Survey of Software Interception Techniques in the Context of Cloud Applications

Joseph Cabrita
Univ Rennes
F-35000, France

Clément Courageux-Sudan
Univ Rennes
F-35000, France

Jean-Michel Gorius
Univ Rennes
F-35000, France

Quentin Le Dilavrec
Univ Rennes
F-35000, France

***Abstract*—Complex computational systems need to be carefully studied so as to understand their behavior. Emulation is one way of achieving this goal. In this paper, we present our attempt at intercepting a large-scale cloud application, namely Ceph, using SimGrid as a simulation backend. We focus our efforts on intercepting execution events at a high level, by directly altering the application's source code. This enables us to refine existing interception techniques and apply them in the context of production-ready cloud applications.**

I. Introduction

Cloud infrastructures have become ubiquitous, as they are used to provide numerous server-based services. From websites to scientific computing, clouds abstract the user away from the underlying hardware and software peculiarities. A typical cloud infrastructure is made out of a large number of service layers. Those layers interact in a complex manner. The complexity underlying the operations of such systems is hard to grasp precisely by using only mathematical models. Moreover, the development of those models is a time consuming and error-prone task.

Studying complex cloud applications is tedious. There is a need for adapted tools designed to abstract the user away from their underlying complexity. To overcome the difficulties implied by the intricate behavior of cloud-based applications, one can resort to software interception.

Software interception techniques enable the user to capture runtime events and react to them according to a given set of rules. There are different possible levels of interception, each with its own limitations and opportunities. The more fine-grained interception methods capture each and every event and notify the user, while more coarse-grained techniques capture global behaviors. In the first case, the user can react to very low-level events, but when faced with a complex application, the event-processing infrastructure can quickly become flooded. In the latter case, coarse-grained interception enables the user to work at a much higher level, and thus closer to the original programmer's intent.

In this paper, we present the methodology we used in order to simulate the execution of a large-scale cloud application by using source-level interception. We emphasize the difficulties implied by such an approach and discuss some possible improvements. The detailed context of our study, as well as the exact problem specification, are exposed in section II. Section III gives a broad overview of existing software interception techniques usable for cloud application study. It follows a low-to-high-level approach. Finally, section IV and section V present our attempt at intercepting a large-scale cloud application and discuss the results we obtained.

II. Context

This section introduces methods which can be used to study the behavior of complex cloud-based applications (section II-A). It focuses especially on software execution environment emulation (section II-B). Section II-C explicitly specifies the problem covered by our work.

A. Behavioral study of cloud applications

The study of the behavior and evolution of complex distributed and layered cloud systems can follow three main approaches [1]: real platform execution, simulation and emulation.

In the first case, the application is executed on an existing cloud infrastructure and its behavior is studied under real execution conditions. This approach needs a large computing infrastructure to execute even the simplest of tests. Existing production platforms such as IGridA [2] or IDRIS suffer from numerous drawbacks. Experiments led on those infrastructures are difficult to reproduce, as they are subject to external noise caused by applications running concurrently on the same platform. A thorough understanding of an application's execution can sometimes be hard even on specialized experimental platforms such as GRID'5000 [3].

The second possible approach to cloud-based application study, namely simulation, makes use of an application model plugged into a simulator. In the end, the simulator executes the application's model instead of the real application. A simulator like SimGrid [4] is capable of handling models involving hundreds of computation nodes. Simulation makes experiments highly reproducible but its major drawback is the inability to capture the entire and exact behavior of an application.

The last possible technique which can be used to study a cloud-based application's behavior is emulation. The emulated application is executed in a virtualized environment provided by an *emulator*. The emulator intercepts particular application-emitted messages and/or signals at a given abstraction level. Emulation aims at overcoming the limitations of simulation and real execution. More precisely, the code of the application

is actually run on a host machine but, thanks to the virtualization process, the application is tricked into believing it is executing on a different host. Emulation makes it possible to run applications designed to be run on multiple hosts at once on a single machine. This eases the observation and analysis of complex programs, as experiments can be run on a personal computer.

B. Software emulation

In the following, we consider cloud-based application study using the standpoint of emulation. This approach easily overcomes the difficulties implied by heterogeneous execution infrastructures. Such infrastructures are similar to those typically found in cloud software-and hardware stacks. Emulation abstracts away the need for a real distributed infrastructure.

One can use an on-line emulator to virtualize a cloud-based application's execution environment and study its behavior and its evolution. An on-line emulator uses a variety of *hooks* inserted around the application's code to capture numerous events happening during execution. These hooks can be inserted at different levels in the software stack and can take numerous forms. The interception methods discussed in section III can be used to build up such hooks.

C. Problem specification

Our work aims at presenting a proof of concept showing that it is possible to apply software interception techniques to emulate cloud-based applications. We focus our efforts on emulating Ceph.

Ceph is a widely used cloud-based application providing storage services which are able to handle very large amounts of data [5]. It is an open source project with an extensive documentation.

To emulate Ceph, we choose to use SimGrid [4], and its remote command execution API, *Remote SimGrid* (RSG). SimGrid is a scientific instrument used to study the behavior of large-scale distributed systems such as compute-grids, clouds or HPC systems. It can be used to evaluate heuristics, prototype applications or even assess legacy MPI (*Message Passing Interface*) applications.

Due to the lack of a reliable interception interface, cloud-based application study using SimGrid is currently limited to simulation. We aim at overcoming this limitation by introducing a new way to study cloud-based applications using RSG. We study numerous possible software interception techniques which could allow us to pass runtime events to SimGrid using RSG. Those techniques exhibit various levels of granularity and can be implemented at different layers in the software and/or hardware stack.

III. Software interception techniques

Software interception is a set of means by which an external program can capture events happening during the execution of another application. This kind of capture can be used to monitor or profile applications. It can also be used to alter the execution environment of an executing application. The latter scenario is the one we discuss in the rest of this paper. We particularly focus on software interception of network communications, which are essential to every cloud-based application. Each of the following sections discusses a particular level of interception, i.e. a layer in the software or hardware stack at which events can be captured, from a low-level to a high-level standpoint.

A. Network-level interception

The lowest-level approach to intercept network communication events is the one located directly at the network hardware layer. Using a hardware-level virtual machine, one can trick a running application into using a filtered virtual Ethernet port for all of its network communication operations.

This approach is one of many taken by ns-3, a discrete-event network simulator for Internet systems [6]. This widely used simulator makes use of a `qemu` virtual machine (VM) to execute an application. This VM sets up a virtual Ethernet port through which every network communication issued by the executed application is established. This port is in turn connected to a physical Ethernet port through a network bridge. This configuration allows an external network packet analysis tool such as *Wireshark* [7] to intercept the communication packets issued by the application under study. It can then notify other systems of captured communication events and even get the exact content of each message. This interception method is completely transparent to the underlying VM and application, as every behavioral observation is made entirely outside of the execution environment.

A major drawback to this interception technique is its extremely fine-grained event capture. By intercepting every network packet emitted by the executed application through the VM's virtual Ethernet port, one has a very local view of this application's behavior and actions. In fact, even initiating a simple network communication can take several dozens of packets. One cannot focus her attention at such a fine level of detail. A potential solution to this difficulty is to go up one layer of abstraction and intercept events at the driver level.

B. Driver-level interception

Located just above the raw network layer, drivers are the means used by operating systems to communicate with the network hardware, typically a network card on the host machine. It is possible to intercept network communication events on the driver level by using the hardware abstraction layer provided by an operating system (OS) or a *type 1 hypervisor*.

An hypervisor or *virtual machine monitor* is a program designed to create, run and manage a set of virtual machines. There are two types of hypervisors, namely *type 1* and *type 2* hypervisors. Type 1 hypervisors run directly on the hardware of the host machine and act as a kind of operating system. They manage system calls and interact directly with the hardware using drivers. Type 2 hypervisors, or hosted hypervisors, run on a host OS. They forward system calls to the underlying OS, which in turn is in charge of interacting with the hardware.

By writing a custom network driver, one can use it to notify an external program of every network event. This in turn can be used to emulate an execution environment and propagate the effects of the intercepted network operations on it. However, this approach is hard to implement in practice, as one has to write a low-level network driver in order to achieve successful interception. This driver-based method also has a fine interception granularity, as it notifies the emulator of each network transaction initiated with the network card. As is the case for network-level interception, this level of detail is too high in comparison of the execution time and operational complexity of cloud systems. In the next section, we go further up in the interception layer hierarchy and focus on kernel-level interception.

C. Kernel-level interception

The kernel is a program located at the heart of an OS. It is the one responsible for managing every possible interactions between programs and/or hardware. Intercepting events at the kernel level can be achieved by dynamically interacting with the kernel. The main approach used to achieve such behavior is the use of a *libOS*. A libOS is an operating system kernel (e.g. a given Linux kernel) packaged as a dynamic library. This dynamic library can be used by applications to interact directly with kernel-level functionalities.

A first approach used to intercept network communication events at the kernel level using a libOS-based solution is presented by Zhang et al. [8] in the form of the KylinX project. Their goal is to sandbox an application to avoid malicious behavior on cloud infrastructures. They use virtual machines managed in a process-like fashion by the server's hypervisor to control the behavior of each and every running application. This method introduces the concept of *process-like VMs* (pVMs) and treats the hypervisor as a low-level operating system. Those pVMs run a *unikernel*, which in turn is responsible for executing a given application. Unikernels are operating systems in which each running program shares the same address space as the other programs running at the same time. Each process-like VM loads the unikernel as a shared library, using the libOS pattern, and intercepts system calls at the kernel level. The intercepted calls are then filtered and transmitted to the underlying hypervisor if needed.

Another example of the use of a libOS to achieve interception of an application's events is DCE [9]. DCE, which stands for *Direct Code Execution*, aims at executing unmodified applications within the ns-3 simulator [6]. To achieve that goal, the network stack implementation of the Linux kernel is copied and modified at the lowest level to interact with the ns-3 simulator. This code is embedded into DCE as a libOS. This library can then be used by the simulated application to communicate with the simulator instead of the real network. The simulated application doesn't have to be modified because it uses the classical implementations of sockets found in the kernel. This makes it possible for an application to communicate transparently with the ns-3 simulator.

The level of detail offered by kernel-level interception offers more flexibility in the context of application emulation, as it gives the emulator a bit more semantic information about the events captured during execution. For example, the interception of the `open` system call informs the emulator that the application under study wishes to open a file descriptor. However, this information is often not usable, as system calls are ubiquitous when calling higher-level functions. It can be useful to go up a level in the software interception hierarchy and intercept events at the binary-level to ease the interpretation of captured events.

D. Binary-level interception

Executable binaries offer a higher level of abstraction when compared to a kernel. There exist numerous ways of intercepting events at this level, including libOSes, `LD_PRELOAD`, the `ptrace` system call or a dynamic call capture method implemented in the DCE [9] project. In the next paragraphs, we describe some projects which make use of those techniques to intercept execution events.

1) Dynamic library pre-loading: One of the main techniques used to intercept events at the binary level is library call interception. On a Linux host, this can be achieved by using the `LD_PRELOAD` environment variable. `LD_PRELOAD` is a variable that can be used to specify shared libraries that will be loaded with a higher priority at runtime. When launching an application, the dynamic linker loads the libraries specified by `LD_PRELOAD` before any other. As loaded symbols are remembered by order of first load, this technique enables a user to effectively replace functions by others dynamically. This feature can easily be used to perform interception by overwriting a function used by an application.

As an example, we can consider the POSIX function `getimeofday` which returns the system time. One could make his own implementation of the function, which would return a simulated time instead of the real time, and embed it into a shared library. `LD_PRELOAD` can then be set to use this new library before running a program that uses this function. It has the effect of redirecting every call to `gettimeofday` when running the program to the new implementation.

One project using this library pre-loading trick is *MicroGrid* [10]. Microgrid is a grid emulator used to analyze the behaviour of real grid applications inside virtual environments. To achieve that goal, Microgrid makes use of `LD_PRELOAD` to intercept some function calls through a custom made library, namely `libmgrid`. Two different resources are intercepted with this library. In the first place, calls to network services (e.g. `send` or `recv`) are redirected to a network simulator. Then, functions related to the system time are intercepted to return the time of the virtual environment.

2) Process-tracing system call: Dynamic library pre-loading is not the only way one can intercept events during a process' execution. In fact, the Linux kernel provides a system call named `ptrace`, which allows a program to interact with another by using *signals*. Signals are similar to commands emitted and received by and from programs. They allow one

program to change the behavior of another one by using a very simple communication scheme. The `ptrace` system call makes use of such signals to manipulate the execution behavior of a program called the *tracee* based on given events, for example the call of a given library function or system function. Using `ptrace`, one can intercept this call and, for instance, insert code inside the currently executed code section before resuming normal execution. The inserted code will then eventually be executed as if it were part of the original application.

The `ptrace` system call can sometimes be difficult to deal with. To ease the filtering and modification of system calls issued by a running application, one can use a `ptrace` extension called `seccomp-bpf` (`seccomp` Berkeley Packet Filtering). This extension is based upon a Linux kernel feature known as `seccomp`, which performs process isolation and sandboxing.

The *MBOX* [11] sandboxing project makes use of this solution. MBOX allows an application to run inside a sandboxed file system. The system calls that interact with files are intercepted and modified to use the sandboxed environment instead of the real file system of the host machine. Using `seccomp-bpf` combined with `ptrace` greatly reduces the overhead introduced by the interception with `ptrace` only, by intercepting the calls of interest and ignoring the others.

3) DCE library-level interception: The DCE project discussed in section III-C also performs binary-level interception. It implements a custom standard C library which is used by the emulated applications instead of the default one. As for Microgrid, these new implementations are mainly used to propagate the effects of function calls on the simulated environment instead of the real machine.

The process of rewriting libraries has one major drawback. It is very hard to detect and replace all of the functions that are using some kind of resources inside a program. In addition to the need of finding all the functions to replace, the process of rewriting them is very time consuming. As an example, the DCE project has rewritten over 400 methods of the standard C library, but might need to cover more in the future to be able to run more applications.

Even though binary-level interception gives great control over functions called by the application under study, it could be better to consider the application's source code directly. This, of course, assumes the latter is available to the user willing to emulate the application's behavior.

E. Source-level interception

One can use what we call *source-level interception* to intercept events happening during program execution, given that the source code of the application to emulate is available and compilable. Source-level interception consists of a set of source code modifications targeting key locations of the code. Those key locations could for example be the functions responsible for establishing a network communication or for initiating a data exchange transaction between two processes of the same program. This kind of interception heavily relies on user annotations in the source code. In fact, numerous source-level interception toolkits make use of dedicated compiler *pragmas*, i.e. indications to the compiler on how to treat a given section of the code. Even though inserting those pragmas can be tedious, this approach to runtime event interception allows the user to chose the exact level of detail at which she wants to observe the behavior of the modified application. A common example of source-level interception in the world of HPC simulation is its use to modify certain parts of the source code of MPI applications. These applications use the MPI (*Message Passing Interface*) API to communicate between numerous computation nodes on which they are executing.

One of the major MPI application simulation toolkits is SST Macro [12]. SST Macro is an MPI application simulator which makes heavy use of preprocessor directives (i.e. *pragmas*) to execute an MPI application on a single host while tricking it into believing it is executing on multiple nodes concurrently.

An alternative to SST Macro is SMPI [13]. SMPI is an MPI application simulation toolkit built on top of SimGrid [4]. It makes use of source-level interception in the form of source code annotation using macros and pragmas to intercept MPI calls. SMPI provides a custom implementation of a large part of the most common MPI functions. Those functions are statically replaced in the simulated application's source code by a pre-compilation script.

Source code modification using macros and/or pragmas can be a good way to intercept events happening during the execution of an application. It allows the user to precisely select the level of granularity at which the observations of the runtime behavior are made. This is the approach we choose to follow and that will be exposed in the following section.

IV. Contribution

The main goal of our work was to integrate Remote SimGrid (RSG) with Ceph through source-level interception. Section IV-A presents the method we used to achieve this integration, whereas section IV-B gives an overview of some key simulation milestones achieved thanks to our approach.

A. Methodology

The Ceph codebase contains a large quantity of code, which makes it practically impossible to completely understand each and every detail of the Ceph infrastructure. By following a step-by-step approach, we reduced the amount of application-specific knowledge required to successfully intercept Ceph. The following sections give more insight into the steps we took.

1) Linking to RSG: Integrating an application with Remote SimGrid requires us to link it to the RSG dynamic library, `librsg.so`. Once this linking is done, the application becomes a SimGrid client. When launching a simulation using SimGrid, the user has to specify the number and the name of each client that will interact during the simulation process. The simulator then starts the simulation and waits for each client to notify its startup. This constrains the execution of cloud applications like Ceph, which are often split into multiple

interacting binaries. In the case of Ceph, some binaries are launched to prepare the cluster by creating file systems, registering metadata and ensuring everything is setup before executing the main Ceph infrastructure. A running instance of Ceph is then divided into multiple components, mainly cluster monitors for system metrics monitoring, metadata storage handlers for filesystem metadata manipulation and storage, and object storage handlers, which interact with the filesystem to store files and serialized objects. At the time of this writing, there is no simple way to dynamically add a client to a SimGrid simulation by using RSG, and the total number of clients has to be known upfront. This limitation has led us to consider only instances of Ceph monitors, bypassing the complexity induced by the large amount of interacting binaries inside Ceph.

2) Finding the weak point: Cloud-based applications make heavy use of network communications and data exchange protocols. Ceph has a well-defined messaging API that allows all components of the infrastructure to communicate using a unified interface. Integrating RSG into this part of the application requires only minor source code moifications, mainly to accomodate for the overloaded implementation of the network sending and receiving routines. For example, every call to the `send` system call has to be replaced with a call to the corresponding RSG implementation, `rsg::send`.

The Ceph communication protocol is mainly defined by using a common `Messenger` class. By subclassing it and providing our own version of a messenger, we were able to plug it directly into Ceph with very little additional code modifications. However, it is worth noting that overlooking a call to `send` or `recv` can lead to a deadlock in the simulation, as a call to `rsg::send` for example will not be able to transfer data to the standard `recv` system call.

3) Adapting the interception to SimGrid: SimGrid is able to precisely model network communications only if the exchanged packets exceed a given size. By working at a high level in the source code, e.g. the `Messenger` level, we are able to aggregate successive small messages and transmit them all at once. Additionnally, there is a more direct mapping between the concepts used by the Ceph communication protocol and the standard communication protocol implemented by SimGrid. The latter is built upon a simple model based on *actors* and *mailboxes*. Each time a message has to be sent over the network by the Ceph `Messenger` class, it can instead be posted to a mailbox by an actor, associated with the given `Messenger`. On the other side of the communication channel, another actor reads from the same mailbox and receives the message.

Our custom `Messenger` implements the same interface as the standard Ceph messengers. It wraps the calls to RSG (and therefore SimGrid) and makes their usage transparent to the rest of the application.

4) Intercepting other parts of the application: SimGrid exposes many of the functionalities provided by an operating system. In particular, SimGrid uses its own process management system and its own scheduler. For our use case, this scheduler transforms the execution flow of the simulated program by sequentializing multi-threaded parts of the application. For this transformation to work, we have to intercept not only the network communications, but also all the process synchronization and thread management calls. Performing such interception can be fairly trivial in some cases, but it can require much more work in some places of the code base.

As simulated and real-world threads cannot interact, the simulation can proceed only when all the thread management calls are intercepted. The Ceph code base contains various thread implementations scattered throughout multiple source files and even inside library dependencies. Unlike network communications, multi-threading is not implemented in a common place and it therefore requires more search and replace work. In addition to that, as for many production-ready cloud applications, Ceph relies heavily on libraries and other forms of code dependencies. All those dependencies have also to be intercepted in order for the simulation to work properly. Missing a call to `std::thread::join` in one library's code can lead to a simulation deadlock.

In the end, we intercept a large part of the threading calls inside of a Ceph monitor.

B. Results

Instrumenting and intercepting the entire Ceph infrastructure requires a lot of work, notably because of the large number of obstacles and difficulties mentionned in the previous section. Because of that, we chose to intercept only a subset of the application, by focusing only on Ceph monitors. Monitors are essential components of a Ceph cluster that perform filesystem and system metrics monitoring and maintaining a map of the current state of the cluster.

In the first place, we focused on intercepting network communications inside Ceph by implementing our own `Messenger` variant and plugging it into the Ceph messaging infrastructure. This messenger redirects messaging function calls to SimGrid's messaging protocol API and lets the simulator handle all communications between the interacting entities of a running Ceph cluster.

Part of the process management system has also been intercepted. This part of the interception procedure is very tedious, especially because of the multiple source code locations where hooks have to be inserted. Some Ceph dependencies have also been intercepted and adapted in order to integrate with RSG.

Ceph monitors can now be executed on top of a SimGrid simulation and they can initiate communications by using the communication protocol exposed by the simulator. Moreover, some parts of Ceph's unit testing suite have been successfully instrumented and can be run on top of SimGrid.

V. Discussion

In the end, neither the low-level interception approach nor the high-level source code interception techniques are entirely satisfying. Low-level interception keeps the user too far away from the program semantics and leads to an increased amount

of intercepted calls to simulate. On the other hand, source-level interception allows the user to better get the grasp of what exactly she is intercepting. However, during the simulation of a program, a number of different domains of interception overlap and interfere with one another. For example, the simulator has to be notified whenever there is a network call, a system call that could change the disk state and it should even be the one handling synchronization in multi-threaded applications. It quickly becomes hard to keep up with all the possible ways to express the same idea in a given program.

Let us consider the case of thread synchronization inside of the Ceph infrastructure as an example. Ceph is a large project with many dependencies and each one has its own way of manipulating threads. Some use custom wrappers, others directly use the thread implementation provided by the C++ standard library and others sometimes even refer directly to the underlying OS-dependent implementation (e.g. the pthread library under POSIX-compliant systems). For high-level interception to be effective in such a context, the user wanting to simulate Ceph has to track down each individual interface which manipulates threads and adapt it to insert interception hooks, without breaking the overall publicly exposed API. For small projects with little to no dependencies, such code adaption is merely a matter of finding the right spot in the code and changing a few lines of code. But for real-world cloud applications like Ceph, there are thousands of such locations scattered throughout the entire code base.

A possible solution to this problem could reside in the use of a mixed approach to software interception. Instead of focusing only on low-level details or on high-level constructs to intercept, one could follow an incremental workflow. Starting from a basic low-level interception framework which captures and redirects all system calls to a simulator without considering program semantics or behavioral correctness, the user can then build higher-level source interception hooks and insert them into the source code. She will not have to worry about forgetting some part of, for example, the thread synchronization code, as the low-level interception will catch it if it goes through the high-level interception without being noticed.

Figure 1 gives an illustration of the multiple interception layers that exists within the same application, like a Ceph monitor. Let us assume that the user implemented her own version of a Ceph `Messenger`, `RsgMessenger`, with built-in hooks to the Remote SimGrid API. Each time a call to `RsgMessenger::create` is issued, the RSG server will be notified and it will transmit the relevant information to the simulation platform. If we further assume that the user has overseen a call to a system function (e.g. a `recv` system call) in some other place of the code base, then this call will be intercepted and redirected to the simulator, but this time through a low-level interception framework. The simulator will be notified that a particular system call occured, but it will have to resort to a default behavior until the user also adds interception code at the call's location, if she wants to give more information to the simulation backend.

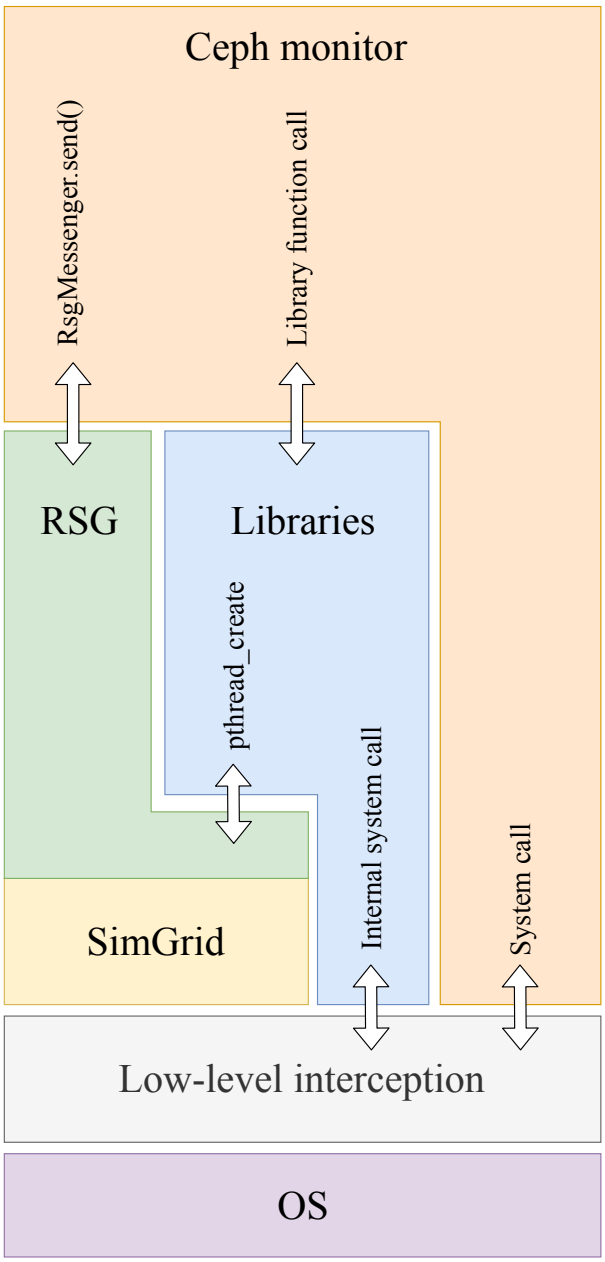

Figure 1. Multiple interception layers.

By using an hybrid interception approach, we can ensure that the user can work iteratively, adding a small element to the interception code and testing it before moving on to the next one. All the calls intercepted by the low-level framework would follow a default behavior when encountered by the simulator and, as such, might not follow the original program semantics. In the end, the user will have to go over each of these calls to check that the behavior used by the simulator is the right one, but this is essentially a tradeoff between usability and program correctness preservation. As more and more user-crafted hooks are added to the code, the behavior of the simulated program should converge to its real execution behavior.

Using the work presented in this paper, it should be possible to build such an hybrid interception framework. Some promising results have been obtained when intercepting low-level system calls and redirecting them to Simgrid by using CWRAP [14]. CWRAP was primarily designed as a client-server testing toolset for unpriviledged applications and was part of the Samba "torture" testing suite. It makes use of the `LD_PRELOAD` strategy described in section III-D1 and is able to intercept all system calls made by a given application. By adapting CWRAP to integrate with Simgrid, it could serve as the low-level interception framework discussed earlier in this section. With the aid of such a tool, one could then follow the same approach as we did and integrate RSG into a cloud application like Ceph but without having to go through the software engineering phase required to ensure that each and every call to the system gets notified to the simulator.

VI. Conclusion and future work

In this paper, we established a proof of concept, showing that it is possible to integrate the SimGrid simulation backend with a cloud application by using the Remote SimGrid API. We used source level interception techniques to intercept the application's network communications at key locations and redirect the intercepted calls to SimGrid. Moreover, we partially intercepted the process synchronization and thread management code of the application and part of its dependencies.

We emphasized the importance of an incremental approach to cloud application emulation and we gave some insights about possible improvements that could be made to our methodology. Those improvements use an hybrid interception approach, combining binary-level interception using dynamic library pre-loading and higher level source interception. This combined point of view enables a user to focus on particular locations of the source code and to let a low-level framework catch and redirect all other system calls to the simulator.

Emulating cloud applications can be of great use to understand and study the behavior of those large software infrastructures. Large-scale cloud applications can be run on a single resource-limited experimental host (e.g. a personal laptop).

In the future, we plan to continue the work on a low-level interception framework based on CWRAP. This will allow us to bootstrap a more effective high level interception methodology. Moreover, Ceph is only a single study case. By further improving the automation of source level interception, it will be possible to instrument nearly any cloud application and run it on the SimGrid simulation backend.

Acknowledgements

We would like to thank Martin Quinson[1]and Millian Poquet[1] for their guidance and their precious advice on our work.

Source code

All the source code is available on GitHub under the following links:

- Ceph (instrumented version): https://github.com/Kayjukh/ceph
- rocksdb (a modified Ceph dependency): https://github.com/quentinLeDilavrec/rocksdb
- Remote SimGrid (slightly adapted): https://github.com/klementc/remote-simgrid

References

[1] J. Gustedt, E. Jeannot, and M. Quinson, "Experimental Validation in Large-Scale Systems: a Survey of Methodologies," *Parallel Processing Letters*, p. 16, 2009. [Online]. Available: https://hal.inria.fr/inria-00364180

[2] I. IRISA. The igrida computing grid. [Online]. Available: http://igrida.gforge.inria.fr

[3] F. Cappello, F. Desprez, M. Dayde, E. Jeannot, Y. Jégou, S. Lanteri, N. Melab, R. Namyst, P. Primet, O. Richard, E. Caron, J. Leduc, and G. Mornet, "Grid'5000: a large scale, reconfigurable, controlable and monitorable Grid platform," in *6th IEEE/ACM International Workshop on Grid Computing - GRID 2005*, Seattle, USA, United States, Nov. 2005, grid 2005 held in conjunction with SC'05, the International Conference for High Performance Computing, Networking and Storage. [Online]. Available: https://hal.inria.fr/inria-00000284

[4] H. Casanova, A. Giersch, A. Legrand, M. Quinson, and F. Suter, "Versatile, Scalable, and Accurate Simulation of Distributed Applications and Platforms," *Journal of Parallel and Distributed Computing*, vol. 74, no. 10, pp. 2899–2917, Jun. 2014. [Online]. Available: https://hal.inria.fr/hal-01017319

[5] S. A. Weil, S. A. Brandt, E. L. Miller, D. D. E. Long, and C. Maltzahn, "Ceph: A scalable, high-performance distributed file system," in *Proceedings of the 7th Symposium on Operating Systems Design and Implementation*, ser. OSDI '06. Berkeley, CA, USA: USENIX Association, 2006, pp. 307–320. [Online]. Available: http://dl.acm.org/citation.cfm?id=1298455.1298485

[6] T. R. Henderson, M. Lacage, G. F. Riley, C. Dowell, and J. Kopena, "Network simulations with the ns-3 simulator," *SIGCOMM demonstration*, vol. 14, no. 14, p. 527, 2008.

[7] G. Combs *et al.*, "Wireshark-network protocol analyzer," *Version 0.99*, vol. 5, 2008.

[8] Y. Zhang, J. Crowcroft, D. Li, C. Zhang, H. Li, Y. Wang, K. Yu, Y. Xiong, and G. Chen, "Kylinx: A dynamic library operating system for simplified and efficient cloud virtualization," in *2018 USENIX Annual Technical Conference (USENIX ATC 18)*. Boston, MA: USENIX Association, 2018, pp. 173–186. [Online]. Available: https://www.usenix.org/conference/atc18/presentation/zhang-yiming

[9] H. Tazaki, F. Urbani, E. Mancini, M. Lacage, D. Camara, T. Turletti, and W. Dabbous, "Direct Code Execution: Revisiting Library OS Architecture for Reproducible Network Experiments," in *The 9th International Conference on emerging Networking EXperiments and Technologies (CoNEXT)*, Santa Barbara, United States, Dec. 2013. [Online]. Available: https://hal.inria.fr/hal-00880870

[10] H. J. Song, X. Liu, D. Jakobsen, R. Bhagwan, X. Zhang, K. Taura, and A. Chien, "The microgrid: A scientific tool for modeling computational grids," *Sci. Program.*, vol. 8, no. 3, pp. 127–141, Aug. 2000. [Online]. Available: http://dx.doi.org/10.1155/2000/481921

[11] T. Kim and N. Zeldovich, "Practical and effective sandboxing for non-root users," in *Presented as part of the 2013 USENIX Annual Technical Conference (USENIX ATC 13)*. San Jose, CA: USENIX, 2013, pp. 139–144. [Online]. Available: https://www.usenix.org/conference/atc13/technical-sessions/presentation/kim

[12] H. Adalsteinsson, S. Cranford, D. A. Evensky, J. P. Kenny, J. Mayo, A. Pinar, and C. L. Janssen, "A simulator for large-scale parallel computer architectures," *Int. J. Distrib. Syst. Technol.*, vol. 1, no. 2, pp. 57–73, Apr. 2010. [Online]. Available: http://dx.doi.org/10.4018/jdst.2010040104

[13] A. Degomme, A. Legrand, G. Markomanolis, M. Quinson, M. L. Stillwell, and F. Suter, "Simulating MPI applications: the SMPI approach," *IEEE Transactions on Parallel and Distributed Systems*, vol. 28, no. 8, p. 14, Aug. 2017. [Online]. Available: https://hal.inria.fr/hal-01415484

[1]Myriads team, IRISA, Univ Rennes, France

[14] The Samba Team. Cwrap - a toolset for client-server testing. [Online]. Available: https://cwrap.org/