SMPD PMI Wire Protocol Reference Manual
Version 0.1
DRAFT of March 2, 2005
Mathematics and Computer Science Division
Argonne National Laboratory

David Ashton

March 2, 2005

Contents

1 Introduction **1**

2 SMPD manager topology **1**

3 Child process environment **3**

4 SMPD wire commands **4**

4.1 Comand Format 4

4.2 Commands 5

4.3 spawn command 6

5 string format **7**

1 Introduction

When a user builds MPICH2 they have the option to choose the SMPD process manager to launch and manage processes in MPICH2 jobs. MPICH2 provides an implementation of smpd and mpiexec to launch MPICH2 jobs. MPICH2 applications communicate with the process manager using the PMI interface. The PMI library for smpd provides an implementation of PMI for communicating with SMPD process managers. This document describes the environment and wire protocol between the MPICH2 application and the SMPD manager.

If a process manager implementor replicates the environment and protocol described in this document, they would be able to launch and manage MPICH2 jobs compiled for SMPD.

An SMPD manager communicates with its child process through environment variables and a socket. This document describes the environment and the wire protocol on that socket.

2 SMPD manager topology

This section describes how SMPD is organized in MPICH2. An implementation of a process manager that uses the protocol described in this document is not required to use this topology. It is provided for reference.

In the idle state, SMPDs reside on each node unconnected. When a new job is to be launched, mpiexec first selects a list of hosts to launch a job on. Then it connects to the SMPDs and they fork or spawn new managers resulting in a connected tree with mpiexec at the root. See figure 1. This tree remains for the duration of the job. It can grow as a result of spawn commands. Each SMPD manager has a id in the tree used to route commands. Each manager can manage multiple child processes. The control socket connections between the SMPD manager and the child processes are referenced by context ids. The SMPD manager provides the context id to the child when it launches a process.

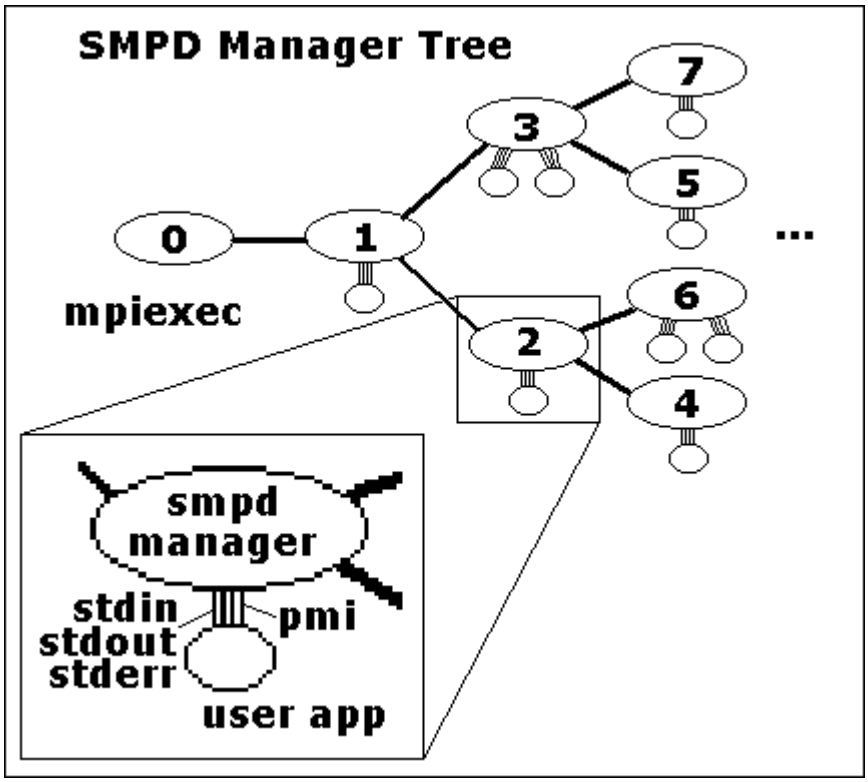

Figure 1: SMPD Manager tree

3 Child process environment

SMPD managers launch and manage child processes in an MPICH2 job. MPICH2 processes compiled with the SMPD PMI library expect the following environment variables to be set:

PMI_RANK = my rank in the process group (0 to N-1)

PMI_SIZE = process group size (N)

PMI_KVS = my keyval space name unique to my process group

PMI_DOMAIN = my keyval space domain name

*PMI_CLIQUE = my node neighbors in the form of a clique. A clique is a comma separated list of ranges and numbers. Example: 0,2..4,7

PMI_SMPD_ID = my smpd manager node id

PMI_SMPD_KEY = ctx_key value to be included with PMI commands from this process.

PMI_SPAWN = 0 or 1 if this process was started by a PMI_Spawn_multiple command.

PMI_APPNUM = index of the command that started the local process

**PMI_SMPD_FD = file descriptor/handle to convert into the PMI socket context.

**PMI_HOST = host description as specified by the MPIDU_Sock interface

**PMI_PORT = port PMI_Host is listening on

**PMI_ROOT_HOST = root host to connect to to establish the PMI socket context.

**PMI_ROOT_PORT = root listening port number

**PMI_ROOT_LOCAL = 0 or 1 if the root process is to act as the root smpd manager. If PMI_ROOT_LOCAL is specfied and it is 1, the root MPICH process starts a separate thread or process to act as the smpd manager. This manager listens on the specified port for pmi socket contexts to connect from all the processes in the job and handles smpd pmi commands for them. It is an error if PMI_ROOT_HOST is not the same as the host where rank 0 is launched.

* If not specified, default clique contains only the local process.

** Only one set of options may be specified.

The three options, PMI_SMPD_FD, PMI_ROOT_HOST/PMI_ROOT_PORT and PMI_HOST/PMI_PORT are mutually exclusive. If PMI_SMPD_FD exists then the process uses that handle as its connection to the SMPD manager otherwise it makes a socket connection to the host/port described by PMI_ROOT_HOST/PMI_ROOT_PORT or PMI_HOST/PMI_PORT.

4 SMPD wire commands

This section describes the wire protocol for PMI commands from the child process to the smpd manager.

4.1 Comand Format

Commands are variable length. Each command begins with a 13 byte header. The header is a NULL terminated ascii string representation of the length of the command to follow the header. After the header is a string of the length described by the header. Both the header and the command are NULL terminated. The header is always 13 bytes no matter where the NULL character falls. The command string begins at the 14th byte and the length of the command must include the NULL character.

Commands contain `key=value` strings to describe the components of the command. All commands will have the following keys:

- `cmd=command`
- `src=my_smpd_id`
- `dest=dest_smpd_id`
- `tag=command_tag`
- `ctx_key=pmi_smpd_key`

Additional command specific keys are described in the following section.

4.2 Commands

`done`

No more PMI commands, close the context. This command is sent from the child directly to its SMPD manager and does not receive a reply.

Example: `cmd=done src=3 dest=3 tag=14 ctx_key=0`

`exit_on_done`

The root smpd manager can and should exit when all done commands are received. This command is sent by the root process.

Example: `cmd=exit_on_done src=1 dest=1 tag=13 ctx_key=0`

Is this command necessary? Shouldn't the root smpd know that it is a root smpd and exit automatically when all its pmi contexts close?

`barrier`

Barrier across a set of processes. Add `name=barrier_name value=number_of_participants`. The result command returns SUCCESS or FAIL.

Example: `cmd=barrier src=2 dest=1 tag=3 ctx_key=1 name=kvsname value=2`

`dbcreate`

Create a new keyval space. If `name=kvsname` is added to the command then the keyval space is created with the provided name, otherwise the implementation chooses a name. The result command returns SUCCESS or FAIL and `name=kvsname`.

Example: `cmd=dbcreate src=1 dest=1 tag=100 ctx_key=0`

`dbdestroy`

Destroy a keyval space. Add `name=kvsname`. The result command returns SUCCESS or FAIL.

Example: `cmd=dbdestroy src=4 dest=1 tag=13 ctx_key=1 name=kvsname`

`dbput`

Put a keyval into a kvs space. Add `name=kvsname key=user_key value=user_value`. The result command returns SUCCESS or FAIL.

Example `cmd=dbput src=3 dest=1 tag=100 ctx_key=0 name=kvsname key=foo value=bar`

`dbget`
: Get a kevyal from a kvs space. Add `name=kvsname key=user_key`. The result command returns SUCCESS or FAIL and `value=val`.

  Example: `cmd=dbget src=4 dest=1 tag=0 ctx_key=0 name=kvsname key=foo`

`dbfirst`
: Start the keyval space iterator. Add `name=kvsname`. The result command returns SUCCESS or FAIL and `key=key value=val`.

  Example: `cmd=dbfirst src=1 dest=1 tag=22 ctx_key=0 name=kvsname`

`dbnext`
: Get the next keyval from the iterator. Add `name=kvsname`. The result command returns SUCCESS or FAIL and `key=key value=val`.

  Example: `cmd=dbnext src=2 dest=1 tag=12 ctx_key=0 name=kvsname`

`spawn`
: Spawn a new process group. See the next section for a complete description.

`result`
: The result of a previous command. Result commands will always have two fields, `cmd_tag=command_tag` and `result=result_string`. The `command_tag` matches the tag of the command the result command refers to. The `result_string` is SUCCESS or a failure message. Other return fields will be present as specified by the issued command.

4.3 spawn command

The `spawn` command is issued by a single node to launch a set of processes in a new process group.

The `spawn` command is used to implement PMI_Spawn_multiple.

The keys to the `spawn` command are the following:

```
ncmds = x number of commands

cmd0 = command

cmd1 = command
```

```
...
argv0 = string1 string2 string3 ...
argv1 = string1 string2 string3 ...
...
maxprocs = n0 n1 n2 ...  nx-1
nkeyvals = n0 n1 n2 ...  nx-1
keyvals0 = ‘‘0=\‘‘key=val\’’ 1=\‘‘key=val\’’ ...  n0-1=\‘‘key=val\’’’’
keyvals1 = ‘‘0=\‘‘key=val\’’ 1=\‘‘key=val\’’ ...  n1-1=\‘‘key=val\’’’’
...
npreput = number of preput keyvals
preput = ‘‘0=\‘‘key=val\’’ 1=\‘‘key=val\’’ ...  n-1=\‘‘key=val\’’’’
```

The `ncmds` key represents the size of the rest of the vector arguments. There will be `ncmds` `cmd` and `argv` keys. `maxprocs` and `nkeyvals` will contain `ncmds` entries. The values in `maxprocs` represent the requested number of processes to launch for the corresponding `cmd` command. There will be `ncmds` `keyvals` keys and each `keyvals` key will contain nx keys where nx is the corresponding value in `nkeyvals`. `npreput` represents the number of keys in the `preput` key. The keys in the `preput` key are to be put in the keyval space of the spawned process group before any of the processes are launched.

Example: `cmd=spawn src=3 dest=0 tag=4 ctx_key=0 ncmds=1 cmd0=myapp argv0=‘‘one \‘‘two args\’’ three’’ maxprocs=4 nkeyvals=2 keyvals0=‘‘0=\‘‘host=toad\’’ 1=\‘‘path=/home/me\’’’’ npreput=1 preput=‘‘0=\‘‘port=1244\’’’’`

5 string format

This section describes the format of key=value elements in a stream.

stream := frame | frame stream

frame := element frame_char | element separ_char frame

element := key delim_char value

key := string

value := string

string := literal | quoted

literal := array of chars without separators, deliminators, or quotes

quoted := quote_char array-of-escaped-characters quote_char

chars := ascii characters

escapted chars := ascii characters with escaped quote_chars and escape_chars

quote_char := ”

escape_char := \

delim_char := =

separ_char := ’ ’

frame_char := ’\0’

Example:

a=b ”my name”=”David Ashton” foo=”He said, \”Hi there.\””

References