Native LaunchMON Prototype Support on FLUX

7/25/2013

Dong H. Ahn, Livermore Computing Division, Lawrence Livermore National Laboratory, ahn1@llnl.gov

Goal

This prototyping effort aims to demonstrate that FLUX can natively support LaunchMON. The LaunchMON software layer that natively resides in this resource manager (RM) can leverage the underlying RM mechanisms to enhance its capability and performance. This layer will then form a basis within FLUX for providing seamless daemon launching to many existing run-time tools and infrastructure, and further make it extremely easy to create new run-time tools. Rich support for run-time tools is essential for FLUX, as its workloads are expected to be increasingly larger, more diverse and complex. Thus, the native LaunchMON support will position FLUX to meet some of the emerging usability challenges with scalable and easily-composable run-time tools and middleware.

Background

Many run-time tools that target parallel and distributed environments must co-locate and synchronize a set of daemons with the distributed processes of the target application. However, efficient and portable deployment of these daemons on large-scale systems had been an unsolved problem. Since 2008, we have responded to this gap with LaunchMON, a scalable and portable general-purpose infrastructure for launching tool daemons. Its API allows tools to identify all processes of a target application, and to launch daemons on the relevant nodes and control daemon interaction.

Since its inception, the LaunchMON project has been highly successful. Many run-time tools and middleware have used this API so that they do not have to rely on ad hoc launching mechanisms, which often present both portability and scalability challenges. With LaunchMON support, tools/middleware builders could quickly focus on the core functionality that they wanted and needed to innovate. Further, as we have ported LaunchMON to an increasingly wider range of RMs and platforms—e.g., SLURM, OpenRTE, Cray ALPS and the Blue Gene family, the client tools have gained portability and scalability for free on high-end machines.

Client software that uses LaunchMON includes code-development tools like LLNL's Stack Trace Analysis Tool (STAT), Allinea DDT and Rogue Wave Software ThreadSpotter, as well as middleware like a Scalable Checkpoint/Restart (SCR) prototype, and SPINDLE (Scalable Parallel Input Network for Dynamic Environment).

Problems

As part of its core services, LaunchMON provides the tool with compute-nodes information relevant to the target application processes and also synchronizes the co-located tool processes with these target processes. For high portability, we decided, very early on, to use a de factor standard interface called the MPIR process acquisition interface, which provides most of the desired functionality portably on many RMs.

Unfortunately, the portability of this interface comes with several disadvantages. In particular, it is designed as a debug interface and hence requires its user to trace (i.e., via the ptrace system call) the states of the RM starter process (e.g., srun) through spawning and controlling the target application. Because ptrace keeps multiple tracing processes from attaching to a same target process, it is often not easy to coordinate and combine the tools that use this mechanism. Tools must coordinate the use of this mechanism among themselves through an ad hoc attach-and-detach scheme by which each takes turns to to trace the starter process. Because this approach is error-prone, only few tools that use LaunchMON currently implement it.

Approach

It is our view that the current LaunchMON implementation is a stepping stone towards a more ideal service form whereby its API is more deeply integrated into the RM software and hence directly uses the native RM mechanisms. However, such tight interactions require that LaunchMON and the RM must be designed and implemented together to be effective. Through co-design efforts, the RM will implement and provide some of missing native mechanisms that LaunchMON needs to address its deficiencies such as composability issues and to improve its performance and scalability further. With the FLUX project newly started, we have a unique opportunity to realize this potential.

As part of this prototyping effort, therefore, we will modify LaunchMON to have a deeper RM integration. Using the native FLUX mechanisms, we will get rid of needing to trace the RM starter process. With this approach, we will be able to better combine and coordinate distinct, yet complementary, tools/middleware sessions together. Further, we can enhance the overall performance and memory utilization of LaunchMON.

More specifically, LaunchMON will collect the global MPIR process table by gathering through CMB the relevant information distributed across the key-vale stores. This distributed MPIR process table associated with the target application will rid LaunchMON front-end of needing to broadcast the entire global process table, which can become relatively big at large scale, to the back-end daemons. Each back-end will also discover the local process table by directly querying the key-value stores. Finally, LaunchMON will use the native APIs of FLUX to spawn back-end daemons, to co-locate them with the application processes, and to monitor and detect the liveliness and failures of these daemons.

The following further details the FLUX mechanisms the new approach requires as well as a concrete modification proposal based on the LaunchMON 1.0 release branch.

New Engine

In our current implementation, LaunchMON Engine is the main bridge between the target RM and the distributed APIs of LaunchMON. On one hand, the Engine gets co-located with the RM starter process, traces the states of this process, and converts low-level debug events generated from the tracing into a set of well-defined RM events. On the other hand, the Engine also establishes a TCP/IP socket connection with the run-time of the front-end API, which runs in a POSIX thread of the client software's front-end. Through a communication protocol called LMONP, the Engine receives commands from the front-end and also sends RM state information back to the front-end.

LaunchMON Engine code is quite complex mainly because it must trace the RM starter process, and to do this its code has debugger internals such as symbol table parsing and management, process-tracing through ptrace and various machine models. Further, it must also abstract out several key differences in the MPIR process acquisition interface implementations as different RMs can provide different interface extensions. (Note that a recently published, MPI forum-sanctioned document called the MPIR Process Acquisition Interface Version 1.0 describes all of the extensions.) To deal with the complexity, the Engine uses complex object-oriented design patterns such as class templates to parameterize the Engine code with different machine models and class inheritance to package up the basic Engine actions into an abstract base class etc.

Events (lmonp_fe_to_fe_msg_e)	Direction	**Description**	**RM**
lmonp_conn_ack_no_error	Engine→FE	Connect-ack with no parse error	No
lmonp_conn_ack_parse_error	Engine→FE	Connect-ack with parse errors found	No
lmonp_stop_at_launch_bp_spawned	Engine→FE	RM starter process hits the MPIR_Breakpoint function with MPIR_DEBUG_SPAWNED state	Yes
lmonp_rminfo	Engine→FE	rminfo is filled (e.g., RM type etc)	Yes
lmonp_stop_at_launch_bp_abort	Engine→FE	RM starter hits the MPIR_Breakpoint function with MPIR_DEBUG_ABORTING state	Yes
lmonp_proctable_avail	Engine→FE	MPIR process table filled–typically right after lmonp_stop_at_launch_bp_spawned	Yes
lmonp_resourcehandle_avail	Engine→FE	Resource handle info filled–typically right after lmonp_stop_at_launch_bp_spawned	Yes
lmonp_stop_at_first_exec lmonp_stop_at_first_attach lmonp_stop_at_loader_bp lmonp_stop_at_thread_creation lmonp_stop_at_thread_death lmonp_stop_at_fork_bp lmonp_stop_not_interested		These are not communicated thus far. But we may need some RM support to implement their equivalent handlers. (Further reviews are needed)	Maybe
lmonp_terminated	Engine→FE	RM starter terminated	Yes
lmonp_exited	Engine→FE	The main thread of the RM starter exited	Yes
lmonp_detach_done	Engine→FE	Detach done	Yes
lmonp_kill_done	Engine→FE	Kill done	Yes
lmonp_stop_tracing	Engine→FE	Engine failed and done its cleanup	Yes
lmonp_bedmon_exited	Engine→FE	Back-end daemons exited	Yes
lmonp_mwdmon_exited	Engine→FE	Middleware daemons exited	Yes
lmonp_detach	FE→Engine	Please-detach cmd	Yes
lmonp_kill	FE→Engine	Please-kill cmd	Yes
lmonp_shutdownbe	FE→Engine	Please-shutdown-back-ends cmd	Yes
lmonp_cont_launch_bp	FE→Engine	Please-continue-from-MPIR_Breakpoint cmd	Yes

Table 1: Communication protocol between LaunchMON Engine and Front-End (FE)

As part of our modification plan, we propose to create a new Engine implementation that directly uses native RM APIs instead of the MPIR debug interface, while still communicating with the front-end through the same wire protocol (LMONP). This proposal has three main advantages in contrast to modifying the LaunchMON API implementation : 1) it will significantly reduce the code complexity of the Engine, which then can improve the robustness of LaunchMON under FLUX; 2) it will minimize the changes needed for the LaunchMON front-end API and thus reduce backward-compatible and portability concerns: i.e., we desire the same code base of the LaunchMON front-end API work well on other RMs as well; and 3) it will then still continue to allow the client software's front-end to be run on any arbitrary location with TCP/IP connectivity (e.g., a user's desktop).

Clearly, this new Engine implementation will need alternative mechanisms using FLUX-provided primitives to be able to continue to serve as the bridge between the front-end and FLUX. To show the interactions between the Engine and the front-end, Table 1 lists the current wire protocol defined between these two components, which is a part of LMONP. The new Engine will use FLUX mechanisms including launch, control, query and monitor APIs to enable this commands and control communication structure. In this table, the events shaded in blue will likely require new RM APIs within the new Engine. The events in yellow need further reviews. The lmonp_fe_to_fe_msg_e enumerator is currently defined in a file: launchmon/src/lmon_api/lmon_lmonp_msg.h#l93.

Failure Handling Semantics Enforcement

LaunchMON defines a rigorous failure handing semantics when one or more LaunchMON components failed, which is described in one of the README files: README.ERROR_HANDLING. Some of the handling mechanisms will need support from the RM. For example, when back-end tool processes fail, LaunchMON wants to get notified of this event to react to this properly. Some RMs do provide these mechanisms better than others, and certainly we want FLUX to provide all of the mechanisms that enable us to detect a failure of any component in a tool session as well as in the target application.

Proposed FLUX Mechanisms and Interfaces

This section describes some of the RM mechanisms and interfaces that the new approach will need. At a high level, the new scheme will require new mechanisms to launch, control, and destroy the target application processes and tool daemons, monitoring or notification mechanisms to synchronize between them, and query mechanisms to fetch information about these processes. We propose the following API set as a starting point for this co-design process.

Data Types

/ *! MPIR_PROCDESC_EXT is an extension to the vanilla mpir process descriptor type */

```
typedef struct {
        char *host_name; / * Something we can pass to inet_addr */
        char *executable_name; / * The name of the image */
        int pid; / * The pid of the process */
} MPIR_PROCDESC;

typedef struct {
        MPIR_PROCDESC pd;
        int mpirank;
        int cnodeid;
} MPIR_PROCDESC_EXT;
```

enum flux_rc_e defines various return code.

Query, Update and Monitor API

/ *! Interface that creates a lightweight job (LWJ) job context */

```
flux_rc_e
FLUX_update_createLWJCxt (flux_lwj_id_t *lwj)
```

/ *! Interface that converts the pid of the RM starter process to its LWJ id. If the tool wants to work on an LWJ that is already running, this will come in handy. */

```
flux_rc_e
FLUX_query_pid2LWJId (const char *hn, pid_t pid, flux_lwj_id_t *lwj)
```

/ *! Interface that converts the target LWJ id to the LWJ information that includes the information on the RM starter process. flux_lwj_info_t needs to be defined */

```
flux_rc_e
FLUX_query_LWJId2JobInfo (const flux_lwj_id_t *lwj, flux_lwj_info_t *info)
```

/ *! Interface that returns the size of the global MPIR process table */

flux_rc_e

FLUX_query_globalProcTableSize (const flux_lwj_id_t *lwj, size_t *count)

/ *! Interface that returns the global MPIR process table associated with lwj into ptab */

flux_rc_e

FLUX_query_globalProcTable (const flux_lwj_id_t *lwj, MPIR_PROCDESC_EXT* ptab, size_t count);

/ *! Interface that returns the size of the local MPIR process table associated with lwj into count based on the hostname (hn) or where this call is made if hn is NULL */

flux_rc_e

FLUX_query_localProcTableSize (flux_lwj_id_t *lwj, const char *hn, size_t *count)

/ *! Interface that returns the local MPIR process table associated with lwj into pbuf based on the hostname (hn) or where this call is made if hn is NULL. */

flux_rc_e

FLUX_query_localProcTable (const flux_lwj_id_t *lwj, const char *hn, MPIR_PROCDESC_EXT* buf, size_t count);

/ *! Interface that allows the caller to register a status callback function, which is invoked whenever the status of lwj is changed – e.g., just created and stopped for sync; lwj terminiated (we will need to design the state changes of an lwj and notify many state changes to the callback with some mask support). If we don't want the callback style as this might require the API implementation to be muli-process or multi-threaded, we can alternatively design a polling mechanism */

flux_rc_e

FLUX_monitor_registerStatusCb (const flux_lwj_id_t *lwj, int (*cb) (int *status))

Launch and Co-locate API

/ *! Interface that launches target application or tool daemons given an executable and a list of arguments. If the sync flag is true, this interface should spawn the processes and leave them stopped. If the target LWJ is not NULL, this will co-locate the specified executable (e.g., tool daemon path) with the processes of the "target" LWJ. Note that we need to determine the rest of the options/ arguments so that we can overload this interface for launching of both application processes and tool/ middleware daemons. For the co-location, we need to express subset vs. full-set launching etc. */

flux_rc_e

FLUX_launch_spawn (const flux_lwj_id_t *me, int sync, const flux_lwj_id_t *target, int nn, int np)

Control API

/ *! Interface that kills and cleans up all of the processes associated with the target LWJs */

flux_rc_e

FLUX_control_killLWJs (const flux_lwj_id_t target[], int size)