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.

[*]A full version of this paper is available as *Author's Guide to Preparing ACM SIG Proceedings Using LaTeX2$_\epsilon$ and BibTeX* at www.acm.org/eaddress.htm

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.

```
struct _IO_FILE_plus
{
  FILE file;
  const struct _IO_jump_t *vtable;
};
```

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.

```
struct _IO_jump_t
```

```
{
...
ssize_t(*) __read (FILE *, void *, ssize_t)
    ;
ssize_t(*) __write (FILE *, const void *,
    ssize_t);
off64_t (*) __seek (FILE *, off64_t , int);
int (*) __close (FILE *);
int (*) __stat (FILE *, void *);
...
};
```

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

Apart from certain functionalities which are initialized only in a user OS such as HW drivers and devices, such as network buses, NICs, etc. LKL spins up an LKL high level CPU lock controlled thread, which makes calls to start_kernel. LKL being an uniprocessor system, initializes kernel on a singular LKL thread, locks of which are synced with Linux scheduler calls, which are called when the scheduler decides to switch execution control to kernel level tasks for preemption, and other tasks, which require certain memory level moderation to achieve atomic operation. Also, since the LKL CPU thread needs to be initialized before any linux functionalities can be use by the application thats using the LKL library, it eventually disrupts the flow of DCE scripts, which works on a scheduling algorithm, which works on specific ns-3 task context switch paradigms, different from the Linux kernel, making it wait for LKL's internal Linux kernel opertions to finish, before it can schedule other ns-3 Light weight threads, making simulation results differ a lot from real world observations.

LibOS, does not depend on the actual start_kernel, particularly because as opposed to LKL which sets up an environment for the Linux kernel all the basic requirements for the kernel to assume an actual linux workspace making it easy for LKL to access most of Linux functionalities without any change to kernel internals, LibOS on the other hand makes use of it's own lib_init fuction which calls specific setups calls required by the network stack of the kernel to work, such as proc, VFS, ramfs, scheduler, etc. It also overrrides scheduler member functions, syncing it with ns-3 scheduler's. LibOS creates an ns-3 task copy for each kernel task which is created by the copy_process, create_process, etc. kernel functions. Each such task maps with itself a callback function which should be called once the wait time for a task is over, or has been invoked as a part of a regular scheduling process. Once tasks are processed, it is also popped off the ns-3 task queue.

2.3.3.3 Maintenance

LKL, was designed to be a low maintenace Linux-As-A-Library interface, which exposes Linux subsystems through overided UNIX system calls. Since, LKL is an architecture level port of the Linux kernel, isolating LKL specific code to the arch and tools Linux folders, moving on to newer or custom releases induces less cost as compared to LibOS, which works on dynamic component compiling & linking build system through a custom Makefile, where are Linux files and modules to be compiled and linked to the final nuse shared object file, should be explicitly mentioned, and this listing should be exhaustive, assuring no function calls are being made by any critical component to another component which does not exist, and if it does, we need to fullfill the depedency graph for all such modules and file. This makes migrating to newer versions of Linux kernel, or for that matter, simply debugging kernel internals, requires more effort as one is not sure if it's the glue code that's malfunctioning, or is it the allocator or the ns-3 synced scheduler causing trouble.

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

Bake works on a build configuration script called bakconf.xml . The bakeconf.xml file is converted to bakefile.xml after the configuration file is parsed by the Bake module. Based on the bakeconf.xml file creates a build Dependency graph, maintaining build steps, parameters and post build commmands for each Dependency node. Bake dynamimcallly binds Dependency modules based on configuration options specified for each build module defined in the bakeconf.xml. The three mahor steps of a Bake build are, configure, download and build. Bake supports almost all major types of source code fetching methods, ranging from git, mercurial to standard publicly available zipped archive files. In the build step, dependencies are built in the order with modules having no dependencies on other modules, to the module having all dependencies already satisfied. To support the custom glibc build, the dependency graph looked something like this :

ToDo : Attach Bake Dependency graph ?

Bake has a source configuration option named patch, which can safely aplly a patch, without re-applying if it has already been applied before. Using this option, the custom DCE specfic libc patch is applied, which disables the security checks on vtable mangling, and also disables the pie object checks for dlmopen position independent executable loading. The glibc is then build using standrd build steps. The linux kernel headers files are then installed into /usr directory of a custom glibc's sysroot. This leaves with a standard Linux like system root to use for building DCE without any issues.

3.3 Docker environenment for DCE

3.4 net-next-nuse-5.10

Net-Next-Nuse is an architecture port of the Linux Kernel, which resorts to selective kernel module linking and takes control over critical kernel components such as the slab allocator, task scheduler, workqueue-waitqueue handling, timer based function invocation, as well as some kernel utilities such as jiffies and random number generators. It also sets up emulations of the network system calls for both general socket networking operations as well netdevice based operations. All of this is exposed through an API initialised by a simulator initialisation call, sim_init, which does a bidirectional mapping of the imported(DCE → Linux) and exported(Linux → DCE) function utilities.

On the other hand, LKL, implements some out of the box simple interfaces for relatively complex utilities, which in net-next-nuse, are mostly copied directly from the kernel code, making it vulnerable to possible code breaks, if certain kernel structures change over the years. But since LKL makes use of standard kernel calls and standard mount-read/write-remove workflow, it's more versatile and would,possibly, wouldn't break with new releases. Inspirations could be drawn from them and an net-next-nuse-LKL Hybrid could be developed.

The net-next-nuse-LKL Hybrid (Currently under continuous development) is discussed below : (The below details are subject to change as we move forward with the GSoC program)

3.4.1 Background

The implementation of net-next-nuse strictly considers only the use case of DCE and thus might not find application outside of DCE unlike LKL which is designed for running a plethora of networking applications, but it fits in really well when it comes to DCE.

Let's start with the Makefile design of net-next-nuse, because that's where the story begins. When Linux builds, apart from all other files, it generates two important files vmlinuz and initramfs. The file vmlinux is where all the kernel code(/kernel /mm /fs /net /security ...) is compiled and linked. When it comes to LKL, all it does it does a raw objcopy on the generated object files vmlinux and copies all symbols from vmlinux along with the global functions exported from the LKL arch port, and combines them into a final library called lkl.o, which is then passed on to the tools/lkl, to link the host, netdevice, utility, filesystem, and networking handler codes to make the final liblkl.so shared library. Seems good right ? Well, it did seem to me when I wrote my proposal, until I identified how it could actually become a bottleneck.

Now let's see how net-next-nuse does it. net-next-nuse understands that the way Linux Kernel behaves on the host, and the way the kernel interacts with the task scheduler and how the scheduler operates on the tasks, is completely different from how DCE manages fibers(called task in DCE terminologies). DCE is based on context based light weight threads called fibers, which require a custom scheduler, called TaskManager in DCE. On an ordinary host machine when you run any application, and then you spin up normal threads, you would not need to bother about scheduling the threads on the different cores of your cpu and maintaining mutex locks of the cpu, making the kernel responsible for jumping from one thread to another. But when you use fibers, you can jump from one thread to another only when one of the fibers yields to another fiber, and this context switch is done by a custom scheduler, which is written in TaskManager::Schedule(). Wonder how that is good ? Because when you use ordinary threads, the number of context switches the CPU has to do (push your thread data stack on to you address space, in and out), is a lot more expensive than making switches in Fibers.

That seems good though, but now we have a problem. We have a scheduler inside DCE, but now when we load a linux kernel library, we also have a scheduler that comes along with it. So, every new operation that you make inside the kernel creates something called a (virtual) kernel thread. But, how do we manage those threads, because the DCE TaskManager cannot see them because it's virtual. That's where net-next-nuse comes into play, it does a selective linking of kernel modules and utilities and avoids linking of the scheduler, workqueue, waitqueue and other such linked components, and rather implements them using DCE imported functions which maps to TaskManager utilities. So everytime some process wants to preempt and block, the kernel calls schedule() or schedule_timeout(), and we pass control to DCE to take care of it.

3.4.2 Slib Allocator

When the kernel is booted up by net-next-nuse in lib_init(...), it calls some of the required initialisation functions, and all, included, initcalls. Most of these initcalls require us to create a memory cache object called kmem_cache. These memory slabs are used to allocate memory to child objects. It uses functions like kmem_cache_alloc(...) and functions as simple as kmalloc(...) to allocate space for a pointer object. Now, the kernel already has slab allocators like, SLUB and SLOB, but all of them allocate space internally, giving no control to DCE. We thus use a custom SLAB allocator called SLIB, which calls internal DCE malloc(...) functions and sets up kmem_cache data structures and ctor's etc. It also implements compound and single page operations like _put_page(...), kfree(...) and array space allocation.

3.4.3 Virtual Kernel Task and its Real Fiber Equivalent

Let's talk a little about how threads are internally managed in the kernel. For every new task inside the kernel, for example, a syscall, creates an internal kernel thread. Now originally when a system boots up, there has to be some initial task that invokes the start_kernel(...) function. This is called the init_task. Whenever a new kernel thread is to be created, we make a call to kernel_thread(...). Along with the function that is to be invoked in the thread, and the arguments to be passed, we also need to pass along one more argument, known as clone flags. Internally, the kernel will try to clone a previous task (or you can say the init_task), as much as possible. These flags basically determine how far the kernel should go, as in cloning the structures. Some of the flags are CLONE_FS, CLONE_VM, CLONE_FILES, etc. kernel_clone(...) and passes a special clone flags data structure. This function further calls copy_process(...). copy_process(...) is responsible for checking which flags are enabled using a bitwise & operator and calling the corresponding copy utility, for example copy_fs(...), copy_files(...), copy_cred(...), It also makes a call to dup_task_struct(...)

which will bind changes of the current task_struct to a new one and return it back. After we get back a proper task_struct, the kernel_clone(...) schedules a fork for the newly created tNNask.

But the point is, we don't need most of it ? Why ? Because it's irrelevant to us. DCE is not a complete architecture level port and thus lacks a clear definition of init_task. So, for the very first kernel_thread the current task would be init_task, so not having a proper definition of it, creates a domino effect making all subsequent taks having incomplete structure, leading to possible SEGFAULTS. Another reason is, we do not use the kernel scheduler, rather we need to fork a kernel thread in such a way that the DCE scheduler can see it, and every time TaskManager::TaskWait(...) is called on a particular task, we can put the current job to sleep, and give the other tasks/fibers a chance to execute their set of functions.

So, to take control over the kernel thread creation, we rewrite the kernel_thread(...) function to call lib_task_start(...), which calls the internal TaskManager::Start(...) to create a DCE task, fills up the task_struct, sets the SimTask context with the task_struct and returns back the pid of the task. Also current as previously referred to before, is not a variable. It's a MACRO, which calls get_current(...) which we hijack and lead to TaskManager::TaskCurrent(...), which checks if the current task has a SimTask context, if it does, it returns it back. If it does not, it calls TaskManager::Start(...) to set it up.

3.4.4 Scheduler Workqueue/Waitqueue Implementation

Most of the scheduling code can be found in the /kernel directory of the Linux Kernel. This is very important to us and so we never compile the original kernel implementation for the scheduler. We rather implement all the key functionalities used by the networking stack. So, question is, if we are not running any applications inside the kernel, why do we even need to schedule anything ? The answer to this is blocking network calls. Imagine you creating a socket, binding and listening to it and then you make an accept(...) call, and the client application doesn't seem to ever come up. The system would go to a standstill if it did not schedule the client application for it to issue a corresponding connect(...) call. And that's one of the issues with LKL [4.2]. That is why we avoid linking those files, and rather pass majority of the control to DCE, but without making a single change in code, because DCE also has to work when simulation scripts use the default ns-3 network stack. We thus rewrite certain functions. Below a few function and concepts have been discussed : schedule() : schedule_timwout() : lib_task_wait() : lib_task_yield() : add_timer() : mod_timer() : init_timer_key() : do_softirq() : open_softirq() : queue_work_on() :

3.4.5 Jiffies Timer

The Linux kernel management of time is based on the use of the global jiffies variable which contains a 32bit integer that reports the number of ticks elapsed since the boot of the kernel. The duration of each tick depends on the way the kernel was configured but nowadays, it is usually configured to be one millisecond. The jiffies variable is normally increased whenever the kernel timer interrupt is triggered. This timer interrupt is also responsible for executing any expired kernel timers every ten millisecond, etc. [22, 19] both deal with this by executing periodically a per-node event that increases the jiffies variable and relies on the Linux kernel code to deal with its internal timers. This approach suffers sadly from one major drawback: even if there are no timers scheduled to expire for the next 10 or 80 milliseconds, the simulation will keep running and executing events, just for the sake of incrementally increasing the value of this jiffies variable. Rather than waste time to do this, we instead configure the Linux kernel to not use periodic ticks with CONFIG NOHZ and then replace entirely the kernel timer facility to schedule simulation events for each kernel timer instead of keeping track of the kernel events in a data structure separate from the main simulation event list. The resulting kernel network stack thus runs in tickless mode and does not waste time scheduling unnecessary events.

3.4.6 Kernel Initialisation

The Kernel initialisers are divided into base init functions and roughly 8 levels of initcalls and exports a start and end pointer of the list. Init functions are the ones directly called in the start_kernel(...) or by certain device drivers, whereas initcalls are the ones which are stored in a special .initcall.init section of the final linked library. These are then copied to their respective positions through the linker.lds script. In out lib_init function I call specific init functions, namely : vfs_caches_init_early rcu_init devices_init buses_init radix_tree_init timekeeping_init cred_init uts_ns_init vfs_caches_init seq_file_init proc_root_init I also changed the initcall invocation loop of lib_init to this :

```
 initcall_entry_t *call;
 extern initcall_entry_t __initcall_start
  [], __initcall_end[];

 call = __initcall_start;
 do {
     initcall_from_entry(call)();
     call++;
 } while (call < __initcall_end);

Before we call all the init functions, I
  also added these two lines of code :

 files_cachep = kmem_cache_create("
  files_cache",
          sizeof(struct files_struct), 0,
          SLAB_HWCACHE_ALIGN|SLAB_PANIC|
  SLAB_ACCOUNT,
          NULL);

 fs_cachep = kmem_cache_create("fs_cache"
  ,
          sizeof(struct fs_struct), 0,
          SLAB_HWCACHE_ALIGN|SLAB_PANIC|
  SLAB_ACCOUNT,
          NULL);
```

These caches are required while setting up the fs_struct and files_struct for each new task in it's corresponding task_struct. Please see [5.9].

3.4.7 Security LSM Module

The security module of the Linux kernel depends on a special configuration called the CONFIG_LSM. The LSM module requires a few architecture defined variables. Surprisingly, you cannot find them in the source code(.c or .h files) anywhere, rather they are placed in a specific section of the library/binary called .init.data section, through ASM code. net-next-nuse makes use of a linker script which can help us position variables inside the library under specific sections. I had to put this code in the linker.lds to initialise the start and end of the LSM table.

```
  . = ALIGN(CONSTANT (MAXPAGESIZE));
  .init.data : AT(ADDR(.init.data)) {
     __start_lsm_info = .;
     KEEP(*(.lsm_info.init))
     __end_lsm_info = .;
     __start_early_lsm_info = .;
     KEEP(*(.early_lsm_info.init))
     __end_early_lsm_info = .;
  }
```

vfs_kern_mount(...) validates and parses params using security_fs_context_parse_param(...), which checks if the requested operation was blocked in the LSM tables. If this module is not enabled, it would return back ENOPARAM by default and fail.

3.4.8 Kernel Code - net-next-nuse Alignment

I had to remove the panic_on_taint symbol, which got exported while linking kernel/panic.c(we need to have this file for acknowledging kernel runtime errors) and accessing it in kernel/sysctl.c caused errors because it couldn't find it back. Also, having this made no sense, as its value defaults to 0 and never changes during program execution. When the kernel gets initialized, it called mnt_init, which is responsible for setting up the kernel file system (kernfs), the sysfs, ramfs, etc. and then makes a call to init_mount_tree, which sets up the required namespace for the all the mounts. This function will then make a corresponding call to vfs_kern_mount, which will use the vfs setup to initialise the file systems we need. vfs_kern_mount will set up the file context and mount it. In this process it checks for the security module and the LSM entries, to check if the kernel was not previously configured to avoid declaration of the specified namespace. If all checks pass, it returns back a vfsmount pointer. We then use this pointer to set up a mount namespace which initialises the data structure mt_namespace, using alloc_mnt_ns . To initialize the mnt_namespace object, we require the current tasks nsproxy, which is the user namespace proxy member. Every task is required to define some initial maximum value for each namespace it might require during the execution of the task. Similarly, for allocating a mount namespace it increases the UCOUNT_MNT_NAMESPACES of the current task's nsproxy which has an upper bound of init_task 's value. Since, we are not doing a complete arch port (we don't need to), this value is not defined, and would through a SEGFAULT, when READ_ONCE(...) tries to make an atomic operation on it to access it and increase the value by 1 if doing so doesn't exceed the max limit, using the atomic_inc_below(..) macro. We thus manually set the value for it, during task initialisation in lib_task_start(...).

```
ucounts->ns->ucount_max[
    UCOUNT_MNT_NAMESPACES] = MNS_MAX;
```

But, we did set up the mount namespace, for the init task, but what for the other tasks ? We thus create a global variable def_mnt_ns and store a backup of the first mount namespace using an extern variable and then later on set it up in the lib_task_start(...)

```
struct nsproxy *ns;
ns->uts_ns = 0;
ns->ipc_ns = 0;
ns->mnt_ns = def_mnt_ns;
ns->pid_ns_for_children = 0;
ns->net_ns = &init_net; // global struct *
    net
task->kernel_task.nsproxy = ns;
```

This seems good right ? But what is def_root ? Ok, I had to hack this again XD. Remember init_mount_tree, we discussed it in the pointer above. What does it do ? It sets up a vfs mount. Okay, does that mean a path somewhere in the kernel ? Yes it should have. I also found another line of code there : init_task.nsproxy->mnt_ns = ns;

Okay, does this mean that the first task has the mount namespace generated while we mount and allocate the mount tree and acquire an mnt_namespace. What else can we observe here ? How does the Linux Kernel create tasks then ? Do you remember doing these when you create processes on your OS ? No, right ? Which means, do we reuse some of the previous data from the init_task and replicate them in new tasks. Yes, we do. Whenever we create a kernel_thread(..) it calls copy_process(...) which copies some of the required parameters from the host task, depending upon the flags you pass it. For example, If you had set the CLONE_FS flag, it would copy the fs_struct from the init_task to the newly made task, by calling copy_fs(..) in kernel/fork.c. Which means, if we never have diverse task member requirements we can just reuse the previous task, implying the first task ? Right ! So, we can then just declare a global def_root and then by declaring an extern variable for it, and keep reusing !

The netif_napi_add tests for the NAPI_STATE_LISTED bit to be enabled in the state of the napi_struct passed on to it from cgroup init calls. We never set this flag directly, so this check was removed from the kernel code. sock_init is one of the many initcalls which are called when sim_init is invoked by DCE. First of all before we proceed with anything, it is necessary that we have initialised the proc file system completely, so that the sysctl interface could be initialised by the net_sysctl_init(...). We then register the sockfs filesystem using register_filesystem(...) and go for a kern_mount(...) which invokes the VFS kern mount function and as discussed above, it requires the mnt_init(...) to correctly setup the mount environment. Initially, in net-next-nuse-4.4.0, mnt_init(...) was made a blank function, and we have reasons to support it. In older kernel releases, sockfs file system init did not depend on a file system context, and required a mount function (sockfs_mount) which would directly mount the file system onto the kernel, but in commit [6.1], the socket file system mounting process was handed over to the internal VFS mounting mechanism, thus breaking our setup. So, now we need to have the mnt_init(...) and put in patches and glue codes wherever needed. ToDo include/arm changes to files, for example, atomic, barrier, ptrace, user, user32/64. . . . ToDo Rump kernel header file double include, adding flag to disable linux include

3.4.9 Native Kernel NetDevices

Certain components of ns-3 require to setup custom NetDevices which require a custom mtu and other flags such as should it be a multicast device, or a P2P device, or should it broadcast etc. These flags could be put togethers and we can allocate a netdevice for each such requirement using the alloc_netdev(...), passing all the configurations needed for mtu,address length, and destructors, along with a register_netdev(...) call. This would return back a forward declared struct SimDevice which is (probably) casted to a NetDevice object inside ns-3.

- 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.