Architecture of DMTCP

Gene Cooperman

March, 2015

This is intended as a gentle introduction to the architecture of DMTCP. Shorter descriptions are possible. For a low-level description with references to code in the implementation, see `doc/restart_algorithm.txt` . For the general use of plugins, see `doc/plugin-tutorial.pdf` . DMTCP uses plugins internally from `DMTCP_ROOT/src/plugin`. DMTCP also offers optional plugins (not enabled by default) to end users. Thes come in two flavors: `DMTCP_ROOT/plugin` (fully supported) and `DMTCP_ROOT/contrib` (newer plugins, possibly contributed by third parties).

1. Usage of DTMCP:

```
1: dmtcp_launch a.out
2: dmtcp_command --checkpoint
     ↪ ckpt_a.out.*.dmtcp
3: dmtcp_restart ckpt_a.out.*.dmtcp
```

DMTCP offers a `--help` command line option along with additional options both for `configure` and for the individual DMTCP commands. To use DMTCP, just prefix your command line with `dmtcp_launch`.

2. dmtcp_launch a.out

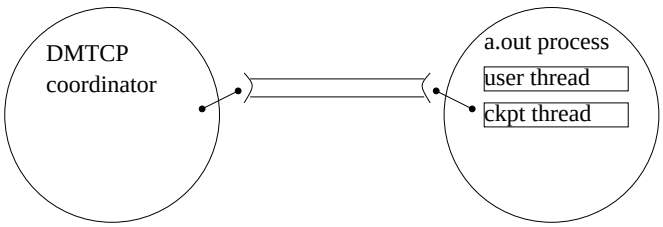
The command `dmtcp_launch a.out` is roughly equivalent to:

1: `dmtcp_coordinator --background` (if not already running)
2: `LD_PRELOAD=libdmtcp.so a.out`
(`LD_PRELOAD` causes `libdmtcp.so` to be loaded first, and initialized before the call to `main` in `a.out`.)

The `dmtcp_launch` command will cause a coordinator process to be launched on the local host with the default DMTCP port (if one is not already available).

DMTCP includes a special coordinator process so that DMTCP can also checkpoint distributed computations across many computers. The user can issue a command to the coordinator, which will then relay the command to each of the user processes of the distributed computation.

Note that a *DMTCP computation* is defined to be a coordinator process and the set of user processes connected to that coordinator. Therefore, one can have more than one DMTCP computation on a single computer, each computation having its own unique coordinator. Each coordinator is defined by its unique network address, "(hostname, port)", which defaults to "(localhost, 7779)".



The coordinator is *stateless*. If the computation is ever killed, one needs only to start an entirely new coordinator, and then restart using the latest checkpoint images for each user process.

LD_PRELOAD is a special environment variable known to the loader. When the loader tries to load a binary (a.out, in this case), the loader will first check if LD_PRELOAD is set (see 'man ld.so'). If it is set, the loader will load the binary (a.out) and then the preload library (libdmtcp.so) before running the 'main()' routine in a.out. (In fact, dmtcp_launch may also preload some plugin libraries, such as pidvirt.so (starting with DMTCP-2.0) and set some environment variables such as DMTCP_DLSYM_OFFSET.)

When the library libdmtcp.so is loaded, any top-level variables are initialized, before calling the user main(). If the top-level variable is a C++ object, then the C++ constructor is called before the user main. In DMTCP, the first code to execute is the code below, inside libdmtcp.so:

`dmtcp::DmtcpWorker dmtcp::DmtcpWorker::theInstance;`

This initializes the global variable, `theInstance` via a call to `new dmtcp::DmtcpWorker::DmtcpWorker()`. (Here, `dmtcp` is a C++ namespace, and the first `DmtcpWorker` is the class name, while the second `DmtcpWorker` is the constructor function. If DMTCP were not using C++, then it would use GNU gcc attributes to directly run a constructor function.)

Note that DMTCP is organized in at least two layers. The lowest layer is MTCP (mtcp subdirectory), which handles single-process checkpointing. The higher layer is again called DMTCP (dmtcp subdirectory), and delegates to MTCP when it needs to checkpoint one process. MTCP does not require a separate DMTCP coordinator.

So, at startup, we see:

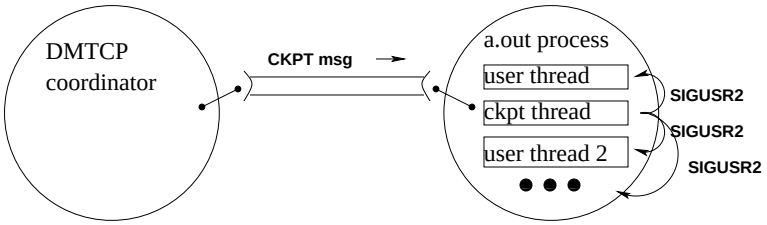
1: a.out process:
2: primary user thread:
3: new DmtcpWorker():
4: Create a socket to the coordinator
5: Register the signal handler that will be used for checkpoints.
The signal handler is src/threadlist.cpp:stopthisthread(sig)
The default signal is src/siginfo.cpp:STOPSIGNAL (default: SIGUSR2)
The signal handler for STOPSIGNAL is registered by `SigInfo::setupCkptSigHandler(&stopthisthread)` in src/threadlist.cpp
6: Create the checkpoint thread:
Call `pthread_create (&checkpointhreadid, NULL, checkpointhread, NULL)` in src/threadlist.cpp
7: Wait until the checkpoint thread has initialized.

8: checkpoint thread:
9: checkpointhread(dummy): [from src/threadlist.cpp]
10: Register a.out process with coordinator
11: Tell user thread we're done
12: Call select() on socket to coordinator

13: primary user thread:
14: new DmtcpWorker(): [continued from above invocation]
15: Checkpoint thread is now initialized. Return.
16: main() [User code now executes.]

PRINCIPLE: At any given moment either the user threads are active and the checkpoint thread is blocked on `select()`, or the checkpoint thread is active and the user threads are blocked inside the signal handler, stopthisthread().

3. Execute a Checkpoint:



1: checkpoint thread:
2: return from `DmtcpWorker::waitForCoordinatorMsg()` in src/dmtcpworker.cpp
3: receive CKPT message
4: send STOPSIGNAL (SIGUSR2) to each user thread
See: `THREAD_TGKILL(motherpid, thread->tid, SigInfo::ckptSignal()` in the section labelled `case ST_RUNNING:` in src/threadlist.cpp
See `Thread_UpdateState(curThread, ST_SUSPINPROG, ST_SIGNALED)` and `Thread_UpdateState(curThread, ST_SUSPENDED, ST_SUSPINPROG)` in src/threadlist.cpp
5: Recall that dmtcpWorker created a signal handler, `stopthisthread()`, for STOPSIGNAL (default: SIGUSR2)
6: Each user thread in that signal handler will execute `sem_wait(&semWaitForCkptThreadSignal)` and block.
7: The checkpoint thread does the checkpoint.

8: Release each thread from its mutex by calling sem_post(&semWaitForCkptThreadSignal) inside `resumeThreads()`.
9: Each thread updates its state from `ST_SUSPENDED` to `ST_RUNNING`:
See `Thread_UpdateState(curThread, ST_RUNNING, ST_SUSPENDED)` in src/threadlist.cpp
10: Call `DmtcpWorker::waitForCoordinatorMsg()` using the socket to the coordinator and again wait for messages from the coordinator.

4. Checkpoint strategy (overview)

1: Quiesce all user threads (using STOPSIGNAL (SIGUSR2), as above)
2: Drain sockets
(a) Sender sends a "cookie"
(b) Receiver receives until it sees the "cookie"
Note: Usually all sockets are "internal" — within the current computation. Heuristics are provided for the case of "external" sockets.
3: Interrogate kernel state (open file descriptors, file descriptor offset, etc.)
4: Save register values using setcontext (similar to setjmp) in mtcp/mtcp.c
5: Copy all user-space memory to checkpoint image To find all user-space memory, one can execute:
`cat /proc/self/maps`
6: Unblock all use threads

5. Restart strategy (overview)

This section will be revised in the future.

The key ideas are that `dmtcp_restart` exec's into `mtcp_restart`. The program `mtcp_restart` (source code in src/mtcp/) is created specially so that the process will consist of a single relocatable memory segment. The program relocates itself into a "hole" in memory not occupied by the checkpoint image. It then calls `src/mtcp/mtcp_restart.c:restorememoryareas()`. This maps the memory areas of the checkpoint image into their original memory addresses. ("Bits are bits.")

While still in `restorememoryareas()`, it calls restore_libc() to update the information in memory about the different threads. Finally, it calls a function pointer, `restore_info.post_restart`, which in fact is bound to `ThreadList::postRestart()` in src/threadlist.cpp.

It is then the job of `ThreadList::postRestart()` to create the remaining threads and set their state correctly. Each remaining thread finds itself inside the signal handler again (After all, "bits are bits."), and the checkpoint thread then releases each thread, as described in the previous section.

6. Principle: DMTCP is contagious

New Linux "tasks" are created in one of three ways:

1. new thread: created by pthread_create() or clone()
2. new process: created by fork()
3. new remote process: typically created by the 'system' system call:
   `system("ssh REMOTE_HOST a.out");`

DMTCP makes sure to load itself using wrapper functions.

7. Principle: One DMTCP Coordinator for each DMTCP Computation

One may wish to run multiple DMTCP-based computations on a single host. This is easily done by using the `--host` and `--port` flags of `dmtcp_launch` or of `dmtcp_coordinator`. If not specified, the default values are localhost and port 7779. By using `dmtcp_command`, one can communicate a checkpoint or other request to one's preferred coordinator (again by specifying host and port, if one is not using the default coordinator).

In the simplest case, the user invokes only `dmtcp_launch`, with no flags. The `dmtcp_launch` command then looks for an existing coordinator at localhost:7779. If none is found, `dmtcp_launch` invokes `dmtcp_coordinator` with those default values, localhost:7779.

Thus, an occasional issue occurs when two users on the same host each invoke `dmtcp_launch` with default parameters. They cannot both use the same coordinator. Similarly, a single user may want to launch two independent computations, and independently checkpoint them. If the user invokes `dmtcp_launch` (default parameters) for both computation, then there will only be one coordinator. So, in the view of DMTCP, there will only be a single computation, and a checkpoint command will checkpoint the processes of both computations.

8. Plugins and other End-User Customizations

DMTCP offers a rich set of features for customizing the behavior of DMTCP. In this short overview, we will point to examples that can easily be modified by an end-user. See `doc/plugin-tutorial.pdf` for a more extensive description of plugins.

DMTCP Plugins. *DMTCP plugins* are the most general way to customize DMTCP. Examples are in `DMTCP_ROOT/test/plugin/`. A dynamic library (`*.so`) file is created to modify the behavior of DMTCP. The library can write additional wrapper functions (and even define wrappers around previous wrapper functions). The library can also register to be notified of *DMTCP events*. In this case, DMTCP will call any plugin functions registered for each event. Examples of important events are e.g. prior to checkpoint, after resume, and after restart from a checkpoint image. As of this writing, there is no central list of all DMTCP events, and names of events are still subject to change.

Plugin libraries are preloaded after libdmtcp.so. As with all preloaded libraries, they can initialize themselves before the user's "main" function, and at run-time, plugin wrapper functions will be found in standard Linux library search order prior to ordinary library functions in libc.so and elsewhere.

For example, the sleep2 plugin example uses two plugins. After building the plugins, it might be called as follows:

```
dmtcp_launch --with-plugin \
  PATH/sleep1/dmtcp_sleep1hijack.so:PATH/sleep2/dmtcp_sleep2hijack.so a.out
```

where `PATH` is `DMTCP_ROOT/test/plugin`

In a more involved example, whenever `./configure --enable-ptrace-support` is specified, then DMTCP will use the plugin `DMTCP_ROOT/plugin/ptrace/ptracehijack.so`. A new plugin to provide PID/TID virtualization is currently planned. As with support for ptrace, a more modular structure for PID/TID virtualization will be easier to maintain.

Dmtcpaware. The dmtcpaware interface is an obsolete customization mechanism that is no longer supported in DMTCP-2.2 and beyond.

MTCP. In DMTCP-2.1 and earlier, the MTCP component of DMTCP could be compiled to run standalone, with opportunities for hook functions using weak symbols. MTCP has now been almost entirely re-written, and is now tightly integrated with DMTCP. *For thos who wished to use the prior MTCP architecture (just the checkpoint thread, but no separate coordinator), a plugin with those features is planned for the future.*

9. Implementation of Plugins

The implementation techniques of wrapper functions and pid/tid virtualization we part of the DMTCP implementation not too long after the initial offering of DMTCP. More recently, this functionality was wrapped into a high level abstraction, plugins. This section emphasizes the implementation of these features. For information on using plugins, and writing your own, see `doc/plugin-tutorial.pdf` .

a. Wrapper functions

Wrapper functions are functions around functions. DMTCP creates functions around libc.so functions. Wrapper functions are typically created using dlopen/dlsym. For example, to define a wrapper around libc:fork(), one defines a function fork() in libdmtcp.so (see `extern "C" pid_t fork()` in execwrappers.cpp).

Continuing this example, if the user code calls fork(), then we see the following progression.

a.out:call to fork() $\longrightarrow$ libdmtcp.so:fork() $\longrightarrow$ libc.so:fork()

The symbol libdmtcp.so:fork appears before libc.so:fork in the library search order because libdmtcp.so was loaded before libc.so (due to LD_PRELOAD).

Next, the wrapper around `pthread_create` remembers the thread id of the new thread created. The wrapper around `fork` ensures that the environment variable LD_PRELOAD is still set to libdmtcp.so. If LD_PRELOAD does not currently include libdmtcp.so, then it is reset to include libdmtcp.so before the call to fork(), and then LD_PRELOAD is reset to the original user value after fork().

The wrapper around `system` (in the case of creating remote processes) is perhaps the most interesting one. See `'man system'` for a description of the call `system`. It looks at an argument, for example `"ssh REMOTE_HOST a.out"`, and then edits the argument to `"ssh REMOTE_HOST dmtcp_launch a.out"` before calling `system`. Of course, this works only if dmtcp_launch is in the user's path on `REMOTE_HOST`. This is the responsibility of the user or the system administrator.

b. PID/TID Virtualization

Any system calls that refer to a process id (pid) or thread id (tid) requires a wrapper. DMTCP then translates between a virtual pid/tid an the real pid/tid. The user code always sees the virtual pid/tid, while the kernel always sees the real pid/tid.

c. Publish/Subscribe

Plugins also offer a publish/subscribe service for situations where a DMTCP computation contains more than one process, and the user processes must coordinate with each other. Details are in `doc/plugin-tutorial.pdf` .