Architecture of DMTCP

Gene Cooperman

January, 2012

This is intended as a gentle introduction to the architecture of DMTCP. Shorter descriptions are possible.

1. Usage:

1: `dmtcp_checkpoint 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_checkpoint`.

2. dmtcp_checkpoint a.out

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

1: `dmtcp_coordinator --background` (if not already running)
2: `LD_PRELOAD=dmtcphijack.so a.out`

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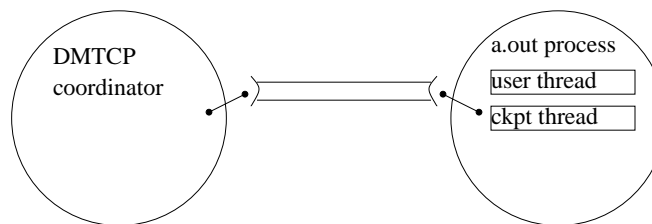
The coordinator is *stateless*. If the computation is ever killed, one needs only to start an entirely new coordinator, and then restart using the latest checkpoint images for each user process.

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 (dmtcphijack.so) before running the 'main()' routine in a.out.

When the library dmtcphijack.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 dmtcphijack.so:

`dmtcp::DmtcpWorker dmtcp::DmtcpWorker::theInstance ( true );`

This initializes the global variable, `theInstance` via a call to `new dmtcp::DmtcpWorker::DmtcpWorker(true)`. (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 a direct way to 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(true):
 4:         Create a socket to the coordinator
 5:         Register the signal handler that will be used for checkpoints.
              The signal handler is mtcp/mtcp.c:stopthisthread()
              The default signal is STOPSIGNAL (default: SIGUSR2)
              The signal handler is registered by mtcp_sigaction(STOPSIGNAL, &act, &old_act) in mtcp/mtcp.c
 6:       Create the checkpoint thread         Call pthread_create (&checkpointthreadid, NULL, checkpointthread,
         NULL) in mtcp/mtcp.c
 7:         Wait until the checkpoint thread has initialized.

 8:     checkpoint thread:
 9:       checkpointhread(dummy): [from mtcp/mtcp.c]
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(true): [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:
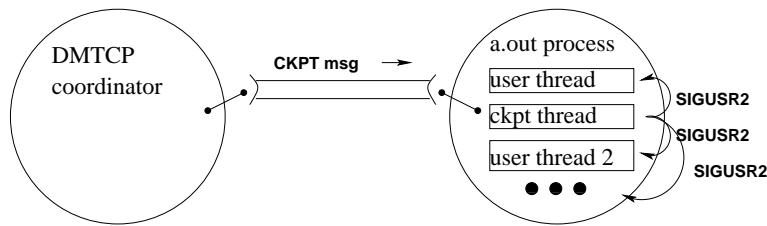
1: checkpoint thread:
2: return from select()
3: receive CKPT message
4: send STOPSIGNAL (SIGUSR2) to each user thread
5: See: retval = mtcp_sys_kernel_tgkill(motherpid, thread->tid, STOPSIGNAL); in mtcp/mtcp.c See `mtcp_state_set` `(&(thread -> state), ST_SUSPINPROG, ST_SIGENABLED)` and `mtcp_state_set (&(thread -> state),` `ST_SUSPENDED, ST_SUSPINPROG)` in mtcp/mtcp.c
6: `mtcp_state_set(MtcpState * state, int value, int oldval)` is defined in mtcp/mtcp_state.c, and does its work through a Linux per-thread futex (similar to a mutex).
7: [Recall that dmtcpWorker created a signal handler, STOPSIGNAL (default: SIGUSR2)
8: Each user thread in the signal handler for STOPSIGNAL will block on the per-thread futex.
9: The checkpoint thread does the checkpoint.

10: Release each thread from its per-thread futex. See `mtcp_state_set (&(thread -> state), ST_RUNENABLED,` `ST_SUSPENDED)` in mtcp/mtcp.c
11: Call select() on 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. 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.

6. 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 dmtcphijack.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() $\longrightarrow$ dmtcphijack.so:fork() $\longrightarrow$ libc.so:fork()

The symbol dmtcphijack.so:fork appears before libc.so:fork in the library search order because dmtcphijack.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 dmtcphijack.so. If LD_PRELOAD does not currently include dmtcphijack.so, then it is reset to include dmtcphijack.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_checkpoint a.out"` before calling `system`. Of course, this works only if dmtcp_checkpoint is in the user's path on REMOTE_HOST. This is the responsibility of the user or the system administrator.

7. 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.

8. Modules, Dmtcpaware, 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.

DMTCP Modules. *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.

Module libraries are preloaded after dmtcphijack.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_checkpoint --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. A second customization facility is provided by *dmtcpaware*. This allows an end user to register the three most important events discussed earlier (pre-checkpoint, post-resume, post-restart). Dmtcpaware is implemented through the use of *weak symbols*. When weak and ordinary symbols are both defined, a loader will load an ordinary symbol in preference to a weak symbol of the same name. If an ordinary symbol is not defined, then the weak symbol will be loaded. Hence, a weak symbol is a kind of *default definition* of a symbol.

For further information on the implementation, see the source files, dmtcpawareapi.cpp, dmtcpaware.c, and dmtcpaware.h in DMTCP_ROOT/dmtcp/src.

For examples of dmtcpaware programs, see `DMTCP_ROOT/test/dmtcpaware1.c` and similarly for dmtcpaware2.c and dmtcpaware3.c. These programs are compiled by linking with `DMTCP_ROOT/dmtcpaware/libdmtcpaware.a`. They are invoked in the usual way. For example, `dmtcp_checkpoint dmtcpaware1`.

MTCP. For standalone MTCP programs, there is also a mechanism based on weak symbols. For an example of its use, see: `DMTCP_ROOT/mtcp/testmtcp.c`