Boot System Services Interface (SSI) Modules for LAM/MPI
API Version 1.0.0 / SSI Version 1.0.0

Jeffrey M. Squyres
Brian Barrett
Andrew Lumsdaine
http://www.lam-mpi.org/

Open Systems Laboratory
Pervasive Technologies Labs
Indiana University
CS TR576

August 4, 2003

Contents

1 Overview

Before reading this document, readers are strongly encouraged to read the general LAM/MPI System Services Interface (SSI) overview ([2]). This document uses the terminology and structure defined in that document.

The boot SSI kind is used to start a LAM universe. Its primary responsibility is to start processes on local and remote nodes that either will constitute the LAM run-time environment (RTE) or function outside the LAM RTE.

The most commonly understood paradigm for this is using `rsh` or `ssh` to start processes on remote nodes. However, there are many other environments where using `rsh`/`ssh` is superfluous, not possible, or subverts other RTEs. Examples include (but are not limited to):

- Batch queuing systems. PBS,[1] for example, has the Task Management (TM) API that can be used for launching processes on nodes that the job owns. Indeed, using TM instead of `rsh`/`ssh` will allows PBS to guarantee clean up when a job is done and to provide better accounting of resources. This same principle generally holds true for other batch queuing systems as well.
- BProc systems. BProc clusters use multiple machines to emulate a single, large system. As such, there is really only one "machine", and using `rsh`/`ssh` not only "seems odd" in this situation, it actually creates problems. Using the native BProc-based controls is a much better fit.
- Condor.[2] Condor already has an RTE with extensive process control mechanisms; there is no need to use `rsh`/`ssh` in such environments.
- Globus.[3] Globus is used to connect multiple sites; using `rsh`/`ssh` may not be possible between them. Globus also has an RTE with extensive process control mechanisms that should be used instead of `rsh`/`ssh`.

Hence, the boot SSI abstracts the process of starting LAM RTE processes on local and remote nodes and provides and interface such that different RTEs can provide modules for "native" LAM booting in each environment.

1.1 General Scheme

There are actually two components of the boot SSI module: starting LAM RTE processes and executing a startup protocol.

1.1.1 Starting LAM RTE Executables

The boot SSI is used in the following LAM executables:

- `lamboot`: Starts up a set of "LAM daemons" (each of which may be one or multiple processes) on a node.
- `lamgrow`: Expand a currently-running LAM universe by adding another "LAM daemon" on a remote node.

[1] See `http://www.openpbs.org/`.
[2] See `http://www.cs.wisc.edu/condor/`.
[3] See `http://www.globus.org/`.

- `recon`: Executes a test program on a set of nodes to verify functionality.
- `wipe`: Forcibly shut down a LAM RTE by executing the LAM command `tkill` on a set of nodes.

The sequence of boot API calls when starting LAM RTE executables on remote nodes is listed in Table 1.

Time	Booting agent
1	`open_module()`
2	`init()`
3	Module is selected
4	`parse_options()`
5	`allocate_nodes()`
6	`verify_nodes()`
7	`prepare_boot()`
8	`start_rte_procs()`
9	Launch RTE procs on remote nodes
If LAM-provided algorithm used:	
9.1	`start_rte_proc()`
9.2	`start_rte_application()`
10	`deallocate_nodes()`
11	`finalize()`
12	`close_module()`

Table 1: Sequence of boot API calls over the life of a selected module (excluding the startup protocols). Steps 9.1 and 9.2 are only invoked if one of the LAM-provided boot algorithms is used. All of step 9 will be repeated for as many nodes need to be booted.

1.1.2 Exchanging Startup Protocols

`lamboot` and `lamgrow` must exchange information to make the newly-spawned LAM daemons aware of all of its peers. This is what is referred to as "executing the startup protocols." The startup protocols are *only* necessary when starting LAM daemons on remote nodes; `recon` and `wipe` do not use these startup protocols.

The booting agent will start one or more LAM daemons. Each of these newly-spawned LAM daemons will connect back to the booting agent and send its location information (e.g., its UDP port). The booting agent will gather all of this information and, once all the LAM daemons have reported in, will connect back to each of them and send the union of all the information. The union represents the entire LAM universe, and is how the LAM daemons become aware of their peers. Table 2 shows this sequence of events.

1.2 Booting Algorithms

Each boot module is responsible for launching LAM executables on remote nodes. The algorithm used to start execution over a set of nodes is up to the module; a module can provide its own functionality or use one of the LAM-provided generic algorithm frameworks. The LAM-provided algorithm frameworks provide all the structure and bookkeeping necessary to launch across a set of nodes (including error detection).

Time	Booting agent	LAM daemon
1	`open_srv_connection()`	
2	Launch remote LAM daemon	
3	`receive_lamd_info()`	`send_lamd_info()`
4	`close_srv_connection()`	
5	`send_universe_info()`	`receive_universe_info()`

Table 2: Sequence of boot API calls during the startup protocols for one LAM daemon. If more than one LAM daemon was booted, steps 3 and 5 would be repeated for each.

If a module chooses to utilize the built-in algorithms, it provides function pointers for callbacks before invoking the algorithm function. The algorithm function will then iterate over all nodes in the set, invoking the module's callback functions to launch executables on remote nodes.

The callback functions required by the provided algorithms are marked as "Algorithm Callback" and are described in Section 3 (page 16). If a module chooses not to use these algorithms, the callback functions do not need to be provided, and NULL should be specified for these function pointers.

The following generic algorithm frameworks are provided in LAM/MPI:

- Linear: A linear approach is used to launch a set of processes: each process is individually started and, if requested, the startup protocols are executed. The next process is not started until the current process has been fully established. This is the simplest algorithm.
- Linear / windowed: Well-suited for remote agents that do not need to wait for remote processes to complete launching, this algorithm uses a linear approach with a sliding window for remote process callbacks – never allowing more than N callbacks to be outstanding at any given time. This algorithm is typically only useful for launching LAM daemons (i.e., processes that require callbacks to the booting agent) when hiding the latency involved in remote launching and/or startup protocols is noticeable, and will fall through to the normal linear algorithm if startup protocols are not required.
- Tree: A tree-based approach is used to launch a set of processes. Interior nodes in the tree will have "helper" processes launched on them in addition to the target RTE process (the "helper" is part of the boot SSI framework and does not need to be provided by the module). The helper process will launch more helpers and/or target RTE processes. The out degree of each node is determined at run time. Note that the tree algorithm is still linear at each node – even though N children processes will be started, the same general linear algorithm is followed (wait for each process to "fully start" before starting the next one).

  Note: As of this writing, the tree algorithm has not yet been implemented.
- Thread: Same as the tree algorithm, except that multiple threads are used, and boot SSI API calls may be overlapped. The big difference between thread and tree is that boot SSI modules that support the thread algorithm must not only be thread safe, but also provide a high level of concurrency when its API calls are simultaneously invoked with orthogonal sets of parameters.

  Note: As of this writing, the thread algorithm has not yet been implemented.

1.3 Error Handling

Any errors that occur within API functions are expected to be handled by that API call, to include outputting error messages. Returning failure statuses from API calls will generally propagate up to the top-level and cause an overall failure, unless otherwise specified.

2 Services Provided by the boot SSI

Several services are provided by the boot SSI that are available to all boot modules.

2.1 Header Files

The following header files must be included (in order) in all module source files that want to use any of the common boot SSI services described in this document:

```
#include <lam−ssi.h>
#include <lam−ssi−boot.h>
```

Both of these files are included in the same location: `share/ssi/include`. If using GNU Automake and the `top_lam_srcdir` macro as recommended in [2], the following can be added to the `AM_CPPFLAGS` macro (remember that `LAM_BUILDING` must be defined to be 1):

```
AM_CPPFLAGS = \
        −I$(top_lam_builddir)/share/include \
        −I$(top_lam_srcdir)/share/include \
        −I$(top_lam_srcdir)/share/ssi/include
```

All three flags are necessary to obtain the necessary header files (note that the build directory is explicitly included in order to support VPATH builds properly, even though it will be redundant in non-VPATH builds).

2.2 Module Selection Mechanism

The selection of which boot SSI module to use persists through the life of the LAM universe. This is not only goverend by the fact that the LAM daemons will make a boot module selection when initially launched and keep using that selection until the end of the universe; it simply does not make sense to change the boot module selection after the LAM universe has been established.

Hence, since all LAM processes will only exist within the timeframe that the LAM universe, the scop of the boot module selection is the life of the process (in the case of the LAM daemon, this coincides with the life of the universe). As such, there will only ever be one module selected during a given process. Selection typically occurs during the initialization of the process. All unselected modules will immediately have their finalize API function invoked (if their initialize API function was invoked), followed by their close function (if provided). They will then be ignored for the duration of the process.

No LAM-provided communication is available between boot modules of peer processes because by definition, there is no LAM run-time environment when the boot modules are initialized. Hence, selection consensus must be able to be achieved independently, or utilize communication channels that are provided by the underlying boot mechanism.

boot module authors should be particularly careful about using environment variables to communicate values between modules. Some underlying boot mechanisms will automatically copy the environment to the remote process (e.g., TM/PBS), but others may not (e.g., `rsh`/`ssh`).

2.3 Types

Some types are used in different API calls and throughout the boot SSI.

2.3.1 `struct lamnode`

This type is used to describe nodes in the LAM universe. It is prototyped in `<lamnet.h>`.

```
struct lamnode {
  int4 lnd_nodeid;
  int4 lnd_type;
  int4 lnd_ncpus;
  int lnd_bootport;
  char *lnd_hname;
  char *lnd_uname;
  struct sockaddr_in lnd_addr;
  LIST *lnd_keyval;
  struct lnd_ssi_boot_nodeinfo *lnd_ssi;
};
```

The individual elements are:

- `lnd_nodeid`: A unique integer identifying a node, from 0 to $(N - 1)$, where N is the total number of nodes in the universe.
- `lnd_type`: A set of bit flags indicating attributes about that node. The most important flags to boot modules are:
  - `NT_BOOT`: Indicates that a node is supposed to be booted.
  - `NT_ME`: Indicates that this node is the local node.
  - `NT_ORIGIN`: Indicates that this node is the origin node.
  - `NT_WASTE`: Indicates that this node should not be used for default scheduling by `mpirun` and `lamexec`. For example, nodes marked with this attribute will not be used for "`mpirun C a.out`".
- `lnd_ncpus`: Number of CPUs on that node.
- `lnd_bootport`: TCP port number used in the startup protocols.
- `lnd_hname`: String name for the node, usually parsed from the boot schema file.
- `lnd_uname`: String username to be used to login on the remote node, or NULL if unnecessary.
- `lnd_addr`: Binary representation of the TCP address of the node.
- `lnd_keyval`: List of key-value pairs parsed from the boot schema file. These key-value pairs provide an extensible method to obtain module-specific infomration from the boot schema file. The `bhostparse()` utility function is typically used to parse boot schema files (see Section 2.5.1), and will fill the `lnd_keyval` list with every "key=value" pair found in the boot schema. For example:

```
inky.cluster.example.com cpu=2
pinky.cluster.example.com cpu=4
blinky.cluster.example.com cpu=4
clyde.cluster.example.com cpu=2 user=jsmith
```

Each `struct lamdnode` instance will have its `lnd_keyval` filled with a list of the "key=value" pairs from the boot schema listed above. The `struct lamdnode` instance for `clyde` will have two entries which each of the others will have one. All keys and values are represented as strings.

Although all "key=value" pairs will be parsed by `bhostparse()` and placed in the `lnd_keyval` list, commonly used keys include:

  - `hostname=<host>`: Specifies the target node's name or IP address. The first token on each line in the boot schema file is implicitly the hostname.
  - `cpu=<NUM>`: Specifies the number of CPUs that LAM may use on the target node.
  - `user=<username>`: Specifies the login name which can be used to remotely login to the node (if different than the username of process owner).
  - `prefix=<path>`: Specifies the path where LAM binaries are installed on the target node.
  - `schedule=(yes|no)`: Specifies whether this node needs to be scheduled for running jobs or not.

- `lnd_ssi`: "Extra" information that each boot module can define. Each module must provide its own definition for the type `lnd_ssi_boot_nodeinfo` (but not necessarily before including the `lamnet.h` file).

2.3.2 `struct psc`

This type is returned in a `LIST` by the `hbootparse()` function (see Section 2.5.2). It contains a list of `argv`-style arrays of processes to start on a target node.

```
struct psc {
  char **psc_argv;
  int4 psc_argc;
  int4 psc_delay;
  int4 psc_flags;
};
```

The members are:

- `psc_argv`: NULL-terminated array of command line tokens to start on the target node.
- `psc_argc`: Length of `psc_argv`.
- `psc_delay`: Delay this many seconds after starting.
- `psc_flags`: Currently unused; reserved for future expansion.

2.3.3 `lam_ssi_boot_proc_t`

This type is used as an argument to boot API functions, indicating which LAM RTE process to start.

```
typedef enum {
  LAM_SSI_BOOT_PROC_LAMD,
  LAM_SSI_BOOT_PROC_RECON,
  LAM_SSI_BOOT_PROC_WIPE,
```

```
  LAM_SSI_BOOT_PROC_MAX
} lam_ssi_boot_proc_t;
```

2.4 Global Variables

Several global variables are available to all boot modules. These variables are `extern`'ed in `<lam-ssi-boot.h>`.

2.4.1 `int lam_ssi_boot_base_server_port`

This `int` is defined and set by the TCP startup protocol functions, and is described in Section 2.5.12.

2.4.2 `int lam_ssi_boot_did`

This `int` is set by the boot SSI initialization function, and will therefore be usable by every boot API function (including `open`). It is the debug stream ID specific to the boot SSI modules (see [1] for a description of LAM/MPI debug streams). If boot modules do not create their own debug streams, they should use `lam_ssi_boot_did`.

Debug streams should be used in conjunction with `lam_ssi_boot_verbose`. Note, however, than debug streams should be used with care (and/or "compiled out" with preprocessor directives when not in use) because even if their output is not displayed, they still invoke a function call and may generate overhead at run-time.

2.4.3 `int lam_ssi_boot_verbose`

The `lam_ssi_boot_verbose` variable will be set by the boot SSI initialization function, and will therefore be usable by every boot API function.

`lam_ssi_boot_verbose` is used to indicate how many "status" messages should be displayed. This does not pertain to error messages that the module may need to print – only messages that are superfluous "what's going on" kind of messages.

The value of this variable should be interpreted as following:

- Less than zero: do not output any status messages.
- Zero: print minimal amounts of status messages. This typically corresponds to the "`-v`" flag on various LAM commands.
- Greater than zero: print status messages – potentially more than if the value were zero; exact meaning is left up to the module. A value of 1000 typically corresponds to the "`-d`" flag on various LAM commands.

2.4.4 `int lam_ssi_boot_optd`

The `lam_ssi_boot_opt` variable is of type `OPT*`. It will be set to a non-`NULL` value after the call to the boot SSI's main open call (i.e., before any API functions of boot modules). It contains the parsed arguments from the command line. See the `all_opt(3)` man page for details on how to use the `OPT` type.

Most boot modules will not need this variable. One of the API functions (described in Section 3.4) receives the `OPT*` variable as a parameter. If later API functions require information from it, the module can save a local copy that can be shared throughout the module. The main purpose of this variable is for utility routines provided by the boot SSI that can be used as API functions, but are not part of individual modules (described in Section 2.5).

2.5 Functions

Several common functions are provided to all boot SSI modules.

2.5.1 bhostparse()

```
#include <boot.h>
int bhostparse(char *filename, struct lamnode **nodes, int *nnodes);
```

Parses a boot schema and returns an array of `struct lamnode` instances. `filename` is the filename of the file to parse. `nnodes` will be allocated and filled by this function (it is the caller's responsibility to free the `nodes` array later); `nnodes` is set to the length of `nnodes`.

`bhostparse()` will parse all key-value pairs and place them in the `lnd_keyval` member on the corresponding entries in the `nodes` array. `bhostparse()` also sets/clears NT_WASTE depending upon whether the key-value pair `schedule=yes/no` has been defined. See section 2.3.1 for information on NT_WASTE.

This function is typically invoked in the `allocate_nodes` API call (see Section 3.5).

2.5.2 hbootparse()

```
#include <boot.h>
int hbootparse(int debug, OPT *ad, char *inet_topo, char *rtr_topo, LIST **proc_list);
```

Find the LAM RTE configuration file in the command line parameters and parse it, doing variable substitution (if `inet_topo` or `rtr_topo` are non-NULL), and return a `LIST` of `argv` arrays containing LAM RTE processes to start on remote nodes. The default LAM RTE configuration file specifies only a single `lamd` (with some associated command line parameters). Other configurations are also possible (such as starting multiple processes on the target node).

The returned `LIST` contains a list of `struct psc` instances. This list can be processed by the boot module to actually start the specified process(es) on the target node(s).

If the boot module uses one of the built-in booting algorithms, this function is typically invoked by the `start_rte_proc` internal API function (see Section 3.11) in conjunction with the `lam_ssi_boot_inet_topo()` function when starting the `lamd` RTE process. For example (many details omitted):

```
int lam_ssi_boot_tm_start_rte_proc(struct lamnode *node, lam_ssi_boot_proc_t which) {
  if (which == LAM_SSI_BOOT_PROC_LAMD) {
    struct psc *p;
    char *inet_buf = lam_ssi_boot_build_inet_topo(node, origin_lamnode, origin_id)

    hbootparse(lam_ssi_boot_did, lam_ssi_boot_optd, inet_buf, NULL, &bootlist);
    for (p = al_top(bootlist); p; p = al_next(bootlist, p)) {
      /* Must not modify the contents of the list items; make duplicates to work with */
      av_cmd = sfh_argv_dup(p->psc_argv);
      ac_cmd = p->psc_argc;
      remote_launch(&av_cmd, &ac_cmd, node);
    }
    free(inet_buf);
    al_free(bootlist);
```

```
  }
}
```

2.5.3 lam_deallocate_nodes()

```
#include <boot.h>
int lam_deallocate_nodes(struct lamnode **nodes, int *nnodes);
```

Utility function to deallocate an array of `struct lamnode`s. It is *not* sufficient to simply `free()` the `nodes` array; this function is provided because each item in the array may contain additional memory that must be specifically freed (e.g., the key=value pairs). Upon return from this function, `*nodes` will be set to NULL, and `*nnodes` will be set to 0.

This function is typically invoked from the `deallocate_nodes()` API function.

2.5.4 lam_ssi_boot_base_check_priority()

```
int lam_ssi_boot_base_check_priority(char *module_name, int base, int want_default, int *priority);
```

A utility function that performs some mundane tasks:

- Assign `priority` to the base priority.
- If the `want_default` flag is set to 1, set `priority` to 75.
- If `module_name` is not NULL and if the environment variable `LAM_MPI_SSI_boot_NAME_priority` is set, set `priority` to the integer value of that variable.

The end result is that `priority` will have a valid priority assigned to it when the function returns.

2.5.5 lam_ssi_boot_base_find_boot_schema()

```
char *lam_ssi_boot_base_find_boot_schema(OPT *args);
```

Analyzes `argc` and `argv` to find the boot schema and verify that it exists.

Note that `OPT*` is an internal LAM type for holding command line parameters. Its use is documented in the `all_opt(3)` man page.

`opt` is not modified by this function. The return value will be a string representing the filename of the found boot schema, or NULL if nothing was found (indicating an error). If nothing is found, an appropriate error message will be printed.

If an absolute pathname is found, it is used. If a relative pathname is found, it is checked against the present directory, the `$TROLLIUSHOME/etc` directory, the `$LAMHOME/etc` directory, and finally the LAM `$sysconf` directory (selected at configuration time).

A `lam_ssi_boot_verbose` value of one or larger will trigger output to the `lam_ssi_boot_did` debug stream.

This function is typically invoked in the `parse_options` API call (see Section 3.4).

2.5.6 lam_ssi_boot_base_find_hostname()

```
char *lam_ssi_boot_base_find_hostname(OPT *args);
```

This function is used when the `lamgrow` command was used to invoke the boot SSI. This is because the `lamgrow` command does not provide a boot schema file – a single hostname is provided on the command line for growing the current LAM universe. This function analyzes `argc` and `argv` to find a string hostname or IP address and verify that it exists.

Note that `OPT*` is an internal LAM type for holding command line parameters. Its use is documented in the `all_opt(3)` man page.

`opt` is not modified by this function. The return value will be a string representing the found hostname/address or `NULL` if nothing was found (indicating an error). If nothing is found, an appropriate error message will be printed.

A `lam_ssi_boot_verbose` value of one or larger will trigger output to the `lam_ssi_boot_did` debug stream.

This function is typically invoked in the `parse_options` API call (see Section 3.4).

2.5.7 lam_ssi_boot_base_lamgrow()

```
char *lam_ssi_boot_base_lamgrow(char *hostname, struct lamnode **nodes,
                                int *nnodes, int *origin);
```

This function is used when the `lamgrow` command was used to invoke the boot SSI. This is because the `lamgrow` command does not provide a boot schema file – a single hostname is provided on the command line for growing the current LAM universe. This function analyzes the current LAM universe and generates an array of `struct lamnode` instances based on its contents, to include an instance for the new node to be booted. Only the entry for the new node will have the `NT_BOOT` flag set.

`hostname` is the string host name or IP address of the node to be booted. `nodes` is allocated and filled by this function (it is the caller's responsibility to free the `nodes` array later); `nnodes` is set to the length of `nodes`. The origin argument is filled with the origin node's ID. Note that it is not necessarily the same node as the node that is invoking `lamgrow`.

Depending on how `lamgrow` was invoked, it is possible that the array of `struct lamnode` instances may contain entries for "invalid" nodes (see the `lamgrow(1)` man page for more details). Such entries will have a node ID of `NOTNODEID`, and all their other data will be invalid. Although these nodes must be skipped by the booting algorithms (all the provided algorithms properly skip them), space must be allocated for them in all internal arrays and tables.

This function is typically invoked in the `allocate_nodes` API call (see Section 3.5).

2.5.8 lam_ssi_boot_base_ioexecvp()

```
int lam_ssi_boot_base_ioexecvp(char **cmdv, int showout, char *outbuff, int outbuffsize);
```

This function is used to execute a command. It is typically used to execute a command that starts LAM daemons or starts a proxy process which then starts LAM daemons.

`cmdv` contains the command to be executed. The function can direct command stdout to buffer and/or stdout. `showout` is used to control this. If command stdout is to be directed to output buffer, then `outbuff` should point to the buffer for stdout data and `outbuffsize` should contain size of this buffer.

This function is typically indirectly invoked in the `start_application` API call (see Section 3.10).

2.5.9 `lam_ssi_boot_build_inet_topo()`

```
char *lam_ssi_boot_build_inet_topo(struct lamnode *dest_node, struct lamnode origin_node,
                                   int origin);
```

This function is typically invoked before calling `hbootparse()`. It examines the command line arguments (in `lam_ssi_boot_optd`) and builds a string suitable for replacement as the `$inet_topo` in LAM RTE configuration files. This string is typically later passed to the `hbootparse()` function for this exact purpose.

The string that is returned from this function is `malloc`'ed memory; it must later be freed by the caller.

2.5.10 `lam_ssi_boot_do_common_args()`

```
int lam_ssi_boot_do_common_args(OPT *aod, int *argc, char ***argv);
```

Utility function to handle some mundane argument handling (e.g., adding "-v" and/or "-d" to `argv` if they are found in `aod`). It is typically invoked with the `argv` of a process to start on a remote node, allowing "-v" and "-d" to propogate to remote processes.

2.5.11 Built-in Algorithms

The boot SSI framework provides several generalized algorithms to launch processes across a set of nodes. These algorithms are generally invoked from within boot module API calls. The algorithms, in turn, will make callbacks into the module to perform the actual work (e.g., launch a process). The algorithms perform all the necessary bookkeeping and timing to execute the entire set of tasks as well as exchange all startup protocol information (if necessary). Note, however, that these functions will all skip nodes that are not either not marked with the `NT_BOOT` flag or have a node ID that is equal to `NOTNODEID`.

Note that the use of these functions is not mandatory. They are simply provided as drop-in algorithms so that modules do not need to write their own.

Section 1.2 generally describes the available algorithms. Their names are long because of the SSI prefix rule. Each of the functions below have the same signature:

```
int algorithm(struct lamnode *nodes, int nnodes, int want_startup_protocol,
              lam_ssi_boot_proc_t which, int *num_started);
```

The arguments are as follows:

- IN: `nodes` is an array of nodes to boot across.
- IN: `nnodes` is the length of the `nodes` array.
- IN: `want_startup_protocol` is a flag indicating whether the function should invoke the module's startup protocol functions at the apporpriate times during the boot process. It should only be set to 1 when booting LAM daemons; 0 all other times.
- IN: `which` is an enum indicating what kind of process to launch (see Section 2.3.3).
- OUT: `num_started` will be filled in by the algorithm indicating how many nodes were actually booted.

The provided algorithm functions are:

- `lam_ssi_boot_base_alg_linear()`: Simple linear process-launching algorithm.
- `lam_ssi_boot_base_alg_linear_windowed()`: Linear algorithm with a sliding window for the startup protocols. This is especially well-suited for boot environments where remote process invocation latency can be hidden by not waiting for a remote action to finish before progressing onto the next action. This algorithm guarantees that there will never be more than N outstanding agents waiting to exchange startup protocol information.

  This algorithm is especially relevant if the built-in TCP startup protocols are used (described in Section 2.5.12), because at least some operating system TCP stacks only allow a limited number of clients to be pending on a listening socket. Hence, using the windowed algorithm will guarantee that that operating system limit is never exceeded.

  The default window value is 5, and can be changed by setting the `boot_base_linear_win_size` SSI run-time parameter.
- `lam_ssi_boot_base_alg_tree()`: N-way tree-based process-launching algorithm (N is determined at run-time). This algorithm will first launch a tree of "helper" executables – one on each interior node (and potentially exterior nodes) in the tree. After the helper tree is established, the target LAM RTE process will be launched, and any necessary startup information will be exchanged.

  Note: This function is not yet implemented.
- `lam_ssi_boot_base_alg_thread()`: Same as the tree-based algorithm, except that each interior node will use multiple threads to perform its actions and may invoke the boot API functions multiple times simultaneously within a single process. boot modules that use the thread algorithm must not only be thread safe, but also provide a high degree of concurrency when its API calls are simultaneously invoked with orthogonal sets of parameters.

  Note: This function is not yet implemented.

2.5.12 TCP-Based Startup Protocols

Most (if not all) boot modules will be able to use the generalized TCP startup protocol functions since TCP is likely able to be used for such meta-information exchanges regardless of the underlying communication network. If TCP connectivity is not available, the boot module will need to provide startup protocols itself.

These functions eliminate the need for most boot modules to provide their own functions for several of the boot API calls. Since these functions can be used for the corresponding API functions, only their names are listed below – their signatures and behavior are described in Section 3:

- `lam_ssi_boot_base_open_srv_connection()`
- `lam_ssi_boot_base_send_lamd_info()`
- `lam_ssi_boot_base_receive_lamd_info()`
- `lam_ssi_boot_base_close_srv_connection()`
- `lam_ssi_boot_base_send_universe_info()`
- `lam_ssi_boot_base_receive_universe_info()`

Note that these functions are subject to operating system limits such as how many pending clients can be held on a listening socket. Some operating systems have a surprisingly low backlog limit. Modules that utilize booting algorithms that could have multiple clients simultaneously calling the server back should be aware of this limitation, and either use multiple servers or some kind of windowed protocol (e.g., the linear windowed algorithm described in Section 2.5.11).

The information that must be exchanged is:

- From the `lamd` to `lamboot`, send the following:
  - UDP port number that the `lamd` will use for normal operations
  - Any other information required to call the `lamd` back to pass the universe information
- From `lamboot` to the `lamd`, loop sending the following information to each `lamd` in the LAM universe:
  - Integer node identifier of that `lamd` (from 0 to $N - 1$), or `NOTNODEID` if it is not a valid node
  - Either the byte-packed IP address of the `lamd` or the string hostname of the `lamd`
  - Integer UDP port number that the `lamd` is listening on
  - Integer (bit flags) for the node that the `lamd` is running on (see Section 2.3.1)
  - Integer number of CPUs that the `lamd` thinks that the node has

3 boot SSI Module API

This is version 1.0.0 of the boot SSI module API.

Each boot SSI module must export a `struct lam_ssi_boot_1_0_0` named `lam_ssi_boot_<name>_module`. This type is defined in Figure 1. This `struct` contains a small number of items, one of which is a function pointer that may return a pointer to the `struct` shown in Figure 2.

```
typedef struct lam_ssi_boot_1_0_0 {
  lam_ssi_1_0_0_t lsb_meta_info;

  /* Initialize / finalize functions */

  lam_ssi_boot_init_fn_t lsb_init;
  lam_ssi_boot_finalize_fn_t lsb_finalize;
} lam_ssi_boot_1_0_0_t;
```

Figure 1: The boot type for exporting the initialization and finalization API function pointers.

The majority of the elements in Figures 1 and 2 are function pointer types; each is discussed in detail below. When describing the function prototypes, the parameters are marked in one of three ways:

- IN: The parameter is read – but not modified – by the function.
- OUT: The parameter, or the element pointed to by the parameter may be modified by the function.
- IN/OUT: The parameter, or the element pointed to by the parameter is read by, and may be modified by the function.

```
typedef struct lam_ssi_boot_actions_1_0_0 {

  /* Boot API function pointers */

  lam_ssi_boot_parse_options_fn_t lsba_parse_options;
  lam_ssi_boot_allocate_nodes_fn_t lsba_allocate_nodes;
  lam_ssi_boot_verify_nodes_fn_t lsba_verify_nodes;
  lam_ssi_boot_prepare_boot_fn_t lsba_prepare_boot;
  lam_ssi_boot_start_rte_procs_fn_t lsba_start_rte_procs;
  lam_ssi_boot_deallocate_nodes_fn_t lsba_deallocate_nodes;

  /* Algorithm callback functions (optional) */

  lam_ssi_boot_start_application_fn_t lsba_start_application;
  lam_ssi_boot_start_rte_proc_fn_t lsba_start_rte_proc;

  /* Startup protocol: sending individual lamd info */

  lam_ssi_boot_open_srv_connection_fn_t lsba_open_srv_connection;
  lam_ssi_boot_send_lamd_info_fn_t lsba_send_lamd_info;
  lam_ssi_boot_receive_lamd_info_fn_t lsba_receive_lamd_info;
  lam_ssi_boot_close_srv_connection_fn_t lsba_close_srv_connection;

  /* Startup protocol: broadcasting universe info */

  lam_ssi_boot_send_universe_info_fn_t lsba_send_universe_info;
  lam_ssi_boot_receive_universe_info_t lsba_receive_universe_info;

} lam_ssi_boot_actions_1_0_0_t;
```

Figure 2: The boot type for exporting the main action API function pointers.

Function prototypes are also marked as one of three classes:

- Boot API: This function is part of the set of functions that start processes on nodes. This function will be invoked by the LAM infrastructure.
- Algorithm Callback: This function is invoked as a callback from the LAM-provided boot algorithm function frameworks. Although these functions serve as useful abstractions, they are only required if the LAM-provided boot algorithms are used.
- Protocol: This function is part of the set of functions that exchange startup protocol information.

boot module writers looking for insight into how the API is used should also look at the source code for `lamboot`, `recon`, and `wipe`.

3.1 Data Item: `lsb_meta_info`

`lsb_meta_info` is the SSI-mandated element contains meta-information about the module. See [2] for more information about this element.

3.2 API Function Call: `lsb_init`

- Type: `lam_ssi_boot_init_fn_t`

```
typedef const lam_ssi_boot_actions_t *(*lam_ssi_boot_init_fn_t)
  (lam_ssi_boot_location_t where, int *priority);
```

- Arguments:
  - IN: `where` is an `enum` indicating where this module is being initialized. The value will be one of the following:
    - `LAM_SSI_BOOT_LOCATION_ROOT`: The module is being invoked on the root of the boot. This typically means a user-level command such as `lamboot`, `recon`, `wipe`, or `lamgrow`.
    - `LAM_SSI_BOOT_LOCATION_INTERIOR`: This module is being invoked in an interior node of the boot. This may mean that a hierarchical boot algorithm is being used, and that this process is a "helper" launching application. It directly implies that this node has both a parent and one or more children.
    - `LAM_SSI_BOOT_LOCATION_LEAF`: All other cases. This includes the LAM daemon itself (see below).
  - OUT: `priority` is the priority of this module, and is used to choose which module will be selected from the set of available modules at run time.
- Return value: Either `NULL` or a pointer to the `struct` shown in Figure 2.
- Description: If the module wants to be considered for selection, it must return a pointer to the `struct` shown in Figure 2 that is filled with relevant data and assign an associated priority to `priority`. See [2] for more details on the priority system and how modules are selected at run time.

  If the module does not want to be considered during the negotiation for this communicator, it should return `NULL` (the value in `priority` is then ignored).

  Note that the LAM daemon itself must also initialize the boot SSI and come to the same selection conclusion as its peers. Although the LAM daemon will not use any of boot API functions to launch remote processes, it will use the startup protocol functions to exchange location information with a peer.

3.3 API Function Call: `lsb_finalize`

- Type: `lam_ssi_boot_finalize_fn_t`

```
typedef int (*lam_ssi_boot_finalize_fn_t)(void);
```

- Arguments: None.
- Return value: Zero on success, nonzero otherwise.

- Description: Finalize the use of this module. It is the last function to be called in the scope of this module's selection before the module close function. It should release any resources allocated during the life of this scope.

3.4 API Function Call: `lsba_parse_options`

- Type: `lam_ssi_boot_parse_options_fn_t`

  typedef int (∗lam_ssi_boot_parse_options_fn_t)(OPT ∗args, **int** bhost_schema_args);

- Arguments:
  - IN: `args` contains the command line arguments.
  - IN: `bhost_schema_args` is 1 if the `argc`/`argv` pair contains a boot schema filename (e.g., from `lamboot`, `recon`, and `wipe`), and 0 if the pair contains a string hostname/IP address (e.g., from `lamgrow`).
- Return value: Zero on success, nonzero otherwise.
- Description: The module can examine the command line parameters (see the `all_opt(3)` man page for details on how to use the `OPT` type).

  This API function typically makes use of the two utility functions `lam_ssi_boot_base_find_boot_schema()` and `lam_ssi_boot_base_find_hostname()` (described in Sections 2.5.5 and 2.5.6, respectively).

3.5 API Function Call: `lsba_allocate_nodes`

- Type: `lam_ssi_boot_allocate_nodes_fn_t`

  typedef int (∗lam_ssi_boot_allocate_nodes_fn_t)(**struct** lamnode ∗∗nodes, **int** ∗nnodes, **int** ∗origin);

- Arguments:
  - OUT: `nodes` is a pointer to a `struct lamnode` that this function is expected to fill with an array of `struct lamnode`s.
  - OUT: `nnodes` is a pointer to an `int` that this function is expected to fill with the length of `nodes` array.
  - OUT: `origin` is a pointer to an `int` that this function is expected to fill with an index into the `nodes` array representing the element for this node.
- Return value: Zero on success, nonzero otherwise.
- Description: Create and fill in a `lamnode` structure for each node to be booted. There are few requirements on the completeness of the structure, but all unused fields should be zeroed out before returning. Additionally, the `lnd_type` field for the origin member should have the `NT_ORIGIN` and `NT_ME` flags set. Additionally, the total number of nodes must be correct.

  Note that this function's actions may be determined by the value of the `bhost_schema_args` flag to the `parse_options()` API call.

The `deallocate_nodes()` API call should later be used to free the memory associated with the nodes list.

This API function typically makes use of the two utility functions `bhostparse()` and `lam_ssi_boot_base_lamgrow()` (described in Sections 2.5.1 and 2.5.7, respectively).

3.6 API Function Call: `lsba_verify_nodes`

- Type: `lam_ssi_boot_verify_nodes_fn_t`

```
typedef int (*lam_ssi_boot_verify_nodes_fn_t)(struct lamnode *nodes, int nnodes);
```

- Arguments:
  - IN: `nodes` is the array of `struct lamnode`s returned by the `allocate_nodes()` API call.
  - IN: `nnodes` length of the `nodes` array.
- Return value: Zero on success, nonzero otherwise.
- Description: Last sanity check on the node list. If possible, check the node list for conditions such as (but not limited to):
  - Existence of node (e.g., try to resolve IP names)
  - Permission to execute on node
  - Ensure that the local node is in the list
  - If the number of nodes is greater than one, ensure that the local address is not 127.0.0.1 if using standard IP-passing scheme

  After this call, `lamnodes` should be filled in with enough information for the boot SSI to contact each of the target notes. As such, it needs to determine which entry in the array is the origin (this may have been determined by the `allocate_nodes()` API call, but in some cases, it is not possible to determine it until here in `verify_nodes()`).

3.7 API Function Call: `lsba_prepare_boot`

- Type: `lam_ssi_boot_prepare_boot_fn_t`

```
typedef int (*lam_ssi_boot_prepare_boot_fn_t)(void);
```

- Arguments: None.
- Return value: Zero on success, nonzero otherwise.
- Description: Perform any setup work that might be needed by the `start_rte_procs()` API call, but that only needs to be done once. For example, on BProc architectures, boot modules may generate the `argv` arrays for starting up the LAM daemons.

3.8 API Function Call: lsba_start_rte_procs

- Type: lam_ssi_boot_start_rte_procs_fn_t

```
typedef int (*lam_ssi_boot_start_rte_procs_fn_t)
  (struct lamnode *nodes, int nnodes, lam_ssi_boot_proc_t which, int *num_started);
```

- Arguments:
  - IN: nodes is the array of nodes to start processes on
  - IN: nnodes is the length of the nodes array.
  - IN: which is an enum specifying which LAM RTE process to start.
  - OUT: num_started is a pointer to an int indicating how many processes were successfully started.
- Return value: Zero on success, nonzero otherwise.
- Description: Takes a nodes array and starts a LAM RTE process on each node.

  The function must only launch on nodes that have the NT_BOOT flag set on their type and do not have a node ID of NOTNODEID. All other nodes must be skipped.

  Note that there is both a return code from this function (indicating overall success or failure) and a separate count of how many processes were started. This is for the case where *some* processes may start properly, but others fail. The num_started argument tells the caller how many processes now need to be cleaned up. The nodes array can be examined to find out exactly which nodes were successfully booted (NT_BOOT must be reset on the lnd_type of nodes that were successfully started).

  If any of the LAM-provided boot algorithms are used, this is the function that typically invokes them.

3.9 API Function Call: lsba_deallocate_nodes

- Type: lam_ssi_boot_deallocate_nodes_fn_t

```
typedef int (*lam_ssi_boot_deallocate_nodes_fn_t)(struct lamnode **nodes, int *nnodes);
```

- Arguments:
  - IN/OUT: nodes is an array of nodes.
  - IN/OUT: nnodes is the length of the nodes array.
- Return value: Zero on success, nonzero otherwise.
- Description: Clean up any memory associated with the node allocation step. Although not required, modules are encouraged to reset nodes and nnodes to NULL and 0, respectively, when the function returns. This function will be called only after the nodes information is no longer needed. This function typically invokes lam_deallocate_nodes() (see Section 2.5.3).

3.10 Algorithm Callback Function Call: `lsba_start_application`

- Type: `lam_ssi_boot_start_application_fn_t`

```
typedef int (*lam_ssi_boot_start_application_fn_t)
  (char ***argv, int *argc, int num_procs, struct lamnode *node);
```

- Arguments:
  - IN: `argv` is an array of `argv`-style command line arguments; i.e., an array of commands to start.
  - IN: `argc` is an array of indicating how long each array is in the `argv` array.
  - IN: `num_procs` is the length of the first dimension of `argv`.
  - IN: `node` is a pointer to a single `struct lamnode` indicating which node to start on.
- Return value: Number of processes successfully started.
- Description: Launch the specified processes on the specified node. Return the number of processes successfully booted. Hence, if the return value is equal to `num_procs`, the function completed successfully. There is no mandate that processes be started in the order they exist in `argv`.

  The return value is explicitly vague so that modules can get "even more parallelism" if they happen to use a remote startup agent that provides a high degree of parallelism.

  It is incorrect to use this function directly from a boot algorithm to launch a LAM RTE process (e.g., `lamd`, `recon`, `wipe`). Use the `start_rte_proc()` API function instead. This function should only be used by a boot algorithm to start up other instances of the boot algorithm (i.e., "helper" executables).

3.11 Algorithm Callback Function Call: `lsba_start_rte_proc`

- Type: `lam_ssi_boot_start_rte_proc_fn_t`

```
typedef int (*lam_ssi_boot_start_rte_proc_fn_t)
  (struct lamnode *node, lam_ssi_boot_proc_t which);
```

- Arguments:
  - IN: `node` is a pointer to a single `struct lamnode` indicating where the process should be started.
  - IN: `which` is an `enum` indicating what kind of LAM RTE process should be started (see Section 2.3.3).
- Return value: Zero on success, nonzero otherwise.
- Description: Start a LAM RTE process on the specified node. This can use the `start_application()` API function (in fact, it is encouraged).

  This function exists because the boot algorithm should not need to know any of the details about starting a LAM RTE process. This function provides an upcall to give the boot algorithm an abstract mechanism to launch a LAM RTE process.

3.12 Protocol Function Call: lsba_open_srv_connection

- Type: lam_ssi_boot_open_srv_connection_fn_t

  typedef int (∗lam_ssi_boot_open_srv_connection_fn_t)(**struct** lamnode ∗nodes, **int** nnodes);

- Arguments:
  - IN: nodes is a pointer to an array of struct lamnodes that are expected to connect to this process.
  - IN: nnodes is the length of the nodes array.
- Return value: Zero on success, nonzero otherwise.
- Description: Open a private, server-side communication endpoint (i.e., the channel will only be used within the boot SSI) that the LAM daemon will connect back to. This function will only be called once per instance of the module, meaning that if you use BOOT_LINEAR, it will only be called once (on the origin node) but on BOOT_TREE it may be called multiple times (one for each helper process).

  Note that the addresses given in nodes may or may not be the actual clients that connect. There are some valid network architectures where connections may seem to come from addresses other than what are listed in the nodes array. It is suggested to boot module authors that unless the special "boot promiscuous mode" is enabled in LAM, only accept connections from the addresses listed in the nodes array (when possible). However, when "promiscuous mode" is enabled, accept connections from anyone, and rely on the connector to identify themselves in the boot protocol.

3.13 Protocol Function Call: lsba_send_lamd_info

- Type: lam_ssi_boot_send_lamd_info_fn_t

  typedef int (∗lam_ssi_boot_send_lamd_info_fn_t)(OPT ∗args, **int** dli_port);

- Arguments:
  - IN: args contains the command line parameters.
  - IN: dli_port is the UDP port number that the local LAM daemon is listening on for normal operations.
- Return value: Zero on success, nonzero otherwise.
- Description: Open a connection back to the booting agent, send relevant location information (e.g., the LAM's UDP port number), and then closes the connection. It is assumed that the information necessary to connect back to the invoking agent is either in the command line arguments (see the all_opt(3) man page for details on how to access the OPT type) or in the environment.

3.14 Protocol Function Call: lsba_receive_lamd_info

- Type: lam_ssi_boot_receive_lamd_info_fn_t

  typedef int (∗lam_ssi_boot_receive_lamd_info_fn_t)(**struct** lamnode ∗nodes, **int** nnodes);

- Arguments:
  - IN/OUT: `nodes` is a pointer to an array of `struct lamnode`s that were successfully started.
  - IN: `nnodes` is the length of the `nodes` array.
- Return value: Zero on success, nonzero otherwise.
- Description: Accept a connection from a LAM daemon and receive the information it sends back. The function is provided with an array of `struct lamnode` entries, one of which will correspond to the LAM daemon that will be contacting it. It is up to the function to figure out which one is responding. When finished, close the connection. A new connection is used to broadcast the information at a later time.

  In the case that only one LAM daemon can be communicating with the function, (for example, when the boot algorithm is linear), then `nnodes` will one and the job of searching is much easier.

3.15 Protocol Function Call: `lsba_close_srv_connection`

- Type: `lam_ssi_boot_close_srv_connection_fn_t`

```
typedef int (*lam_ssi_boot_close_srv_connection_fn_t)(void);
```

- Arguments: None.
- Return value: Zero on success, nonzero otherwise.
- Description: Close the channel opened during the `open_srv_connection()` API function.

3.16 Protocol Function Call: `lsba_send_universe_info`

- Type: `lam_ssi_boot_send_universe_info_fn_t`

```
typedef int (*lam_ssi_boot_send_universe_info_fn_t)(struct lamnode *nodes, int nnodes, int which);
```

- Arguments:
  - IN: `nodes` is an array of nodes that the information needs to be broadcast to.
  - IN: `nnodes` is the length of the `nodes` array.
  - IN: `which` is an index into the `nodes` array indicating which node to connect and send the information to.
- Return value: Zero on success, nonzero otherwise.
- Description: Connect to LAM daemon and send the union of all the LAM location information (i.e., send information about all the peer LAM daemons that comprise the LAM universe).

  This function opens a connection to a target LAM daemon, sends the information, and disconnects.

3.17 Protocol Function Call: `lsba_receive_universe_info`

- Type: `lam_ssi_boot_receive_universe_info_t`

  typedef int (∗lam_ssi_boot_receive_universe_info_t)(**struct** lamnode ∗∗universe, **int** ∗universe_size);

- Arguments:
  - OUT: `universe` is a pointer to an (as yet unallocated) array of information that will be received.
  - OUT: `universe_size` is a pointer to an `int` that will be filled to be the length of the `universe` array.
- Return value: Zero on success, nonzero otherwise.
- Description: After the LAM daemon communicates its port information to the booting process, it waits for information about the entire run-time universe. This function is where it waits for that information. It returns the information returned about all neighbors.

  Similar to the `receive_lamd_info()` API function, this function should accept the connection, read the information, and close the connection when finished reading.

4 To Be Determined

Things that still need to be addressed:

- Tree algorithm needs to be implemented.
- Thread algorithm needs to be implemented.
- It is likely that future versions of this API will need to adjust some of the API calls to allow for the tree and thread algorithms – allowing arrays of lamd and universe info to be passed, etc.

5 Acknowledgements

This work was supported by a grant from the Lily Endowment National Science Foundation grant 0116050, and used research and development resources of the University of Pennsylvania Liniac Project.

References

[1] Brian Barrett, Jeff Squyres, and Andrew Lumsdaine. *LAM/MPI Design Document*. Open Systems Laboratory, Pervasive Technology Labs, Indiana University, Bloomington, IN. See `http://www.lam-mpi.org/`.

[2] Jeffrey M. Squyres, Brian Barrett, and Andrew Lumsdaine. The system services interface (SSI) to LAM/MPI. Technical Report TR575, Indiana University, Computer Science Department, 2003.