

Tutorial for DMTCP Plugins

March, 2013

Contents

1	Introduction	1
2	Anatomy of a plugin	2
3	Writing Plugins	2
3.1	Invoking a plugin	2
3.2	The plugin mechanisms	2
3.2.1	Plugin events	3
3.2.2	Plugin wrapper functions	3
3.2.3	Plugin coordination among multiple or distributed processes	4
3.2.4	Using plugins to virtualize ids	4
4	Writing Plugins that Virtualize IDs and Other Names	5
5	Caveats	5
A	Appendix: Plugin Manual	5
A.1	Plugin events	5
A.1.1	dmtcp_process_event	5
A.1.2	NEXT_DMTCP_PROCESS_EVENT	6
A.1.3	Event Names	6
A.2	Publish/Subscribe	8
A.3	Wrapper functions	8
A.4	Miscellaneous utility functions	8

1 Introduction

This is a reminder that there are more boldface below here.

Also, see the comments in the LaTeX file for smaller issues for the 2.0 release.

Also, must then create plugin-tutorial.pdf, and add to svn

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 have contributed to be used as part of DMTCP. Several of these packages are based on DMTCP plugins. These include:

1. a plugin to adapt DMTCP to automatically checkpoint and restart in conjunction with the TORQUE batch queue system.
2. **ADD WHEN COMPLETE:** a plugin to adapt DMTCP to automatically checkpoint and restart in conjunction with the Condor system for high throughput computing.
3. **ADD WHEN COMPLETE:** a plugin that allows DMTCP to checkpoint QEMU/KVM.
4. **ADD WHEN COMPLETE:** record-replay plugin (later)
5. **ADD WHEN COMPLETE:** a package to adapt I/O to different environments, when a DMTCP checkpoint image is restarted with a modified filesystem, on a new host, etc.
6. **ADD WHEN COMPLETE:** a plugin that allows a Python program to call a Python function to checkpoint itself.
7. **ADD WHEN COMPLETE:** a plugin that enables one to checkpoint over the network to a remote file.

Plugin code is expressive, while requiring only a modest number of lines of code. The plugins in the contrib directory vary in size from **FILL IN** to 1500 lines of code to implement a plugin for the Torque batch queue.

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, **FILL IN**) 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 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_checkpoint --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 ins 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_process_event`. When a DMTCP plugin event occurs, DMTCP will call the function `dmtcp_process_event` for each plugin. This function is required only if the plugin will handle plugin events. See Appendix A for further details.

```
void dmtcp_process_event(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("*** The plugin has now been checkpointed. ***\n");
        break;
    case DMTCP_EVENT_THREADS_RESUME:
        if (data->resumeInfo.isRestart) {
            printf("The plugin is now resuming or restarting from checkpointing.\n");
        } else {
            printf("The process is now resuming after checkpoint.\n");
        }
        break;
    ...
    default:
        break;
    }
    NEXT_DMTCP_PROCESS_EVENT(event, data);
}
```

ACTUALLY, I THINK MY EXAMPLE plugin was calling the wrong event name. I’ll look at this more carefully later. – Gene

I’LL FINISH WRITING LATER.

Plugin events:

*** The model for events: When a DMTCP event occurs, DMTCP calls ROUTINE in each plugin, in the order that the plugins were loaded, offering each plugin a chance to handle the event. If ROUTINE is not defined in a plugin, DMTCP skips calling that plugin. When ROUTINE is called, it is given the unique event id, and a switch statement can decide whether to take any special action. If no action is taken, ROUTINE returns XXX, and the next plugin is offered a chance to handle the event. If a plugin does handle the event, a typical user code fragment will: A. optionally carry out any pre-processing steps B. optionally ask DMTCP to invoke the next event handler C. optionally carry out any post-processing steps

If all three steps are invoked, this effectively creates a wrapper function around any later plugins that handle the same event. If step B is omitted, no further plugins will be offered the opportunity to handle the event.

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
```

Plugin wrappers:

Use sleep1/sleep2 for the example.

(see paper for other; mention tech. report ??, if ONWARD allows i.t)

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 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 the directory, `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

In this section, we consider a further complication. If the user code or run-time library has cached that initial id, then this presents a problem on restart. Rather than create an independent mechanism, this section shows how to handle this problem using existing tools.

Kapil, you said that Torque had a good example of this. What is it?

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. If the user code or run-time library has cached that initial id, then this presents a problem on restart. Each process must re-discover its peers at restart time, without knowing the pre-checkpoint ids.

The solution is to virtualize the id. This mechanism is used internally in DMTCP to virtualize the many ids provided by the kernel, by network host ids, and so on. This section describes how your own plugin can take advantage of the same mechanism.

A good example of the use of virtualization occurs in the Torque plugin at `DMTCP_ROOT/contrib/torque`. **IS THIS TRUE? WHAT IS BEING VIRTUALIZED?**

4 Writing Plugins that Virtualize IDs and Other Names

Most writers of plugins can skip this section. **Virtualization of names is required if**

5 Caveats

CAVEATS: Does your plugin break normal DMTCP? to test this, modify DMTCP, and copy your plugin into `DMTCP_ROOT/lib`, and then run 'make check' for DMTCP as usual.

SHARED MEMORY REGIONS: If two or more processes share a memory region, then the plugin writer must be clear on whether DMTCP or the plugin has responsibility for restoring the shared memory region. Currently, **EXPLAIN**

Virtualizing long-lived objects: **HOWTO**

INTERACTION OF MULTIPLE PLUGINS: For simple plugins, this issue can be ignored. But if your plugin has talbes with long-lived data, other wrappers may create additional instantiations. It is reasonable for them to do this for temporary data structures at the time of checkpoint or at the time of restart. But normally, such an object, created when the checkpoint event occurs, should be destroyed before creating the actual checkpoint image. Similarly, at restart time, if new instances are created, they should be destroyed before returning control to the user threads. It is polite for a plugin to check the above restrictions. If it is violated, the plugin should print a warning about this. This will help others, who accidentally create long-lived objects at checkpoint- or restart-time, without intending to. If the other plugin intends this unusual behavior, one can add a whitelist feature for other plugins to declare such intentions. This small effort will provide a well-defined protocol that limits the interaction between distinct plugins. Your effort helps others to debug their plugins, and a similar effort on their part will help you to debug your own plugin.

Putting a `printf` inside a plugin at the time of checkpoint is dangerous. This is because `printf` indirectly invokes a lock to prevent two threads from printing simultaneously. This causes the checkpoint thread to call a `printf`. **See README in test/plugin.**

At checkpoint time, the DMTCP user thread will stop on that same lock. This causes the two threads to deadlock.

This use of conflicting locks is a bug in DMTCP as of DMTCP-2.0. It will be fixed in the future version of DMTCP.

A Appendix: Plugin Manual

A.1 Plugin events

A.1.1 `dmtcp_process_event`

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

NAME

`dmtcp_process_event` - Handle plugin events for this plugin

SYNOPSIS

```
#include "dmtcp/plugin.h"

void dmtcp_process_event(DmtcpEvent_t event, DmtcpEventData_t *data)
```

DESCRIPTION

When a plugin event occurs, DMTCP will look for the symbol `dmtcp_process_event` 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_process_event` in later plugin libraries will be called only if each previous function had invoked `NEXT_DMTCP_PROCESS_EVENT`. 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

`NEXT_DMTCP_PROCESS_EVENT`

A.1.2 `NEXT_DMTCP_PROCESS_EVENT`

A typical definition of `dmtcp_process_event` will invoke `NEXT_DMTCP_PROCESS_EVENT`.

NAME

`NEXT_DMTCP_PROCESS_EVENT` - call `dmtcp_process_event` in next plugin library

SYNOPSIS

```
#include "dmtcp/plugin.h"

void NEXT_DMTCP_PROCESS_EVENT(event, data)
```

DESCRIPTION

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

`NEXT_DMTCP_PROCESS_EVENT` 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 `NEXT_DMTCP_PROCESS_EVENT` 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_process_event`

If user-installed package, compile with `-IDMTCP_ROOT/dmtcp/include`.

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_process_event` function of the plugin when the specified event occurs. If an event is thread-specific (**GIVE EXAMPLES**), DMTCP guarantees to call `dmtcp_process_event` within the same thread.

DO I HAVE ALL THE THREAD-SPECIFIC EVENTS?

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. Currently, the plugins that use the data parameter **use it to test if this is restart or resume??. In this case, why don't we have a single function that every plugin can call to test if this is during a restart or resume?**

Note that the events

`RESTART` / `RESUME` / `REFILL` / `REGISTER_NAME_SERVICE_DATA` / `SEND_QUERIES`

should all be processed after the call to `NEXT_DMTCP_PROCESS_EVENT()` in order to guarantee that the internal DMTCP plugins have 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-grained 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);`

should we have separate DMTCP_EVENT_THREADS_RESUME? — I vote yes.

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)

should we have separate DMTCP_EVENT_RESTART_USER_THREAD? — I vote yes.

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_create

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 `refsec:publishSubscribe` 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 the directory, `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

fill in

A.4 Miscellaneous utility functions

Numerous DMTCP utility functions are provided that can be called from within `dmtcp_process_event()`. 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.