Invite Research Project

Summer 2009- Final Report

Waseem Ilahi – wki2001@columbia.edu

Professor Gail Kaiser

Chris Murphy

Abstract

There is a limit to how many tests one can perform on a software before deploying it. This limitation makes it possible for some of the bugs in the code to go unchecked. Thus, the system/software is released with those bugs still present and unknown to user and developer. Our research is based on solving this problem in the manner that even after the software has been released to the general public; there can be a way to keep testing it continuously, without disrupting the original order of execution. This can be achieved by branching off the original path of execution and let the software go on, while we perform tests on this branched version. To make sure, we don't hurt the user's system by performing these tests, we use the virtualization software. Instead of forking the process and running the process under the same system, we "checkpoint" the process, ZAP virtualization software, and put this process in a separate "sandbox" we call a "pod". This way the tests are performed, without any effects on the original execution. So far, we did not have a way for the code to perform a checkpoint on itself, by which it means on the pod it is in. The process had to call an external manager that would do this task. This was done over TCP/IP. My job, in Summer 09, was to implement a function, that can be called by the program and when called, it takes the checkpoint of the pod the process is running in. I then had to integrate this new functionality with the InVite code, which originally executed a fork and ran on the same system, without any virtualization. To achieve this goal, I had to go through the source code for ZAP and the manager that is built on top of it, called DEJAVIEW. This paper provides a description of the source code for both the software as well as the code that I wrote. I also included the path (line numbers and the file names) of execution for some of the major commands that ZAP executes. In this paper, I will try to include all my findings during Summer 09, so the future project students can use this paper to avoid going through what I did, when I was going through the source code for the two modules.

1. Introduction

The traditional testing is never able to find all the bugs in the program. Guanyin team realizes this fact and has been trying to investigate different testing methodologies; one of them is known as "In Vivo Testing". This method performs the tests continuously, even after the deployment of the software. The functionality that we had so far, for this suite was that it forked the original process and then the test was run on this child thread. To avoid system crashes and major changes in the execution, a virtualization technique was used. Guanyin team started using the ZAP/POD system developed in Professor Nieh's lab. Using ZAP software to take a snapshot of the entire "system" and then branch off at that point, put the branched process in a new "pod"(virtual system), and then run the branch along with the original, Guanyin team was able to achieve that goal of preserving the system variables, etc. However, there was no way of check pointing the pod, the process was in, from within the pod. This is where this summer's work comes in. I had to implement that functionality. This paper describes the findings I made during this semester, while researching the ZAP and DEJAVIEW source code, in order to write the function call that takes the checkpoint of a pod from within the pod. Section 2 gives a little description about the ZAP, DEJAVIEW and INVITE and what their link is to my work for the semester. Section 3 talks about the DEJVIEW source code. Section 4 will go over my findings, while working with the ZAP source code. Section 5 will talk about the function call that I wrote and why it's a major step towards utilizing the Dr. Nieh's ZAP system even further. Then there will be the documentation for going through the ZAP code, in case future students need some guide for going through the ZAP source. After that, there will be a list and a little explanation of each file in my source code, which I wrote over the semester. The conclusion will be done by lessons learned while working on this project and the future directions.

2. ZAP, DEJAVIEW & INVITE:

ZAP is the virtualization system, developed in Dr. Nieh's lab, by Oren Laadan. Dejaview is the "manager" system that encapsulates the ZAP system, to make it easier for the user to use the ZAP system. DEJAVIEW is written as python scripts. When a command is given to the dejaview, it sets up some variables and pulls up the appropriate entry from its database, representing the session/pod in question. It sets the appropriate variables, then it sets the "command" to give to the ZAP module and then hands over the control to ZAP, which then performs the actual function. DEJAVIEW is more like the mediator between the user and the ZAP system. The commands that ZAP requires are quite complex, but DEJAVIEW simplifies that. For example, when you execute a
dejaview add_proc command, you don't need to give it input output file names or many other flags that zap requires, because if you were to give the same command to zap, you would need to give it the names for standard input output and the flags for various other things.

For how to work with DEJAVIEW, you can look at my friend Moses Vaughan's final report for his work in spring 09, in which he gives a step by step tutorial on creating a pod and then adding processes and check pointing etc.

As I mentioned above, ZAP provides a thin virtualization layer on top of the original OS. It employs the technique of sandboxing all the processes with the virtual "Systems", called pods. Each pod is independent originally and whatever happens in one pod doesn't affect the other pod. So, in our case, where we have to test a system, without making any changes to the original, we put the process in a pod and then checkpoint the pod. This creates a checkpoint file that contains everything required to "branch" off from the original pod, which we do. Then we resume the new pod. This pod can run the tests, while the original pod contains the original process.

Invite on the other had is a testing suite, that performs the tests on the software, even after they have been deployed. This continuous testing ensures the possibility to catch most of the bugs. The InVite software forks off of the point where the test is to be performed and then the child performs the test, while the parent continues without being affected.

Now, all the technology that we had, didn't give us a way to check point the pod from within the pod itself. ZAP source does have the resources that can we used to achieve this functionality. My job for this semester, as I mentioned earlier, was to explore the code, in an effort to find a way to do that.

Then use that new functionality to give INVITE the ability to checkpoint and use the pods, rather than a mere fork. For that I explored the ZAP and DEJAVIEW sources, before I started programming. Below, I will provide a brief discussion on my findings, about both the systems, before I provide the description of my function call.

I would like to thank Mr. Oren Laadan for his help, and being patient with me when I asked him question while learning the inner workings of the software.

3. DEJAVIEW SOURCE

Dejaview is written as python scripts. There are four main files that do most of the work, namely dejaview.py, session.py, pod.py, zap.py. They call upon other variables that belong to other files and classes in those files. When the user executes any command like 'dejaview command', a function inside the dejaview.py file is called. This function first of all checks the argument to make sure they are valid, if not it prints out the "Usage" output. Then it gets the session that the command is about, if its 'new' command, it creates a new entry in the list. Now that it has the session, it calls a function specific to the session, this function is defined in session.py file, which is one of the most important, especially for the purposes of my work during summer. Now session.py has the session class that holds major information about the sessions. For example, name, id, children, ckptfiles for its checkpoints. It's all in a form of dictionary that it calls cpentry. This is where the session is initialized. I had to deal with the session listckpt, session branch, session resume functions, while creating the function for check pointing. Session.py then calls pod.py which performs the pod related functions. This is where the pods are allocated and de-allocated for the sessions. This is also where the session is mounted or un-mounted. It sets up some variables and based on what is required it calls the appropriate function in the zap.py file. Now this is where the interaction between ZAP and DEJAVIEW happens. The function in zap.py sets up the variables, like the checkpoint file name directory paths, etc. Then it sets up the important 'cmd' string. This is the string that gets passed to the zap module, which then executes it, to do whatever the user asked for. A few examples for various commands are given below: (quotes excluded)

dejaview ckpt sess1: cmd = "sudo /dejaview/zap/zap/utils/zap -ve zap p0 -bl -S / --inpod -o /var/cache/djvw/checkpoints/ck.1"

dejaview add_proc –u root sess1 /tmp/testckpt: cmd = " sudo /dejaview/zap/zap/utils/zap addproc p0 --stdout=/tmp/testckpt.out --stderr=/tmp/testckpt.err --append -- /usr/bin/sudo -H -u root DISPLAY=:0.0 -- /tmp/testckpt "

dejaview resume sess2: cmd = " sudo /dejaview/zap/zap/utils/zap -vseg unzap p1 --inpod -i /var/cache/djvw/checkpoints/ck.1 -f --port=20000:20001,25000:25001 "

(refer to the functions in zap.py for corresponding variables)

remember p0 , p1 are pods' names, use your own pod names when testing the commands. This is how dejaview makes it easier for user to do all those tasks. However, not all of the commands go the zap module, e.g., dejaview new, creates a new session without needing the zap module.

<u>The locations for the above files in the dejaview source are as follows:</u>

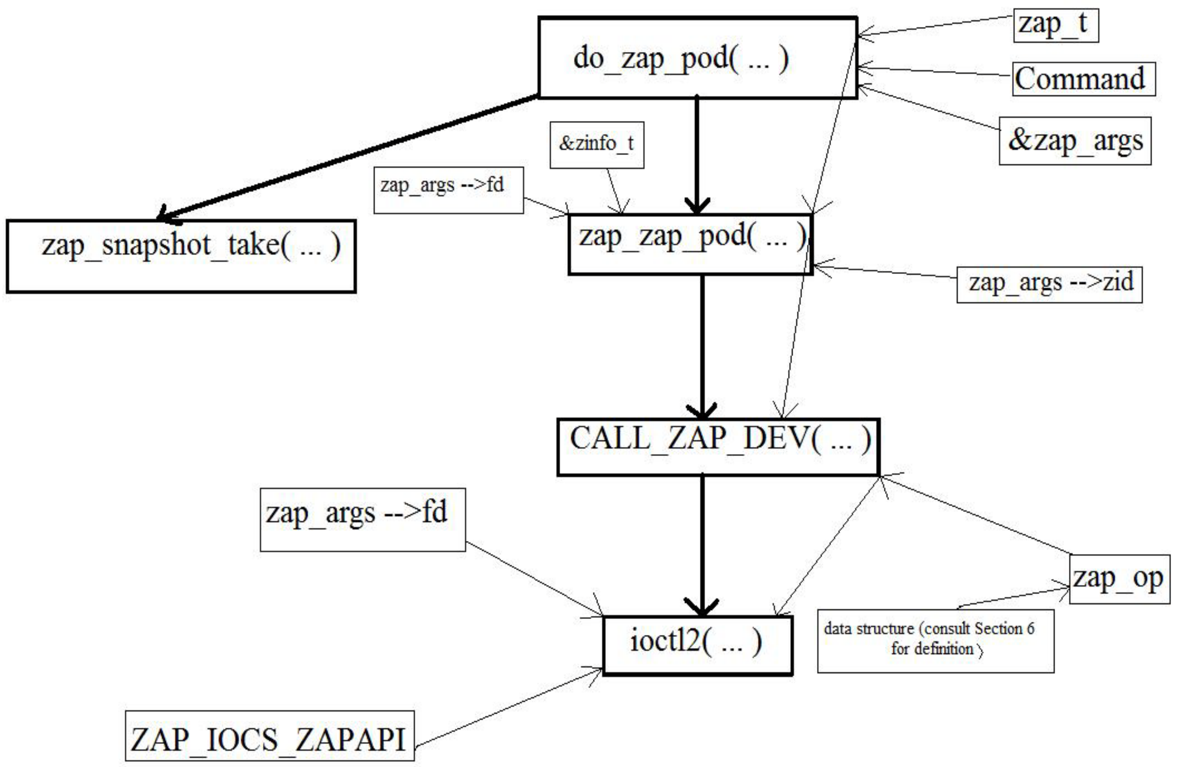
dejaview.py →	/dejaview/utils/manage/dejaview.py
session.py →	/dejaview/utils/manage/djvw_manage/utils/session.py
pod.py →	/dejaview/utils/manage/djvw_manage/utils/library/pod.py
zap.py →	/dejaview/utils/manage/djvw_manage/utils/library/zap.py

4. ZAP SOURCE

The file that I was asked to look into was the tbd.c file inside the /zap/utils/ directory. However, in an effort to investigate the source, I ended up going through some other very important files. tbd.c is the source for a thinkback daemon, that takes checjpoints at some intervals. The important thing in the file was the call to the zap library function

do_zap_pod(…). Now this function, once called with the right parameters, does the work for you. You simply call this function after setting up a couple of structures and passing them as its arguments, and it takes the checkpoint for you. I have the index of these important functions in the section 6, for anyone who wants to have a look at it. To look at how I used do_zap_pod( ) have a look at Section 5, subsection do_zap_pod( ).

P – T – O →



Path of Execution, through the ZAP source, for a checkpoint

To work with the function as well as the rest of the structures etc, I had to go in the source code, follow the examples and also look into their definitions. From my investigations, I found out that at the relatively high level of using the zap source code and wrapping around it, like myself, you need to know about the following files. The rest were not that important at the level I needed to go into the source. The files are:

zap.h	→	/zap/zap.h
zap.c	→	/zap/utils/zap.c
utils.c	→	/zap/utils/utils.c
zutils.c	→	/zap/utils/zutils.c
utils.h	→	/zap/utils/utils.h

zutils.h → /zap/utils/zutils.h

zaplib.h → /zap/lib/zaplib.h

zaplib.c → /zap/lib/zaplib.c

zap_api.h → /zap/zap_api.h

zap_api.c → /zap/zap_api.c

and finally

mig.c → /zap/utils/mig.c

The do_zap_pod(…) function is declared in the mig.c file. While calls yet another function zap_zap_pod(..) which is defined in zaplib.c. What's important to mention here is that, even though I needed only do_zap_pod(..) and follow its path to the lowest level to figure out its functioning, every file mentioned above defines the functions for each zap command and these functions call other functions, that are defined in the same file. What I am trying to get at is that, the hierarchy follows the same path, to the most part, through the files, no matter which zap command we are trying to execute. All the low level functions are defined inside the zaplib.c file for mostly all the zap commands. mig.c is also important because it defines the wrapper functions that can be used by the programmers like myself for our own purpose.

The important function, structures and variables, and their place in the appropriate files is mentions in Section 6.

P – T – O →

5. Checkpoint Function Call and Integration with InVite { ckpt ( ) }

After I was finished with the catching up and understanding the zap & dejaview source code, I started programming for the function call: int ckpt(void);

The function ckpt( ) takes no arguments but returns three kinds of values, negative, zero and positive values.

Return Values

This function returns negative value for a failed attempt at the checkpoint. The failure can occur on many different positions in the execution. Since, the check pointing in the ZAP heavily relies on files, if a file fails to open then we have to return -1 showing that the function did not complete the task in its entirety. Among those files are the counter file, for the checkpoint file name and the log file, named tbd.log. Also, if the do_zap_pod( ) function returns a negative value ckpt( ) also returns a negative value. Therefore, unless everything gets completed successfully, we get a negative return value from ckpt( ).

ckpt( ) returns 0 if the process is the parent process. This means that the current process is the original process that called the ckpt( ) function.

ckpt( ) return a positive (non zero) value, corresponding to the child's Id, if the returning value is the child process. For example, if the session is check pointed and then resumed three times, the function will return 1, 2, 3 respectively for each of the child process and zero for the parent process. The checkpoint in zap is basically a file with all the information in it.

do zap pod( )

The do_zap_pod( ) function gets three arguments. The one in the middle is very simple. It is ZAP_ZAP_POD, the command that tells ZAP to take a checkpoint. It is used in the lower level function, to tell them what the caller has asked to do. The other two arguments are structures of the type zap_t and zap_args respectively. Section 6 below tells us where they are defined in the ZAP source, so the user can have a look at them. Now the first of the two structures, zap_t, gets filled by the do_zap_open( ) function, which I will talk about a lot in Section 6. Apparently, this function opens the “zap” device to read/write to it. The second structure is very important, as it holds the flags and the file descriptor for the checkpoint file.

Now, in ckpt( ) I first set the zap_args’ flags as ZAP_POD_CONT | ZAP_MIG_INPOD, which means to let the session keep running after the checkpoint has been taken and the checkpoint is being taken from inside the pod. Other than that, I set a few variables necessary to take the complete checkpoint. Then after the name for the checkpoint has been established, I open it and assign its file descriptor to the fd field of zap_args. I get the zname and zid fields of zap_args from the name and zid fields of the zinfo structure (see Section 6 for information). Now we have all three arguments ready, so we pass them along to do_zap_pod( ), which does its thing and returns the result to us.

Logfile and checkpoint Name

When we run the command **dejaview listckpt session_name**, the dejaview manager calls the getAvailableCheckoints(self) function inside the session.py file. The said function calls another function called __concat_tbd_checkpoints( ) and then it returns the list of checkpoints on the current session. Now the __concat_tbd_checkpoints( ) function looks into the

/dejaview/fs/cache/sess.”id”/checkpoints/ directory and sees if it has a tbd.log file. This file is suppose to be a log file, that has the checkpoint information listed in it. __concat_tbd_checkpoints( ) then reads the file entry by entry and then matches each entry to a predefined expression. And stores the information, it extracts from the entries it reads from the file, inside a list. Then it assigns these values to the appropriate variable. Now it checks if the entry that it just read already exists in the system, if it does then it continues, else it assigns all the values to a dictionary entry named cpentry. One of the keys of this dictionary is the checkpoint file name. Now after setting the keys and the values, it appends the cpentry to the list of checkpoints for the session. This continues for all the entries in the tbd.log file. Then the __concat_tbd_checkpoints( ) saves the session udates and returns.

Because of the above stated facts we need to populate the tbd.log file after do_zap_pod( ) returns a positive (non zero) value. This is because we need to update the log file only for the parent process. So after the the do_zap_pod( ) returns, we check, if the return value was zero then we update the log file, with the information we get from zinfo_t as well as the time the checkpoint was taken and the checkpoint number that we get from the counter file that I mentioned earlier.

Now, the __concat_tbd_checkpoints( ) function, when enters the checkpoint name in cpentry, it names it tbd. “#”, so we have to make sure that we name the checkpoint file the same way and that our tbd.log entry is consistant with the checkpoint file’s name.

Child ID

This brings us to the final important part inside the ckpt( ) function, which is also related to the get_childID( ) function. When do_zap_pod( ) returns, we check to see if the value was negative and if were, we simple return that value, but if it was not then we check to see if it were zero, and if it is zero then we know that the current process is the child process, so we return the return value of the get_childID( ) function.

The get_childID( ) function opens up the file named “ChildIdentifier” and reads the value stored in it. This value is the child Id of the current process (child). The value gets updated when a session is branched by dejaview branch command. So the unique child id is relative to the time the child was branched and the time it was resumed. You can branch two children and resume them in the opposite order, but they one branched first will have the id =1 and the second will have 2. More detailed explanation; we consult the Section 7 of this report, where I talk more about session.py, which holds the code more this feature.

Integration with InVite

For integration of this function ckpt( ) with Invite, I merely had to replace the fork command inside the precompile.pl with ckpt( ). For more details on this also, look at the Section 7, paragraph three, where I talk about the changes made to InVite code.

The code is commented itself and very self explanatory, what is important to mention here is the steps to take for the whole process. I do have various scripts written that test the function call in various ways, but I will write a step by step process in testing the function. You can see the usage for the scripts in the Section number 7.

P – T – O →

Demo

These are the steps I take to test the function: (make sure you are a super user for it to work)

I will number the commands that I run:

i) sudo su /* become the super user. */

ii) cd /home/CKPT/ /* move to the directory of the source code. */

iii) make /* makes the target “testckpt” → calls the ckpt( ) to take checkpoint */

iv) touch empty /* Script for creating an empty pod. */

v) chmod 755 empty /* change its permissions. */

vi) dejaview new –s demo1 –nilfs –size=1024 /* create a new session. */

vii) dejaview start –start=/home/CKPT/empty sess1 /* start the session with the empty script.*/

viii) cp testckpt /zap/p0/root/tmp/testckpt /* copy the test to the pod. */

ix) dejaview add_proc –u root sess1 /tmp/testckpt /* add the process to the pod “run it”*/

x) dejaview listckpt sess1 /* see the list of the checkpoints on the session. */

xi) dejaview branch –c 1 sess1 sess2 /* branch off at the point of the checkpoint. */

xii) dejaview resume sess2 /* resume the check pointed session. */

xiii) more /zap/p0/root/tmp/Parent_Output /* see the parent’s output. */

xiv) more /zap/p1/root/tmp/Child_Output /* see the child’s output. */

You can repeat the xi and xii steps to create as many children as your want. Make sure you keep track of their names or ids to know which is which, and you can check their child ids by repeating the step number xiv but with the appropriate pod number.

To further understand the functionality, read the source code, provided with this final report and also look at the Section 7 below. The material in section 7 is also available as a README with the source.

6. Zap Source Hierarchy

This section is for the use of the programmers that intend to write wrapper functions and use the ZAP source to add new functionality to the ZAP system or to improve the already existing functionality.

In the following sections, I provide the important variables and data structure that ZAP source utilizes and its functions need and arguments or set for further use. Then, I provide the hierarchy of the functions that get called, when some of the important ZAP commands get executed. I will try to document the more significant functions, which can/may be used for above mentioned future developers.

Helper variables, structures and functions:

This data structure gets set when we call the do_zap_open() function. It contains all the important information about the session/pod that we are going to work with. The above mentioned function opens up the /dev/zap and then sets the the zapt_t structure after allocating space for it.

struct zap_struct → declared line 64 – 116 in < …/zap/zap.h >

typedef struct zap_struct   zap_t   line 33   in < …/zap/zap.h >

extern zap_t zaph; line 34   of   < ../zap/utils/utils.h >

The following variable is used for zap ID.

zid_t   /* is   an   int : */

extern zid_t zid; line 33   of   < ../zap/utils/utils.h >

The variable to hold ZAP_DEVICE.

extern char *dev;   line 30   of   < ../zap/utils/utils.h >

This is an important data structure. It holds the information about the pod/session. We can set it by passing it to the function, zap_query_pod(…) , which as its name says looks for the pod and if it finds it, then sets the zinfo data structure with is values. This function is defined inside the zaplic.c file.

extern zinfo_t zinfo; line 31   of   < ../zap/utils/utils.h >

This is just a global variable that holds the pod name in it.

extern char *zname;   line 32   of   < ../zap/utils/utils.h >

These two macros can be used for ease of use by the programmers.

```
#define PATH_LEN            4096                  < ../zap/utils/utils.h >   line 174

#define ZAP_DEVICE          "/dev/zap"       < ../zap/utils/utils.h >    line 175
```

The following function logs the error , prints it and then kills the process.

extern void die(const char *fmt, ...);   /* log/print err, die (1) */ line 45   < ../zap/utils/utils.h > (defined in utils.c line number 125)

Following are the commands that differentiate among various things that ZAP is trying to do. For Example, for checkpointing normally, we use ZAP_ZAP_POD.

/* zap commands */ file : < zap/zap_api.h > lines 61 – 82

```
enum {
        ZAP_CREATE_POD = 1,
        ZAP_REMOVE_POD,
        ZAP_SIGNAL_POD,
        ZAP_ADD_PROC,
        ZAP_GET_RPID,
        ZAP_GET_VPID,
        ZAP_QUERY_POD,
        ZAP_CMD_POD,
        ZAP_ZAP_POD,
        ZAP_ZAP_NET,
        ZAP_ZAP_DIST,
        ZAP_UNZAP_POD,
        ZAP_UNZAP_NET,
        ZAP_UNZAP_DIST,
        ZAP_SET_OPTION,
        ZAP_GET_OPTION,
        ZAP_ZAP_SELF,
        ZAP_UNZAP_SELF,
        ZAP_COMMAND_MAX
    };
```

This structure gets set by the functions that are defined in the zaplib.c. And it is this structure that is passed to the system call and the values stored in its elements tell the system call what to do.

```
    struct zap_op {  /* file : zap_api.h   line  # 50. */
        int action;
        zid_t zid;
        long arg1;
        long arg2;
        long arg3;
        long arg4;
        long arg5;
    };
```

Now this is an important structure. This structure needs to be set to be gived to the library function, that is used for taking a checkpoint, actually it is passed to other major high level library functions as well. These functions then set different things according to the values the elements of this structure hold, e.g., the 'flags' filed, lets you know which flags are set, etc.

struct zap_args lines 85 – 135 file zutils.h

static char const *__opt_s;   //line 223   file   utlis.c

Functions Used by ZAP API

This is simply a descent into the code as various commands are executed by ZAP. This section can be used as a guide to understand, what ZAP is doing, when we try to execute a certain command. This is not a particular if else decent, it simply lists all the possible branches in one list, in each subsection, I will try to document it appropriately to make it clear. I will try commenting the use of each function, to the right of that function.

Important Note: The subsections and their subsections etc, list the steps each function takes and the steps the function called takes and so on so forth.

1) Process Management:

i) Add Process:

This command tries to add a process into the existing pod.

This function add_proc( ) first declares a few variables, then allocates space for zinfo.root and zinfo.name, then it queries the pod, I already explained the functionality of zap_query_pod( ) function. If the function fails at any point then the die(..) function is called, which is also explained in the above sus section.

a. static int add_proc(zap_t zh, zid_t zid, char *name, char *fin, char *fout, char *ferr, int fapp, char *cwd, char **argv, char *envp[]) line 448 – 566 < …/zap/utils/zap.c >

i. zap_query_pod(zh, zid, zname, &zinfo) /* explained above. */

ii. die(….) /* if the above function fails. */

iii. chroot(zinfo.root) //change process's root into pod's root

iv. assert_fs()//defined on lines 441 – 446 //warn user about /lib/tls(futex)

v. chdir("/") //change working directory to inside the pod root

vi. zap_un_string(zh, pfd) file < zap/lib/zaplib.c> line 368 – 468 (fork two times) return value means its either granddad or the grandchild?)

vii. STATUS(who); //prints out the status (defined earlier in zap.c)

viii. zap_add_proc(zh, zid, name, 0, 0) //the main function that adds the //process file < zap/lib/zaplib.c> line 470

zap_add_proc( ) sets the zap_op struct according to the parameters it gets and the passes it along to the CALL_ZAP_DEV(..) macro as 'op' argument

CALL_ZAP_DEV(zh, op); //same file line 67 (very low // level ; after this we trap the system calls and use ioctl)

ix. int zap_close(zap_t zh) //close the zap file descriptor.

x. /* open "redirected" input/output/error */ This part sets the stdout, stdin and stderr streams to the appropriate files.

xi. execve(argv[0], argv, envp) //now execute the program After everything is set, we perform the execution.

P – T – O →

2) Pod Management:

i. ADD POD:

Well as the name states, this command adds a new pod to the system.

static void command_add(int argc, char *argv[], char *envp[]) //This is the main function, that gets executed, when the flag for add pod is noted. It does the following important things.

The following structure corresponds to all the options that the 'cmd' could have in it.

a. static struct option opts[] = { //c structure for command line args

```
          { "help",     0, NULL, 'h' },
          { "root",     1, NULL, 'r' },
          { "ip",       1, NULL, 'i' },
          { "noproc",   0, NULL, '1' },
          { "nopty",    0, NULL, '2' },
          { NULL,       0, NULL, 0 }
      };
```

The following parses the argument list and returns the appropriate flags and values.

b. __getopt_long(argc, argv, "+hr:", opts, NULL) line 226 file utils.c

Set the root variable for the current pod.

c. if (root == NULL) {

```
          sprintf(zbuf, "/zap/%s/root", zname);

          root = zbuf;

      }
```

Following function is as described in the section above, opens the pod/session we need to work with, as it sets all the structures for us.

d. zaph = do_zap_open(); line 248 file zutils.c

   i. zap_t zh = zap_open(dev) // line 513 file zaplib.c //This is the library function that does the actual work.

      1. version = zap_get_option(zh, 0, NULL, ZAP_OPT_VERSION, NULL, NULL); line 230 same file

   ii. zap_valid(zh) line 103 file zaplib.c

   iii. zap_verbose(zh, 1); line 93 file zaplib.c

As the name suggests it creates a new pod. It is at the same level as zap_zap_pod(..).

e. zap_create_pod(zaph, zname, root, 0) line 134 file zaplib.c

   i. CALL_ZAP_DEV(zh, op); //same file defined earlier

This step is optional, it sets up the ip address for the pod, if it is required.

f. zap_setopt_pod(zaph, 0, zname, ZAP_PODOPT_NETIP,

   (void *) &ip, NULL) < 0) line 108 file //zaplib.h

i. zap_set_option(zh, zid, zname, opt, (void *) (arg1), (void *) (arg2))

   //gets called by the above mentioned function.

   // line 212 file zaplib.c

The following function mounts the pod on the system. It calls sub functions, that do the work for it.

g. do_mount_pod_proc(zname, root) < 0)//line 77 // file zutils.h

   i. do_mount_pod(zname, root, "proc", "proc") //line 141 file zutils.c

1. mount(source, target, NULL, MS_BIND, NULL)

/* mount /dev/pts */ (honestly: that is all the documentation for this function, there was in the source as well)

h. do_mount_pod_pty(zname, root)//line 80 file zutils.h

   i. do_mount_pod(zname, root, "proc", "proc")//line 141 file zutils.c

      1. mount(source, target, NULL, MS_BIND, NULL)

i. zap_close(zaph);) //close the zap file descriptor.

j. STATUS(ret) //check the status.

3) Checkpoint/Restart:

i) Checkpoint/Zap

This command takes the checkpoint of the pod/session. command_zap(..) is the function that gets called, after the arg list has been parsed and we know what the user asked for, from the main function. Then it calls __comand_zap that does all the work.

a. static void command_zap(int argc, char *argv[], char *envp[]) //line 1724

   i. __command_zap(argc, argv, envp, ZAP_ZAP_POD);//line1308 zap.c

   First of all it declares a bunch of variables. There is a lot that needs to be done for this particular command. Then it initializes a bunch of structures and variables.

   //the options correspoing to the flags passed down.

   1. static struct option opts[]static struct option opts_d[]

2. static struct option opts_r[]
3. static char sopts[] = "+a:bfFhiklno:r:sS:" //set the flag lists
4. static char sopts_d[] = "+a:bfFhkm:no:sr:
5. static char sopts_r[] = "+bfFhiklr:sS:"
6. get_zname(&argv, &argc);// line 568 file zap.c get the pod’s name
7. __getopt_long(argc, argv, getopt_str, getopt_opt, NULL);
   As mentioned before this above function parses the argument list and returns the list of values.//line 226 file utils.c
8. The Lines 1431 – 1625 check the parses list and set the flags accordingly.
9. memset((void *) &za, 0, sizeof(struct zap_args)); // clear out the zap_args structure. (the structure is explained in the ‘helper variables’ section.
10. ftruncate(fd, 0) /* if it's a retry --truncate the checkpoint files*/
11. /* only if not a retry and not inpod -- open the output files */
    __zap_open_fds(&za, out, aux, meta, force?1:2, 1); // line 1297
    // file zap.c
12. za.procfd = zap_open_stats(zid, zname);//line 325 file zutils.c open the stats for the pod.
13. Set all the entries for the members of the zap_args structure. (line : 1675 – 1689)
14. do_zap_open( ) // line 248 file zutils.c

15. /* only do_zap_prep( ) on first attempt */Prepares the pod for the snapshot. It calls the query pod function that checks the pod for you

    do_zap_prep(zaph, &za) // line 1011 file utils/mig.c

    i. zap_query_pod(zh, za->zid, za->zname, &zinfo)
    ii. chroot(zinfo.root)

16. __zap_open_fds(&za, out, aux, meta, force?1:2, 1); // line 1297 file zap.c . set the file descriptor for the checkpoint file.
    a. do_open_file(out, STDOUT_FILENO, mode, rw) //line 107 file zutils.c

The actual checkpoint This is the function, I use to take the inpod checkpoint. Now this function does a whole bunch of things. It then calls the zap_zap_pod(…) function that passes the control to the macro that calls the ioctl system call to take the checkpoint. But before zap_zap_pod( ) is called do_zap_pod( ) does a few other things.

17. do_zap_pod(zaph, cmd, &za) // line 1048 file utlis/mig.c
    First it declares a few variables then it clears out the zinfo structure that it just declared. It then sets the flags and the name fields for the zinfo data structure.

If it was asked to take a preliminary snapshot to reduce the down time, it calls the following function to take initial snapshot.

a. zap_snapshot_take(zh, za, za->snapshot, 0):

   ←// the snapshot line (989) file :mig.c

   i. ret = ioctl(fd, inc ? NILFS_SYNC_INC : NILFS_SYNC, &snapid); ← the actuall system call that takes the snapshot.

Now this function actually takes the checkpoint. As mentioned above, it sets the zap_op structure and then passes it along to the system call/

b. zhid = zap_zap_pod(zh, za->zid, fd, &zi); ←//standalone checkpoint line 249 file zaplib.c

c. if the command stated to checkpoint network state : zap_zap_net(zh, zid, auxfd, &zi) // line 278 file zaplib.c // similar to zap_zap_pod( ) but with different intentions.

Snapshot the file system.

d. za->snapid = zap_snapshot_take(zh, za, za->snapshot, 1)// the snapshot line (989) file :mig.c

   i. ret = ioctl(fd, inc ? NILFS_SYNC_INC : NILFS_SYNC, &snapid); ←

If the flag say to kill the pod after checkpoint. If ZAP_POD_KILL is requested

e. : zap_remove_pod(zh, za->zid, za->zname, act)// line 162 file zaplib.c

If a (ZAP_ZAP_DIST) is ordered

f. zap_zap_dist(zh, za->zid, auxfd, &zi) //line 278 // file zaplib.c

18. Distributed Checkpoint: do_zap_remote(spec, &za)
19. ESTATUS(&za, "checkpoint", ret);//Check the status
20. STATUS(ret);//again check the status

<u>Last Updated August 25, 2009</u>

P – T – O →

7. The Source Code:

As my goal was to write a wrapper function around the original ZAP source code to achieve the inpod checkpoint functionality, I did not ‘touch’ the ZAP source code. However, since it is a wrapper function, it does rely on the existing code in the ZAP source. The most important part is the use of do_zap_pod(zap_t, ZAP_ZAP_POD, &zap_args) function. This function is defined inside the mig.c file, so I needed to keep the mig.o file in the same directory. As we can see, this function takes three arguments, the first is the data structure, that gets set by the do_zap_open( ) function, defined in the zutils.c file. Hence, the zutils.o is placed in the same directory. This tree goes up and we end up needed mig.o, utils.o, zutils.o and zaplibo in the same directory as our source code i.e., ckpt.c . We also need to include utils.h and zutils.h in the header file to use the datastructures and make the above mentioned system calls. However, I did not change any bit of ZAP source code.

In the dejaview source code, the only file I manipulated was the session.py file. It contains a class named session, which holds the information about each session. I added one more field to this class. This field gets changed at the ‘branch’ time, meaning, we add 1 to it and assign the resulting value to the same field of the child session (just created). I named this field ‘children’. When the session resumes and the resume function inside the session.py is called, before the pod.start(..) is called, I take the value from self.children of the resumed session and out it into a file named ChildIdentifier. The get_childID( ) function can read this information and know what id the child has.

For the Invite code to work properly, I had to change the precompile.pl file. I replaced the place where is forked and created a pipe to simply call the ckpt( ) and use the return value to run the test for the child process and continue if it were the parent itself. This even simplifies the code. The output that precompile.pl produces is in the file named ckpt_output.c.

Other than the above mentioned mutation, I did not change anything in the existing source code.

The List of Files and Directories:

a. The CKPT directory

session.py → goes in the : **/dejaview/utils/manage/djvw_manage/utils/** deirectory.

It is important to replace existing file with this one, for this version makes it possible for the child to know its ID. Since the original manager (dejaview) doesn't offer to run any scripts while resuming, there is no sure fire way to know the child id other than this.

mig.o utils.o zaplib.o zutils.o → needed for use of ZAP structures and functions in ckpt( )

ckpt.h & ckpt.c → The header file declares the function **ckpt( )** and the C code file defines it. The .c file also contains another local function that returns the child id. They are well documented and describe what they are doing. The documentation for the ckpt( ) and get_childID( ) can also be seen in the Section 5 where I talk about both of them.

testckpt.c → this is the test file that calls the ckpt( ) function and if the return value is zero, it writes 'I am the Parent' to the Parent_Output file, and loops forever. However, if the return value is the positive number, it copies the return value (child ID) in a variable and writes 'I am the Child # childID" and loops on it forever.

treeckpt.c → this test creates a tree like output, where the main function initializes two variables top and bottom to zero and then calls ckpt( ) and if the return value is > 0 it sets top =

child ID. And then it goes on to ckpt( ) again and this time for return value > 0 it sets bottom = child ID. So after branch and resume we get different values in the output file for each.

Makefile → makes all the targets to the copied to the appropriate pods.

b. The CKPT_Invite Directory

testinviteckpt.c testinviteckpt.ic ckpt.h and { ckpt.c (slightly different form the original) }

These are the same files that existed in the original invite code as demo.c etc. Just the name is different. (The changes were made to precompile.pl, for **InVite**'s integration with the **ckpt( )**)

./precompile.pl testinviteckpt.c testinviteckpt.ic

Gives us ckpt_outpt.c that is the final C source file that has the **ckpt( )** functionality built in.

Makefile → makes the final binary that can be run inside the pod to test the functionality.

c. The Test Scripts:

start → starts of three pods/sessions and adds the treeckpt, testckpt and ckpt_output one in each. (Now we have three pods that have already check pointed)

restart_test new_session_name → branches off of the testckpt session and resumes the child

Parent_Output_test → prints out the output of the parent process for testckpt.

Child_Output_test pod_number → prints out the output of the child inside the pod specified (Works for testckpt)

restart_tree → branches off of both the checkpoints taken by treeckpt and resumes them, then it branches off of the first child and resumes that child of the child.

Tree_Output pod_number pod_number pod_number → prints the output produced by the entire tree, given the pod numbers for the children.

restart_invite parent_session_name parent_pod_number → branches off of the ckpt_output into three child sessions and resumes them

Invite_Parent_Output → prints the output of the parent process only

Invite_Output pod_number pod_number pod_number → give it the pod numbers for the child processes and it prints the output for the parent as well as the children.

delete_session session_name → stop then delete any session

delete_tree → delete the tree created by the restart_tree script

delete_invite → delete the child sessions created by the restart_invite

delete_tests → delete the sessions created by the start script

addtest addtree ./CKPT_Invite/addinvite are three helper scripts that the user doesn't need to call directly.

*** Important note: Make Sure the permissions for all the scripts are set to 755 (chmod 755 script_name and put the CKPT directory in home directory, for example : /home/CKPT.**

****One Last Thing: ZAP is limited to the number of pods per system runs/mounts. Meaning, if you restart too many checkpoints, you will have to reboot the system for the pods to start smoothly.**

Remember that changing the order of execution of these scripts between output and delete may not always work.

8. Future Direction

Since, we have a way to checkpoint from within the pod, pragmatically and we have also successfully integrated this functionality with InVite, the next step in development of this project is to achieve the ability to migrate these and any other checkpoints taken, to different systems/pod on the same machine as well as over the network. For this capability, we will have to keep in mind that the two pods/systems must be compatible, and as Oren mentioned, we have to assume the base file systems for the two pods are equivalent as well. We will also need as external manager, which mediates the transfers to and from these different pods. Right now the manager for branch and resume, even inside the same pod is the human user.

9. Lessons Learned

First of all, because of this project, I learned a new language (to the most part), Python. Also, I was able to understand the workings of the virtualization systems and the concept behind this important topic. Now, by the end of the semester, I know most of the zap and dejaview source code. I was also able to work with InVite and understand the important concept that defines it. I am pleased that I was given the opportunity to work with such an interesting and important project as this one. I need to give my thanks to my project coordinators Chris Murphy and Dr. Gail Kaiser, for providing me the opportunity to work on such an interesting subject, because of which, I was able to contribute to the research, which made me feel proud and satisfied.

This is one of the few learning opportunities, in which I had to give the equal time in learning the prerequisites, before I could start programming. I am planning to continue on this project during the next semester as well and I hope I will do even better and achieve more milestones in the future.

10. References:

[1] The ZAP and Dejaview source code.   “Oren Laadan” (NCL)

[2] InVite source code. “Chris Murphy”   (PSL)

[3] Moses Vaughan final report S09   (PSL)