TANSIV-Docker: a lightweight and accurate native distributed application simulator

HUGO DEPUYDT

ACM Reference Format:
Hugo Depuydt. 2024. TANSIV-Docker: a lightweight and accurate native distributed application simulator. 1, 1 (April 2024), 11 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

Distributed computing, the use of multiple computers to achieve a goal, has additional challenges compared to writing standard or parallel applications. In particular, synchronizing instances of the program running on different computers, without a reliable global clock, or gracefully handling some instances' failures, are difficult problems. A distributed system must at some point be tested to ensure that the implementation reaches the expected goals in terms of functionality and performance.

Distributed systems exist in many different forms, including standard programs using standard operating system network features, such as BSD sockets, but can also be implemented inside a kernel, or as a bare-metal program, as can be found in some embedded systems. They can be based on frameworks, including low-level ones designed to bypass the kernel's handling of the network, such as DPDK (Data Plane Development Kit).

Because of this diversity, it can be difficult to reliably test these applications at scale. Testing on real hardware is long, costly, and difficult to scale. Emulation through virtual limitation of capabilities make for simpler to deploy tests, but isn't generalizable to all hardware (and especially network) configurations. Most network simulators only allow running a model of the application, written using simulator's framework. Network simulators that allow running native applications exist, such as ns-3's Direct Code Execution (DCE) framework, but may not be accurate enough for specific applications, especially those combining CPU use with network timings. In fact, ns-3's DCE can, and has been used to test both native applications and in-kernel network algorithms.

The TANSIV project attempts to reconcile these issues, by running applications in virtual machines, through a network simulator, with high accuracy of network compared to processor speeds. This is therefore compatible with all previously-mentioned application types, including in-kernel systems or embedded programs, as long as the associated hardware can be accurately simulated/emulated in a virtual machine. Additionally, this allows for tests on a very broad spectrum of simulated hardware, including complex network topologies, while remaining accurate.

Our goal was to extend TANSIV's architecture to light virtualization, which reduces the types of application it can run, but uses less resources, is easier to set up, and, being a program running on the host operating system, should allow for easier inspection, manipulation or debugging of the program compared to it running inside a virtual machine (which would possibly require the use of a helper program inside the virtual machine's guest operating system). The main contributions of this work are: an architecture allowing an accurate simulation of network and CPU resources, as with TANSIV, and a prototype, to evaluate the feasibility of this idea using Linux Docker containers, and in particular

Author's address: Hugo Depuydt.

2024. Manuscript submitted to ACM

Linux's `cgroup.freeze` interface. We evaluate the prototype's result in a simple simulation, to determine whether the prototype allows for realistic results (or, similar enough to TANSIV's results).

The rest of this document is organized as follows. We present some background information in section 2. We overview related work in section 3. We present our architecture and prototype in section 4. We evaluate the prototype in section 5. Finally, we present our conclusion in section 6.

2 BACKGROUND

2.1 Virtualization

Virtualization allows running applications (often called *guests*) in a simulated or isolated environment, and is often used to run multiple identical instances of said applications on a single host. There are multiple types of virtualization, depending on how the applications run or how much is virtualized.

Full virtualization simulates an entire computer: it reads the instructions like a CPU would, runs its instructions according to their intended behavior, and simulates peripherals (like a hard-drive, network interface, or graphics card) along with input/output operations. This type of virtualization allows running almost any application, as long as its required hardware can be simulated as well. However, it requires storing all of the software stack in memory, including another kernel if the application is a standard userland program. It also adds some overhead as the emulator and the guest have to both conform to the shared standard of a real computer, which can be inefficient for tasks such as reading and writing files. For example, disk storage is provided through a virtual hard drive, where the guest will create a standard filesystem, as on a real hard drive. There are methods to reduce disk space used on the host by these hard drive images: avoid allocating unwritten disk space (zeroes) on the virtual hard drive's file, or run multiple virtual machines off of the same disk, and consider written sectors separately for each VM, but these aren't as efficient as working at the file level (if it was possible).

There are two possible implementations of this virtualization type: the machine can be simulated using software that will act as the virtual CPU, or the CPU itself can have virtualization features to run most of the instructions of the fully virtualized guest. Software emulation can run applications designed for other Instruction Set Architectures (ISAs), and allows complete control over the execution of the guest, but is a lot slower than hardware virtualization. QEMU is an example of software that can do software full virtualization. Hardware-accelerated virtualization on the other hand, has the host CPU running the code mostly unsupervised, with some instructions going back to the hypervisor (to emulate hard drive access for example). This makes hardware-accelerated virtualization a lot faster than software emulation, but can only run code for the host ISA. For x86 and amd64, VT-x (Intel) and AMD-V (AMD) instructions can be used for this, combined with hypervisors such as KVM on Linux, or HYPER-V on Windows, among others.

Paravirtualization involves making interfaces between the host and guest more efficient for virtualization. This can be done partially, for example with device drivers only. On Linux, QEMU/KVM has virtio drivers, that can be used notably for network access, disk access, or image display. So, where a fully virtualized guest would output SATA[1] commands to access the hard drive, with the host having to emulate an actual hard drive, both can communicate using a simpler format. There is also the possibility of running the whole guest in a paravirtualized way, by modifying it to replace hard to virtualize instructions such as I/O access with explicit communication with the hypervisor. Paravirtualization is implemented in the Xen hypervisor, as an example. Paravirtualization is less general than full virtualization however,

[1]A common bus interface for hard drives.

as it requires the guest to be designed for it: either with specific drivers as in the former case, or fully built for its instructions to run paravirtualized as in the latter.

Light virtualization makes applications run on the the host operating system. Instead of simulating a computer, it simply isolates the application, which can run as though it was the only one running on the host,[2] on its own dedicated filesystem. It can also be restricted to separate network interfaces, including virtual ones, among other features. This has some advantages of standard virtualization, including the possibility of running multiple identical instances of applications, but is also limited to running applications compatible with the host operating system. When running multiple applications, all of them share the same kernel: this means less memory used, and even less translation necessary than running a paravirtualized kernel. This makes it more efficient than standard virtualization in terms of memory and processor usage: applications have almost the same performance as any other application running on the host. On Linux, light virtualization can be accomplished using cgroups and namespaces (PID, network, ...).[3]

A common framework for light virtualization is containers. Containers, as are offered by Docker, among others, are small images of filesystems representing a minimal distribution or application. Containers can build on the filesystem of other containers. For example, the simgrid Docker container image will contain simgrid binaries and libraries, and include the rest of the filesystem from the debian container image. Other containers can be built from common images, with images only being stored once. When running a container, a filesystem is created from the image, and written files are added to a filesystem specific to the container. In other words, the filesystem isn't shared among containers of the same image, only non-modified files are: this allows running multiple containers from the same image and store different data on each of them.

2.2 Network simulation

As experiments on real hardware can be expensive, multiple methods for simulating networks have been developed to conduct experiments in a more accessible way. These methods need to consider, for given network topologies, the bandwidth and latency of each link, and the behavior of routers (such as congestion, etc.).

Network emulation sets virtual latency and bandwidth values between applications by modifying the behavior of actual network packets. This can be done by using traffic shaping features on network interfaces (physical or virtual). This is generally regarded as giving very accurate results. However, this requires more resources than network simulation, and is therefore less scalable. In addition to this, network emulation makes it difficult and inefficient to simulate complex topologies, as all intermediate links would need to be emulated.

Network simulation uses a simulator to predict the behavior of packets. This allows it to simulate arbitrary network topologies, without having to run the real packets through all the emulated links. Examples include ns-3, SimGrid, and OMNet++. They are often used with models of applications, programmed using the simulator's framework. However, some have support for native applications: ns-3 has the DCE (Direct Code Execution) framework, which supports userspace and kernelspace applications. SimGrid supports applications using the MPI (Message Passing Interface) standard, and simulates them at a higher level than ns-3's DCE, as it considers messages, and not individual network packets. However, simulating native applications is more difficult as packets need to be delivered on time, and simulators need to run in real-time. The accuracy can therefore be decreased for very low-latency networks, or high-bandwidth ones. Network simulation is more scalable than emulation in general, but there are multiple approaches that differ in scalability.

[2]Another way to see it is that the guest runs as if the kernel booted on its filesystem.

Packet-level network simulation simulates each individual packet's flow in the network. Packets going through a link will arrive after the link's latency, or delayed if the link's bandwidth is already saturated. Packets going through a router will be put in its queue and delivered as a real router would deliver them. As the full path of packets is simulated, packets going through the internet would be simulated by simulating each router the packet would have to go through. This approach is the most detailed, but also requires the most resources, and is therefore limited to simulating smaller networks. ns-3 and OMNet++ are packet-level network simulators.

Flow-level network simulation can be conceived of as similar to simulating fluids going through pipes. Equations determine the arrival time of data, given some state of the network, such as how "full" the network is at a given point, or the capacity of links. This approach uses less resources than packet-level network simulation, which allows it to simulate much bigger networks. However, it is less detailed and may give slightly less accurate results. SimGrid is an example of a flow-level network simulator.

2.3 TANSIV

TANSIV[1] combines full virtualization and network simulaton to simulate distributed applications. It makes all packets go through the SimGrid network simulator, but is also designed to handle "faster" networks than would be possible to simulate in real-time from SimGrid or ns-3's DCE alone, while maintaining a realistic execution of the program.[3] This also means that more detailed simulations are possible, as there is no speed constraint for the simulator. To do this, the applications (Virtual Machines) are paused when the network simulator should run, and resumed after the simulator is done processing current packets. After the network simulator runs, the applications will be allowed to run for a set amount of time before being paused again. In order to process packets, TANSIV can receive packets sent by applications at any time, and records their sending times. These packets, along with their timestamps, will be analyzed by the simulator the next time it runs. On the other hand, packets that are received by applications are sent by the simulator only when applications are paused.

The time applications are allowed to run is determined as follows:

- If there are packets in transit, the applications run up until the next time a packet is supposed to be received (including by intermediary nodes).
- If there are no packets in transit, the applications run for a time equivalent to the minimum latency in the simulated network.

In all cases, this ensures that packets sent at any time in a run window (including at the very beginning) can be delivered by the simulator at the right time.

In summary, at each application pause, the simulator

(1) gets information on packets sent in the previous application run window.
(2) predicts the packets' arrival times.
(3) sends any packets that must be received by applications at this time.
(4) outputs the time applications are allowed to run.

[3]ns-3's DCE can modify the program's perception and execution of time, but this may become inconsistent with the number of instructions the program executed. This inconsistency is exploited for sandbox detection in some malware: one thread runs a loop that increases a counter, another waits, and the results can be compared to what is expected on some standard processors.

3 SIMILAR WORK

Turret[2] is a framework designed to find performance vulnerabilities in arbitrary distributed applications automatically. It does so by running applications in virtual machines, connected through ns-3's DCE, and using virtual machine snapshotting and snapshotting of ns-3 to simulate multiple executions with modified messages. Its architecture allows studying arbitrary distributed applications (any that will run in a virtual machine). But the use of virtual machines limits the number of nodes as they require significant resources. The performance of ns-3's DCE might also limit the accuracy of the simulation of very low-latency or high-bandwidth networks, as it needs to simulate these in real-time.

Distem[4] is designed to study distributed applications on some specified (emulated) target hardware, from different (real) hardware, with an emphasis on ease of use. In particular, this allows testing distributed applications on more nodes than physically available, or on heterogeneous hardware when the available hardware is homogeneous. It does so by running Linux containers (LXC) and reducing their performance through several Linux features: it operates on CPU power and network speeds for the most part. Containers are more lightweight than virtual machines, but this does limit it to running applications that can run in Linux containers, which is less general. It is also designed to run containers on multiple machines, in order to simulate more virtual nodes. However, this approach can't be used to simulate more powerful machines, or network speeds between two virtual nodes that are greater than the link between their hosting physical nodes.

TANSIVTx is another framework for studying distributed applications, with an emphasis on network accuracy. It uses virtual machines, which allows for running arbitrary applications. Like Turret, it connects these virtual machines through a network simulator (but uses SimGrid instead of ns-3). Its main feature is the ability to pause virtual machines to run the network simulator: this allows simulating networks that couldn't otherwise be simulated in real-time, and increases accuracy. But, even though it removes the need for a fast network simulation, the use of virtual machines limits the number of nodes on the network that can run applications, as virtual machines will use more memory.

4 CONTRIBUTION

4.1 Constraints

As stated before, our goal is to extend TANSIV to light virtualization, to make it easier to use, and to reduce its resource requirements. This requires a light virtualization framework that allows routing the application's network traffic through a network simulator, and pause the execution of applications in a way that is invisible to them. The chosen solutions should have a minimal impact on performance when the process runs (but can take time when the application is paused), and be as accurate with regards to time as possible.

4.2 Solutions

To route a container's network traffic to a network simulator, one possibility is to have a program read all raw network packets coming from the container, and translate this to the network simulator's API.[4] Then, when a packet is supposed to be received, the program will build a network packet from the received data and send it to the application. Raw ethernet frames should be sent to the network simulator, for the bandwidth to be accurate for an ethernet link. The other option would have been to modify the API used by the application and translate this to the network simulator's API, but this is less general as this would require implementing this for each of the several different APIs available for network access.

[4]Application Programming Interface.

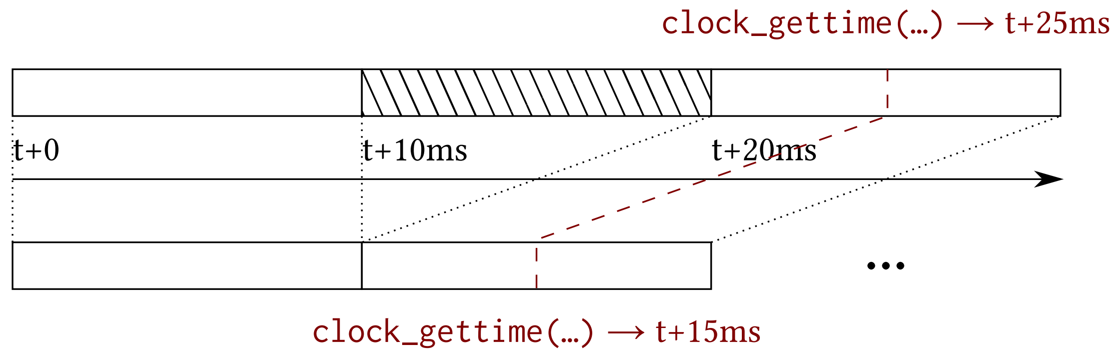


Fig. 1. Illustration of the problem with pausing applications through `cgroup.freeze` (shown as striped rectangle). Functions like `clock_gettime` will give away the real time, therefore it needs to be changed before being given by the application in order for the application to "see" the virtual "application time". The `LD_PRELOAD` library is in charge of doing this translation, and simply substracts the total amount of time the application spent paused from the real time.

Pausing applications can be done by preventing the execution of processes and threads for some time, but this alone won't be invisible to the application, as functions to get the current time[5] or to wait for a given time, such as sleep functions, will still operate based on real time. The clock would need to be paused and restarted as well. We can fix this problem by keeping track of the total time the applications spend paused, and substracting this from every request for time. For performance, this value can be stored in a shared memory area, accessible by the piece of software modifying the results of time-related functions, and updated when the application in paused (which avoids race conditions). Sleeping functions can be called again when they expire with the time the application still needs to sleep in its virtual time, if they were interrupted by a pause (which can be detected by a change in the shared memory offset). Other common functions involving time include file timestamps, timers, and any function with a timeout.

Pausing of applications is done in a separate process (one per application), running in "high priority" or "real-time" mode, to increase precision of the pausing and resuming times. All this process does is

(1) get the time of the next pause from the main process,
(2) wait until this time arrives, then pause the application,
(3) inform the main process that the application is paused, so that the network simulator can run,
(4) wait for the network simulator to finish processing packets, then update the total time the application spent paused, stored in shared memory, and resume the execution of the attached application.

4.3 Implementation

We chose to use Docker on Linux as a light virtualization solution, but the same design should be implementable on any such solution. In particular, any container system using Linux cgroups and able to restrict access of the container to a single tap interface should work.

[5]Such as `clock_gettime` on Linux, or the `RDTSC` instruction on x86.

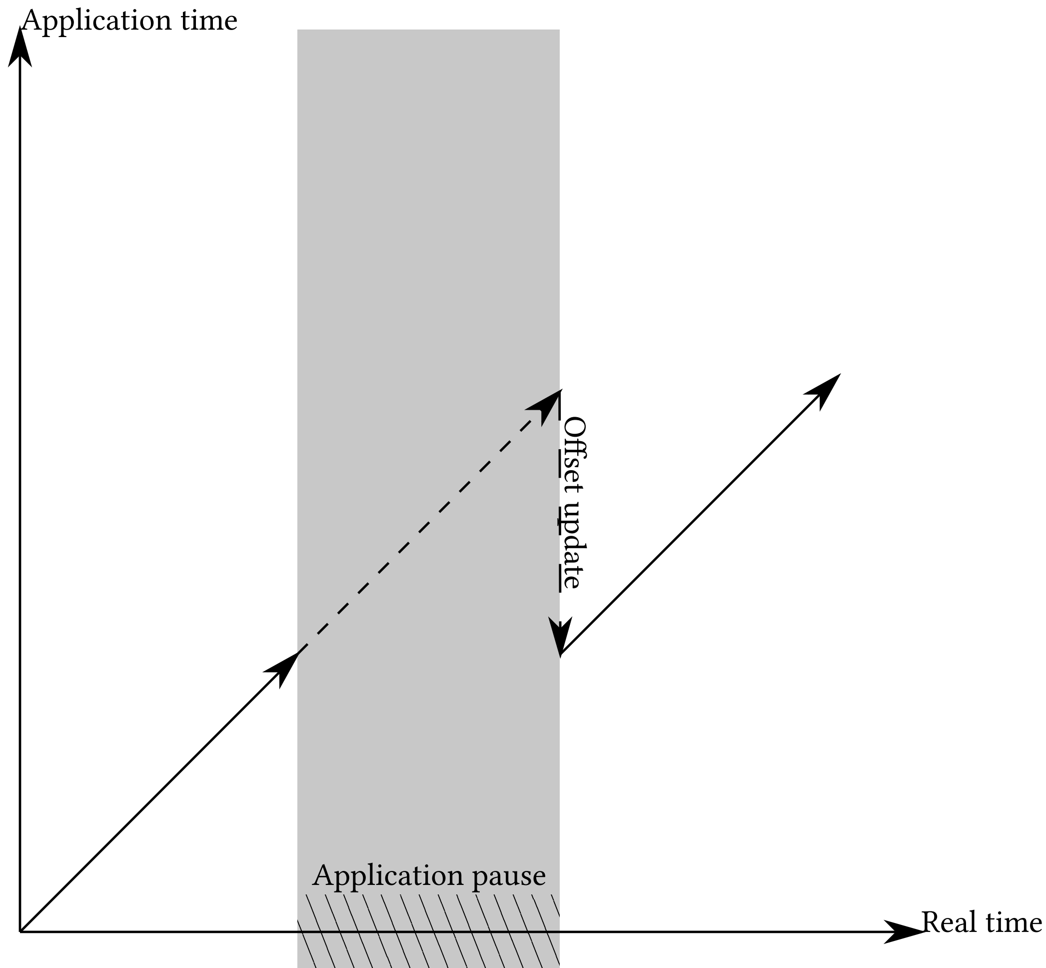
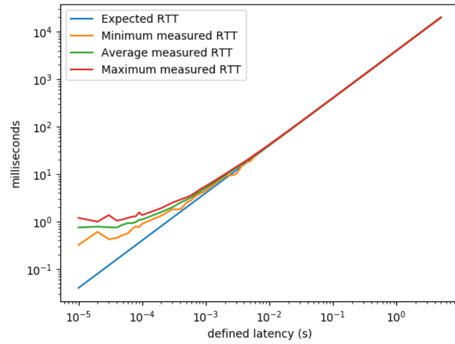


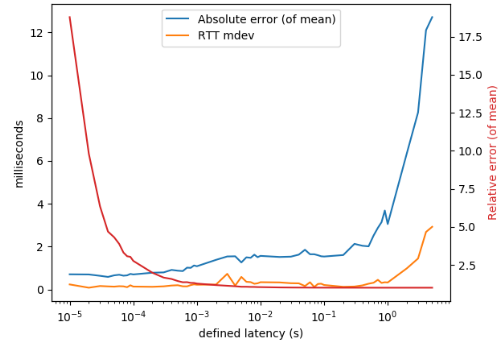
Fig. 2. Illustration of time correction using the stopper process and `LD_PRELOAD` library, through an offset in shared memory.

The interception of network communications in the container was achieved through a tap interface: a special virtual network interface type that allows reading sent packets and writing packets to be received through a file-like interface. Note that in order for the packet's send times to be accurately measured by storing the current time on each read, the packets must be read fast enough, but TANSIV's architecture makes this separable from the network simulator's performance. Containers can be limited to a single outside network interface through Linux's network namespaces, and, in this case, using Docker networks: we created a macvlan network based on an outside tap interface, which makes the routing configuration affect only the container, but makes it accessible for reading and writing outside of it. All ethernet packets were transmitted through SimGrid using the already-existing TANSIV framework.

The pausing of containers can be done through the Linux `cgroup.freeze` interface, which allows "freezing" and "thawing" the execution of a cgroup (which Docker creates for each container), but does no specific handling of the clock: containers always see the current time, and sleep functions consider paused time as slept time. This works in a similar way to `SIGSTOP` and `SIGCONT`, but works at the cgroup level, and signals can't be seen by the applications.



(a) Graph of measured latency based on simulated latency (log scale)



(b) Graph of absolute and relative error of the mean of the measured latency compared to the round-trip time, along with standard deviation of measurements, based on simulated latency

We therefore had to modify the behavior of time-related functions by using a library loaded through `LD_PRELOAD`, which makes dynamically-linked programs use this library's functions before any other library functions: this allows replacing, for example, C standard library functions, or functions of some more specific libraries. This solution is simple, efficient, and allows modifying many time-related functions that don't induce a system call, such as `clock_gettime`, which are stored in the Linux vDSO[6] However, this will not modify the behavior of system calls, which may be done directly by some applications, or statically-linked ones. Interception of system calls is also possible (for example through seccomp), but wasn't explored here. Interception of processor instructions that give the current time, such as `RDTSC` on x86, also wasn't considered. This is because this is rare in practice with common applications.

On Linux, a process can be run in real-time priority by setting its scheduling policy to `SCHED_FIFO`, which will pre-empt running threads as necessary to run the high-priority process as soon as possible. This should make the containers' pauses more accurate compared to the scheduled time.

5 EVALUATION

We evaluated our prototype on accuracy of latency and bandwidth simulation, based on a simple star topology involving two containers, with varying latency and bandwidth for the node to router link. All experiments were run on an ASUS R505CB laptop, with an Intel Core i3-3217U (1.80GHz) CPU, running Ubuntu 20.04. Most values come from a single test, because of the large number of parameter values tested.

5.1 Latency

We evaluated the accuracy of latency simulation in our implementation, and which latencies can be acceptably simulated. We used the `ping` command from iputils, in user-to-user mode (`-U` option) to measure latencies. This mode makes `ping` measure the time when it receives the response, instead of getting the received packet's timestamp from the kernel (through `SIOCGSTAMP` or `SO_TIMESTAMP`). This adds a little more time to the measured round trip time, but was done because the translation of packet timestamps wasn't implemented in the prototype. We will estimate that the expected

[6]Meaning "virtual dynamic shared object", which makes some system calls available as functions for performance reason. This is used for functions which get the current time because they may be called many times in short intervals.

round-trip time is supposed to be four times the latency of the node to router link. Higher latencies weren't tested as `ping` would consider delayed packets as lost.

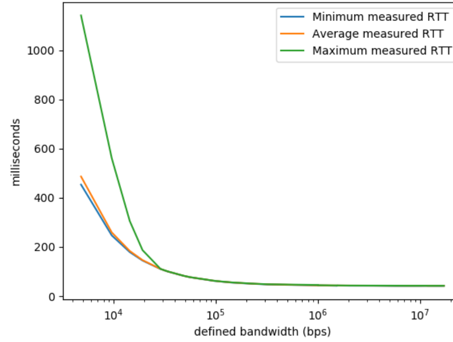
Figure 3a shows how well measurements from `ping -U` fit with the target roundtrip time. It appears to fit well, except for very low latencies (<0.5ms). Figure 3b shows the deviation more clearly, with the absolute error staying between 0.5ms and 2ms for the most part. For large latencies (on the order of a second and more), the absolute error goes up, reaching almost 13ms for a simulated latency of 5s. This is correlated to an increase in the standard deviation, which goes up to 3ms for a latency of 5s, while it's between 0.08ms and 0.5ms for most of the lower latency samples. For smaller latencies, absolute error seems to very slightly increase when the latency does. As the absolute error doesn't go below roughly 0.4ms, the relative error is high for very low latencies, but low for the rest of simulated latencies. This includes the higher latencies, as the increase in absolute error is small compared to the high latency. Some latency is expected because of the network stack: on the same machine, a `ping -U -c30 ::1` command gives a mean measured latency of 0.13ms. Considering this, the prototype seems suited for simulating most kinds of standard network links, as far as latency is concerned..

5.2 Bandwidth

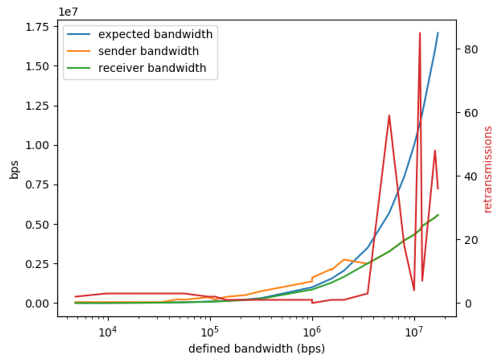
We wanted to see how accurate the prototype was on simulating bandwidth, and for which networks this accuracy would be acceptable to run simulations on. Bandwidths were measured using `iperf3`, in TCP mode. There is both a "sender" and "receiver" bandwidth: the sender bandwidth corresponds to the amount of data the client could send to the server, and the receiver bandwidth is the amount of data that was actually received. It is possible that because of packet loss (or big delays), connections drop and some data isn't actually transferred. In this case, `iperf3` creates a new connection: the final bandwidth values are the average over each connection. When using `iperf3` in UDP mode, the sender and receiver bandwidths can simply differ because of packet loss, with the receiver bandwidth being the bandwidth of non-lost packets.

Figure 4b shows that the sender and receiver bandwidths follow the expected bandwidth at low bandwidths, with the sender bandwidth detaching starting with around 40kbps, becoming higher than the expected bandwith, and receiver bandwidth starting to detach from the expected bandwidth at around 2Mbps, with retransmissions increasing starting with 4Mbps. For bandwidths lower than 4Mbps, retransmissions are in the range of 0 to 3, but starting with 4Mbps, retransmissions can reach very high values, with some samples still having low retransmissions values (4-7). The sender bandwidth can be seen higher than the expected bandwidth in most places, but becomes almost equal to the receiver bandwidth starting with around 3.5Mbps. This is also true in an extended version of the graph, where we can see that the receiver and sender bandwidths start to increase logarithmically from this point, instead of linearly (to follow the expected bandwidth), until it stops increasing and becomes more chaotic. Figure 4c shows the absolute and relative errors, and is consistent with the previous observations. It has higher relative error on small bandwidths, and high error at higher bandwidths, where the measured bandwidth doesn't keep up with the expected one. The minimum relative error is 1.13: this may simply be the difference between physical and logical bandwidth, as "expected bandwidth" here takes into account the ethernet frames, and measured bandwidth likely ignores IP and TCP headers. Additionally, TCP doesn't use the full network link bandwidth: it starts at a low bandwidth and doesn't increase up to the maximum bandwidth, as would be possible by carefully tuning a UDP flow using the known parameters of the current networks.

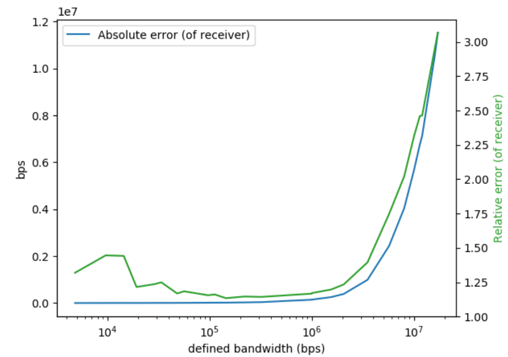
One hypothesis for why bandwidth results show the prototype struggling with moderately high bandwidth is that packets aren't read fast enough, which makes them have inaccurate and sometimes overlapping or even out of order timestamps. This in turns, could create large delays, sometimes seen as packet loss by applications. This is because, in



(a) Graph of measured latency based on simulated bandwidth



(b) Graph of measured bandwidth based on simulated bandwidth



(c) Graph of absolute and relative error of the measured receiver bandwidth compared to the simulated bandwidth, based on simulated bandwidth

the prototype, packets are timestamped as they are read, and sent to the simulator before reading the next one, as was done in the framework. Tests done with `iperf3` in UDP mode seem consistent with this hypothesis being one factor in this issue.

Therefore, the prototype in its current state can only study relatively low-bandwidth links, such as distributed applications running behind ADSL consumer connections. But, considering TANSIV achieves much greater bandwidths through QEMU, this is likely to be a weakness in the prototype and not with the overall architecture.

6 CONCLUSION

We showed an architecture allowing for the generalization of TANSIV to Linux containers, along with a working prototype. The prototype shows that it's possible to accurately simulate very low latencies using this approach. However, bandwidth results weren't very conclusive as the prototype ran into problems with high bandwidth flows. Results with lower bandwidths tend to show that the simulation is somewhat accurate.

The current implementation uses tap interfaces with the Linux TCP/IP stack, which doesn't use the simulated time. The only place time is supposed to be used is in two TCP features: timeouts and delayed ACKs. Timeouts (for packet

loss detection) will only have an effect if a timeout does occur, which shouldn't happen very often, and delayed ACK is an optional feature, but its behavior may change if containers spend a significant amount of time frozen. Note that there are variants of the TCP Congestion Avoidance algorithm, such as TCP Vegas that make use of slight round-trip time changes: these algorithms can give very different results when used with the current prototype, compared with a real network. Options to explore to avoid this problem include userspace TCP/IP stacks (such as `lwip`) or replacing the tap interface with a socket API that communicates with the network simulator (which is closer to what ns-3's DCE does).

Assuming that the amount of time containers can spend in a frozen state isn't limited, it would be possible to extend the architecture to do other tasks when containers are frozen. As an example, applications could be inspected when frozen, or debugged by implementing a `gdb` server, with all applications being paused while the user interacts with the debugger. Compared with standard `gdb`, this would make distributed applications easier to debug as user interactions wouldn't affect the application's behavior regarding network and time. All instances of the application could be stopped when debugging one, in addition to the network, which makes the user's inspection and debugging invisible to the whole simulation.

Another possibility would be to process the applications' run windows sequentially instead of in parallel: this can make for somewhat accurate simulation of a larger number of applications (significantly more than the amount of available processor cores). This is made very easy by the fact that packets can only be received outside of run windows, so they only need to be run after all the windows in the previous round are.

REFERENCES

[1] Léo Cosseron. Tansivtx: time-accurate network simulation interconnecting vms with hardware virtualization towards stealth analysis. Master research Internship, (2022).

[2] Hyojeong Lee, Jeff Seibert, Endadul Hoque, Charles Killian, and Cristina Nita-Rotaru. 2014. Turret: a platform for automated attack finding in unmodified distributed system implementations. In *2014 IEEE 34th International Conference on Distributed Computing Systems*, 660–669. DOI: 10.1109/ICDCS.2014.73.

[3] Paul B. Menage. 2007. Adding generic process containers to the linux kernel. In *Proceedings of the Linux Symposium*, 45–58.

[4] Luc Sarzyniec, Tomasz Buchert, Emmanuel Jeanvoine, and Lucas Nussbaum. 2013. Design and Evaluation of a Virtual Experimental Environment for Distributed Systems. In *PDP2013 - 21st Euromicro International Conference on Parallel, Distributed and Network-Based Processing* (21st Euromicro International Conference on Parallel, Distributed and Network-Based Processing). IEEE, Belfast, United Kingdom, (Feb. 2013), 172–179. DOI: 10.1109/PDP.2013.32.