MPD will serve as a primitive scheduler itself if it is not instructed to start processes on specific hosts, the system administrator, for example, can execute maintenance jobs through the Process Manager without going through the Queue Manager or Scheduler components.

Our primary testbed has been the Chiba City testbed [3] at Argonne National Laboratory. Some experiments on starting both MPI and non-MPI jobs follow, to illustrate the scalability of the MPD-based implementation of the Process Manager. These tests were carried out on Chiba City.

6.1 Testing the MPD ring

The first test is just of the MPD ring's ability to forward messages around the ring. Here the ring had 206 hosts in it. We simulated larger rings by sending a message around multiple times.

times around ring	time in seconds
1	.13
10	.89
100	8.93
1000	89.44

This is linear, as we would expect from a ring. However, it is so fast, because the MPD daemons are pre-connected, that the time to get a message around the ring is not significant. The second line of the above table represents more than 2000 hops in less than a second. Therefore the ring does not need to be replaced by a more scalable structure such as a broadcast tree. We prefer the ring structure since it is symmetric (no single point of failure) and can be made more fault tolerant.

6.2 Starting Non-MPI jobs

We ran `hostname` on each node and collected the output through MPD's tree of connections for `stdout`. Times were as follows:

number of hosts	time in seconds
1	.83
4	.86
8	.92
16	1.06
32	1.33
64	1.80
128	2.71
192	3.78
200	3.85

This is quite sublinear, indicating that the MPD daemons are benefitting from being able to get the startup message quickly (see Section 6.1) and then start the processes largely in parallel.

6.3 Old MPI Startup vs. New MPI Startup

In earlier versions of MPICH, run under previous process managers, the first MPI process had to start the others in order to exchange information with them that allowed for later MPI connections to be made. This mechanism was not scalable. Now that the process manager provides a route for contact information to be exchanged, this process is much faster. Times below are for starting a short MPI job (the classical `cpi` example from the MPICH distribution), which does need to establish MPI communication among many of its processes, in order to carry out an `MPI_Broadcast` and `MPI_Reduce`. Times are in seconds.

number of processes	old startup time	new startup time
1	.4	.63
4	5.6	.67
8	14.4	.73
16	30.9	.86
32	96.9	1.01
64		1.90
128		3.50

Note that the improvement is quite significant, and that startup time is nearly constant for jobs of size up to 32. We are exploring improvements in the MPD implementation to increase scalability further, but it already has had a large impact on the starting of MPI jobs, particularly interactive ones.

7 Conclusion

Our primary conclusion is that the component idea for scalable system software architecture can be a useful approach. We have described how defining process management abstractly led to a capable process management interface and we have provided a prototype implementation that scales well.

References

1. R. Butler, W. Gropp, and E. Lusk. A scalable process-management environment for parallel programs. In Jack Dongarra, Peter Kacsuk, and Norbert Podhorszki, editors, *Recent Advances in Parallel Virutal Machine and Message Passing Interface*, number 1908 in Springer Lecture Notes in Computer Science, pages 168–175, September 2000.
2. R. Butler, W. Gropp, and E. Lusk. Components and interfaces of a process management system for parallel programs. *Parallel Computing*, 27:1417–1429, 2001.
3. Chiba City home page. http://www.mcs.anl.gov/chiba.

4. Narayan Desai, Andrew Lusk, Ewing Lusk, and John-Paul Navarro. The configuration manager and other infrastructure components of a scalable systems software environment. Technical Report ANL/MCS-P988-0802, Argonne National Laboratory, 2002.
5. William Gropp, Ewing Lusk, Nathan Doss, and Anthony Skjellum. A high-performance, portable implementation of the MPI Message-Passing Interface standard. *Parallel Computing*, 22(6):789–828, 1996.
6. Scalable systems software center home page. `http://www.scidac.org/scalablesystems`.

Appendix: XML Schemas

The XML schema for messages to the Process Manager is as follows:

```
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema" xml:lang="en">
  <xsd:annotation>
    <xsd:documentation>
      Process Manager component inbound schema
      SciDAC SSS project, 2002 Andrew Lusk alusk@mcs.anl.gov
    </xsd:documentation>
  </xsd:annotation>

  <xsd:include schemaLocation="pm-types.xsd"/>

  <xsd:complexType name="createpgType">
    <xsd:choice minOccurs="1" maxOccurs="unbounded">
      <xsd:element name="process-spec" type="pg-spec"/>
      <xsd:element name="host-spec" type="xsd:string"/>
    </xsd:choice>
    <xsd:attribute name="submitter" type="xsd:string" use="required"/>
    <xsd:attribute name="totalprocs" type="xsd:string" use="required"/>
    <xsd:attribute name="output" type="xsd:string" use="required"/>
  </xsd:complexType>

  <xsd:element name="create-process-group" type="createpgType"/>

  <xsd:element name="get-process-group-info">
    <xsd:complexType>
      <xsd:choice minOccurs="1" maxOccurs="unbounded">
        <xsd:element name="process-group" type="pgRestrictionType"/>
      </xsd:choice>
    </xsd:complexType>
  </xsd:element>

  <xsd:element name="del-process-group-info">
    <xsd:complexType>
      <xsd:choice minOccurs="1" maxOccurs="unbounded">
        <xsd:element name="process-group" type="pgRestrictionType"/>
      </xsd:choice>
    </xsd:complexType>
```

```
  </xsd:element>

  <xsd:element name="signal-process-group">
    <xsd:complexType>
      <xsd:choice minOccurs="1" maxOccurs="unbounded">
        <xsd:element name="process-group" type="pgRestrictionType"/>
      </xsd:choice>
      <xsd:attribute name="signal" type="xsd:string" use="required"/>
    </xsd:complexType>
  </xsd:element>

  <xsd:element name="kill-process-group">
    <xsd:complexType>
      <xsd:choice minOccurs="1" maxOccurs="unbounded">
        <xsd:element name="process-group" type="pgRestrictionType"/>
      </xsd:choice>
    </xsd:complexType>
  </xsd:element>

  <xsd:element name="checkpoint-process-group">
    <xsd:complexType>
      <xsd:choice minOccurs="1" maxOccurs="unbounded">
        <xsd:element name="process-group" type="pgRestrictionType"/>
      </xsd:choice>
    </xsd:complexType>
  </xsd:element>

</xsd:schema>
```

The XML schema for messages from the Process Manager is as follows:

```
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema" xml:lang="en">
  <xsd:annotation>
    <xsd:documentation>
      Process Manager component outbound schema
      SciDAC SSS project, 2002 Andrew Lusk alusk@mcs.anl.gov
    </xsd:documentation>
  </xsd:annotation>

  <xsd:include schemaLocation="pm-types.xsd"/>
  <xsd:include schemaLocation="sss-error.xsd"/>

  <xsd:element name="process-groups">
    <xsd:complexType>
      <xsd:choice minOccurs='0' maxOccurs='unbounded'>
        <xsd:element name="process-group" type="pgType"/>
      </xsd:choice>
    </xsd:complexType>
  </xsd:element>
```

```
  <xsd:element name="process-group" type="pgRestrictionType"/>

  <xsd:element name="error" type="SSSError"/>

</xsd:schema>
```

The above schemas use the following types:

```
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema" xml:lang="en">
  <xsd:annotation>
    <xsd:documentation>
      Process Manager component schema
      SciDAC SSS project, 2002 Andrew Lusk alusk@mcs.anl.gov
    </xsd:documentation>
  </xsd:annotation>

  <xsd:complexType name="argType">
    <xsd:attribute name="idx" type="xsd:string" use="required"/>
    <xsd:attribute name="value" type="xsd:string" use="required"/>
  </xsd:complexType>

  <xsd:complexType name="envType">
    <xsd:attribute name="name" type="xsd:string" use="required"/>
    <xsd:attribute name="value" type="xsd:string" use="required"/>
  </xsd:complexType>

  <xsd:complexType name="pg-spec">
    <xsd:choice minOccurs='0' maxOccurs='unbounded'>
      <xsd:element name="arg" type="argType"/>
      <xsd:element name="env" type="envType"/>
    </xsd:choice>
    <xsd:attribute name="range" type="xsd:string"/>
    <xsd:attribute name="user" type="xsd:string"/>
    <xsd:attribute name="co-process" type="xsd:string"/>
    <xsd:attribute name="exec" type="xsd:string" use="required"/>
    <xsd:attribute name="cwd" type="xsd:string" use="required"/>
    <xsd:attribute name="path" type="xsd:string" use="required"/>
  </xsd:complexType>

  <xsd:complexType name="procType">
    <xsd:attribute name="host" type="xsd:string" use="required"/>
    <xsd:attribute name="pid" type="xsd:string" use="required"/>
    <xsd:attribute name="exec" type="xsd:string" use="required"/>
    <xsd:attribute name="session" type="xsd:string" use="required"/>
  </xsd:complexType>

  <xsd:complexType name="procRestrictionType">
    <xsd:attribute name="host" type="xsd:string"/>
    <xsd:attribute name="pid" type="xsd:string"/>
    <xsd:attribute name="exec" type="xsd:string"/>
```

```
  </xsd:complexType>

  <xsd:complexType name="pgType">
    <xsd:choice minOccurs="1" maxOccurs="unbounded">
      <xsd:element name="process" type="procType"/>
    </xsd:choice>
    <xsd:choice minOccurs='0' maxOccurs='1'>
      <xsd:element name='output' type='xsd:string'/>
    </xsd:choice>
    <xsd:attribute name="pgid" type="xsd:string" use="required"/>
    <xsd:attribute name="submitter" type="xsd:string" use="required"/>
    <xsd:attribute name="totalprocs" type="xsd:string" use="required"/>
    <xsd:attribute name="output" type="xsd:string" use="required"/>
  </xsd:complexType>

  <xsd:complexType name="pgRestrictionType">
    <xsd:choice minOccurs="0" maxOccurs="unbounded">
      <xsd:element name="process" type="procRestrictionType"/>
    </xsd:choice>
    <xsd:attribute name="pgid" type="xsd:string"/>
    <xsd:attribute name="submitter" type="xsd:string"/>
    <xsd:attribute name="totalprocs" type="xsd:string"/>
  </xsd:complexType>

</xsd:schema>
```