Tutorial for DMTCP Plugins

March, 2013

Contents

1 Introduction

Plugins enable one to modify the behavior of DMTCP. Two of the most common uses of plugins are:

1. to execute an additional action at the time of checkpoint, resume, or restart.

2. to add a wrapper function around a call to a library function (including wrappers around system calls).

Plugins are used for a variety of purposes. The `DMTCP_ROOT/contrib` directory contains packages that users and developers have contributed to be optionally loaded into DMTCP.

Plugin code is expressive, while requiring only a modest number of lines of code. The plugins in the contrib directory vary in size from 400 lines to 3000 lines of code.

Beginning with DMTCP version 2.0, much of DMTCP itself is also now a plugin. In this new design, the core DMTCP code is responsible primarily for copying all of user space memory to a checkpoint image file. The remaining functions of DMTCP are handled by plugins, found in `DMTCP_ROOT/plugin`. Each plugin abstracts the essentials of a different subsystem of the operating system and modifies its behavior to accommodate checkpoint and restart. Some of the subsystems for which plugins have been

written are: virtualization of process and thread ids; files(open/close/dup/fopen/fclose/mmap/pty); events (eventfd/epoll/poll/inotify/signalfd); System V IPC constructs (shmget/semget/msgget); TCP/IP sockets (socket/connect/bind/listen/accept); and timers (timer_create/clock_gettime). (The indicated system calls are examples only and not all-inclusive.)

2 Anatomy of a plugin

There are three primary mechanisms by which a plugin can modify the behavior of either DMTCP or a target application.

Wrapper functions: One declares a wrapper function with the same name as an existing library function (including system calls in the run-time library). The wrapper function can execute some prolog code, pass control to the "real" function, and then execute some epilog code. Several plugins can wrap the same function in a nested manner. One can also omit passing control to the "real" function, in order to shadow that function with an alternate behavior.

Events: It is frequently useful to execute additional code at the time of checkpoint, or resume, or restart. Plugins provide hook functions to be called during these three events and numerous other important events in the life of a process.

Coordinated checkpoint of distributed processes: DMTCP transparently checkpoints distributed computations across many nodes. At the time of checkpoint or restart, it may be necessary to coordinate information among the distributed processes. For example, at restart time, an internal plugin of DMTCP allows the newly re-created processes to "talk" to their peers to discover the new network addresses of their peers. This is important since a distributed computation may be restarted on a different cluster than its original one.

Virtualization of ids: Ids (process id, timer id, sysv ipc id, etc.) are assigned by the kernel, by a peer process, and by remote processes. Upon restart, the external agent may wish to assign a different id than the one assigned prior to checkpoint. Techniques for virtualization of ids are described in Section 3.2.4.

3 Writing Plugins

3.1 Invoking a plugin

Plugins are just dynamic run-time libraries (.so files). They are invoked at the beginning of a DMTCP computation as command-line options:

```
dmtcp_launch --with-plugin PLUGIN1.so:PLUGIN2.so myapp
```

Note that one can invoke multiple plugins as a colon-separated list. One should either specify a full path for each plugin (each .so library), or else to define LD_LIBRARY_PATH to include your own plugin directory.

3.2 The plugin mechanisms

The mechanisms of plugins are most easily described through examples. This tutorial will rely on the examples in `DMTCP_ROOT/test/plugin`. To get a feeling for the plugins, one can "cd" into each of the subdirectories and execute: "`make check`".

3.2.1 Plugin events

For context, please scan the code of `DMTCP_ROOT/plugin/example/example.c`. Executing "`make check`" will demonstrate the intended behavior. Plugin events are handled by including the function `dmtcp_event_hook`. When a DMTCP plugin event occurs, DMTCP will call the function `dmtcp_event_hook` for each plugin. This function is required only if the plugin will handle plugin events. See Appendix A for further details.

```
void dmtcp_event_hook(DmtcpEvent_t event, DmtcpEventData_t *data)
{
  switch (event) {
  case DMTCP_EVENT_WRITE_CKPT:
    printf("\n*** The plugin is being called before checkpointing. ***\n");
    break;
  case DMTCP_EVENT_RESUME:
    printf("*** Resume: the plugin has now been checkpointed. ***\n");
    break;
  case DMTCP_EVENT_RESTART:
    printf("*** The plugin is now being restarted. ***\n");
    break;
  ...
  default:
    break;
  }
  DMTCP_NEXT_EVENT_HOOK(event, data);
}
```

3.2.2 Plugin wrapper functions

In its simplest form, a wrapper function can be written as follows:

```
unsigned int sleep(unsigned int seconds) {
  static unsigned int (*next_fnc)() = NULL; /* Same type signature as sleep */
  struct timeval oldtv, tv;
  gettimeofday(&oldtv, NULL);
  time_t secs = val.tv_sec;
  printf("sleep1: "); print_time(); printf(" ... ");
  unsigned int result = NEXT_FNC(sleep)(seconds);
  gettimeofday(&tv, NULL);
  printf("Time elapsed:  %f\n",
         (1e6*(val.tv_sec-oldval.tv_sec) + 1.0*(val.tv_usec-oldval.tv_usec)) / 1e6);
  print_time(); printf("\n");

  return result;
}
```

In the above example, we could also shadow the standard "sleep" function by our own implementation, if we omit the call to "NEXT_FNC".
To see a related example, try:

```
cd DMTCP_ROOT/test/plugin/sleep1; make check
```

Wrapper functions from distinct plugins can also be nested. To see a nesting of plugin sleep2 around sleep1, do:

```
cd DMTCP_ROOT/test/plugin; make; cd sleep2; make check
```

3.2.3 Plugin coordination among multiple or distributed processes

It is often the case that an external agent will assign a particular initial id to your process, but later assign a different id on restart. Each process must re-discover its peers at restart time, without knowing the pre-checkpoint ids.

DMTCP provides a "Publish/Subscribe" feature to enable communication among peer processes. Two plugin events allow user plugins to discover peers and pass information among peers. The two events are: `DMTCP_EVENT_REGISTER_NAME_SERVICE_DATA` and `DMTCP_EVENT_SEND_QUERIES`. DMTCP guarantees to provide a global barrier between the two events.

An example of how to use the Publish/Subscribe feature is contained in `DMTCP_ROOT/test/plugin/example-db` . The explanation below is best understood in conjunction with reading that example.

A plugin processing `DMTCP_EVENT_REGISTER_NAME_SERVICE_DATA` should invoke:
int dmtcp_send_key_val_pair_to_coordinator(const void *key, size_t key_len, const void *val, size_t val_len).

A plugin processing `DMTCP_EVENT_SEND_QUERIES` should invoke:
int dmtcp_send_query_to_coordinator(const void *key, size_t key_len, void *val, size_t *val_len).

3.2.4 Using plugins to virtualize ids and other names

Often an id or name will change between checkpoint and restart. For example, on restart, the real pid of a process will change from its pid prior to checkpoint. Some DMTCP internal plugins maintain a translation table in order to translate between a virtualized id passed to the user code and a real id maintained inside the kernel. The utility to maintain this translation table can also be used within third-party plugins. For an example of adding virtualization to a plugin, see the plugin in plugin/ipc/timer.

In some less common cases, it can happen that a virtualized id is passed to a library function by the target application. Yet, that same library function may be passed a real id by a second function from within the same library. In these cases, it is the responsibility of the plugin implementor to choose a scheme that allows the first library function to distinguish whether its argument is a virtual id (passed from the target application) or a real id (passed from within the same library).

4 Application-Initiated Checkpoints

Application-initiated checkpoints are even simpler than full-featured plugins. In the simplest form, the following code can be executed both with `dmtcp_launch` and without.:

```
#include <stdio.h>
#include "dmtcpplugin.h"
int main() {
  if (dmtcpCheckpoint() == DMTCP_NOT_PRESENT) {
    printf("dmtcpcheckpoint:  DMTCP not present.  No checkpoint is taken.\n");
  }
  return 0;
}
```

The most useful functions are:

`int dmtcpIsEnabled()` — returns 1 when running with DMTCP; 0 otherwise.

`int dmtcpCheckpoint()` — returns DMTCP_AFTER_CHECKPOINT, DMTCP_AFTER_RESTART, or DMTCP_NOT_PRESENT.

`int dmtcpDelayCheckpointsLock()` — DMTCP will block any checkpoint requests.

`int dmtcpDelayCheckpointsUnlock()` — DMTCP will execute any blocked checkpoint requests, and will permit new checkpoint requests.

The last two functions follow the common pattern of return 0 on success and DMTCP_NOT_PRESENT if DMTCP is not present. See the subdirectories DMTCP_ROOT/test/plugin/applic-initiated-ckpt and DMTCP_ROOT/test/plugin/applic-delayed-ckpt, where one can execute make check for a live demonstration.

A Appendix: Plugin Manual

A.1 Plugin events

A.1.1 dmtcp_event_hook

In order to handle DMTCP plugin events, a plugin must define dmtcp_event_hook.

```
NAME
       dmtcp_event_hook - Handle plugin events for this plugin

SYNOPSIS
       #include "dmtcp/plugin.h"

       void dmtcp_event_hook(DmtcpEvent_t event, DmtcpEventData_t *data)

DESCRIPTION
       When a plugin event occurs, DMTCP will look for the symbol dmtcp_event_hook
       in each plugin library.  If the symbol is found, that function will be called
       for the given plugin library.  DMTCP guarantees only to invoke the first such
       plugin library found in library search order.  Occurences of
       dmtcp_event_hook in later plugin libraries will be called only if
       each previous function had invoked DMTCP_NEXT_EVENT_HOOK.
       The argument, <event>, will be bound to the event being declared by
       DMTCP.  The argument, <data>, is required only for certain events.
       See the following section, ``Plugin Events'' for a list of all
       events.

SEE ALSO
       DMTCP_NEXT_EVENT_HOOK
```

A.1.2 DMTCP_NEXT_EVENT_HOOK

A typical definition of dmtcp_event_hook will invoke DMTCP_NEXT_EVENT_HOOK.

```
NAME
       DMTCP_NEXT_EVENT_HOOK - call dmtcp_event_hook in next plugin library

SYNOPSIS
       #include "dmtcp/plugin.h"

       void DMTCP_NEXT_EVENT_HOOK(event, data)

DESCRIPTION
       This function must be invoked from within a plugin function library
       called dmtcp_event_hook.  The arguments <event> and <data> should
       normally be the same arguments passed to dmtcp_event_hook.
```

```
       DMTCP_NEXT_EVENT_HOOK may be called zero or one times.  If invoked zero
       times, no further plugin libraries will be called to handle events.
       The behavior is undefined  if DMTCP_NEXT_EVENT_HOOK is invoked more than
       once.  The typical usage of this function is to create a wrapper around
       the handling of the same event by later plugins.

SEE ALSO
       dmtcp_event_hook
```

There are examples of compiling a plugin in the examples in `DMTCP_ROOT/test/plugin`.

A.1.3 Event Names

The rest of this section defines plugin events. The complete list of plugin events is always contained in `DMTCP_ROOT/dmtcp/include/dmtcp/plugin.h` .

DMTCP guarantees to call the dmtcp_event_hook function of the plugin when the specified event occurs.

Plugins that pass significant data through the data parameter are marked with an asterisk: *. Most plugin events do not pass data through the data parameter.

Note that the events
RESTART / RESUME / REFILL / REGISTER_NAME_SERVICE_DATA / SEND_QUERIES
should all be processed after the call to DMTCP_NEXT_EVENT_HOOK() in order to guarantee that the internal DMTCP plugins have first restored full functionality.

Checkpoint-Restart

DMTCP_EVENT_WRITE_CKPT — Invoked at final barrier before writing checkpoint

DMTCP_EVENT_RESTART — Invoked at first barrier during restart of new process

DMTCP_EVENT_RESUME — Invoked at first barrier during resume following checkpoint

Coordination of Multiple or Distributed Processes during Restart (see Appendix A.2. Publish/Subscribe)

DMTCP_EVENT_REGISTER_NAME_SERVICE_DATA* restart/resume

DMTCP_EVENT_SEND_QUERIES* restart/resume

WARNING: EXPERTS ONLY FOR REMAINING EVENTS

Init/Fork/Exec/Exit

DMTCP_EVENT_INIT — Invoked before main (in both the original program and any new program called via exec)

DMTCP_EVENT_EXIT — Invoked on call to exit/_exit/_Exit **return from main?**;

DMTCP_EVENT_PRE_EXEC — Invoked prior to call to exec

DMTCP_EVENT_POST_EXEC — Invoked before DMTCP_EVENT_INIT in new program

DMTCP_EVENT_ATFORK_PREPARE — Invoked before fork (see POSIX pthread_atfork)

DMTCP_EVENT_ATFORK_PARENT — Invoked after fork by parent (see POSIX pthread_atfork)

DMTCP_EVENT_ATFORK_CHILD — Invoked after fork by child (see POSIX pthread_atfork)

Barriers (finer-grainded control during checkpoint-restart)

DMTCP_EVENT_WAIT_FOR_SUSPEND_MSG — Invoked at barrier during coordinated checkpoint

DMTCP_EVENT_SUSPENDED — Invoked at barrier during coordinated checkpoint

DMTCP_EVENT_LEADER_ELECTION — Invoked at barrier during coordinated checkpoint

DMTCP_EVENT_DRAIN — Invoked at barrier during coordinated checkpoint

DMTCP_EVENT_REFILL — Invoked at first barrier during resume/restart of new process

Threads

DMTCP_EVENT_THREADS_SUSPEND — Invoked within checkpoint thread when all user threads have been suspended

DMTCP_EVENT_THREADS_RESUME — Invoked within checkpoint thread before any user threads are resumed.
For debugging, consider calling the following code for this event: `static int x = 1; while(x);`

DMTCP_EVENT_PRE_SUSPEND_USER_THREAD — Each user thread invokes this prior to being suspended for a checkpoint

DMTCP_EVENT_RESUME_USER_THREAD — Each user thread invokes this immediately after a resume or restart (`isRestart()` available to plugin)

DMTCP_EVENT_THREAD_START — Invoked before start function given by clone

DMTCP_EVENT_THREAD_CREATED — Invoked within parent thread when clone call returns (like parent for fork)

DMTCP_EVENT_PTHREAD_START — Invoked before start function given by pthread_created

DMTCP_EVENT_PTHREAD_EXIT — Invoked before call to pthread_exit

DMTCP_EVENT_PTHREAD_RETURN — Invoked in child thread when thread start function of pthread_create returns

A.2 Publish/Subscribe

Section 3.2.3 provides an explanation of the Publish/Subscribe feature for coordination among peer processes at resume- or restart-time. An example of how to use the Publish/Subscribe feature is contained in `DMTCP_ROOT/test/plugin/example-db` .

The primary events and functions used in this feature are:

`DMTCP_EVENT_REGISTER_NAME_SERVICE_DATA`

int dmtcp_send_key_val_pair_to_coordinator(const void *key, size_t key_len, const void *val, size_t val_len)

`DMTCP_EVENT_SEND_QUERIES`

int dmtcp_send_query_to_coordinator(const void *key, size_t key_len, void *val, size_t *val_len)

A.3 Wrapper functions

For a description of including wrapper functions in a plugin, see Section 3.2.2.

A.4 Miscellaneous utility functions

Numerous DMTCP utility functions are provided that can be called from within dmtcp_event_hook(). For a complete list, see `DMTCP_ROOT/dmtcp/include/dmtcp/plugin.h` . The utility functions are still under active development, and may change in small ways. Some of the more commonly used utility functions follow. Functions that return "char *" will not allocate memory, but instead will return a pointer to a canonical string, which should not be changed.

```
void dmtcp_get_local_ip_addr(struct in_addr *in);
const char* dmtcp_get_tmpdir(); /* given by --tmpdir, or DMTCP_TMPDIR, or TMPDIR */
const char* dmtcp_get_ckpt_dir();
   /* given by --ckptdir, or DMTCP_CHECKPOINT_DIR, or curr dir at ckpt time */
const char* dmtcp_get_ckpt_files_subdir();
int  dmtcp_get_ckpt_signal(); /* given by --mtcp-checkpoint-signal */
const char* dmtcp_get_uniquepid_str();
const char* dmtcp_get_computation_id_str();
uint64_t dmtcp_get_coordinator_timestamp();
uint32_t dmtcp_get_generation(); /* number of ckpt/restart sequences encountered */
const char* dmtcp_get_executable_path();
int dmtcp_get_restart_env(char *name, char *value, int maxvaluelen);
  /* For 'name' in environment, copy its value into 'value' param, but with
   * at most length 'maxvaluelen'.
   * Return 0 for success, and return code for various errors */
   */
```