Revitalizing DCE

[Extended Abstract] *

AuthorName
Institute for Clarity in Documentation
mail@mail.com

ABSTRACT

This paper describes the design, implementation and validation of the ns-3 model of the Licklider Transmission Protocol, the standard transport protocol used to provide transmission reliability in Delay Tolerant Networks (DTNs). DTNs are an emerging field whose principles are used to provide communications in extreme and performance-challenged environments, such as spacrecraft,underwater, or disaster response scenarios. Evaluation of such environments requires the use of simulation tools. As of now, there is a lack of precise simulation models of these protocols, and concretely within the ns-3 simulator. The ns-3 model presented in this paper accurately models the LTP protocol and offers ...

Categories and Subject Descriptors

C.2.2 [**Computer-Communication Networks**]: Network Protocols—*Protocol architecture*; I.6.5 [**Simulation and Modeling**]: Model Development

General Terms

Theory

Keywords

ACM proceedings, LaTeX, text tagging

1 Introduction

The rest of this paper is organized as follows: Section 2 provides an overview on Delay Tolerant Networks and its transmission protocol standards. Sections 3 describes the design and implementation of the Licklider Transmission Protocol ns-3 module. Section 4 presents the testing approach procedure. Section 5 shows the validation procedure for achieving interoperability against existing implementations. Lastly, section 6 offers the conclusions and future work.

2 Challenges

2.1 libio vtable mangling

The vtable is abstractly a table maintainig references to functions called for virtual functions defined for a class or an entity. These functions can be overriden dynamically by user defined functions and the respective call for the cooresponding virtual function in a derived clas object can be bound to that function at runtime, unlike pre-defined functions which are static and fixed, and can be determined during compile time. The libc on Linux provides this highly flexible feature for all other user defined classes but the case with the FILE structure is not the same.

The FILE structure is a library defined structure which defines the overall organization, orientation and properties of any file I/O stream opened by the host application. It maintains different parameters to store useful operational fields like the UNIX based file descriptor number of the opened stream, the read/write offsets and buffer addresses of the stream. The pseudoname for the FILE structure as seen inside libc is *_IO_FILE*. Since, FILE is a library defined entity, the library provides it's own set of implementation for all possible operations on an open FILE stream. Whenever an *_IO_FILE* stream is allocated by the kernel, a contiguous memory location is allocated as a block called *_IO_FILE_plus*. The *_IO_FILE_plus* structure looks like this.

```
1 struct _IO_FILE_plus
2 {
3   FILE file;
4   const struct _IO_jump_t *vtable;
5 };
```

Now, by nature of implementation of the kernel's memory allocation processes, the contents of a struct are allocated in contiguous memoory locations. This can be veriied by the *sizeof* operation of C to verify that the sum of sizes of the individual members of a struct is equal to the size of the struct object. Similarly, the FILE and the *_IO_jump_t* objects are allocated in contiguous memory locations. Specifically, the *_IO_jump_t* areas is interesting to us, as it defines the callbacks or refference pointers to the functions handlers for each supported file operation. Some of the callback which are interesting to DCE and it's use cases are highlighed below.

```
1 struct _IO_jump_t
```

*A full version of this paper is available as *Author's Guide to Preparing ACM SIG Proceedings Using LaTeX2ε and BibTeX* at `www.acm.org/eaddress.htm`

WOODSTOCK '97 El Paso, Texas USA

```
2 {
3 ...
4 ssize_t(*) __read (FILE *, void *, ssize_t)
      ;
5 ssize_t(*) __write (FILE *, const void *,
      ssize_t);
6 off64_t (*) __seek (FILE *, off64_t, int);
7 int (*) __close (FILE *);
8 int (*) __stat (FILE *, void *);
9 ...
10 };
```

This structure acts like the vtable for the FILE structure, but it does not behave like the oridinary vtable seen when working with virtual functions and derived classes, which are dynamic and supports run time bindings. This vtable is rather expected to behave as a statically bound vtable (there does exist other libc functions like *fopencookie*, to override some of the FILE operation implementations, but not all, and it also does not atach itself to a standard file I/O stream, and rather works with user defined buffers a.k.a cookies).

DCE, which is supports simulating real applications on top of the ns-3 stack and also provides different networking stack choices (Linux, ns-3, FreeBSD), for the host application being run to get a simulated real world execution environenment and results. To support such an implementation and to sync application operations like system calls, file I/O operations, networking system calls, etc. it need to hijack all such calls and re-route it through corresponding handlers based on the application logic and simulation script implementation. Considering file I/O operations, DCE needs to have control over read/write/close/seek/stat operations of each open file, which requires us to overwrite the vtable handlers with the corresponding handlers defined in DCE's stdio definition source files.

Taking advantage of the contiguous memory allocation of the FILE and *_IO_jump_t*, we can execute a buffer overflow attack on the FILE object to overwrite the vtable with our custom vtable definition for all the operation we would want to overwrite. We can make a dummy *_IO_FILE_plus* pointer point to the explicitly casted FILE object. *memcpy* the existing vtable to a local copy, modify and overwrite the stream operations with our custom written implementation, and then re-point the vtable field of our dummy *_IO_FILE_plus* to the local modified vtable, and now we have control over those stream operations, which can now be routed through and and to behave as ns-3 streams, Unix FD streams, etc. based on the type of file descriptor that is defined. This is one of the productive uses of an buffer overflow attack to leverage control over FILE streams to regulate stream buffer flushing and data redirection, but the same could be used for use cases which might pose as potential security threats, as it lets penetration testers to make use of tools like pwntools etc. to gain control over application execution and important run time CPU register values such as the *rip* which is used for the relative addressing of application components(which is also how position-independent-executables work), which is more secure as compared to static addressing, where fixed address values of symbols and pointers could be gained by static analysis tools for run time application exploitations.

Post libc-2.25, security features have been implemented to glibc to identify such buffer overflow attack. Whenever any FILE operation is executed, glibc would verify if the FILE object's vtable could be trusted and is not corruputed or manipulated. To verify this, it makes a call to *_IO_validate_vtable*. Every libio vtable is defined in a unique section called *libio_IO_vtables*. By definition, libc would trust the vtable if the vtable of the current FILE object lies within this section. It checks if the offsets of this vtable lies between *__stop__libc_IO_vtables* and *__start__libc_IO_vtables*, if it does, we can continue with the operation, if not, libc conducts a final check by calling *_IO_vtable_check* which makes final checks on the FILE vtable pointer location, namespace and edge cases where FILE * objects are passed to threads which are not in the currently linked executable. Since, when we oveflow the *_IO_FILE_plus* and overwrite the *_IO_jump_t*, it does not lie in *libio_IO_vtables* section and it also does not pass the pointer mangling sanity checks, leading to a *__libc_fatal (Fatal error: glibc detected an invalid stdio handle);*

2.2 PIE loading and usage

PIE or position-independent-executables are applications compiled with special compiler flags, which allow the application to be loaded into random memory address, not depending on absolute symbol addresses, avoiding exploits which hijack the call stack by referencing constant function/symbol addresses. In the case of a PIE, every memory address is accessed with refference to what is called the *%rip*. The *%rip* is computed at the time of execution when the application is loaded into virtual memory. This makes it difficult for attackers to determine symbol location in memory.

In DCE, we support the execution of real host application in simulation, bridging the networking layer between the host and the specified networking stack, and also other system calls made by the host applications. Since, we might have to load several applications into memory, and alsp have control over the position of the main symbol of the loaded application, we need to have position independent executables, so that when they are loaded into memory, the symbol positions in memory are dynamic, giving us control over when an application is launched in a simulation which can be configured in the script using available ns-3 programming constructs for the *DceApplicationManager* class. To implement this, DCE used the *CoojaLoader* which uses dlmopen udner the hood to load the executable into memory. In glibc version newer than 2.25, security checks have been introduced to identify such PIE objects being loaded through dlmopen, and in case it finds the **DF_1_PIE** flag in it's ELF Dynamic headers, if would abort with an error.

2.3 Linux Networking Stack for DCE

DCE is by nature of ns-3 designed for writing network simulations and achieve real world like results. It makes is necessary for it to provide script writers with freedom to choose which networking stack they would want to make use of for their simulation script. We currently support the following networking stacks : Linux, ns-3 and FreeeBSD. Script writers can configure the *DceManager* class by setting the network stack to what they preffer. Currenly, the Linux network stack is based on net-next-nuse-4.4.0, which is built on top of base Linux kernel version 4.4.0. net-next-nuse is a library port of the Linux kernel, which does selective linking of requried kernel modules and components such as the networking, VFS, MMU, etc. layers, patching the holes with

glue code, exporting DCE useful callback structures, abstracting the internal network data flow, also syncing Linux kernel synchronization, process creating, and DCE-kernel task, IRQ and tasklet scheduling and synchronization, to expose a Linux like execution bed for host applications.

Since Linux kernel 4.4.0, the later Linux kernel releases have had major developements in almost all kernel compoenents, but for our specific use case the networing stack has seen major changes in several components such as the TCP Timer, Jiffies and clock HZ usage, napi working and internal enums and state definitions and checks, Packet flow, checksum and offloading hardennings, newer congestion control algorithms, ucounts API, etc. The current Linux kernel networking stack does suffice for now, but hasn't kept pace with the current research needs.

2.3.1 LKL

LKL also known as Linux Kernel Library, is a library port of the Linux kernel which through some pre-shipped helper shell scripts can be used to hijack all system calls made by the host application and map it through the ported Linux kernel rather than system defined implementations. It also allows one to setup network interfaces such as TAP, RAW, VDE , etc. with custom gateways, masks, IPs, etc. with the help of json configuration files. These helper shell scripts make use of LD_PRELOAD to reorder library loading to LKL written system calls to take control in place of the libc defined routines.

2.3.2 LibOS

LibOS can also be reffered to what internal architecture net-next-nuse uses under the hood. It is also a Lixnu kernel port which works on selective kernel module linking and patching on the go working princinple. It defines special link time constructs to include only specific kernel files and symbols which are needed for executing on top of the Linux kernel with *nuse*, which works similar to LKL in terms of hijacking system calls and rerouting them through nuse and kernel defined routines. It only links kernel components which lets LibOS start the kernel and run all *__initcall*(s) which are defined with a *__init* and registered as as a initcall using special macros. Thes routines are linked into special *.INIT* setions of the final linked library. These components include critical parts such *kernel*, *net* and other selective parts of *proc*, *mm*, *fs* , *drivers*, etc.

2.3.3 LKL vs. LibOS

Below is a parameter wise comparison of LKL and LibOS on the various design decisions which become critical for a complex application framework like DCE with very specific demands from it's underying network stack.

2.3.3.1 Linux Kernel Support

LKL, which was primarily designed to work as Linux-as-Library interface for application to be dynamically linked to at runtime, is built on top of Linux 5.2. The kernel port design of LKL, favours kernel version upgradation with little to no efforts. Abstractly, the kernel upgradation process would include a git rebase on the kernel version we would want to use, and project should compile with no major issues to deal with (some minor compiler, and Linux kernel header definition changes might come up, which should be resolvable with a bit of effors).

LibOS, which makes use of dynamic selective Linux kernel linking, bridges the gap between application workspace and Linux kernel networking stack with the help of glue code, kernel component connector code, and user application provided exported functions and callbacks for proper execution. This architecture required LibOS to modify some of the internal Linux kernel files for additional compoenets such as the slab allocator, which requires LibOS to setup preprocessor directives to select out SLAB allocator for specific Kconfig defined compiler directive, to pass on control to LibOS routines whenever required. Currenly LibOS supports Linux kernel 4.4.0. Upgrading to a newer Linux kernel version might be an intense process requiring one to deal with issues from header file changes, to complete changes in kernel components like the networking layer, memory maqnagement, namespace manager and kernel boot process.

2.3.3.2 In-Library Kernel Boot Order

sdsd

2.3.3.3 Compatibility

sdsd

2.3.3.4 Maintenance

sdsds

3 Solutions

3.1 Custom glibc-2.31 Based Build

On a Linux environements, multiple library systems could be selected to be linked to. A few options are libc are musl. The post-linking structure of the generated binary and the libary vary, in terms of the number of static linkages, symbol table, etc. For instance, musl works on the idea of single static linkage, in which, the library or executable compiled with musl-gcc, is statically linked to only one musl linked library called ld-musl.so.1, which defines all the symbols required by the application. This reduces the size of the executable to a huge extent, but also does more harm than good to DCE. The initial build step of DCE includes calling a script named dcemakeversion.c. This script is responsible for extracting the symbol table of the libc currently being linked to. All symbols for the libraries libc, libpthread, librt, lib and libdl. These libraries are the various modular extensions of the glibc providing features such as pthreads, math, dynamic library loading and the base libc library as well. The symbols are read from the Elf Headers of the respective shared library .so file, and stored in a local .version .

The symbol table for all these libraries are important for us, as DCE as a build outout generates what is called the libc-ns3.so. This is a shared library which is a wrapper for the local libc and DCE implementations, on top of which host applications are executed. This shared library defines all symbols which are defined in the local libc, as NATIVE and all the features implemented inside DCE as DCE. All other symbols which are new to DCE and not implemented by us, but is a part of the local system libc, is then refferenced in the .version files. These symbols are then defined as well, to avoid any runtime symbol lookup errors. It then generates preprocessor mappings for all the symbols. All DCE defined symbols will natively be mapped to DCE implemented versions of them, rather than the systel libc implementation, and all NATIVE defined symbols will be

mapped to global namespace implementations, which are the ones already implemented in the system libc. Users would compile their applications on top of the system libc itself, but with an extra -fPIC and -pie flag, which allows us to load the applications dynamically into out process address space. We then load out libc-ns3.so shared object file, and call out libc_setup function which initialiazes the system call mappings. We also subsequently load other libraries which we have generated in a similar way, i.e, libpthread-ns3.so , librt-ns3.so , libm-ns3.so and libdl-ns3.so . We then finally load the host application, lookup and call the main function of the application, which starts to now work on top of our custom libraries. musl-gcc on the other hand does not support modular libraries for these features, and for all cases would link the single ld-musl.so.1, making it impossible for use to seggregate the symbols for all our different libraries. glibc, on the other hand, works exactly how we would want the standard library to, and links all the standard libraries we require.

In an attempt to override the vtable mangling security checks, it was necessary to use a libc version that matches the system default libc version so that we dont see any symbol lookup errors at runtime because of application built on later libc release and beig loaded into an available namespace using dlmopen. But, we also had to override the security checks on vtable pointer mangling. The gcc compiler and linker options could be used to reconfigure the default build environment to build DCE on top of a custom built patched glibc. The default root directory where gcc starts to look for libraries, header files, etc. is '/' on Linux. We need to re-point this directory to out customg glibc root. This is where the –sysroot option is used to set it to correct bake build directory. We then add the custom glibc prioritized directory for library and header file lookup using the -L and -I option respectively. We then set the rpath and rpath-link paths for ELF executables that could be linked to the shared objects at run time or link time respectively. We then set the dynamic linker to one we have build currently, using the -Wl–dynamic-linker flag. Also, all these changes are placed under an unclosed -Wl –start-group, as DCE requires other linker flags, which when added we insert the ending -Wl –end-group.

3.2 Bake Build Automation

3.3 Docker environenment for DCE

3.4 net-next-nuse-5.10

- Requests are provided in the form of API functions ...

- Notifications are provided as callback functions ...

The LTP may be run at different protocol layers in order to provide support for this we provide Convergence Layer adapters ...

LTP uses engine IDs as its addressing system, we provide a lookup structure in the form of LTP to IP resolution tables ...

4 Results

4.1 Docker vs Native DCE Simulation Tests

4.2 Performance : DCE vs. ns-3

4.3 Google BBR v1 Validation Results

5 Related Work

6 Conclusions

7 Acknowledgments

8 References

[1] Space networks user's guide (snug). Technical Report Revision 9, Goddard Space Flight Center, Greenbelt, Maryland, August 2007.

[2] O. R. Helgason and K. V. Jónsson. Opportunistic networking in omnet++. In *Proceedings of the 1st International Conference on Simulation Tools and Techniques for Communications, Networks and Systems & Workshops*, Simutools '08, pages 82:1–82:8, ICST, Brussels, Belgium, Belgium, 2008. ICST (Institute for Computer Sciences, Social-Informatics and Telecommunications Engineering).

[3] A. Keränen, J. Ott, and T. Kärkkäinen. The one simulator for dtn protocol evaluation. In *Proceedings of the 2Nd International Conference on Simulation Tools and Techniques*, Simutools '09, pages 55:1–55:10, ICST, Brussels, Belgium, Belgium, 2009. ICST (Institute for Computer Sciences, Social-Informatics and Telecommunications Engineering).

[4] Ltplib documentation and test. http://dtn.dsg.cs.tcd.ie/sft/ltplib/.

[5] Ltp-ri specification and implementation. http://irg.cs.ohiou.edu/ocp/ltp.html.

[6] The interplanetary overlay network (ion). http://ion-dtn.sourceforge.net/.

[7] Common open research emulator (core). http://www.nrl.navy.mil/itd/ncs/products/core.

[8] R. Wang, S. C. Burleigh, P. Parikh, C.-J. Lin, and B. Sun. Licklider transmission protocol (ltp)-based dtn for cislunar communications. *IEEE/ACM Trans. Netw.*, 19(2):359–368, Apr. 2011.