**Notes for class on CS 7680 class on Tues 25th Jan.**

Logistics:

* First writing for project due on Friday Feb, 3
* Guest Lecture by Ignazio Clerente [one of the core developers of Open Nebule] on Open Nebula on Jan 31st. [more details soon]

--

1) Open Nebula is a cloud manager implementation similar to open stack which was famous in Europe, during the time when Open stack was making strides in the US.

2) A good paper generally has an interesting story, a good story is required to get the attention of the refree, it also makes the paper stand out.

A good store consists of three parts:

1. What is the problem
2. Why is it important and
3. Why is this work new

Usually answered in the order presented here and fit in one page or less. If one is doing a survey paper the answer to the question to “Why is this work new” should be formed keeping the state of the art at the time when the paper was published in mind. This is sometimes also called a vision statement for your paper.

Here are the excerpts from the discussion of the various course projects from the class:

1. Mass open Cloud:
* Vendor Locking problem in multi landlord cloud
* Baremetal imaging service
* Rapid provisioning of server class machines.
1. Checkpointing on an Infiniband network.
* RDMA networks – Remote direct memory access networks
* Chekpoint restart on RDMA networks has not been done before.
* It Is complicated because lots if protocols are very closely related to the Hardware.
1. Checkpoint/Restart on Microsoft Azure
* Migrating DMTCP to the WSL subsystem of Windows and then
* Interesting because this platform is unexplored and that would bring check-pointing to a completely new platform.
1. Staggered Coordinated Checkpointing:
* Coordinated: all processes start chekpointing at the same time as already done in DMTCP using a coordinator
* Staggered: don’t stop the process that is being chekpointed divide in into subprocesses and overlap the checkpointing and compute.
* Prevents the overwhelming of the backend.

 Process Virtualization in DMTCP:

Is achieved by interposing the DMTCP library on the libgcc library using LD\_PRELOAD environment variable and then using dlsym to call the right system call from within the libgcc library.

However, it is not possible to checkpoint the entire world, and hence one needs to define an interface to the external world which can be used while chekpointing applications which interact with the external world like network sockets, memory accesses etc. To achieve this, in the more recent releases of DMTCP, the concept of plugins has been introduced.

A plugin is just an external library which interposes on some system call and sits above the libdmtcp.so library. Here is a simple plugin interposing on the sleep system calls.

#include <stdio.h>

#include <sys/time.h>

#include "dmtcp.h"

void print\_time()

{

 struct timeval val;

 gettimeofday(&val, NULL);

 printf("%ld %ld", (long)val.tv\_sec, (long)val.tv\_usec);

}

unsigned int sleep(unsigned int seconds)

{

 printf("sleep1: "); print\_time(); printf(" ... ");

 unsigned int result = NEXT\_FNC(sleep)(seconds);

 print\_time(); printf("\n");

 return result;

}

static void checkpoint()

{

 printf("\n\*\*\* The plugin %s is being called before checkpointing. \*\*\*\n",

 \_\_FILE\_\_);

}

static void resume()

{

 printf("\*\*\* The plugin %s has now been checkpointed. \*\*\*\n", \_\_FILE\_\_);

}

static DmtcpBarrier barriers[] = {

 { DMTCP\_GLOBAL\_BARRIER\_PRE\_CKPT, checkpoint, "checkpoint" },

 { DMTCP\_GLOBAL\_BARRIER\_RESUME, resume, "resume" }

};

DmtcpPluginDescriptor\_t sleep1\_plugin = {

 DMTCP\_PLUGIN\_API\_VERSION,

 DMTCP\_PACKAGE\_VERSION,

 "sleep1",

 "DMTCP",

 "dmtcp@ccs.neu.edu",

 "Sleep1 plugin",

 DMTCP\_DECL\_BARRIERS(barriers),

 NULL

};

DMTCP\_DECL\_PLUGIN(sleep1\_plugin);

The library search order and a hidden bug in the DMTCP architecture:

|  |
| --- |
| Libplugin1.so |
| Libdmtcp.so |
| .. |
| .. |
| .. |
| Libgcc.so |

a.out library search order order of initialization order of finalization

The order of initialization is opposite to the library search order, since the initialization of the inner most layer must be done before the initialization of the outermost layer. While destroying/finalization the order should be the same as the library search order. This concept is very prevalent in all on computing and is lso seen in C++ constructor initialization and destruction phases.

This can also be understood if the libraries are seen as layers over one another as concentric circles.

