Newcomer's Guide

Overview

MacPorts, in layman's terms, is a software package manager.

A general question that comes to the mind when we start contributing to MacPorts that how is it different from other package managers?
What's special?

1) MacPorts points to the original source and not to pre-compiled binaries. MacPorts tries the best to ensure you are running the latest release of a software.
   (pre-compiled binary packages can also be generated through macports which make build time short as well)

2) MacPorts ships its own libraries that are to be used, because Apple keeps changing libraries time to time.

3) MacPorts is someway related to FreeBSD port collection.

MacPorts Repositories

Can be accessed here : https://github.com/macports

MacPorts Codebase

So to start, MacPorts main codebase is a combination of two sides:
1)macports-base (https://github.com/macports/macports-base)
2)macports-ports (https://github.com/macports/macports-ports)

To install a software, we define its requisites in portfiles(a tcl script).
The macports-base code acts upon the portfile in order to install it on the system.

macports-base

All the main source code for the base resides in macports-base/src directory.

Before reading this, go through this tcl tutorial:
http://www.tcl.tk/man/tcl8.5/tutorial/tcltutorial.html

<#
	TODO: good introduction to the complete base
	A diagram of architecture on how one file is related to other file with arrows
	describing how the code flows
#>

Trace Mode

Need

To install a port, say x, we use the command,

sudo port install <x>

Now what the base code mainly would do after having the portfile:

1 : Get the dependencies

Lets say, the port <x> depends on some other port (e.g., python2.7).
It has to install that port beforehand.
Dependencies can be anything like a library, a tool or anything.

2 : Install all the dependencies

All the dependencies are installed.

3 : Install the main software

This is when the need of trace mode arises.

Software <x>, which needs to be installed has some build processes.
This build process is the one that requires those dependencies.
Now as the build process needs to use that dependency, it searches certain locations for it. This searching is done generally because of cmake, GNU make, clang, g++, gcc & other shell scripts.

While searching those locations, the build process may try to use the incompatible version of that dependency.
(The user may have installed that dependency in the past, or some other package manager has their own version of that dependency)

This would either lead to more needed dependencies or errors.

This is where trace mode comes into play!

How it works? (In simple terms)

sudo port -t install <x>

When the software was building and trying to search for dependencies, what if we simply hide the incompatible ones? Then the build process won't use the wrong dependency. This is what trace mode does.

Before building the software, two things are done:

1) A server side is setup which can check and provide information about a path. It tells the client if the path it has received as an argument is path to a correct dependency or is path to some incompatible version of the dependency.

2) In order to check a path (by requesting the server), a library is loaded so as to replace all the file operations of libc with our own implementation.
   What it basically does is that during build, when many paths for dependencies are accessed, libc file functions (open, close, rmdir etc) which being responsible for accessing those dependencies are replaced.

Now that being done before build, when the software starts to look for dependencies during the build and tries to access them, the replaced version of the file function is called. (e.g., open gets replaced with _dt_open)
Now what this replaced function would make a connection to the server via sockets and asks the server to check that dependency path.
The server looks the dependency database for the path and returns a flag in order to tell if the dependency is ok or not ok.

If in the replaced version of function, if the server call returns file to be ok, it is calls the appropriate function otherwise it skips the call.
(e.g., If the build was trying to open a file, the replacer function _dt_open will first check the path trying to be opened and if the path is ok, _dt_open will call open, and otherwise proceed further)

Now by this way wrong version of dependencies don't get accessed.

Surfing through the code flow

In order to understand the code, the knowledge of the following is required:

1) Tcl :
   http://www.tcl.tk/man/tcl8.5/tutorial/tcltutorial.html

2) Dynamic library in C & LD_PRELOAD :

http://www.yolinux.com/TUTORIALS/LibraryArchives-StaticAndDynamic.html
(consider the book linked below for a better reference)

3) Unix Sockets with C :
https://www.tutorialspoint.com/unix_sockets/

https://www.youtube.com/watch?v=pdkGTYyvbPQ

(consider the book linked below for a better reference)

4) Processes & threads with C :

https://www.geeksforgeeks.org/multithreading-c-2/

https://www.youtube.com/playlist?list=PL-suslzEBiMrqFeagWE9MMWR9ZiYgWq89

5) File locks & mutexes with C: (Not a must to understand unless you wanna deeply understand the code)

https://www.youtube.com/watch?v=whYnqxKSBBo

6) kqueue() / kevent (Not a must to understand unless you wanna deeply understand the code)

Must refer to this if you donno about kqueue/kevent:
https://wiki.netbsd.org/tutorials/kqueue_tutorial/

https://stackoverflow.com/questions/3354733/triggering-kevent-by-force
(for understanding self-pipe used in code)

7) Atomic Operations (Not a must to understand unless you wanna deeply understand the code)

   https://www.geeksforgeeks.org/g-fact-57/

   All the above topics (except tcl & kevent/kqueue), can be referred from here :

   https://richard.esplins.org/static/downloads/linux_book.pdf

   Above is the pdf of this book : https://books.google.co.in/books/about/Advanced_Linux_Programming_Portable_Docu.html?id=a_DJCgAAQBAJ&printsec=frontcover&source=kp_read_button&redir_esc=y#v=onepage&q&f=false

Starts Here:

Note:
While surfing through the code its recommended to open the code files as listed and keep going as directed. Only when a complete path is specified, switch to the other file.

The user issues the command:

```
sudo port -t install <x>
```

Starts from *macports-base/src/port/**port.tcl***

port.tcl currently has 5000+ loc where most are procs.
It is better to read the script from bottom and find “# Main” by search option of text editor.

For a basic understanding, the code here is just for parsing the arguments in the command, understanding the command type (e.g., install, search etc.) and calling procs of macports.tcl which do the most main tasks.

The main commands called here are mport*(mportinit, mportlookup etc) and all are defined in macports.tcl.

The main points to focus in code in order to understand trace mode is as follows:

Firstly, *mportinit* command execution sets prerequisites before the port that needs to be installed can be looked up in the server.

Followed by that, *mportlookup,* looks for the port that needs to be installed.

After that, *process_cmd* gets executed that basically parses which command is given (install, search, uninstall etc.) and calls the appropriate action.

In *process_cmd,* appropriate action gets called, which in case of “install” is *action_target.*

In action_target, *mportopen* opens the portfile and after that *mportexec* starts the process required to install that port.

At last *mportclose* closes the portfile to prevent any leaks.

These mport* are defined in
*macports-base/src/macports1.0/**macports.tcl***

Now here in the proc *mportinit*, it checks whether to enable the trace mode or not, as per the params passed from port.tcl call.

mportopen will open the portfile.

Note that all this has to happen in a sub interpreter, so that portfile can't change any global variables. The sub interpreter is initialised by worker_init in *mportopen*, and *mportexec* uses that sub interpreter.

Then in *mportexec, eval_targets* gets called.

eval_targets is defined in
*macports-base/src/port1.0/**portutil.tcl***

eval_targets then calls *target_run.*

In target_run proc, search for "#start traclib".
It takes you to a point where the trace mode comes in action.

Main tcl code for trace mode:

This particular block of code in portutil.tcl is responsible to setting up the trace mode. Here firstly, it checks if the trace mode is enabled or is supported and enters the *if* block. In the block its executes *porttrace::trace_start $workpath.* This takes the control to a proc defined in porttrace.tcl.

*macports-base/src/port1.0/**porttrace.tcl***

Now in porttrace.tcl,

As soon as it begins, *create_slave* is execute which begins setting up server side on a separate thread.

Server Side Setup Thread:

A slave thread is created by command *create_slave* and $fifo contains the name which will be given to the server side socket for opening a connection.

create_slave basically creates a new thread which starts working on configuring the server side of trace mode.
We will first discuss complete creation of server and then come back here.

In *create_slave,*

By thread::send, various command are sent to the sub interpreter of slave thread for package inclusion.

Then *slave_init* command is sent to the thread. This is the first use code called.
Followed by this *slave_run* is sent to the other thread.

The create_slave proc ends here and shifts the control back to the *trace_start proc.*

In *slave_init {*
Here sandbox_violation_list, the list responsible for holding the paths which were accessed for dependencies but where were incompatible versions, and sandbox_unknown_list, the list responsible for holding the paths accessed, which were not dependencies & were unknown to macports.

tracelib setname $fifio
After this the socket name is set to hold $fifo.
tracelib is a command defined in pextlib.c which interfaces a C function to be called whenever tracelib command is invoked.
Those C functions reside in tracelib.c which we will discuss in a while.

tracelib opensocket

This makes the socket to start listening to any client requests.

}*slave_init* ends

In *slave_run {*
tracelib run // for accepting data from socket
}slave_run ends

Now this thread is just waiting for calls to be made. The implementation is in tracelib.c (discussed later)
Coming back to **porttrace.tcl**

The main thread simultaneously was setting sandbox bounds.

trace_sandbox is a variable being feeded with a list of such paths that are to be always allowed.

Sandbox basically includes directories, files etc that should always be allowed access.
Some standard directories like /usr/include which have many important headers should never be denied access so they are always allowed.

On the other hand, /usr/local should always be denied access.

The “paths” of correct dependencies should always be allowed access.
(The are checked at build time direct from registry and are **not being set here**)

Also in the same proc,path of a library named darwintrace.dylib is loaded into environment variable DYLD_INSERT_LIBRARIES (in linux the same is called LD_PRELOAD). This process is called dynamically loading a library.
This is explained in the book linked above.

This library is responsible for **Client Side Setup.**

The correct setup is explained later.

After loading this library,
tracelib setsandbox [join **$trace_sandbox** :]

Is called. This just sets up the hardcoded sandbox paths in the server side code.

Now, if you remember, this *trace_start* was called in portutil.tcl.

So after that sandbox set command, the control returns back here.

Now, in here, the dependencies are collected into a list by querying registry.

And it is made available to server side code i.e, tracelib.c by:

tracelib setdeps $deplist

After this the trace mode if block ends.

Immediately after that, this is done:

In here, pre run things get done.

```
if {$result == 0} {
        foreach pre [ditem_key $ditem pre] {
            ……...
            set result [catch {$pre $targetname} errstr]
          …….
        }
    }
```

In here, run starts.

```
if {$result == 0} {
    ……...
    set result [catch {$procedure $targetname} errstr]
    ……...
}
```

This is where trace mode comes into real implementation.
Now when the software starts to build, due to darwintrace.dylib, all file operations that will be conducted by the file to deal with any path are replaced by darwin trace versions.

Now taking an example,
Let's suppose,
The process tries to open a file. For this purpose it will have to call open(2).
This function is replaced by its reimplementation in
*macports-base/src/darwintracelib1.0/**open.c***
Here, consider the code for open.c:

```
static int _dt_open(const char *path, int flags, ...) {
        __darwintrace_setup();
        int result = 0;

        if (!__darwintrace_is_in_sandbox(path, DT_REPORT | DT_ALLOWDIR |
DT_FOLLOWSYMS)) {
                errno = ((flags & O_CREAT) > 0) ? EACCES : ENOENT;
                result = -1;
        } else {
                // Read mode and pass it to the syscall, because we cannot optionally pass
                // parameters to syscalls
                va_list args;
                va_start(args, flags);
                mode_t mode = va_arg(args, int);
                va_end(args);

                result = open(path, flags, mode);
        }
        debug_printf("open(%s) = %d\n", path, result);
        return result;
```

```
}

DARWINTRACE_INTERPOSE(_dt_open, open);
```

Now, in here, __darwintrace_setup() will ope

//bookmark

```
if {$result == 0} {
    foreach post [ditem_key $ditem post] {
        …….
        set result [catch {$post $targetname} errstr]
        ……...
    }
}

# Check dependencies & file creations outside workpath.
if {[tbool ports_trace]
  && $tracing
  && $target ne "clean"
  && $target ne "uninstall"} {

    tracelib closesocket

    porttrace::trace_check_violations

    # End of trace.
    porttrace::trace_stop

    set tracing no
}
```

The process now,