SAARLAND UNIVERSITY

Faculty of Mathematics and Computer Science
Department of Computer Science
Master Thesis

Low-Level End-to-End System-Traces through Modular Full System Simulation

submitted by
Jakob Görgen
Saarbrücken
May 2024

Advisor:
Dr. Antoine Kaufmann
Operating Systems Research Group
Max Planck Institute for Software Systems
66123 Saarbrücken, Germany

Reviewer 1:
Dr. Antoine Kaufmann
Operating Systems Research Group
Max Planck Institute for Software Systems
66123 Saarbrücken, Germany

Reviewer 2:
Dr. Laurent Bindschaedler
Data Systems Group
Max Planck Institute for Software Systems
66123 Saarbrücken, Germany

Saarland University
Faculty of Mathematics and Computer Science
Department of Computer Science
Campus - Building E1.1
66123 Saarbrücken
Germany

Erklärung

Ich erkläre hiermit, dass ich die vorliegende Arbeit selbständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel verwendet habe.

Statement

I hereby confirm that I have written this thesis on my own and that I have not used any other media or materials than the ones referred to in this thesis

Einverständniserklärung

Ich bin damit einverstanden, dass meine (bestandene) Arbeit in beiden Versionen in die Bibliothek der Informatik aufgenommen und damit veröffentlicht wird.

Declaration of Consent

I agree to make both versions of my thesis (with a passing grade) accessible to the public by having them added to the library of the Computer Science Department.

Saarbrücken,_____ _____
(Datum/Date) (Unterschrift/Signature)

Abstract

Analyzing how modern networking systems perform is hard due to their complexity. To meet their requirements, these modern systems rely on many interconnected hardware and software components. In order to gain a deep understanding of such systems, it is crucial to understand the interactions of the many components involved. Existing tools can't trace what hardware components such as network cards do down to the level of e.g. individual memory accesses, even though these can have a significant impact on performance. The information needed to gain detailed insight into hardware itself is too expensive to record and analyze on real hardware in production environments without significantly affecting the systems' performance. Furthermore, the act of tracing itself may influence the system such that the information of interest is simply no longer measurable.

In this work I instead advocate evaluating systems in modular end-to-end simulations to provide deep visibility without affecting system behavior. Simulations offer a powerful alternative to real-world tracing. On the other hand, recording every detail in production systems is expensive and disruptive. Simulations let you examine a system down to the hardware level virtually, without affecting system performance. By modeling individual components and their interactions, simulations provide detailed logs, offering an in-depth view of complex systems. However, it remains a challenge to reconstruct the behavior across the full system using these detailed simulator logs. These log files become quite huge especially when examining hardware. This makes it difficult and impractical to understand the interactions of the involved actions by using simulator log files alone.

This thesis proposes Columbo, a framework aiming to solve this problem by combining simulation with distributed tracing. Simulation provides deep visibility, while distributed tracing helps users to understand the cause-and-effect relationships between events across different simulated components. Columbo uses detailed log files written by different simulators composed into a modular full system simulation. From the information contained in the individual log files the framework creates distributed traces which can be exported to battle tested distributed tracing tools to easily visualize and query traces. This enables users to understand the interactions between different components and to answer questions like "what was the average end-to-end latency within a simulation" and "how was this latency distributed across individual components".

Acknowledgements

I would like to thank my advisor Antoine Kaufmann for providing guidance, feedback and support throughout this project and for making working as part of the OS group at MPI-SWS a great experience.

In addition, I want to thank Hejing and Jonas for the fun discussions and help while working on SimBrick.

Next I would like to thank all my fellow students that made the time at Saarland University fun, especially Marvin, Luca and Felix. I really enjoyed the many lunches and evenings we spend together as well as the countless hours of working together on projects and exercise sheets.

A special thanks also to my friends outside university, in particular I want to thank Maximilian, Karsten, Noah and Jan for the countless hours of welcome distraction from work and their support.

Finally, I thank my family for all the support I received throughout my time at university. My parents Maria and Albert as well as my brother Philipp have been a constant source of support and encouragement.

Contents

Chapter 1
Introduction

1.1 Understanding system performance is hard

Understanding the performance behavior of modern networking systems is hard due to the complex interactions between the involved components. Therefore, tools are needed to help researchers and practitioners reason about the characteristics of a system. Traditional end-host networking stacks struggle to keep up with increasing datacenter access link bandwidths. This situation gets even worse with the slowdown of Moore's Law and the end of Dennard's scaling [38, 54, 31]. As a result, bottlenecks are rarely in the network itself but within end-host hardware- or software-components [44]. In order to deal with these problems, a variety of different solutions have been developed, reaching from Linux network stack optimizations [31, 45, 23, 14] over to hardware offload mechanisms [24, 33], which are by now widely adapted in modern datacenters to meet the high demands for throughput, low latency and low CPU overhead [31, 54]. To leverage these solutions full capabilities and exceptional performance, users need a deep understanding of their systems' complex behaviors [54]. However, building such a deep understanding is difficult due to the size and complexity of these modern stacks, which include many tightly integrated components such as the application, kernel, drivers, network interface card and network that affect the end-to-end behavior of these systems [31, 32]. As the overall performance of such systems is a product of the complex interactions between the involved components, understanding them is necessary.

As a result of these trends, researchers and practitioners alike have a need for performance analysis tools enabling them to understand the complex behavior of these systems end-to-

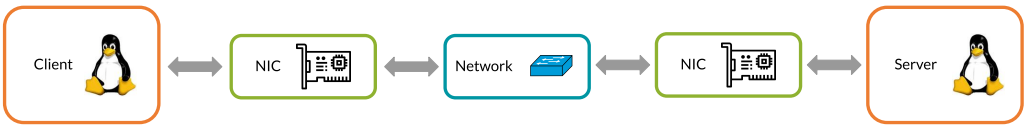


Figure 1.1: Simple Client-Server network topology

end. Consider, for example, a client is sending packets to the server in the simple network topology shown in Figure 1.1. When this packet is received, it may be affected by the client's NIC that is suffering from head-of-line blocking because it's buffer for outgoing packets has become full, thus delaying the sending of a particular packet. Another reason influencing the performance of such a system might be simply that the NIC in such a topology issues lots of interrupts indicating the arrival of packets. Such interrupts have to be handled on host side and incur large overheads.

In particular, being able to explain system behavior by understanding the interplay of different components and gathering inside on how end-to-end latencies are distributed across the different components involved, is crucial.

1.2 Distributed Tracing provides insufficient visibility

The need for correlating and integrating data across components, system and machine boundaries is not new. Users often need to perform distributed tracing for distributed systems, i.e. correlate information/events from one system with those from another to gain the necessary insight into the overall end-to-end behavior of the system. This is especially true if problems span multiple components or systems. To do so, researchers came up with the notion of a context that follows the execution of applications in distributed systems through events, queues, thread pools, caches and messages between different components. Contexts causally connect events on the execution path at runtime and enable a number of different tools [46]. Distributed tracing tools have been developed by researchers and practitioners alike [52, 28, 36, 34]. These tools allow users to causally connect events and logs in different systems across their boundaries, by propagating trace context in the form of request- or event-IDs between the involved systems [46].

Distributed tracing provides, in principle, the concepts and functionality that is needed to gain the visibility wanted as described in Section 1.1. The problem with classical approaches is however that they usually only extend to tracing the involved software components. None of the aforementioned tools gives visibility down to hardware components [52, 28, 36, 34] i.e. one usually cannot see what e.g. was happening inside a NIC down to the PCIe level when using classical distributed tracing approaches. However, the contemporary microsecond-scale network stacks present a challenge in this regard. Performance bottlenecks or anomalies within the hardware components, such as PCIe

interconnects, network interface cards (NICs), network switches or accelerators, frequently have a significant impact on end-to-end performance. [50, 48]. Looking at our example topology in Figure 1.1, it may be necessary to have this level of visibility into, for example, the NIC and its registers or memory in order to fully understand the systems behavior. With classic distributed tracing approaches the visibility would however end at the respective hosts' software stack and not reach into the NICs or the network itself. The reasons for that are diverse. At the fine granularity of individual network packets, PCIe transactions, hardware cycles, recording and transmitting the necessary information for analysis is prohibitively expensive and may affect the behavior and performance of the system under investigation making it infeasible.

1.3 Visibility through Simulation

Simulation is well suited to providing the necessary insight, as described in Section 1.1, into systems. There are two reasons for this:

1. Getting visibility at the fine granularity of individual network packets, PCIe transactions or hardware cycles, recording and transmitting the necessary information for analysis is expensive but sometimes required. This however makes instrumenting real production stacks or instrumenting physical testbeds infeasible due to the large overheads introduced, thereby affecting the behavior and performance of the system that one wants to understand. In a simulation, however, you can get as much information out of a simulation as you need without affecting the simulated system at all.

2. If physical testbeds are not available, because ones work requires cutting edge commercial hardware that is not available at the time of publication, or because hardware extensions to existing proprietary hardware are being developed, or because entirely new ASIC hardware architectures are being proposed, simulation can be a viable alternative, allowing to simulate the required components that are not yet available [44].

Since the ultimate goal is to understand systems across component boundaries in an end-to-end environment, turning to simulation requires simulating each component involved. This becomes feasible by making use of Modular Full System Simulation frameworks like SimBricks that allow to compose end-to-end simulations by attaching different and already existing simulators to each other. That way whole topologies can be simulated end-to-end [44]. Because simulators, such as those used by SimBricks, usually provide deep visibility in the form of log files, one can theoretically easily access all the information that is needed to understand the complex behavior of the components involved.

1.4 Contribution

In this thesis I propose an alternate approach to obtain in depth visibility into system performance: specifically I propose running systems in modular simulations and combine individual simulator logs into detailed full system traces.

When considering the use of modular end-to-end simulation and the visibility provided by the simulators used, it is important to note that huge amounts of data are generated during such simulations. This is especially true when providing visibility down to the hardware level. If users want to understand the behavior end-to-end, they are faced with the tedious and unfeasible task of correlating the data stored in one simulator's log file with the data provided by another simulator. This approach would clearly negate the advantages of modular end-to-end simulation and its visibility. Therefore, tools are needed that allow users to easily correlate and investigate the information provided by the different simulators.

This work aims to address this need by providing Columbo, a framework that allows users to gain the necessary deep insight into systems as described in Section 1.1. To this end, Columbo allows to combine the advantages of modular end-to-end simulations, namely deep visibility, with the advantages of distributed tracing, allowing to understand causal relationships between events to easily reason about the end-to-end behavior of systems.

The key idea is to harness log files written by simulators during a modular end-to-end simulation of a system of interest. Within these log files the detailed behavior of the simulated systems down to the hardware level is recorded without affecting the simulated system at all. These log files are then in a second step used to create events and spans that are then causally connected to form a distributed trace through the simulated system across simulator boundaries.

In the following, we will have a closer look at what distributed tracing is and how context propagation is usually done. We will then discuss modular end-to-end simulation, and in particular SimBricks, a framework that makes it easy to set up such simulations [44]. Then Columbo, its implementation and how context propagation differs from some classical settings are discussed before diving into the evaluation.

Chapter 2
Background

2.1 Distributed Tracing

Modern Internet services are nowadays often realized as distributed systems [28, 36, 52]. Such distributed systems can be found in many successful applications such as web search, social networks, data analytics applications, large-scale machine learning applications, to loosely-coupled microservices, serverless lambdas [46], DNS or simply databases [36]. In order to achieve a high level task multiple components of such a distributed system perform a narrow slice of work. Additionally they communicate across software, process, system and machine boundaries to perform and ultimately fulfill that task. Due to their distributed nature do distributed systems not provide a central point to collect information about their execution unlike standalone applications [46]. This is however a problem as developers need a central point of view that provides deep visibility into such systems in order to understand the interactions between the different components involved across software, process and system boundaries to reason e.g. about performance characteristics or failures of such systems.

Distributed Tracing provides this visibility end-to-end in distributed systems [42]. Distributed tracing achieves this by first recording i.e. logging events within the involved components respectively alongside event-timing information [46] and combining these to form a span. Note that this event creation happens on the critical path of the software. It adds overhead and therefore degrades system performance [42]. This event creation usually happens in software and requires users to instrument the target application they want to trace. These aforementioned spans usually represent individual units of work

that were done inside a component, whereas the events represent the even smaller actions that were done to perform that individual unit of work. The spans created while tracing are then connected to each other to form a tree like data structure in which a parent can have multiple children [1]. This tree like data structure describes the path a request took through the different components of a distributed system as well as performance costs incurred at the visited components and timing between events [46]. Therefore, the causal connections and ordering of these events across software, process and system boundaries are captured [42].

2.1.1 Context Propagation

To be able to establish the causal relationships between events/spans encapsulated within a distributed Trace as described in Section 2.1, developers and researchers generally have two different options:

1. **Metadata Propagation.** One approach is to explicitly propagate metadata through all components of a distributed trace, allowing events and thus spans to be correlated. Such metadata could include a unique identifier for the current execution like e.g. a request ID. In such a case each event is upon generation annotated with these identifiers or more generally with the current context of execution. This attached trace context allows distributed tracing backends to correlate events and the spans they form to each other to ultimately construct the directed acyclic graph like structure that represents a trace. Trace contexts need to be propagated through the whole execution across all system, process or machine boundaries. Consequently, this approach requires non-trivial changes of all components involved at the source code level to allow for the propagation of that context alongside the execution [46]. Propagating trace context across all component and process boundaries alongside requests incurs additional runtime overhead on top of the event generation. Generally this approach was however rather successful in practice and found broad adoption [36, 40, 52].

2. **Inferring Relationships.** The second approach is to not propagate such trace context at all but to infer relationships between events offline by making use of the information contained within the logging statements that create events. Data allowing to infer such relationships may be IP addresses or memory addresses. Alternative approaches for inferring causality may make use of machine learning models or other statistical methods to infer causality between events/spans. Magpie [28] showed for example that causality between events can be inferred after the actual execution already finished. This could for example be the case if start- and end-events generated within the same thread of execution exist, allowing to infer that all intermediary events are causally related. In a similar fashion causality may

be inferred across component boundaries when send and receive events are present on both sides of the boundary. Such approaches avoid the need to propagate trace context completely [46]. As a result they do not require modification of the involved components at the source code level to allow for metadata propagation.

The work presented in this thesis also falls into the category of inferring relationship tracing tools. Similarly, as mentioned above, the framework makes use of the natural boundaries of the components involved and the fact that start/send and end/receive events can be generated during simulation to correlate events across component boundaries. More on that in Chapter 3.

2.1.2 Sampling

Trace creation incurs overhead on several occasions. First, events have to be generated at the components involved/of interest. Depending on the chosen approach for creating causal connections, trace context needs to be propagated between components. As a result the data generated for creating a trace is spread across multiple components. This trace data is in a next step send to tracing backends that receive the data for aggregation and processing. The processing, i.e. the construction of the abstract representation of the traces, also incurs runtime costs. In the end traces are stored in a way that allows users to later on query and analyze them [42].

As a result overheads should be minimized. One approach to do this is called sampling. The idea is that instead of tracing each and every request / execution path through a distributed system, to only apply distributed tracing on a subset of these. An important aspect of this technique is that either a trace is captured in its entirety with all its events, spans and causal connections or not at all. This approach is rather successful as one only has to pay additional costs for trace creation, processing and storing if data is collected and processed at all [42]. Two typical sampling approaches are as follows:

1. **Head-Based Sampling.** The first approach is to make an immediate decision on whether to trace a request or not on its arrival into the system. This approach has been taken, for example, by Googles Dapper [52] or Facebook's Canopy [40] tracing system. This clearly reduces tracing overheads when the system decided to not trace a request before that requests actual execution. One problem with this approach is that if the decision is made before the actual execution has taken place, the decision must be made randomly. This usually means that the set of sampled traces contains to a large extend the traces of the most common execution paths within a system. This is bad as a lot of overlap and redundant information is processed and stored. Edge cases on the other hand will only rarely be sampled, which consequently harms the general usability of such a tracing system [42].

2. **Tail-Based Sampling.** In contrast to head-based sampling, tail-based sampling captures traces for all request entering a distributed system. The decision on whether to keep a trace or not is made once a trace was generated and sent to a tracing backend. This is viable as the trace generation is compared to the actual trace processing, storing and querying cheap in terms of runtime. Paying the trace generation cost has the advantage that such systems can make use of the trace data to make a decision on whether to sample a trace or not. As such that decision can be biased to only further process traces of the largest interest and discard others. This results in a trace set containing traces of higher value which potentially allows sampling overall much fewer traces to receive an equally helpful set of traces as compared to head-based sampling [42].

2.2 Modular Full System Simulation

As mentioned in Section 1.1 are modern end-host networking stacks struggling to keep up with increasing datacenter access link bandwidths. This becomes an even bigger problem considering the slowdown of Moore's Law and the end of Dennard's scaling [38, 54, 31]. In addition, does modern multi-core hardware start to hit a power wall. These trends in combination with a new emphasis on data centric applications processing vasts amount of data are disrupting [25] and result in a situation in which bottlenecks are rarely located in the network itself but within an end-hosts software or hardware components [44].

One way of dealing with this problem is e.g. to optimize the Linux network stack such that it fits the currently required need or to introduce solutions that bypass the OS kernel [31, 45, 23, 14, 49, 39]. Another potential solution is the introduction of hardware offload mechanisms [24, 33]. Generally, there is a trend on using more and more accelerators for anything and everything, both general purpose or embedded computing [25]. By now, hardware offloading mechanisms are widely adapted in modern datacenters to meet the high demands for throughput, low latency and low CPU overhead [31, 54]. It is notable that many of the newly introduced components absorb more and more system level functions such as device controllers or network interfaces [25].

Due to all the variability within these modern system stacks became very to hard understand and make predictions on a systems overall performance/behavior which also heavily depends on an applications characteristics [25]. A deep understanding is however necessary to make full use of these solutions exceptional capabilities and performance due to their complex behaviors [54]. Building up such an understanding is very difficult due to the sheer size, scale and complexity of such systems that comprise many tightly integrated components like the application, kernel, driver, network interface card and the network which influence the end-to-end behavior of these systems [31, 32].

As a result of these trends researchers and practitioners alike have a need for tools aiding them in understanding the complex behaviors of these systems. Ideally such tools shall allow for an arbitrary detailed analysis [25]. This is where simulation comes in, as it can provide the deep insight needed for researchers and engineers to gain an in-depth understanding of these complex systems. When trying to understand such complex systems using actual physical hardware, lots of information, depending on the used hardware, is hidden from the user behind opaque devices or internal buses. In contrast, simulation provides visibility into all the components of a system one is interested in [35]. In simulation, arbitrary complex queries can be executed to gain insight on the simulated system. Additionally, does simulation generally allow to stop the virtual time which in turn enables users to investigate a system at a specific point in time. All this can be achieved by simulation without affecting the simulated system at all[35]. In principle, the user can extract as much information as possible from a simulation and the components involved, without the overhead of doing so being reflected in the simulated time. This is a particular advantage compared to physical testbeds, where gaining this level of insight can introduce significant overheads that also impact performance. Besides, it may not even be possible or feasible to get the information one needs. This can happen, for example, if the hardware of interest is not yet accessible, does not allow the probing of the information of interest, or the probing would generate huge amounts of data even in short periods of time, making it infeasible as the generation of this data would have too great an impact on the timing information of interest.

For these reasons, system architects have turned to simulation through virtual prototyping to evaluate or validate design ideas. Simulations are used in a wide range of areas. It is e.g. used in replicating and simulating a specific architecture in software, allowing architects to explore and validate the functional behavior or the performance of the proposed architecture designs. This approach also enables early software development and allows architects to perform non-intrusive debugging. Therefore, it enables users to shift to software prototyping were the system is entirely simulated in software and away from cumbersome hardware prototyping [43]. Especially the so called *full-system simulation* has proven to be very useful over the past decade. It is a valuable tool aiding developers in the creation of new systems, ranging from embedded system, telecommunication infrastructure to servers and high performance computing solutions [35].

Full-system simulation typically handles two aspects of simulation:

1. **Hardware.** The hardware is modeled completely. That means not just the processor but also it's peripherals are simulated. More specifically does this lead to a simulation simulating the involved hardware complete enough to run the real and unmodified software stack that would also run on the real actual hardware on top of the simulation. There are models for the processor, memory, peripherals, buses, networks or other interconnects depending on the system being simulated,

so that the software cannot tell the difference between the physical and simulated systems [43].

2. **Software.** The full software system is modeled. Considering the software side of a full-system simulation it is often the case that besides the hardware, the software that is needed for running applications on the hardware is modeled completely. That means essentially that not only a user space application is modeled, but also the operating system [43] is modeled such that user space applications can be executed on that operating system without any modifications. This also allows users to gain insight into the actual operating system running within the simulation, rather than only the simulated hardware, which is interesting when considering the interplay between software and hardware.

Because full system simulation models the hardware and its peripherals, as well as the operating system, and allows unmodified application software to run, it is a great tool for researchers and practitioners alike to validate and explore their design ideas. However, the ability to model such systems comes at a cost, as managing all the alternatives of how different components can be modelled within such a simulation requires the development of structures and interfaces that allow users to flexibly and easily compose systems by assembling different components into a variety of simulation configurations [30].

This is where the modularity of *Modular Full System Simulation* comes into play. Modularity and clean interfaces allow users to focus on particular components involved in a simulation without the need to understand the whole system with all its modules/components [29]. This modularity allows e.g. that each device, processor or memory model can be developed more or less independently and be supplied as needed to simulations, allowing for fine-grained control over the features required for a simulation a user is interested in [35]. Another advantage of modularity is that it makes collaboration with other researchers or practitioners much more easy. Depending on the actual realization, a modular design allows the use of an excellent simulation framework by developing and using a module that is currently required, without the need to understand the whole system or to leak any intellectual property [29, 35]. Examples for such modular full system simulators are Gem5[29, 12] and Simics[47].

For these reasons is *Modular Full System Simulation* a valuable tool aiding researchers and practitioners alike in validating, testing and understanding complex modern systems.

2.2.1 SimBricks

As described in Sections 1.1 and 2.2 do researchers and practitioners alike turn to simulation in case physical testbeds might not be available or visibility and flexibility as offered by simulation are required. For this purpose, commonly used tools in the networking domain are for example ns-2 [18], ns-3 [5] or OMNeT++ [53] while hardware

designers might employ Modelsim [16] or Verilator [22], whereas system designers might focus on full system simulators like gem5 [29] or Simics [47]. While all of these are outstanding work, they fail in enabling true end-to-end simulation and evaluation. The reason for this is that none of them is able to simulate all components that might be involved within an end-to-end evaluation of a system: hosts, devices and the network itself [44].

SimBricks solves this problem by providing a framework that allows to easily combine different simulators required for simulating the necessary functionality in order to enable true end-to-end simulation [44]. Unless explicitly stated otherwise, everything within the rest of this section references SimBricks.

SimBricks is based on the idea of using existing or new host, hardware device and network simulators and connecting them to create an end-to-end simulation of a complete system, allowing users to run unmodified operating systems, drivers and applications. This allows to make use of the great ecosystem around simulation which lets users benefit from years of research and engineering efforts. Currently SimBricks does support ns-3 [5], OMNeT++ [53] and the Intel Tofino simulator [13] for network simulation, Verilator [22] for hardware RTL simulation and gem5 [29], Simics [47] and QEMU [19] as host simulators. These simulators allow users to cover a broad range of end-to-end network simulations. To this end, SimBricks is a modular, full-system simulation framework that allows to perform true end-to-end network system simulations. The here presented framework makes use of SimBricks simulations to create *Low-Level End-to-End System-Traces*. Therefore, in the following we will have a look at two important concepts of SimBricks in regard to the framework presented in this work.

SimBricks Modularity and Simulator Interfaces

As mentioned is SimBricks modular. This modularity is achieved by running individual simulators as separate processes in parallel. Each simulator simulates an individual component. To connect them together to a large end-to-end simulation, SimBricks makes use of natural boundaries between components that are involved in such simulations, specifically PCIe and Ethernet. For that reason does SimBricks provide well-defined Interfaces between component simulators. These interfaces specify that the NIC and network simulators are connected via an Ethernet interface, and that the device and host simulators are connected via a PCIe interface. Both of these interfaces are asynchronous and allow specifying propagation delay as in physical systems. In case one wants to integrate a simulator for usage with SimBricks, its necessary to add an adapter that implements the required component interface. The adapters, conceptually placed at the natural boundaries such as PCIe or Ethernet, are used to exchange messages via message passing to send requests and receive responses to and from the adjacent simulator. Now we will have a closer look at the two interface types:

1. **PCIe Interface.** Instead of dealing with the complex details of the PCIe protocol, SimBricks defines the PCIe component interface on PCIes transactional layer used for data operations. Simple settings like link bandwidth and latency are used to abstract away the physical details.

   Using this PCIe interface both host and device can initiate reads and writes that are complected by the other side. For that reason does the SimBricks PCIe interface define Dma-write/read messages as well as Mmio-write/read messages. Dma data transfers are initiated by a simulations' device simulators whereas Mmio accesses are initiated by host simulators. All data transfers are asynchronous like in real PCIe. In case a request has finished, either the device simulator issues a Mmio completed message or the host simulator issues a Dma completed message, using their respective adapters. These messages carry identifiers that allow simulators to match requests to responses.

2. **Ethernet Interface.** Like the PCIe interface does SimBricks Ethernet component interface abstract away low level details of the Ethernet protocol. Instead, only Ethernet frames in the form of packet messages are exposed that carry the payload alongside the packet length. This interface is simpler than the PCI interface as no completion messages or similar are generated. In this case this is not necessary because this information is implicitly captured through the notion of packets. In case e.g. a NIC receives a packet message it receives the packet alongside it, so there is no need to create extra completion messages for that purpose.

The send and receive events present in the PCIe case and the fact that this information is encapsulated in Ethernet packets anyways, are important for the work presented here for the purpose of trace context propagation. More on that in Section 3.6.2.

SimBricks Time Synchronization

SimBricks provides the option to enable time synchronization among simulators without the need for a global synchronization mechanism. The key to this synchronization is that SimBricks enforces message processing times. Within SimBricks all communication between different simulator instances is explicit through message passing. These messages are passed between SimBricks adapters. These adapters are connected point to point at the natural boundaries mentioned beforehand. For this reason it is enough to guarantee that messages are processed at the correct time to ensure synchronization across component boundaries. This is enforced because message senders must tag messages with the time a recipient needs to process them. As SimBricks message passing is point-to-point and statically determined by the simulation structure, it is enough for SimBricks to implement pairwise synchronization.

The messages send through these point-to-point channels have monotonic timestamps and are delivered strictly in order. SimBricks uses the fact that a connection between simulators incurs a propagation latency $\Delta_i > 0$. If a message is sent at time T over interface i, it will arrive at $T + \Delta_i$. Therefore, message timestamps in a channel are monotonic assuming that the attached simulators have monotonic clocks. From this, it follows that no messages with timestamps $< t$ will arrive at that channel in case a message with timestamp t was already received. This allows a simulator to advance its clock to T in case it receives messages with timestamps $\leq T$ from all its peers. With this no further coordination is needed. To ensure liveness however, SimBricks introduces synchronization messages that are sent by simulators if they didn't send any messages for $\delta_i \leq \Delta_i$ time units. This allows simulators to make progress even if no data transfer messages are exchanged.

The option to enable time synchronization across simulator boundaries is important for the work presented here as well. More on that in Section 3.1.

Chapter 3
Design

The work presented here proposes Columbo, a framework that allows systems to be run in modular simulations and individual simulator logs generated during a simulation to be combined into detailed full system traces. Columbo aims to allow doing so while achieving the following goals:

1. **Modularity.** Columbo must be modular to allow easy integration of new components and easy adaptation to changes within already integrated components. The framework makes use of modular simulation, so it is important that the framework presented here is also modular. This is important because modular full system simulation allows the simulation of different components as well as the execution of unmodified operating systems. This allows different types of hardware, or more generally, systems, to be simulated. Columbo should reflect this in a way that it can easily adapt to the use of different simulators or simulator components.

2. **Non-Invasive.** Columbo should be non-invasive, i.e. it should not require any changes to the simulators used, as this would compromise the modularity and easy integration of new functionalities. The goal is to create end-to-end full system traces that establish causal relationships across component boundaries. However, modular full-system simulation allows new components to be added or existing components to be modified easily. The need to instrument simulators or simulator components for use with Columbo would be a tedious and error-prone task for large simulators.

   This means that no instrumentation should be required, as is often the case in classical distributed tracing environments, in order to establish causal connections between events of different simulators. Simulators that are capable of writing a log

file should be usable out-of-the-box, given that the necessary parts of the framework presented here, such as log file parsers, are implemented.

3. **Online.** Columbo must be able to process simulator log files online, i.e. as they are written. This prevents users from persisting large log files, because the information is read and processed while the simulation is still running. Simulations can create large log files. For this reason, it must be possible to run the framework in parallel with the simulation, so that traces can be created online while the simulation is still running. This also speeds up the process of obtaining traces in the first place, since there is no need to run the simulation and trace generation in succession. The ability to analyze traces online means that traces can be analyzed and filtered according to their importance, potentially reducing the amount of data that needs to be retained to capture a meaningful set of traces from a simulation.

4. **Visualization and Querying.** Columbo must allow easy visualization and analysis of traces. After all, end-to-end traces are useless if users cannot easily visualize and query them. The mechanism presented should also be flexible enough to allow users to easily switch to different visualization or querying-tools in the future.

Given these goals we begin the rest this chapter by giving a short overview of the general architecture of Columbo. Then we have a closer look at the general abstractions and building blocks used within Columbo.

3.1 Overall Architecture

As mentioned above, the framework presented here intends to use log files of a modular end-to-end simulation to create low-level end-to-end system traces. These traces should then be made available in a form that allows them to be visualized and analyzed. Conceptually, you can see what that looks like in Figure 3.1. This work focuses on the area framed in red. Simulators and tools for visualizing distributed traces already exist. For the latter, Columbo provides an abstraction that allows the export of traces to such external tools. Therefore, the following is conceptually what should happen: Log files are created by various components within a simulation, the framework presented here reads these log files, analyzes them and creates traces. These traces are then exported to an external tool for visualizing and querying the traces.

Since Columbo is primarily intended for use with SimBricks simulations, we can look at the simple network topology from the beginning of this thesis and see how it could be instantiated using SimBricks, as shown in Figure 3.2. We can see generally the same components as before: two hosts, two NICs and the network itself. When simulated with SimBricks, each of these components would be simulated by an individual simulator instance as shown in Figure 3.2. One could choose to use gem5 [29] to simulate the two

hosts running an unmodified version of Linux, a NIC i40e behavioral model to simulate the NICs and ns3 [5] to simulate a simple network made out of switches. Each of these simulator instances would then be instructed to write its own log file during an end-to-end simulation. These individual log files will be the input data for the framework presented in this work. If a Simulator is not already able to write such a log file, it must be extended by the ability to provide information about the simulation in the form of a log file in order to be usable by this framework.

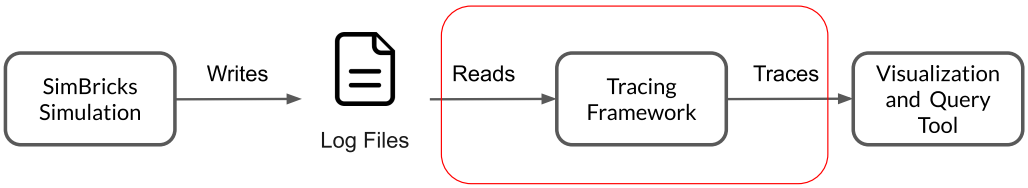


Figure 3.1: Conceptual overview on Columbo relative to simulation and visualization

The log files written by the simulators contain detailed information about the actions performed by the respective simulators at a specific point in time. An important aspect is that SimBricks provides temporal synchronization as described in Section 2.2.1. That means the timestamps attached to actions within the different log files can be ordered globally across the simulator boundaries. This ensures essentially that if an action A in a simulator causes another action B within another simulator, the simulated timestamp of action A is really smaller than the simulated timestamp of action B. This means that when we look at the timing information provided by the simulation, we can be sure that action A really happened before action B. This is important as this work will later make use of this property to implicitly propagate context information to form a trace.

To make use of these log files, users of the framework need to provide a Tracing Script (see Section 4.7.1 for more details). In such a script, a user basically defines how the various tracing components provided by Columbo must be assembled to create traces. This is necessary as the framework itself does not automatically know of the simulated topology and the specific simulators used. A high level view on what a user has to define can be seen in Figure 3.3.

As you can see, a user needs to specify a simulator-specific pipeline, which in turn needs to be connected to a tracer (Section 3.7), which manages the lifetime of the traces and spans created by the pipelines. The tracer itself must be connected to an exporter (Section 3.8), which manages the export of spans (Section 3.3) and traces to external tools for visualization and analysis. The simulator specific pipelines are important for multiple reasons. The log files being written by the simulators used look different depending on the simulators selected. This makes sense as the log file of a NIC simulator is expected to look different from that of a host simulator. As these log files are used to create event

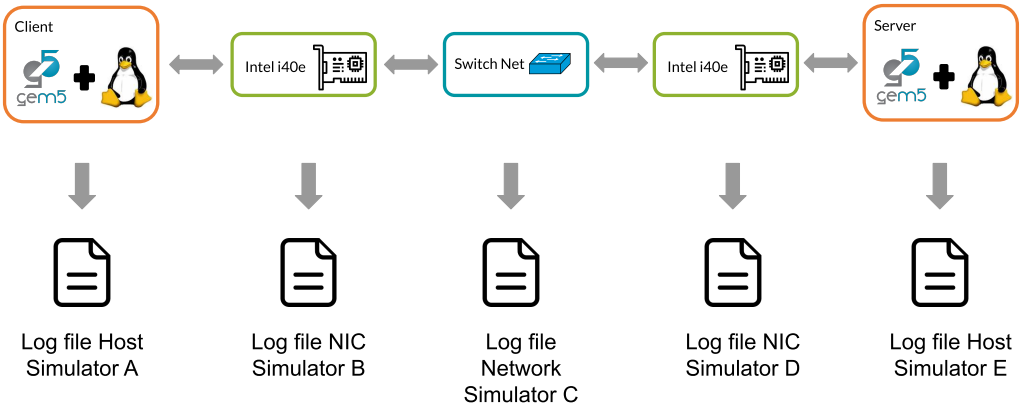


Figure 3.2: Example for a simple network topology using simulation to write multiple log files

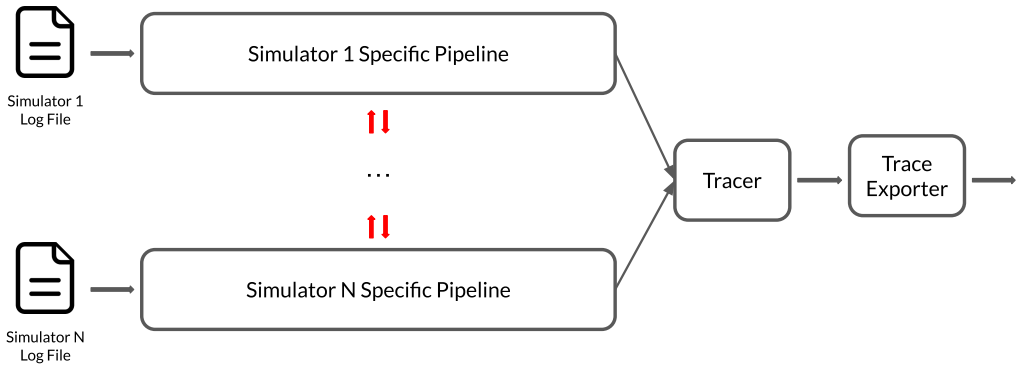


Figure 3.3: Conceptual overview of Columbo's structure used for creating Traces and causal connections for up to N simulators

(Section 3.2) streams, the resulting stream will look different depending on the simulator used. Spans will also look different depending on the simulator used, as they are created from event streams. Therefore, users have to specify simulator agnostic pipelines that contain several and individual stages that handle the parsing of the log files, filtering of the resulting events and the creation of spans. The creation of spans is handled by a component called spanner (Section 3.6) which will be the last stage of each simulator specific pipeline. Once a span is complete a spanner passes a span to the tracer, which then passes it on to the exporter, which sends it to external tools to make it available for analysis by framework users. Causal connections between Spans are also made by Spanners that communicate across pipeline boundaries to send or receive trace context (Section 3.4). All these pipelines will run alongside the actual simulation.

Now that we have established a general overview on how Columbo connects to simulations, we will focus the rest of this chapter on having a closer look at the general abstractions and assumptions that were used in the context of this work. That means the different building blocks used and how these are connected to each other are discussed. Most importantly will we see what a trace context 3.4 is in the context of this work and how it is propagated to causally connect spans 3.3 to form a trace 3.5. Then we will have a look at the tracer 3.7 and the exporter 3.8 abstractions that will handle the trace 3.5 lifetime and allow exporting traces 3.5 to other tools. This is essential for making use of the excellent ecosystem around distributed tracing and hence benefiting from years of prior work in this field. Finally, we will have a look at how to trace "online", meaning to process the simulator log output while the simulation is running and not once it is finished on the resulting log files.

3.2 Events

Events are the smallest units used to create a trace. Within this work they represent operations within a simulator. As a result, they are collected by parsing the event logs generated during modular full system simulation. To do so the simulators participating within such a simulation must be instructed to write these log files as described in Section 3.1. Important is that the events one is interested in are logged. What is of interest depends on the individual application. In general, it can be anything from function calls to invalidating a single cache line.

As a result of the decision to use log files to gain insight into the simulations, there is no need to further instrument any simulators if they already provide the information of interest. Since events are created from these log files, this work provides simulator-specific parsers that generate events by parsing a log file written by the particular simulator the respective parser supports.

Listing (3.1) Example log file entry from *Gem5* [29] representing a call to the function *i40e_lan_xmit_frame* from the Intel I40e device driver

```
6018966965250: system.switch_cpus: T0 : 0xffffffffa002507b
       : push     r15
```

Listing (3.2) Example log file entry from the NIC behavioral model included within *SimBricks* [44] representing the NIC issuing a DMA operation

```
info: main_time = 6018967679249: nicbm: issuing dma op 0
    x55ea69870540 addr 100240060 len 16 pending 0
```

Figure 3.4: Example log file entries that represent individual events

Such events or actions can for example be a host simulator logging that a function was called in the OS kernel as shown in Listing 3.1. Another possibility for an event could be that a simulator generates log output because the corresponding simulator has requested a mmio write operation or is issuing a DMA operation as shown in Listing 3.2. Typically, events correspond to a single line in the log output of a simulator and therefore represent a small operation, although this is not mandatory.

The most important property of an event is the timestamp that each event must have. These must also come from the simulators used. It is important that an event only has a single timestamp and not several. For this reason, an event only has a single point in time when it happened, i.e. it is to a certain extent instantaneous and has no duration, at least from a Trace's perspective.

3.3 Spans

Spans are the next larger unit for creating traces. At their core, spans are a collection of events (Section 3.2). More precisely, they are a set of events ordered by timestamps. For this reason, a span can contain several events of the same type, as long as they differ in a characteristic such as the time at which they occurred.

For example, a span can be the set of events within a host simulator that the corresponding simulator generates when simulating a mmio write operation. Another example is a span that contains all the events that represent a function call within the kernel during a syscall.

Spans therefore represent a logical operation or form a group of logically related events or actions that were generated during the simulation within a simulator. It is important to note that all events within a span must have been occurred within the same simulator. As such, spans on their own do not carry information across simulator boundaries but can give some contextual inside on what happened logically inside a simulator during a certain time.

Spans do have a duration. This duration is determined by the timestamps of the first and the last event inside a span. That means the duration of a span is given by

$$duration = |timestamp_{last} - timestamp_{first}|.$$

Additionally do spans contain a trace context (Section 3.4). The trace context does contain information about the trace a span belongs to as well as information about causal connections of spans with each other. These causal connections form a trace. More on this within the sections on trace context 3.4 and trace 3.5.

3.4 Trace Context

The trace context is a simple building block used by spans. Its purpose is to define the trace a span belongs to as well as to represent causal connections between different spans. To do this, a trace context stores a reference to the trace to which a span belongs as well as an optional reference to another span, namely the parent span. Especially the reference to a potential parent is of interest. Does a span not contain such a reference, the span is a trace starting span. That means it is the first logical set of actions that caused all other sets of actions within the respective simulator and potentially all other simulators that form a trace. Therefore, the trace context does causally connect spans. Accordingly, if a reference to a parent span is set within a spans trace context, it means that the respective span was caused by its parent. Note that a span can only have a single unique parent. When and how these causal connections are made is further discussed in Section 3.6.

3.5 Traces

As mentioned before do spans represent a logical operation. Traces on the other hand do encapsulate these spans and form a tree like structure. An example for this tree like structure can be seen in Figure 3.5. Each span within this structure belongs to the same trace. This tree like structure is created by setting the parent spans within the spans trace context accordingly. Note that a span can only have a single parent, but can have multiple children.A case where a span has multiple children can occur, for example, when a syscall, represented by a span containing all the function call events generated in the kernel during that syscall, causes multiple mmio write accesses to a peripheral device, which are represented by spans themselves. Each of the spans representing such a mmio write Access would have set the syscall span as parent within its respective trace context.

Sending a packet could therefore look like the trace shown in Figure 3.6. Here, the parent of the trace is a syscall that wants to send a network packet. To do this, it triggers a mmio write operation that writes to a register on the NIC connected to the

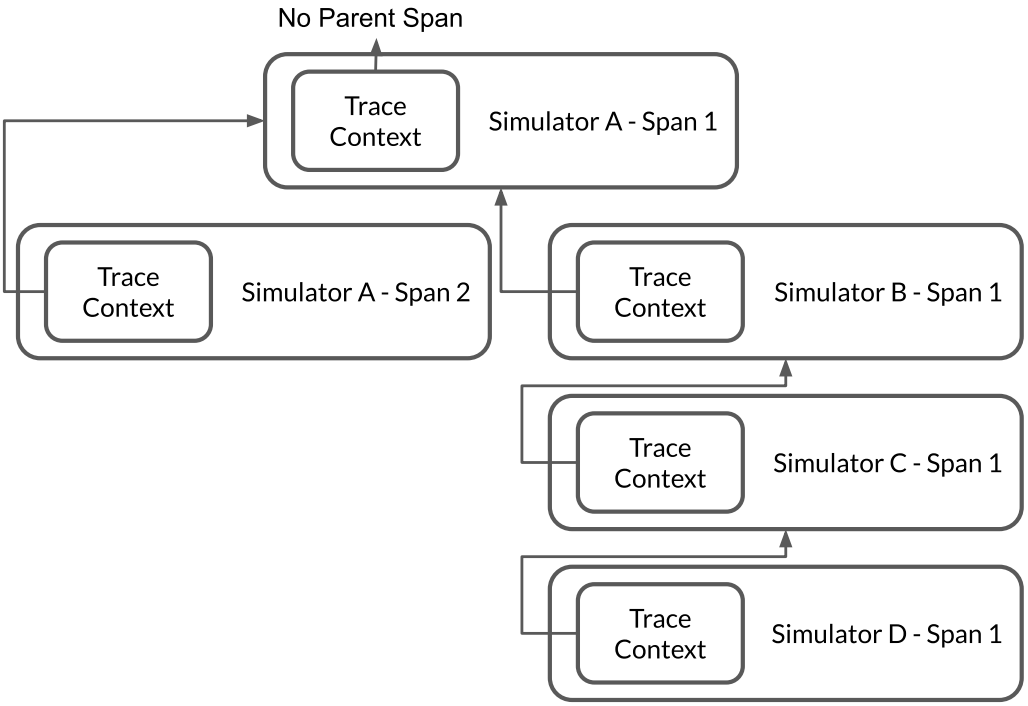


Figure 3.5: Abstract view on the tree like structure that forms a trace

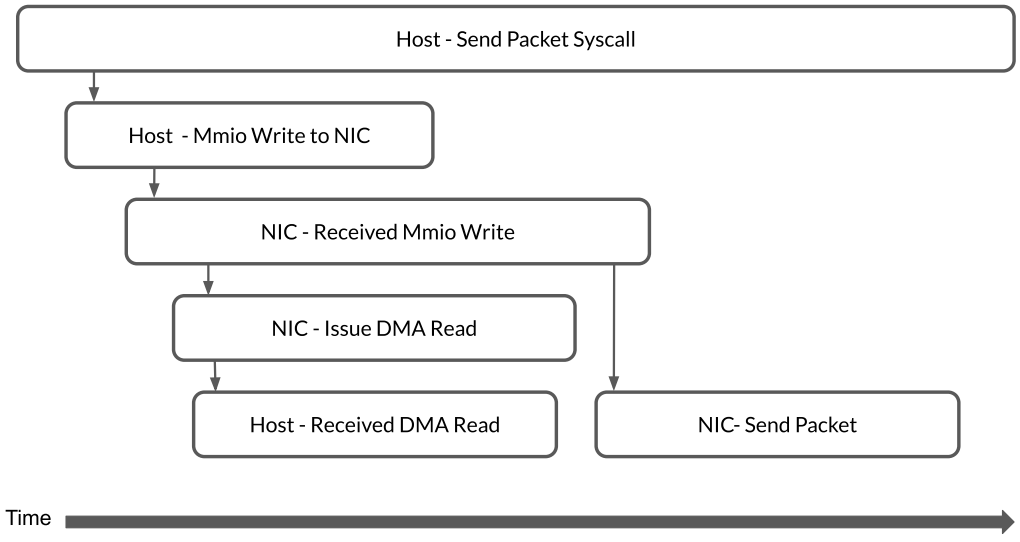


Figure 3.6: Abstract view on a trace. A host triggers to send a network packet and interacts with the NIC.

host. This mmio write operation is the first child of the syscall span. Even such a mmio write operation is a logical block, which can also be found in the log messages of the host simulator, and is represented as a span. This in turn triggers a span in the NIC simulator, i.e. the confirmation that a mmio write access has been made to one of the NIC's registers. The NIC simulator then performs a DMA read to read the data to be sent from the host memory. This DMA read is a child span of the mmio write span on the NIC side. The NIC's DMA read access triggers another span on the host side, the host's acknowledgement that data has been read from memory. Finally, a "packet sent" span is generated on the NIC side, it's parent being the register write span coming from the NIC simulator itself.

3.6 Spanner

The spanner is the most important component when it comes to the creation of traces. spanners are responsible for the creation of spans, and they are also responsible for the correct propagation of trace context, so that spans end up pointing to their respective parents using their trace context.

3.6.1 Span Creation

As mentioned above, one of the tasks of a spanner is to create spans. Spans are an ordered set of events that represent a logical operation. Therefore, a spanner uses an event stream as input to create spans. The event stream is created by parsing a simulator log file. What an event stream looks like depends heavily on the simulator and the configuration of the respective simulators logging capabilities.

Such event streams naturally may look very different depending on the log file the event stream is created from. This makes sense as for example a Simulator for a NIC generates in general a different event stream than a simulator for a host. Therefore, spanners are specialized to a certain degree to the event stream that they receive as input, which in turn depends on the simulator type from whose log file the event stream is created. Theoretically, however, it may also be necessary to create specialized spanners for two different simulators of the same type, depending on the event stream created by parsing the corresponding log file. This could for example be the case for two different host simulators that differ in the amount of information they provide during a simulation. As a result of that, spanners may also be specialized depending on the simulator itself and not only the simulator type. Do similar simulators create a similar or equivalent event stream, the same spanner may be used for both simulators.

Given an event stream, a spanner decides, depending on an event it sees, whether this event creates a new span or whether the event must be added to an already existing span.

This logic is encapsulated within a spanner and again depends on the simulator from which the events are generated and the simulator type of that respective simulator.

3.6.2 Causal Connections

Another task of spanners is it to create causal connections. These causal connections are represented by references from spans to other spans using their respective trace context. The main difference considering the creation of distributed traces compared to a classic setting arises primarily from the fact that in this work simulation and in particular SimBricks simulations are harnessed to create such traces. This choice has implications on the propagation of the trace context. The difficulty lies in the fact that SimBricks can be used to link different simulators together to create large end-to-end simulations. An important aspect is that new simulators can be extended with little effort so that they can be used within SimBricks in the future [44]. This property should be retained in this work. This is one reason why events in different simulators cannot simply be causally linked to each other by passing on trace context explicitly through all components, as it is often done in classical settings (see Metadata Propagation in Section 2.1.1). The problem is that such an approach would result in a substantial implementation effort, since new simulators would have to be extended at all points of interest in order to propagate such a context. Another problem is that attaching context to e.g. network packets might increase the simulated time, thus having an unwanted impact, undermining one of the key advantages of simulation compared to physical testbeds, namely not interfering with the simulated system while providing deep visibility.

An important aspect of being able to create such references is that spanners must know whether a span was triggered by an operation i.e. span within another simulator or not. For example, a spanner that creates spans for the event stream of a NIC simulator must know that a span that bundles mmio write related events must have been caused by an operation i.e. span of a host simulator that issued that mmio write. Another example would be the transmission of a packet through a simulated network. Such a packet must have been transmitted i.e. put on the wire by a NIC. These boundaries do also exist in real operating systems and hence these mechanisms work in the real world the same way [27, 26]. As mentioned in Section 2.2.1 does SimBricks also use these boundaries to interchange messages between simulator instances. Therefore, trace context must be propagated always when these natural interfaces/boundaries between the simulators or in general devices are involved. Columbo makes use of this property and treats the messages SimBricks is exchanging like Start- and end-events (see Inferring Relationships in Section 2.1.1) to know when to propagate trace context. To propagate the trace context a spanner is instantiated for each simulator involved. The actual propagation than happens through a message passing interface between the different spanners. An abstract illustration can be seen in Figures 4.1 and 4.2a. Following the rule that the

need for trace context propagation arises though natural boundaries involved we can distinguish two general cases in which trace context propagation becomes relevant for a spanner:

1. A span must trigger an operation i.e. span within another simulator. Given this case a spanner uses the message passing interface in between the different spanner instances to pass on a reference to the span from which the spanner knows it must cause actions within another simulator. An example for this is a host simulator that issues a mmio write operation to a device register inside an attached NIC. Given such a case, the Spanner responsible for the host simulator must create a span to reflect the operation happening inside the host itself as well as passing on a reference to that specific span to the spanner responsible for the respective NIC simulator. As SimBricks has to send an event to the other Simulator in that case as well (Section 2.2.1), we can easily make use of this property and log that information if not already present within the SimBricks adapter (that must be created to use a simulator alongside SimBricks) in such a case to ensure the respective spanner will see that an e.g. mmio write access was issued.

2. A span must have been triggered by an operation i.e. span within another simulator. Given this case a spanner polls in a blocking fashion from the message passing interface to receive a trace context i.e. reference to the Span that must have been caused the span the current spanner is currently creating. An example for this is a spanner working on a host simulators event stream. Given the case the spanner creates a span related to a DMA access made by an attached device like the NIC. In such a situation the spanner would poll to receive a trace context from the NIC's spanner as the NIC must have been issued that operation i.e. must have created a span on their respective side beforehand. Given the first case this also means the NIC's spanner must have been pushed context to that respective span to the host's spanner. Similar to the first case, SimBricks must have been sent a message to an adapter in such a case as well, that way one could also easily extend the SimBricks adapter on receiver side to log the information if not already present to ensure the respective spanner will see that an e.g. DMA read access was issued.

The above two cases reflect how context propagation works within the presented framework. Note, however, that the above approach works mainly due to the fact that *SimBricks* provides time synchronization. That way a spanner can just push or poll for a trace context when in need and be sure that the span they push or receive has happened before the span they want to make a connection to, this is especially important as it ensures that this also holds from the perspective of simulated time which makes timing information in the resulting traces valid.

3.7 Tracer

The tracer is a simple component. On the one hand, it provides an interface that spanners use to create spans in a simple way. During this creation the tracer ensures that the trace context is correctly set within the spans that are to be created. The context itself must be provided by the spanners respectively. One reason for this design choice is that it reduces the complexity within spanners. On the other hand, the tracer also manages the lifetime of spans. Once a span is finished, i.e. no more events are added to that span, the tracer releases the spans for export and passes them on to the exporter 3.8. Once a span has been exported, the tracer is responsible for releasing the memory of that span so that no memory is wasted.

3.8 Exporter

The exporter is Columbo's gateway to the outside world. It is an important component as it enables already existing tools to be used. It therefore makes it possible to decouple the creation of traces or spans in this particular setting from the analysis by tools of the distributed tracing community. To achieve this, the exporter enables the export of spans to other tools or systems. For this purpose, an exporter offers a simple interface for exporting individual spans. This interface is used by the tracer. The main task of the exporter is to convert spans from the internally used representation, i.e. Columbos internal representation of spans, into the format used by the corresponding target system. Once a span has been converted into the desired format, the span is sent to the desired target system in the new format. In principle, one could have chosen to use an existing format throughout the framework, but a reason for not doing so is that existing formats may not be suitable for the type of hardware that might be simulated. Therefore, the decision was made to introduce this translation layer to be independent of other trace formats. That way one can always plug in a new translation layer that reflects whatever needs a certain usecase might have.

3.9 Pipelines

The pipeline abstraction is used by Columbo to create an event stream and execute operations on it. It is a central building block of the framework presented here and is used to connect various other components with each other. In order to understand what a pipeline is in the context of this work is and how it works, we will first look at channels. Channels are crucial when it comes to enabling pipelines. Furthermore, are they used to allow for message passing as used by spanners to allow for trace context propagation 3.6.2. Once it is clear what channels are, we introduce the individual components that are

connected to form a pipeline. An example on how a pipeline, once all components are connected looks conceptually, can be seen in Figure 3.7.

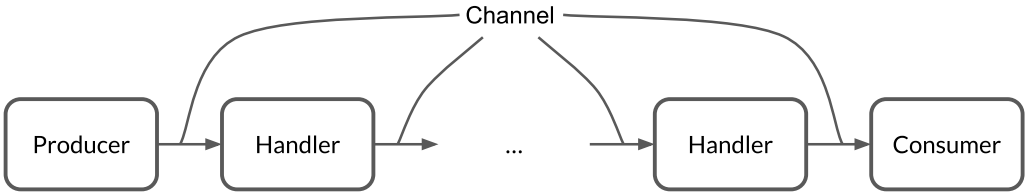


Figure 3.7: Abstract overview on what a pipeline is

3.9.1 Channels

The channels used by Columbo are a simple abstraction that allows messages/values to be sent from one process to another. Conceptually, the channels used by the framework are rather similar, but not equivalent to the channels known from the Go programming language [2]. In general messages/values can be anything. An abstract example on what channels are and how they work can be seen in Figure 3.8.

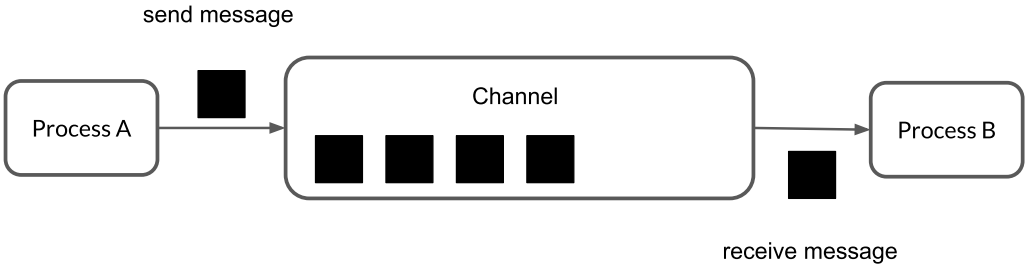


Figure 3.8: Abstract view on channels and how they work

Important properties of channels are the following:

1. Once instantiated, channels can only be used by a specific message/value type. However, this can be freely selected before instantiation.
2. Safe concurrent access. Channels allow pushing and pulling from them safely in concurrent scenarios.
3. Channels allow reading and writing messages/values to them in a blocking and non-blocking fashion.

4. A channel can either have a fixed or infinite capacity.

Columbo makes use of channels in many places. Generally spoken are channels used to pass on messages/values between the different components that form a pipeline. This can be seen in Figure 3.7. In addition, are channels used to propagate trace context between spanner instances (more about this in Section 3.6.2).

3.9.2 Producer

Producers are the first component of a pipeline. They are the starting point of such a pipeline and are responsible for producing values. More specifically are they responsible for producing a sequence of values. These values are passed on to the next pipeline components using Channels.

In the context of this work, producers are usually parsers for the log files of individual Simulators or more specifically wrappers around these parsers. Producers a.k.a. parsers are therefore used within Columbo to generate an event stream on which the pipeline operates.

3.9.3 Handler

Handlers are intermediate steps within a pipeline. They each have an input Channel and an output Channel. From the input Channel they do receive a sequence of values on which they perform an operation. Once the operation to be performed has been completed, values of the same type are passed on to the next stage in the pipeline. Therefore, handlers cannot transform the value sequence with values of type A, which they receive as input, into an output sequence of values with type B. In general, a pipeline can have any number of such handlers as intermediate steps. This is shown in Figure 3.7.

In the context of this thesis, handlers are used to filter event streams, created by a simulator log file parser, to discard unwanted events. They are also used to write event streams to separate files, before the rest of the Pipeline is executed. This can for example be very useful for post-processing the event stream a simulator produces or for debugging purposes.

3.9.4 Consumer

Consumers are the last component of a pipeline. As the name suggests, do they "consume" the value sequence that they receive through an input Channel from the preceding pipeline stages. They do not have an output Channel. Therefore, they cannot pass on the value sequence and the pipeline ends i.e. that a Consumer is a pipelines sink as shown in Figure 3.7.

In the framework presented, consumers are for example a spanner. As mentioned do these use an event stream to create spans. Naturally it would not make sense for them to pass on events any further. Another possibility for a Consumer is a component that writes an event stream similar to the previously mentioned handler into a file with the difference that the events written into the file are not passed on.

Generally do producers, handlers and consumers act as independent tasks. As a result do pipelines allow to suspend the execution of, for example, a consumer in case that consumer issued a blocking operation. While waiting for that operation a pipeline is allowed to resume the execution of e.g. a handler while it is waiting for the consumers blocking operation to be ready. This can be useful if e.g. a spanner pulls a trace context in a blocking way while still allowing the pipeline as a whole to make progress by parsing new events.

3.10 Online Tracing

As already mentioned does Columbo intend to use log files generated by the simulators used in SimBricks simulations to create low-level distributed Traces. One problem with this approach is that these log files can become very large, i.e. several hundred GBs of log file data for a single simulator depending on the simulator and its configuration. An example for this can be found in Section 5.3. Such large log files can be a problem because as a user might simply not have enough disk space available to store them for post-processing. In addition, such simulations can take a long time. On top of that, a reasonable amount of time would be spent on the actual post-processing to create the traces.

For these reasons, the framework offers the creation of traces "online", i.e. traces can be created while the *SimBricks* simulation is still running. This is made possible with the help of "Named Pipes". These are special FIFO files that exist as entities in the file system. These can be read or written by any process like a normal file. The FIFO semantic of these files is special because the bytes that are written first are also read first [15]. Therefore, these named pipes can be given as a log file path to the *SimBricks* simulators so that they write the log output to these named pipes. Making use of this technique also allows for using the simulators out of the box without any changes. Another special feature of named pipe files is that they must be open at both ends [15]. Therefore, for every simulator process that writes to such a file, there must be a Columbo process that reads this file in parallel. That would be the various parser instances.

Chapter 4
Implementation

In Chapter 3 we have seen the abstract design and architecture of Columbo, as well as the general design of the building blocks used to created Low-Level End-to-End System-Traces. In the following we will revisit these components and debate their implementation in more detail. One will see that a lot of design choices were made to allow for easy extension of Columbo in the future as well as to minimize the risk of breaking any abstractions the framework makes when being extended in the future.

Once we have seen the basic building blocks we will discuss how one uses Columbo alongside SimBricks as well as the usage of tools coming from the distributed tracing community. These tools in particular allow to visualize and analyze distributed traces such that users of this framework can benefit from prior work done in the context of distributed tracing. The chapter will conclude by considering how a user could extend Columbo to support new simulators.

4.1 Basic Building Blocks

4.1.1 Events

events are implemented as simple subclasses of an event class. It is important that all events store a name, a timestamp and a type. In the actual implementation they also store a parser name and a parser identifier.

All these attributes inherit events from a common event superclass. Technically speaking are the parser identifier and the parser name not necessary from a functional perspective.

Therefore they could be stripped in builds optimized for runtime. They are however useful as additional information one can facilitate while debugging.

All concrete events, i.e. event subclasses, store in addition to the already mentioned attributes additional information that is important for the respective event. In the case of a mmio-write event, for example, these are a boolean flag whether the write event was a posted write, to which offset and to which base address register the mmio-write was written to. In the case of a function call, additional information includes, for example, the address and the name of a kernel function that was called inside a host simulator given that Columbo was able to resolve the function name given the address of the function.

An important aspect special to function call events on host simulator side is the process of determining function names. To do so the framework does not rely on the simulator used. Gem5, for example, would in principle be able to do this [12]. Instead, Columbo users must use *objdump* to generate symbol tables themselves and make them available to the framework. On the one hand, this enables very fine-grained control over which symbols are resolved by the framework. This can for example be used for filtering purposes. On the other hand does this design decision ensure that symbols can potentially still be resolved based on their address during the simulation, even if the simulator used does not offer this translation. Users make symbol tables available for the framework by setting paths to the files containing the symbol tables as well as a potential address offset within a configuration file. The offset is required to convert the relative addresses within a file created using *objdump* into actual addresses.

4.1.2 Spans

Similar to events, spans are implemented by inheritance. All spans have a unique identifier, a source id, a type, a vector of events that make up the span, and a trace context (Sections 3.4 and 4.1.3). Concrete spans inherit the trace context from the common abstract span superclass. In addition to these attributes do concrete span implementations store additional information that can be derived from the events that a span stores. In the case of a *NicDmaSpan*, this could be the information whether the DMA access represented by the span was a read or write access.

Another unique feature of the span implementation is that all spans must implement the method bool AddToSpan(**const** std::shared_ptr<Event> &event_ptr). This method encapsulates the logic for determining whether an event belongs to a span or not. This essentially ensures that the logic for deciding whether a particular event is really part of a span, and hence the appropriate sanity checks for it, is tied to the span definition itself.

For example, this method encapsulates the logic that a mmio-write event does not belong to a span representing a DMA access. Following the example of a DMA span, this method additionally ensures that a DMA issue event and a DMA access completed event, that

together form a span, really belong together by checking that the respective memory addresses that the events refer to match. These addresses are stored in the individual events. The decision to implement this logic in the spans themselves makes sense. The corresponding properties to be checked depend heavily on the respective span and the events such a span is supposed to bundle. This logic is therefore linked to the specific span.

In addition, this design choice reduces the complexity of spanners, as they can simply add events to a span using the above method. If that method fails, spanners can simply fall back to create a new span. This drastically simplifies the actual implementation of spanners.

4.1.3 Trace Context

Columbo distinguishes between two different trace context implementations.

1. The first implementation is used to store references between spans that describe the tree-like structure that forms a trace, as illustrated in Figure 3.5. That trace context is implemented as a very simple component. It stores an identifier for itself, the trace it belongs to, and a parent, together with a flag indicating whether the parent identifier is set, and a timestamp indicating when the parent started. In particular, the timestamp is not a technical necessity, but allows for sanity checks within spans to ensure that a spans parent is not set to point to a span that occurred after the span in which that parent is set. As it is not a necessity, it may be removed in future versions of Columbo.

2. The second implementation is used for propagating causal connections between spanner instances. It also stores a trace identifier, as well as a parent identifier and a corresponding boolean flag indicating whether that parent identifier is present or not, and a parent timestamp, just like the other trace context implementation. An important difference is that this second context is immutable, to reduce the risk of making errors, which is not the case with the first trace context implementation.

The motivation for making this distinction is to allow Columbo to easily specialize or extend these implementations depending on concrete needs in the future, e.g. if the trace context used for context propagation across spanner instances needs some additional information that would not be needed for just describing a trace tree structure.

4.1.4 Traces

Traces are in principle implicitly described through references from spans to other spans using the trace context that every span stores respectively. The framework, however,

explicitly implements them as a set of spans. Trace instances are managed by the tracer component, which is described below.

The reason for implementing traces as an actual component is that Columbo cannot immediately decide for certain spans whether they start a trace or not. Given a span that represents a syscall. This span will store all function calls made within the kernel during that syscall. This is implemented inside a spanner which creates a new span when it encounters the first function call event of a syscall. The spanner would then continue adding function call events to that span till that respective syscall is completed or a new syscall is started. The problem is, that when creating the span given the first event it can be unclear whether the Syscall will read a received packet or not. If the syscall reads such a packet, it should be connected to a packet-receiving span created by a Nic spanner. If the syscall does not read such a network packet, it is a trace starting span. To deal with this case, such a span is initially always treated as a trace starting span. Does Columbo however detect (while processing events belonging to that span) that the syscall does indeed read a packet, it will alter that spans trace context by setting the traces identifier accordingly, as well as the respective parent span identifier. This is also the reason why the first trace Context implementation (see Section 4.1.3) is not immutable like its counterpart. However, this means that the trace identifier of all potential children of that particular span must also be changed. To easily find all these child-spans the framework maintains the Tratracece data structure explicitly. This is done by the tracer component.

4.2 Spanner

Like the components above, spanners are implemented using inheritance. The idea is that spanners implement individual *handler-methods* which in turn take care of handling events of a certain type. An example of such a *handler-method* is a method within a *host-spanner* that would handle all events that from a host simulator that are somehow related to a mmio access of any kind.

The spanner superclass implements the consumer interface (see Section 3.9) and calls the *handler-methods* specified by the subclasses, depending on the type of an incoming event. The *handler-method* of the subclass that is called takes care of span creation and trace context propagation across spanner/Simulator boundaries.

This design generally allows all events from a specific simulator, that are related to the same type of span, to be handled by a particular *handler-method*, as shown in the example above. This enables easy extension of spanners in the future. It also means that the complexity of handling such events is distributed over small, independently maintainable chunks of code within a spanner implementation.

Since spans, as mentioned in Section 4.1.2, implement an interface and a corresponding method that decides whether an event belongs to a span or not, the implementation of *handler-methods* essentially boils down to two things:

1. The creation of a new span in case an event must start a new one, e.g. when a DMA access is issued that has not yet been seen. In practice this usually means that the spanner was unable to add an event to an already existing span, or that there was no existing span for that type of events.

2. The polling of a trace context if a span/event must have been triggered by another simulator or the propagation/sending of a trace context to another spanner in case a span/event must have triggered actions within another simulator. The polling is done in a blocking manner by spanners as soon as it becomes evident that a span must have been caused by another simulator.

This design enables the effortless addition of new spanner definitions to Columbo or to easily alter already existing spanners in the future.

Another important aspect with regard to polling the trace context is that all *handler-methods* are implemented as coroutines. This enables the spanners *handler-methods* to be suspended in case a trace context must be polled from another spanner through one of Columbo's channels. This polling may be blocked if the corresponding other spanner, from which the trace context needs to be polled, has not yet made this trace context available. Such a scenario may arise where one spanner is required to handle a much larger number of events than another spanner, depending on the simulators to which they are attached. In such a situation, a spanner may need to process numerous events until it can be sure that the trace context needs to be propagated, causing an attached spanners poll to be blocked until the trace context is propagated. This can happen, for example, if a spanner responsible for a NIC is waiting for trace context from a spanner responsible for a host as a result of a mmio access to a NIC register. This is because the spanner responsible for the host will typically have to process considerably more events before it sees the events associated with the mmio access and knows that trace context needs to be propagated. The actual suspension of a blocking process, when polling trace context, is handled automatically by the channels provided by Columbo via their *poll method*. See Section 4.5 for more details.

4.3 Tracer

The tracer is a component that manages the lifespan of traces and, more specifically, spans. A tracer instance is shared by all spanner instances when tracing. The spanner uses the tracer to create traces for which the tracer offers a factory-like interface. This allows the tracer to add the span to an existing trace as soon as it is created, or to create

a new trace using the span. This makes it easy to find spans that are related by the trace they are contained in, should their trace context need to be changed at a later stage, as described in Section 4.1.4. For the altering itself the tracer also provides a method. Therefore, this complexity is hidden from the spanners which just need to use that method in case necessary.

Tracers do also hand finished spans over to an exporter. One challenge when exporting spans is that, depending on the chosen exporter and the underlying protocol/framework, it is not possible to modify spans that have already been exported. As the tracer is responsible for passing spans to an exporter once they were marked as finished by a spanner, the tracer component ensures that spans are only exported once their parent was exported. By doing so, spans will only be exported once Columbo ensured they do not need to be altered afterwards anymore (described in Section 4.1.4). The tracer enforces this by requiring Spanners to explicitly finish spans by calling a method offered by the tracer. Once a span is marked as finished, the tracer uses simple waiting lists as a data structure to check if a span can be exported. If yes it is exported and if not it is put on the waiting list till it can be exported safely.

4.4 Exporter

Exporters implement a simple interface that allows the export of finished spans. Since their task is to export such spans, they must on the one hand handle the sending of such spans to an external tool. On the other hand they must also manage the conversion of spans from Columbos internal format into the format the respective external tool expects/requires.

Columbo currently implements two exporters. One *NOP-exporter* that serves no purpose other than debugging. The second exporter implements an *OTLP-exporter* which converts spans into the Opentelemetry format and uses the OpenTelemetry protocol (OTLP) to export spans to other external tools. Tools supporting this format are for example the *OpenTelemetry Collector* [6], *Jaeger* [4], *Zipkin* [7] or *Grafana* [3]. This enables users to easily benefit from years of prior work.

The overall setup, when utilizing the framework presented here with its *OTLP-exporter* in conjunction with external tools, is described in greater detail in Section 4.7.2.

The choice of the OTLP protocol in the presented framework has one disadvantage for the specific use case of creating low-level end-to-end system traces. Assuming a simulated server receives two distinct packets from a client which were triggered by different unrelated syscalls on the client side. Columbo would create a trace for both of these syscalls respectively. The resulting trace starting syscall spans would each be linked to all the spans they caused in e.g. the NIC. If both packets are, however, read/processed within the same syscall on the server side, that reading syscall span on the server side

would theoretically have two parent spans, namely the two syscall spans on client side. This is not permitted by the OTLP protocol. In order to be able to handle such situations nevertheless, the presented framework creates as many copies of a span as it has parent spans. Afterwards each copy is linked to exactly one of these parent spans. Furthermore, are such span copies labelled as copies such that Columbo users can still distinguish that copies have been made in certain situations.

4.5 Pipelines

As stated in Section 3.9, pipelines are a central component of Columbo. From an implementations point of view are pipelines just a simple wrapper around a producer, a consumer and a set of handlers. To start an actual pipeline the framework provides simple methods that take such a pipeline wrapper as an argument. These methods will handle the proper creation of channels that connect the different components as well as the execution of the pipeline.

An important feature is that all steps of a pipeline, as well as the reading and writing of channels that connect the individual components of a pipeline, are executed in coroutines. This makes it possible for Columbo to suspend such steps if, for example, it is necessary to wait for disc I/O or if a method within a pipeline's producer, consumer or handler attempts to read from an empty channel in a blocking manner or attempts to write to a full channel in a blocking manner. This allows Columbo to treat these methods/coroutines as tasks which are processed by a central thread pool. The thread pool is created automatically by the framework. Users can specify the number of threads that such a thread pool should have through a configuration file. Within this thread pool the individual tasks of the pipeline are written to a work queue, which is processed by that thread pool. If a task is suspended, it is considered processed for the moment so that the thread pool can move on to process the next task. If a task is ready to be resumed again because, for example, there is now space in a channel to write a value to or a value is now available in a previously empty channel, the previously suspended task is added back to the work queue of the thread pool. This allows Columbo to utilize resources efficiently. This is useful, for example, if it is necessary to wait in a blocking fashion for a trace context until it is made available by another spanner. The framework, however, also allows to choose a worker thread model in which case Columbo will automatically create a new thread for each step within a pipeline. In this case potentially more threads are started than in the case described before as no thread pool is created, and the coroutines will each run in a dedicated worker thread. Offering this is no necessity but a choice of freedom the framework offers for users to choose a threading model that fits their individual needs best.

4.5.1 Channel

As already described in Section 3.9.1, channels serve as interface between the individual components of a pipeline. In addition, are channels used during tracing to propagate trace context between spanner instances.

Columbo provides two specific types of channels:

1. Channel with a fixed capacity. These channels implement a ring buffer. These channels with a fixed capacity are used by Columbo to transfer data between pipeline components. In concrete terms, this means that they are used during tracing to send events from a log file parser to a spanner for processing. In the case of passing events through the different stages of a pipeline, Columbo uses channels with fixed capacity. This ensures that once a channel is full, the subsequent pipeline steps are forced to be executed. In case many events are generated while tracing, using channels that have a limited capacity also ensures that memory consumption is limited.

2. Channel with unlimited capacity. These channels store data which is "in flight" inside a linked list without a fixed capacity. Columbo uses these channels to send trace context between individual spanner instances, in order to propagate trace context. In this instance, channels without a fixed capacity are utilized, as trace context is not typically propagated with great frequency in relation to the number of events the framework must handle during tracing. As such, memory usage is not a significant problem in that case. Using channels with unlimited capacity, however, allows spanners to avoid blocking when pushing trace context into such a channel. This allows them to immediately continue to process events after pushing to such a channel.

In both of these channel implementations, the act of reading and writing from and to a channel is realized through the invocation of a coroutine. Similar to the individual steps in a pipeline, these coroutines are processed as tasks by the pipelines thread pool. This allows to suspend the respective coroutine when writing to a full channel or reading from an empty channel. This implies that the associated reading or writing task is temporarily halted in order for the executing thread to potentially process other tasks concurrently. When a task now writes to a channel, any suspended tasks are notified that data is now available. This allows suspended tasks waiting for data to be resumed. Similarly, tasks that wait for data to be read from a channel are notified and resumed once another task reads from the same channel.

Furthermore, all channel implementations are thread-safe and can be used concurrently by multiple readers and writers.

4.5.2 Producer, Handler and Consumer

Producer, handler and consumer are simple interfaces that classes must implement if they are intended to be used within a pipeline. It is also important that the respective methods must be implemented as coroutines, enabling them to be suspended. This can be useful if, for example, a task has to wait for disc I/O. In this case the respective thread of execution can take over other tasks while the corresponding coroutine is suspended waiting for the disc operation to finish.

In the framework presented, for example, a component that provides an event stream is a producer. handlers are typically used as filters to remove unwanted events from the event stream on which a pipeline operates within Columbo. An example of a consumer are spanners that use an event stream, i.e. consume an event stream to create spans.

4.5.3 Online Tracing

As already described in Section 3.10, Columbo utilizes linux named pipes to create traces while the simulators used are still running the actual simulation.

From an implementations point of view, named pipes can be opened and read like normal files. Therefore, no changes are necessary within the simulators used. From the perspective of the framework, there are a few interesting aspects:

1. Named pipes require a reading and a writing process. If this is not the case, opening a named pipe file will cause the program to block. Therefore, it is crucial to start the tracing framework simultaneously with the simulation. In this situation Columbo serves as the reading process, while the simulation serves as the writing process. This ensures that the simulators used can make progress. In addition, when for example using the Gem5 simulator, SimBricks allows you to create checkpoints to speed up the simulation. In this case the SimBricks simulation will attempt to open the named pipes multiple times. Therefore, the processes on the tracing framework side need to be started twice in such a case. The reason for this is that after creating the checkpoint, SimBricks terminates the simulators and thus also closes the corresponding file handles for the named pipes, which in turn causes Columbo to stop as it appears that the simulation is finished.

2. Named pipes have a limited capacity. If they are full of data, write accesses to them are blocked until another process has read enough data from a named pipe until the write access can take place. Suppose the spanner used with a simulator that writes large amounts of data to the named pipe tries to block and poll from another spanner trace context. This polling operation causes the corresponding spanner coroutine to be suspended, as there is no trace Context yet available, thus ultimately stalling the pipeline. Therefore, no more data is read from the corresponding named pipe for

the moment. As a result, the corresponding simulator is also blocked. Blocking this simulator can now cause SimBricks synchronization messages not to be sent as the simulator tries to write data to the named pipe first. This could now cause another simulator to stop progressing because it is waiting for a synchronization message from the blocked simulator. This in turn could cause the simulator's pipeline to stop progressing because the simulator is not writing data to the corresponding named pipe because it is waiting for the synchronization message. This lack of progress could in turn cause the required trace context to never be generated, resulting in a deadlock.

To prevent this, Columbo reads data from the named pipes used. Parses events from the read data and stores these events in a buffer. The size of the buffer must be chosen large enough by the user so that the situation described above does not occur because there is always enough data processed such that the simulators used can make enough progress to send enough synchronization messages.

Another way to avoid this problem is to split the creation of traces into two phases:

(a) In the first phase, one starts the tracing pipeline such that the simulators event logs are read while simulating using named pipes. An event stream is generated, filtered and written into a file without invoking any spanners. This allows for post-processing and creating spans after the actual simulation, using the event streams that were written to files. This is feasible as the pre-processed and filtered event stream is much smaller, compared to the size of the actual raw log output created by simulators (see Section 5.3).

(b) In the second phase, one would use the in the first phase created event stream files to start a pipeline as usual. This time, however, spanners are used to actually create spans. In this phase the possibility of a deadlock is gone as there are no more simulators involved that might block in an attempt to write to a named pipe.

4.6 Supported Simulators

Columbo uses modular full system simulation to generate low-level end-to-end system traces. More specifically, it uses SimBricks simulations. In this section we want to talk a bit about what changes to the supported simulators were necessary to enable tracing. Currently, three of the simulators supported by SimBricks are supported to the extent that tracing is possible. These are Gem5 [29], Ns3 [5] and a behavioral model for an Intel i40e NIC which is provided by SimBricks [44].

In addition to the components already mentioned, such as events and spans also minor changes had to be made to the already existing simulators to enable tracing with Columbo:

1. **Gem5:** Almost no changes were required within Gem5. Gem5 comes out-of-the-box with some command line parameters that can be used with the *Opt* version of Gem5, making Gem5 log more than enough information of interest to create traces [29, 12]. The only changes have been made to the SimBricks adapter of Gem5 to provide additional information in the case of a mmio write access. The additional information required was a flag indicating whether the access is a posted write or not and the base address register used when handling the mmio write. This information is now additionally logged.

2. **Intel i40e NIC:** The NIC behavioral model provided by SimBricks was already able to log all the information on the command line that was important for tracing with the framework presented here. The only changes that were made were to give the model an alternative command line option to write logging information into a file as an alternative to the command line.

3. **ns3:** ns3 provides trace sources that can be used to generate the log output required for use with Columbo. These trace sources can be used, for example, to signal that a packet has been sent or dropped. In order to use trace sources, an ns3 user must connect trace sinks to trace sources [5], which can react to events on trace sources. These trace sinks are used to log messages to a file. To make this possible, helper classes were implemented within ns3 during the course of this work. These allow ns3's config path system to be used to provide an easy way for users to connect trace sinks to trace sources in already existing ns3 components of interest, in order to log information about packets passing through a network simulated by ns3, more specifically a packet passing through the simulated components. In addition, SimBricks ns3 scripts have been adapted to use these helper classes to connect trace sinks to trace sources of interest. Further, additional trace sources have been added to the existing SimBricks adapters so that users of Columbo can also see in the traces when packets have traversed SimBricks adapters.

4.7 Framework Usage

In this section we describe how users would actually use Columbo. That is, we will look at what tracing scripts are in the context of this work and why they are necessary. Once we have looked at that, we will discuss what a possible setup might look like when actually exporting traces/spans to external tools, how in such an exemplary setup simple sampling can be applied to reduce the amount of data and how one could extract further metrics from the exported spans.

4.7.1 Tracing Scripts

In order to run a simulation users must provide a so-called *Tracing Script*. This is necessary because Columbo does not automatically know what the simulated topology looks like and which of the simulated components should be considered during tracing. This information must be provided by the user, i.e. a user must provide this information in the form of a *Tracing Script*. Specifically, this means that the user must define such a script in the form of a C++ program. Within this script, the user defines what the tracing pipeline(s) to be used should look like. Figure 4.1 provides an abstract example of what an end-to-end tracing pipeline for a single simulator looks like.

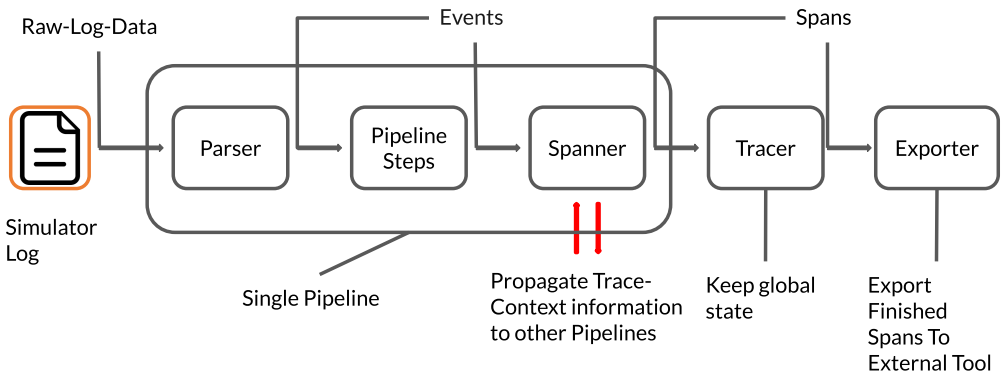
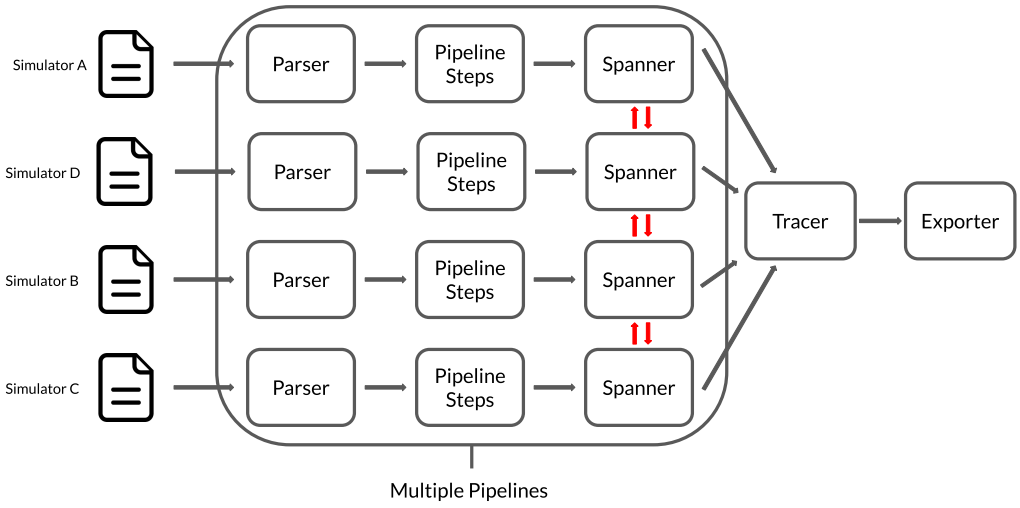


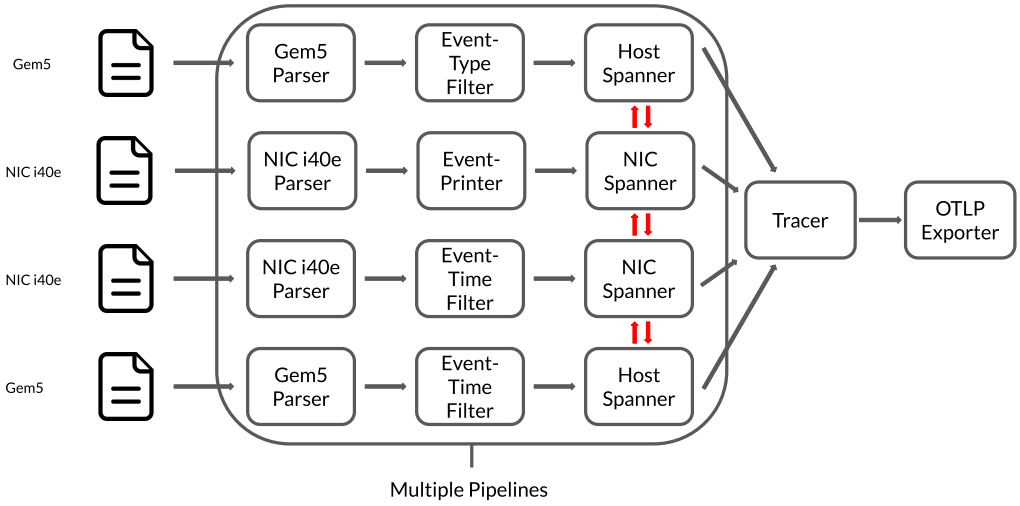
Figure 4.1: Abstract overview of the general composition of a pipeline for a single simulator

Columbo provides components that can be assembled like building blocks to describe the steps to be performed in a pipeline, making it easy for users to write these scripts. Since it is usually not a single simulator that is being traced but several, it is not a single pipeline but several pipelines that need to be defined in a *Tracing Script*, one pipeline for each simulator involved that is of interest. An abstract example of how the structure to be defined in such a script should look in the case of four simulators can be seen in Figure 4.2a.

In an actual script, it is necessary to select concrete types for the individual components of the pipeline, as illustrated in Figure 4.2b. An example of a concrete *Tracing Script* can be found in Appendix A.4.



(a) Abstract view on the composition of multiple pipelines for multiple simulators



(b) Abstract overview of the general composition of multiple pipelines for multiple simulators with concrete types

Figure 4.2

4.7.2 Visualization, Metrics and Sampling

After a user has written a *Tracing Script* and orchestrated a SimBricks simulation, it is possible to trace the corresponding simulation. Simply starting these processes in parallel is however not sufficient, because traces should also be visualized with the option of making requests to filter them. For this reason, a user must define within the *Tracing Script* that one of Columbos exporters (for more details see Sections 3.8 and 4.4) must be used to make spans, and therefore traces, available for visualization and querying.

In order to use the exporter in a meaningful way, the corresponding external tools to which spans and traces are to be exported must be started in addition to the SimBricks simulation and Columbo. An example of such a setup is shown in Figure 4.3. This illustration depicts the general flow of data and interrelationships between the various tools. Data is generated in simulators and written to log files or named pipes. These are then read by the framework presented here. Columbo creates spans and traces which are then sent to, for example, an OpenTelemetry collector. Within the OpenTelemetry collector that data can then conveniently be sent to several other tools. In addition, the OpenTelemetry Collector [6] can be used to further process spans in other desired ways.

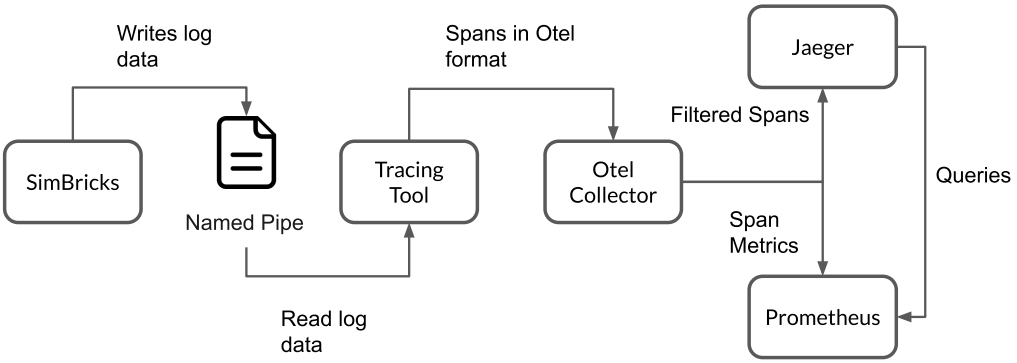


Figure 4.3: General view on interactions between different services when creating and analyzing traces

An example of this would be the automated extraction of span metrics. That means collecting metrics derived directly from single spans of a specific type, such as the duration of a syscall. Another example how spans could be further processed is the filtering of spans that are not of further interest. The OpenTelemetry Collector already provides some tools for this purpose. These tools come in the form of so-called "Processors" whose usage can be conveniently configured through a configuration file.

In the example shown here, the OpenTelemetry Collector then exports the spans to Jaeger [4]. Jaeger allows for browsing and viewing spans in a web interface in detail. Metrics, such as span metrics are sent in parallel to other tools by the OpenTelemetry

Collector. In the example shown here, they are sent to Prometheus [8]. Such potentially interesting temporal information can be partially explored and analyzed through Jaeger or the Prometheus web interface itself. Especially the extraction of more interesting metrics for whole traces like the average end-to-end latency is an interesting direction for future work.

4.8 Framework Extension

Potential users may be interested in adding support for tracing other simulators than those currently supported by Columbo (Section 4.6), or extending support for one of these simulators. Another way in which users may wish to extend the framework is to allow for other export formats than those currently supported, in order to enable the usage of other external tools.

Depending on a users concrete use case, some or all of the following steps may be necessary in order to extend Columbo with the functionality that is needed:

- **SimBricks**. Since Columbo is intended to utilize log files generated by simulators used within a SimBricks simulation, it is necessary to ensure that SimBricks actually supports the simulator of interest. If that is not the case, a first integration step is to integrate such a simulator with the ability to write log files into the SimBricks framework. This is usually done by implementing a small adapter inside the simulator to support the SimBricks protocol [44].
- **Parser**. If a new simulator is added, or if more log output is generated by an already supported simulator, e.g. due to different cli flags or a version upgrade, a parser for the simulator's log files must be provided, or an existing parser must be extended to be able to parse these log files to generate the desired events.
- **Events**. In addition to a parser, it may be necessary to introduce new events or modify existing event definitions within Columbo to reflect the changes in the log output.
- **Spans**. Depending on the simulator and the log output, it may also be necessary to define completely new spans within Columbo, similar to events, or to adapt existing span definitions to reflect that new or different events may be generated by a parser.
- **Spanner**. It is evident that alterations to the spans may also require new spanners to be defined or existing spanners to be adjusted. Depending on what has been changed, spanners may need to define new handler methods or change existing handler methods given new or changed span definitions. It may also be necessary to change when a spanner pushes and polls the trace context given these new or changed span definitions.

- **Exporter**. If a user wishes to use an external tool that does not support the OTLP protocol, the user must define a new exporter that transforms Columbo's internal span/trace representation into the representation required by that tool, as well as implementing the actual export to the external tool.

Chapter 5
Experimental Evaluation

This chapter presents the results of the evaluation. The chapter is divided into three sections, starting with the general setup for the experiments. In Section 5.2 we evaluate the usefulness of the work presented here in a simple experiment, synchronizing two hosts system time with NTP (chrony [9]) and investigating its behavior using the tracing capabilities provided by Columbo. Section 5.3 investigates the general performance of the framework presented here and its performance impact on end-to-end simulations using SimBricks [44].

5.1 Experimental Setup

All experiments were run on a physical host with two Intel(R) Xeon(R) Gold 6336Y CPU @ 2.40GHz, 24 cores each, with 8x 32GiB (256GiB) of memory. The physical host was running Debian GNU/Linux 11 (bullseye) with kernel version 5.15.111.1.amd64-smp. In all experiments, a simple network topology was simulated. The simulation itself was run using SimBricks [44]. The respective topologies used are depicted in more detail in Section 5.2 and Section 5.3. In all cases, the used topologies are composed of two hosts that were simulated using the gem5 [29] simulator. Each of the host simulators itself was attached to SimBricks's Intel i40e NIC behavioral model simulator [44]. These in turn were connected to an instance of ns3 [5] simulating a simple network topology consisting of two network switches connected via a bottleneck link. The simulated hosts had a single core and 8 Gb of memory running ubuntu version 22.04 with kernel version 5.15.93 using only unmodified device drivers and applications. Unneeded features and drivers were disabled to keep the boot times low. Gem5 was configured to use its TimingSimple

CPU model, which simulates an in-order CPU using the timing memory protocol. The CPUs clock frequency was set to 4GHz and DDR4_2400_16x4 memory was used. The PCIe latency and the SimBricks synchronization interval were set to 500ns each. The link latency of the bottleneck link was set to 1μs with a bandwidth of 10 Gbps.

In all experiments where log output was enabled, logging statements were enabled when compiling the Intel i40e NIC behavioral model, ns3 trace sources were attached and used and gem5 was used in the *opt* version together with the debugging flags listed in Appendix A.1 to create the log output necessary to create traces. If no log output was required in an experiment, logging statements were not activated when compiling the Intel i40e NIC behavioral model, trace sources in ns3 were not used and gem5 was used in the *fast* version without logging output enabled.

5.2 Tracing Usage

In this experiment we want to show how Columbo can be used to detect anomalies in a network. For this we use chrony [9] (version 4.2), an NTP (Network Time Protocol [20]) implementation. In the topology shown in Figure 5.1, one of the gem5 hosts is configured to be a NTP server. The server uses its own clock as its reference clock. Thus, the server is always synchronized. The time of this server is used as the reference time, i.e. the ground-truth, throughout the experiment shown here. The second gem5 host is configured with chrony as an NTP client to synchronize its time with that of the server. The respective chrony configurations, for client and server respectively, can be found in Appendix A.2.

To be able to decide whether the server and client are in synchronized, both execute a bash script in parallel to chrony to provide periodic information from the simulation. The bash script is shown in Appendix A.3 and is the same for both client and server. In the bash script a loop is executed 16 times and within this loop the Linux date [11] utility is used to read the Linux system time. This system time is the one to be adjusted by chrony, hereafter referred to as sys-t, given in seconds. After calling the Linux date utility, the bash script uses gem5's m5 tool [12] in the loop to extract the number ticks since the start of the simulation. With the selected simulation settings, the time extracted by m5 is the time that has actually elapsed, regardless of the Linux system time. It is therefore used as the reference time in this experiment and is referred to as real-t in the following and is given in seconds. After using m5, the chrony command line interface [9] is used to request chrony's offset estimate in seconds on the client side in particular. This is referred to as chrony-off in the following. After these three measurements were taken, the bash script sleeps for 40 seconds.

In an ideal scenario, i.e. when the client and server are synchronized, the following conditions should apply to the values that are requested by the above bash script:

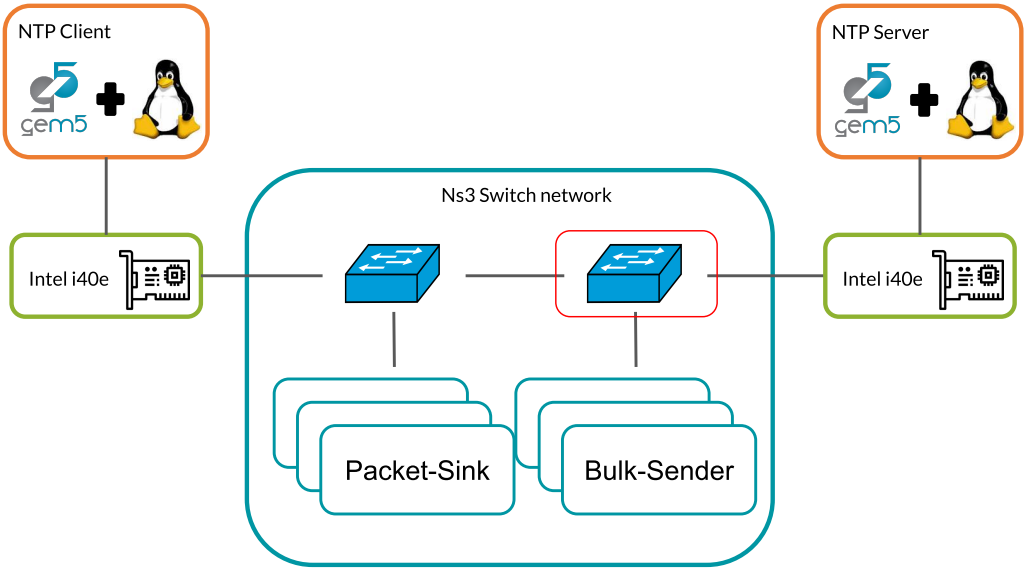


Figure 5.1: Simple dumbbell topology, with bulk send applications exhausting the bottleneck link, instantiated using SimBricks.

1. $|\text{real-t}_{\text{server}} - \text{real-t}_{\text{client}}| = 0s$, i.e. the time difference between client and server in relation to the real elapsed time is 0.
2. $|\text{sys-t}_{\text{server}} - \text{sys-t}_{\text{client}}| = 0s$, i.e. chrony was able to synchronize the Linux system time which is the same for both client and server.
3. $\text{chrony-off}_{\text{client}} = 0s$, i.e. chrony estimates that the client is synchronized with the server time. This should be a consequence of the first two points in an ideal scenario.

If all of these conditions are met, it would mean that the same amount of real time has elapsed on both the client and server sides, with the same system time. In this case not only the system time but also the underlying real time would be the same. The chrony client should also correctly estimate the real offset to be 0 in this case.

Since in the setup presented here the corresponding measured values are obtained using the aforementioned bash script, which is executed within the simulated hosts under Linux, we expect that in the equations shown, the corresponding differences and the chrony estimate are not exactly zero seconds, but only very close to zero seconds, even under ideal conditions. Another reason is that chrony uses software timestamping in the experiments, which is not 100% accurate. This leads to small deviations. So we expect that all measurements shown above are not exactly zero, but very close to zero.

The experiment described above is carried out twice:

1. In the first run, in addition to the two hosts, i.e. the NTP client and server,

there are applications running within ns3. BulkSendApplications are used to send packets to corresponding PacketSinks. The BulkSendApplications send as much data as possible to utilize the available bandwidth completely, i.e. they exhaust the bottleneck link connecting the two switches shown in the topology [5]. PacketSinks, on the other hand, do not send any packets back but consume incoming packets [5].

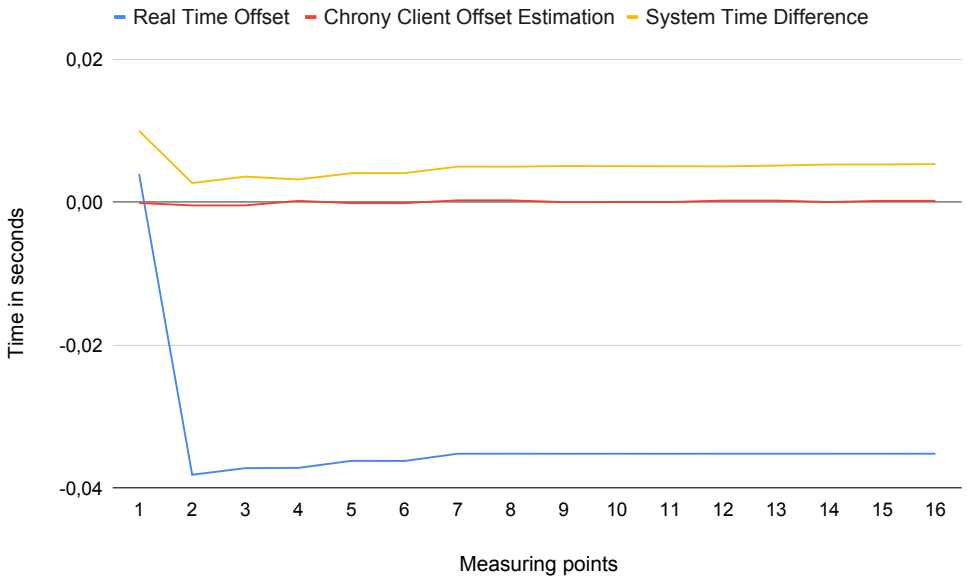
2. In the second test run, the same topology is executed completely without applications within ns3. This test run therefore represents ideal conditions under which chrony should have no difficulty synchronizing the hosts involved.

The results of the first run, i.e. the run in which ns3 applications were used, are shown in Figure 5.2a. The results of the second run, i.e. the run without ns3 applications, are shown in Figure 5.2b. In the respective graphs, the 16 measuring points are listed on the x-axis, which correspond to the 16 iterations of the bash script described above and thus the measuring points at which time information is requested. For both experiments, the following three values are plotted in seconds:

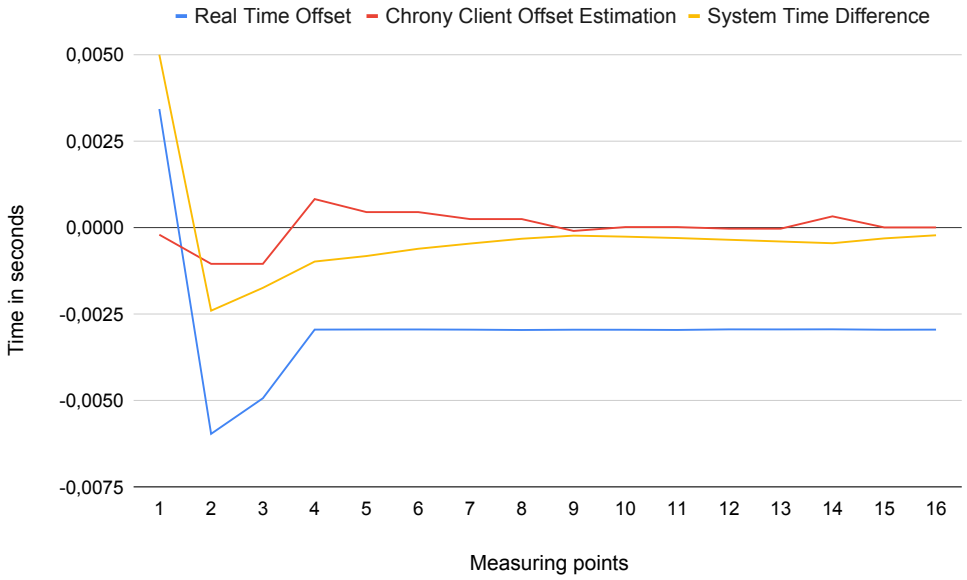
1. Real Time Offset = real-t_{server} − real-t_{client}
2. System Time Difference = sys-t_{server} − sys-t_{client}
3. Chrony Client Offset Estimation = chrony-off_{client}

Let's look at the second run of the experiment, in which no ns3 applications were used to send additional packets over the bottleneck link connecting the switches. Notice that although the conditions for exchanging NTP packets are ideal, the client and server are not fully synchronized. This can be easily seen from the fact that the Real Time Offset $\neq 0s$ and the System Time Difference $\neq 0s$ apply to the shown measurement points. The Real Time Offset averages out to $-0.0028932436s$ while the System Time Difference averages out to $-0.00028319744s$. This happens even though chrony's offset estimate is getting closer and closer to zero seconds, and thus the system time is only slightly or not at all adjusted by chrony as the experiment progresses. It must be noted, however, that the client and server times are very close. The small deviation in time was to be expected, as mentioned above.

Looking at the results of the experiment where ns3 applications were used to exchange packets over the bottleneck link, it is noticeable that, similar to the case where no ns3 applications were used, both the System Time Difference and the Chrony Client Offset Estimation approach the desired value of 0 as the experiment progresses. Unlike the first case, this is not true for the Real Time Offset, which averages out to $-0.033175073722222s$, which is a factor of ≈ 11.5 larger than the setup using ns3 applications to generate background traffic. This basically means that at the same system time, the client and server are at different points in relation to the actual real time. Thus, they are far from being synchronized compared to the first more ideal experiment setup.



(a) Synchronization time results for experiment using ns3 BulkSendApplications. Showing the real time difference between client and server, the difference in system time between client and server as well as the clients time offset estimation.



(b) Synchronization time results for experiment without ns3 applications. Showing the real time difference between client and server, the difference in system time between client and server as well as the clients time offset estimation.

Figure 5.2

The framework presented in this work can now be used to explain this phenomenon. In the case of the two experiments, Columbo was started in addition to the actual simulation in order to generate traces, which were exported to Jaeger using the OTLP-Exporter provided by the framework in order to be able to analyze the generated traces. An important aspect here is that the Tracing Script was configured to generate traces using log messages from all the simulators involved. However, it should be noted that events caused by the respective ns3 applications were filtered out during tracing. Therefore, only packets exchanged between the two gem5 host instances can be seen in the generated traces. In the case of the experiments considered, these are NTP packets.

Tracing allows you to view the resulting traces in Jaeger. In the case of the experiment where no ns3 applications were sending packets, two such traces look as shown in Figure 5.3 after the first search for traces in Jaeger. These two traces belong to the sending of an NTP packet. The lower of the two traces corresponds to an NTP packet being sent from the client to the server, and the upper trace corresponds to an NTP packet being sent back from the server to the client. You can see that both traces have an end-to-end latency of $16\mu s$ and $17\mu s$, respectively. In this case the end-to-end latency is calculated by taking the first timestamp of the trace starting span and the latest timestamp of any events, considering all spans within the trace.

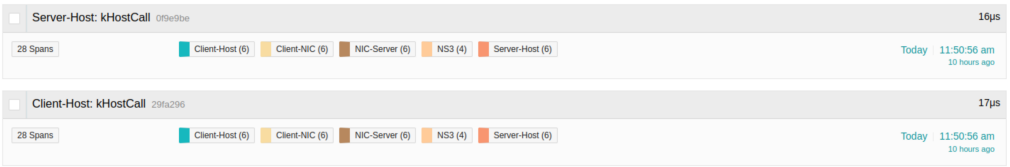


Figure 5.3: Two traces from the no ns3 applications experiment.

Figure 5.4 shows two additional traces. These traces are the result of running the above experiment using ns3 applications to send additional packets over the bottleneck link in addition to the NTP packets send by chrony. One Trace corresponds to the sending of an NTP packet from the client to the server and one Trace corresponds to the return of an NTP packet from the server to the client.

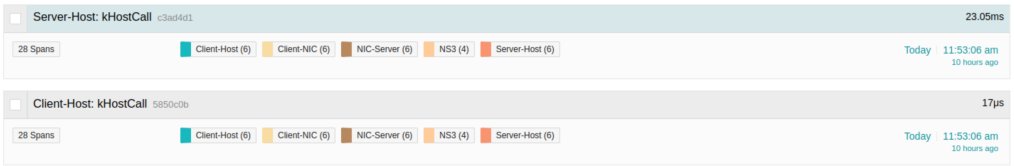


Figure 5.4: Two Traces from the experiment, using ns3 applications.

If we now compare the traces of the experiment where ns3 applications were used with those where no ns3 applications were used, we see that in the first case the end-to-end

latency of the trace where an NTP packet is sent from the server to the client is 23.05ms. Therefore, the latency in this case is significantly higher than in all other cases. Using Jaeger, this trace can now be analyzed in more detail. When opened, the trace looks as shown in Figure 5.5.

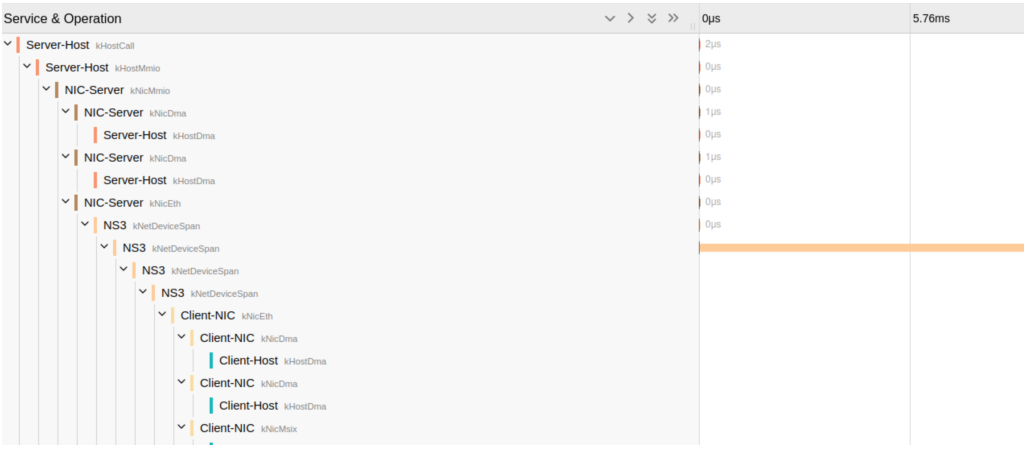


Figure 5.5: Outlier trace. Detailed view within Jaeger.

The individual spans and their causal relationships can be seen in the tree-like view on the left. If one span hangs below another in this tree, the lower span is the child span. For example, in this trace view, you can see the server-side syscall that triggered the return of an NTP packet as the span that started the trace. The immediate child span of that syscall is a mmio write access to one of the registers of the NIC attached to the host. This write also has a counterpart in the NIC simulator. You can see this because the mmio write span has a mmio write span as a child, with the former coming from the host simulator's log file and the latter coming from the NIC simulator's log file. Therefore, to make these connections possible, Columbo performed trace context propagation across simulator boundaries. The mmio span on the NIC side has several child spans. These are the DMA accesses that the NIC performs to read the data to be sent, and a final span that marks that a packet has been sent. The span representing that the NIC sent a packet is then connected to spans coming from the ns3 simulator log file. This is again a point at which Columbo has performed trace context propagation to make the causal connection between the span representing the packet being transmitted and the spans associated with the packet traversing the network. The trace then continues in this style until it is received by the client. On the right side of the view, you can see in color how much time was spent in each span. Notice that a lot of time was spent in one of the *kNetDeviceSpan*s, indicated by the large brown bar, compared to all other spans in the trace shown. In the trace shown, there are four spans of this type, two of which are caused by one of the simulated switches. One span is created when a packet enters a switch, and one span is created when a packet exits a switch. Since the simulated topology consists

of two switches, there are 4 spans. The span that takes the longest to send the NTP packet belongs to the switch marked in red in Figure 5.1. If we look at other traces of the experiment using ns3 BulkSendApplications with the help of Jaeger, we see that all packets sent from the server to the client were delayed by about the same amount of time. In all of them, the red marked switch is the span where the vast majority of time is lost. Users can now drill down into the details of such an interesting span. For example, one can extract the timestamp at which that span started and the timestamp at which that span ended to calculate the actual duration or time spent in that particular span. In the given case, a user can then calculate that $\approx 23ms$ was spent in that span, indicating that the red marked switch is indeed the cause of the high latency of packets sent from server to client.

The reason for this result is that the bottleneck link shown in the topology is exhausted in the server-to-client direction during the experiment using ns3's BulkSendApplications. This is not the case in the client-to-server direction because PacketSinks do not send packets back to the corresponding BulkSendApplications from which they receive packets. In this way, the experiment shown simulates a network in which asymmetric behavior prevails, since the bottleneck link is expected to be exhausted when sending in the server-to-client direction. Given the experimental setup, NTP packets sent by chrony experience additional queuing delay in the switch close to the server when sent from server to client. This is due to the fact that they have to be buffered in the switch, which is marked red in Figure 5.1, until they can be forwarded from the switch using the exhausted bottleneck link. This buffering delay can be easily seen using Columbo, as shown in Figure 5.5. The asymmetry in the network is a problem for chrony, which tries to compensate for asymmetric delays, but cannot do so if there are persistently relatively large asymmetric delays [9]. So we could use Columbo to easily understand and find the problem that causes the client and server to synchronize poorly.

5.3 Tracing Overhead

We will now examine the runtime of Columbo, as well as the size of the log files generated by the simulators compared to the size of the files containing the parsed and filtered events intended to be used to create Spans. The topology shown in Figure 5.6 was used. As a workload for the actual simulation, we use the netperf [17] TCP benchmark to perform a two-second latency test (*TCP_RR*) between the two simulated hosts.

The following configurations are compared trace-wise:

A. **No logging.** In the first setup we ran the experiment using SimBricks without any logging enabled on the simulator side. In this case, tracing is not possible. It allows the simulators to run faster. This configuration is used as a baseline to see the overhead in experiment runtime that is introduced by enabling logging in the

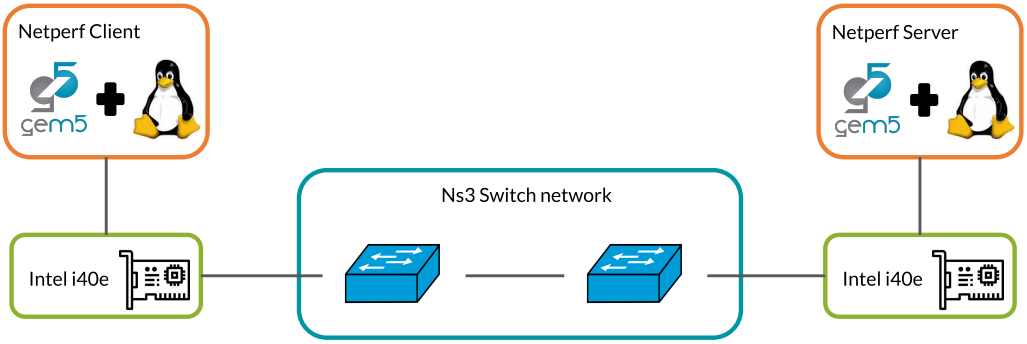


Figure 5.6: Simple dumbbell topology instantiated using SimBricks

simulators.

B. **Logging without tracing.** In the second setup, logging output was enabled for all involved simulators. The simulators were instructed to write log output to a regular file, and no tracing was applied. This shows the overhead of just running the simulators in logging mode to make tracing theoretically feasible and to show how large their respective log outputs are.

C. **Traces from raw log files.** In this experimental setup, the unprocessed, i.e. raw, log files created in Item B. were used to create traces. Therefore, in this case, the simulation was not run in parallel with Columbo, and thus no named pipes were necessary. This reflects the "offline" use of the tracing framework. For the purpose of creating and exporting traces to external tools, a complete tracing pipeline was started, including Exporters and the OpenTelemetry Collector along with Jaeger. This setup was similar to the one shown and described in Section 4.7.2 except that no named pipes were used and no simulation was run alongside Columbo.

D. **Logging and event creation.** In this setup, logging was enabled again for all simulators. The framework presented here ran "online" alongside the actual simulation using the named pipe setup (Sections 3.10 and 4.5.3). The Tracing Script (Section 4.7.1) was written to not use spanners and therefore not use tracers or exporters within the tracing pipeline. The pipelines were configured to read the simulator log output from the named pipes, create events, filter them, and write the resulting event stream to a file for each simulator. Thus, with the given topology, 5 files were created, each containing a simulator instance specific event stream. SimBrick's synchronization events were filtered out from the events and thus from the resulting file.

E. **Traces from pre-processed events.** In this setup, the pre-processed event files written during the setup described in Item D. were used to create traces. A full

tracing pipeline was started utilizing an Exporter and the OpenTelemetry Collector along with Jaeger to export traces similar to Item C.. However, it is important to note that in this case no raw log files were used, but files containing the previously parsed and filtered events needed for trace creation.

F. **Logging and full online trace creation.** In the last setup, both the simulation and the complete tracing pipeline were executed together with external tools in order to be able to export the resulting traces. This means that it was an "online" setup in which the simulators used write their raw log output to named pipes which are then read in parallel by Columbo. This data is then used to create events, which in turn are used to create spans. The spans are then used to create traces, which are eventually sent to an OpenTelemetry Collector using an Exporter. This is the setup described in Section 4.7.2.

In Table 5.1 you can see the results of the different experiment setups in terms of runtime and size of the written files, if any files were written.

Experiment	Runtime	Accumulated file size
No logging A.	1.35 h	×
Logging without tracing B.	7.78 h	1.3 Tb
Traces from raw log files C.	6.4h	×
Logging and event creation D.	7.38h	18 Gb
Traces from pre-processed Events E.	6.11h	×
Logging and full online trace creation F.	7.06h	×

Table 5.1: Tracing Overhead results

5.3.1 File Sizes

One can see that when simulating the topology shown in Figure 5.6, the simulators used, wrote a total of 1.3 Tb of unmodified raw log files. Of these 1.3 Tb, 685 Gb are attributable to the gem5 client, 601 Gb to the gem5 server, 210 Mb each to the Intel i40e NIC behavior models, and 327 Mb to ns3. It is notable that the two gem5 instances in particular generate very large log files. On the one hand, this is due to the fact that they simulate a host system running unmodified Linux and netperf under Linux, i.e. it is to be assumed that these simulators produce larger log files than the other simulators used, since they have to simulate "more". However, with the debugging flags shown in Appendix A.1, they also generate more information than is currently used by Columbo and required for most use cases, such as micro-operations. Therefore, these log files could possibly be made smaller with a more precise selection of the gem5 debug flags depending on the use case, potentially losing some information such as which micro-operations were executed.

If you now take a look at the files containing the pre-processed event streams of the respective simulators, you will see that they require a total of 18 Gb of disk space. Of these, 223 Mb are allocated to the event files of the two Intel i40e NIC behavior models, 381 Mb to the ns3 event file, 8.2 Gb to the gem5 server event file, and 8.5 Gb to the gem5 client event file. The two gem5 simulator event files are particularly noticeable, as they are significantly smaller than the corresponding unprocessed and unmodified log files. The reason for this is that the raw log files contain much more information and actions than Columbo is currently able to parse and convert into events. An example of this are micro-operations that are contained in the simulator's log, but are not converted to events. Since these are not currently converted to events, they are missing in the resulting event files. It is also worth noting that the ns3 event file is slightly larger than the corresponding raw log data. This is due to the fact that the current storage format for events is not optimized to save memory, but to be easily readable by humans.

5.3.2 Runtime

If you look at the runtime of the experiment with the respective logging and tracing combinations, you can see that the experiment with the configuration in which no logging was performed was significantly faster than all other setups. The reason for this is that in this case logging was turned off for all simulators. Therefore, the simulators did not have to execute any logging statements, which improved their performance. The two gem5 instances are the biggest factor here. This can also be seen in Section 5.3.1, as they write by far the largest log files when configured to generate log output. In the case that gem5 is not supposed to generate log output, the *fast* version of gem5 is used in the experiments shown, an optimized binary in which the corresponding logging statements and debug symbols are optimized away. Therefore, gem5 is significantly faster in this case than in the logging case where gem5 is used in the *opt* version. Since gem5 is generally the slowest and thus the limiting factor for the runtime of the entire simulation for the simulators shown, this explains the time difference between the *No logging* setup and the *Logging without tracing*, *Logging and event creation*, and *Logging and full online trace creation* setups. In general, it can be said that simulation times increase significantly depending on the simulators to be traced, regardless of the tracing framework used, in order to generate the log output required for tracing.

In the cases of *Logging and event creation* and *Logging and full online trace creation*, Columbo runs alongside the SimBricks simulation. In the case of *Logging and event creation*, SimBricks and the framework presented here run in parallel to create files that store the parsed and filtered events for each simulator. Regarding the experimental setup of the *Logging and full online trace creation* case, SimBricks and the tracing utility run side-by-side to create Traces and export them to external tools as described in Section 4.7.2. Note that the experiments with the *logging without tracing* setup ran longer

than the *logging and event creation* setup, which itself ran longer than the experiment with the *logging and full online trace creation* setup. The reason for this is that in the *Logging without tracing* and *Logging and full online trace creation* cases, more disk I/O must be performed to persist the raw simulator log files and the event stream files for each simulator, respectively. This is in contrast to the *Logging and full online trace creation* setup, in which all data is kept in memory all the time and no disk I/O is required. This explains the relatively small differences between these setups.

Comparing the runtimes of the above cases with the runtimes of the experiments using the *Traces from raw log files* and *Traces from pre-processed events* setups, you can see that the latter are also significantly faster than the experiments using the *logging without tracing*, *logging and event creation*, and *logging and full online trace creation* setups. This is because the experiments in the latter cases used Columbo "online", i.e. alongside the actual SimBricks simulation using named pipes. In the *Traces from raw log files* and *Traces from pre-processed events* cases, the tracing framework was used "offline", i.e. the actual SimBricks simulation was not running in parallel with the tracing framework. Therefore, Columbo used existing raw and unmodified log files or pre-processed event files as input to create traces and export them to external tools. For this reason, the actual simulation of gem5 is no longer the bottleneck for the runtime of the whole experiment, which explains the general runtime jump between the mentioned experiment setup groups.

Gem5 cannot be the bottleneck, and thus the reason for the long runtimes, in the experiments using the *Traces from raw log files* and *Traces from pre-processed events* setups, because they do not run in parallel with the actual simulation. Instead, they reveal a bottleneck within the actual tracing framework. When looking at how the individual components provided by Columbo are linked together, as shown in Figures 4.2a and 4.2b, you will see that simulator-specific pipelines are instantiated for each simulator, creating spans. In the cases we're considering right now, this comes down to instantiating a pipeline for each raw, unmodified simulator log, or for each file that stores a simulator-specific event stream. The bottleneck now occurs when these spans are ready for export and need to be converted into the required representation to be sent to an external tool. This is done within an Exporter (Sections 3.8 and 4.4). All simulator-specific pipelines share a single Exporter instance and must synchronize their access to it in the current implementation. This synchronization is currently required because pipelines export spans through the tracer. The tracer in turn ensures that spans are only exported if their parent, if present, has already been exported, which requires exclusive access for bookkeeping purposes while a span is being exported as described in Section 4.3. Since exporting and especially converting spans to the OpenTelemetry format used by the currently supported toolchain takes a lot of time compared to the time needed by the simulator specific pipelines to generate spans ready for export, waiting for the Exporter instance to be granted access makes these experiment setups take significantly more time. For example, if the experiment using the *Traces from pre-processed events* setup is run

exactly as before, with the only difference that no Exporter is used (i.e., span creation and context propagation are still done as before), the runtime of the experiment is reduced from 6.11 hours to 27 minutes.

Chapter 6
Related Work

6.1 Simutrace

Simutrace [51] is a tracing framework specifically designed for full system memory tracing. It uses functional full system simulations and captures memory accesses at the hardware level including accesses of user-space programs, the operating system, drivers and direct memory accesses. To enable the tracing of memory accesses in full length without loss during long-running workloads, Simutrace incorporates a fast and capable compressor. As Simutrace uses simulation to gain insight, the resulting traces are free of side effects and thus preserve timing information, similar to Columbo. One distinguishing feature of Simutrace is its ability to track not only the physical or virtual memory addresses being accessed, but also the actual data being written during write operations. This allows for the reconstruction of a system's memory state at a given point in time. Columbo is not capable of doing so, but could in principle be extended to fully trace memory accesses if the simulators used provide sufficient information in the form of a log file. In addition to its primary focus on memory tracing, Simutrace has a flexible and modular design that allows it to interface with a variety of full-system simulators and to trace data other than memory accesses, such as OS introspection events. Simutrace achieves modularity by following a client-server architecture. Clients have two functions: firstly, they trace extensions within a full system simulator and collect information about memory accesses that they send to a server. Secondly, they are peers that connect to a server to query and analyse collected traces. The server is the central component of Simutrace, responsible for storing, compressing/decompressing, receiving and presenting traces to and from clients. Columbo makes only partial use of a client-server architecture. The processing and

creation of traces is handled completely independently of the actual simulation within the framework. It could be argued that this is the client aspect of the work presented. Once a trace is completed, Columbo aims to use tools from the distributed tracing community and thus aims at transforming its internal trace representation into whatever representation is needed by a tool to then send a trace to the respective tool so that the tool takes care of storing and retrieving the respective data, in this case the server. Since Simutrace handles storing the traces itself, it organizes the traces into streams that separate events according to their semantic background and type, providing flexibility and fast random read access without any restrictions on the type of event captured. Eventually, Simutrace writes trace data to persistent storage using a custom format optimized for traces with a large number of entries. This work does not bother to store traces on disk, as they are exported to external tools that are responsible for this if desired. Since Columbo uses log files to receive data, the use of tracing extensions is not required in comparison to Simutrace, which requires the implementation of an integration layer within a simulator, consisting of hooks that communicate the desired information to Simutrace. An aspect that distinguishes this work from Simutrace is that the work presented here supports the tracing of end-to-end simulations involving potentially multiple different and independent simulator instances, while being able to make causal connections between spans across the boundaries of the participating simulators.

6.2 TraceDoctor

In their work *Balancing Accuracy and Evaluation Overhead in Simulation Point Selection* [37] Gottschall et al. present TraceDoctor, a powerful and versatile tracing interface for FireSim [41]. This is also a major difference between the work presented here and TraceDoctor, as this work aims to support arbitrary simulators, while TraceDoctor is specifically designed to be used alongside FireSim. FireSim itself is an FPGA (Field Programmable Gate Array) accelerated full system hardware simulator that can reduce simulation times by two orders of magnitude compared to software simulators for computer architectures. FireSim requires key components of a system, such as the processor core, to be modelled at RTL level. Hardware configurations of such components are then instantiated on FPGAs, allowing all actions for a given clock cycle to be performed in parallel. While this design is beneficial for performance, it makes implementing ideas much harder, and also limits the ability to gather information, as it requires hardware to retrieve information at the level required by for example hardware architects. When running a simulation with FireSim, TraceDoctor runs in parallel on the host using a configurable number of workers, thus minimizing simulation slowdown while analyzing trace data. TraceDoctor is capable of tracing architecture-internal signals in every clock cycle of the simulator. This is achieved by connecting a so-called trace vector to the signals of interest within the simulated hardware to TraceDoctor's hardware component. This is

another general difference between Columbo and TraceDoctor. The design of Columbo is strongly influenced by the decision to modify the simulators as little as possible, or not at all, by using log files as a source of trace data. However, this is not possible in the case of TraceDoctor, as it requires the synthesis of a hardware component on the FPGA side, thus instrumenting the simulation to collect the required data. The trace vectors are transferred to the host via the FPGA's DMA interface. The trace data is processed by software workers running on the host side, which allow the trace data to be analyzed, compressed and filtered. This also helps to keep storage overheads to a minimum. If TraceDoctor is unable to either perform DMA accesses fast enough or to send the previously transferred data, the simulation will eventually stall. Like Columbo does TraceDoctor allow analyzing, filtering or compressing the traced data outside the critical path of the actual simulation. In both cases, however, not performing these actions will potentially cause the simulator to stall, either by not reading from a named pipe in the case of Columbo, or by explicitly preventing the simulation from progressing until a DMA can be performed. TraceDoctor minimizes the risk of simulation stalling by allowing multiple workers to run in parallel on the host side. This also allows several analyses to be performed on the traced data in parallel, if the analyses permit this. The workers are independent of the simulated hardware design, which eliminates the need for FPGA synthesis if changes are made to them. FPGA synthesis is only required by TraceDoctor if the trace vector used changes. Both works allow to change the analyses performed on the trace data without changing the simulators involved. TraceDoctor, however, allows to perform several analyses on the same data in parallel. This is not possible with the work presented here, which allows several analyses to be run, but only one after the other within the same pipeline. As previously stated, the most significant distinction between Columbo and TraceDoctor is that Columbo supports the tracing of end-to-end simulations involving multiple distinct and independent simulator instances, while being able to make causal connections between spans across the involved simulator boundaries.

6.3 Trace Compass

Trace Compass is an open-source application for visualizing and analyzing log files [21]. Trace Compass combines log files from different sources, including the kernel, userspace applications and network devices. It supports a multitude log file formats, including the Common Trace Format (CTF) [10], the Linux FTrace raw binary and text format, as well as PCAP files. These log files are parsed to generate events with timestamps. In general, Trace Compass combines the events parsed from the various log files into a single event stream. This stream contains events that are ordered according to the time at which they occurred, with events that occurred around the same time being packed together closely in the resulting stream. This stream can then be analysed by users to infer relationships

between events and correlate them. To facilitate this, Trace Compass provides a number of different views to show the relationships between events. Trace Compass uses a plugin architecture and is therefore modular and extensible. Developers can add support for new trace types, analysis modules and visualizations. This allows Trace Compass to be easily adapted to different scenarios. In general, Trace Compass is a toolkit that enables the analysis and potential correlation of different log files through different modules. Trace Compass, unlike Columbo, is not designed for use with full system simulation, but it supports a number of specified log file formats. In contrast, the framework presented here is designed to support any simulator log files. Furthermore, the framework presented in this paper attempts to automatically infer relationships between events across simulator boundaries in order to instantly create end-to-end traces. Trace Compass cannot do this in general, but users/developers can add this or similar functionality to Trace Compass in the form of modules. Consequently, Columbo could be incorporated as a module into Trace Compass in the future, thereby enhancing its functionality.

Chapter 7
Conclusion

This thesis presents Columbo, a framework for obtaining in-depth visibility into systems by using log files generated by individual simulators during modular full system simulations. Columbo analyzes the log files generated by such simulations to create comprehensive traces that depict cause-and-effect relationships between actions within different simulators. Traces are created by parsing individual log files to create events. These, in turn, are combined into spans and causally linked to each other to form a distributed trace through the simulated system, across simulator boundaries. Thus, Columbo combines the advantages of modular full system simulation, such as deep visibility, with the advantages of distributed tracing, which enables reasoning about the end-to-end behavior of systems across system boundaries.

The framework approach offers several advantages. Firstly, Columbo seamlessly integrates with existing modular simulations, making it straightforward to adopt and integrate with new simulator components. The pipeline architecture achieves this as outlined in Chapter 3 and Chapter 4. Pipelines allow the creation of small, independent and individual components, such as a new parser for the log format of a simulator. These can then be used within a pipeline in conjunction with already existing components. It is also possible to easily adapt existing pipeline components as they perform small logical tasks within the pipeline, making them easier to maintain and extend.

Secondly, Columbo is non-invasive, requiring no modifications to existing simulators and is functioning out-of-the-box with any simulator that generates log files. Columbo creates end-to-end full system traces, making causal connections across simulator boundaries by utilizing the different log files of the simulators involved. As visibility into the simulated system is an important aspect of simulation in general, the majority of existing simulators

are already capable of providing detailed log files. No further instrumentation is required within a simulator itself by the framework. Therefore, simulators can be integrated into Columbo without any changes. This is also not required for the creation of causal links between spans, as these are inferred from the natural boundaries between simulators. This is possible because simulators generate start- and end-events that allow the creation of spans for a single simulator, such as a syscall entry and return to userspace. On the other hand, send and receive like events are generated between simulator boundaries such as the issuing of a mmio-write on the host side and the confirmation of a mmio-write to a register on the device side, which allow relationships between simulator-specific spans to be inferred across simulator boundaries.

Thirdly, Columbo processes log files concurrently with the simulation, reducing storage requirements and enabling real-time analysis. Given that simulators can create very large amounts of log files, it is possible to run the framework in parallel with the actual simulation using named pipes to create Traces. This prevents the creation of large log files and speeds up the end-to-end process of obtaining Traces.

Finally, Columbo facilitates the exploration and analysis of the generated Traces through the use of visualization tools and querying mechanisms. Columbo comes with a compatibility layer in the form of Exporters (Sections 3.8 and 4.4). Exporters allow the conversion and export of framework Traces to battle-tested external tools from the distributed tracing community. These tools then allow to query and visualize Traces.

By achieving these goals, the framework empowers users to gain deep insight into system behavior across different layers within a modular simulation environment. This paves the way for efficient debugging, performance analysis, and optimization of complex systems.

7.1 Future Work

We now discuss some directions in which the work presented here can be extended in the future.

Tracing Limitations

The framework outlined here represents a first starting point for obtaining low level end-to-end system traces through modular full system simulations and still has limitations that should be addressed in future versions.

There is a first limitation with regard to tracing programs within host simulators. In full system simulations, depending on the use case, it is desirable to run unmodified Linux on a simulated host in order to be able to execute unmodified applications. In addition, simulators such as gem5 can provide information about what happens at CPU level when a program is executed on a host, like the address of a function that is currently being

executed. To make this information more accessible to users of Columbo, it currently attempts to resolve function addresses to their symbol names. However, this is only possible for programs from which a symbol table, that stores a mapping of the relative address offsets of function symbols alongside the required global address offset is provided by the user. This is not easy, especially for userspace programs, for example, due to kernel protection rules and dynamic linking, making a static offset difficult to use. Support for this should be extended in future versions of the framework.

Another general limitation of Columbo results from the choice to infer causal relationships in order to propagate trace context. Suppose we want to simulate a topology where a server communicates with several clients. Assume two clients, A and B, simultaneously send packets to a single server. Consider also that these are processed one after the other by individual syscalls when arriving at the server. Currently, Columbo cannot distinguish in such a scenario whether a syscall is processing the packet originally coming from client A, that from client B or maybe both. This is due to the fact that, depending on the simulator's log file output, only the information that packets are available for processing and the associated DMA access events, which copy the packet data into kernel space, are available. Information about which IP addresses are set in the IP header of these packets, for example, in order to be able to distinguish which of the packets was read in the example, is not available. With only one communication partner this is sufficient to infer causal relationships. In this case, Columbo could uniquely identify the sender of the packets. As this is not possible in the given example, the framework has to "randomly" determine which packet is read, leading to incorrect causal connections between spans. Such limitations should be addressed in future versions of Columbo to make trace context inference more robust.

Tracing Scripts

In order to use Columbo, users currently have to manually write a C++ program using the components provided by the framework. This allows users to define tracing pipelines and assemble them like building blocks. However, this is still very cumbersome and error-prone, as these Tracing Scripts can become quite extensive even with ready-made components (see Section 4.7.1 and appendix A.4).

A possibility for the future is to add functionality to Columbo that allows users to trace simulations without having to write a C++ program. This can be achieved by providing a general purpose program that offers an interface in form of a configuration file that can be used to provide the required information for the framework to automatically initialize tracing pipelines.

Limited Simulator Support

Columbo currently supports only three Simulators: Gem5, Intel i40e NIC behavioral model and ns3 (Section 4.6). As the framework is primarily designed for use with SimBricks, ideally all simulators supported by SimBricks should also be supported by Columbo. Even if some use cases can be covered using the simulators already supported, this is by far not enough if, for example, potential users want to run actual RTL hardware simulations, which SimBricks also enables using Verilator. Therefore, support for more different simulators should be added in future versions.

Exporter Bottleneck

In Section 5.3, we analyzed the performance of the presented framework and its influence on the experiment runtime when used alongside an actual simulation. In the course of this, it was found that the Exporter is a bottleneck in the current pipeline design. This bottleneck significantly increases the time required for trace creation and export as a single exporter instance is used by all pipelines, which need to synchronize their access to it.

This bottleneck should be avoided in future versions of Columbo by providing each pipeline instance with its own Exporter instance during tracing. In this way, access for exporting no longer needs to be synchronized and, due to the increased number of exporters, exporting itself can take place in parallel for each pipeline, thus increasing throughput.

Bibliography

[1] 2023. Complete Guide to Distributed Tracing | New Relic. https://newrelic.com/blog/best-practices/distributed-tracing-guide. (2023). Retrieved Dec 21, 2023.

[2] 2023. The Go Programming Language Specification - The Go Programming Language. https://go.dev/ref/spec#Channel_types. (2023). Retrieved Dec 30, 2023.

[3] 2023. Grafana: The open observability platform | Grafana Labs. https://grafana.com/. (2023). Retrieved Dec 7, 2023.

[4] 2023. Jaeger: open source, distributed tracing platform. https://www.jaegertracing.io/. (2023). Retrieved Dec 7, 2023.

[5] 2023. ns3 | a discrete-event network simulator for internet systems. https://www.nsnam.org/. (2023). Retrieved Dec 7, 2023.

[6] 2023. OpenTelemetry. https://opentelemetry.io/. (2023). Retrieved Dec 7, 2023.

[7] 2023. OpenZipkin · A distributed tracing system. https://zipkin.io/. (2023). Retrieved Dec 7, 2023.

[8] 2023. Prometheus - Monitoring system & time series database. https://prometheus.io/. (2023). Retrieved Dec 7, 2023.

[9] 2024. chrony – Introduction. https://chrony-project.org/. (2024). Retrieved Apr 2, 2024.

[10] 2024. CTF2-SPEC-2.0: Common Trace Format version 2. https://diamon.org/ctf/. (2024). Retrieved Apr 28, 2024.

[11] 2024. date(1) - Linux manual page. https://man7.org/linux/man-pages/man1/date.1.html. (2024). Retrieved Apr 28, 2024.

[12] 2024. gem5: The gem5 simulator system. https://www.gem5.org/. (2024). Retrieved Feb 8, 2024.

[13] 2024. Intel® P4 Studio. https://www.intel.com/content/www/us/en/products/details/network-io/intelligent-fabric-processors/p4-studio.html. (2024). Retrieved Mar 14, 2024.

[14] 2024. JLS2009: Generic receive offload [LWN.net]. https://lwn.net/Articles/358910/. (2024). Retrieved Feb 15, 2024.

[15] 2024. mkfifo(3) - Linux manual page. https://man7.org/linux/man-pages/man3/mkfifo.3.html. (2024). Retrieved Jan 2, 2024.

[16] 2024. ModelSim HDL simulator | Siemens Software. https://eda.sw.siemens.com/en-US/ic/modelsim/. (2024). Retrieved Mar 14, 2024.

[17] 2024. Netperf - Networking performance benchmark. https://hewlettpackard.github.io/netperf/. (2024). Retrieved Apr 2, 2024.

[18] 2024. The Network Simulator - ns-2. https://www.isi.edu/nsnam/ns/. (2024). Retrieved Mar 14, 2024.

[19] 2024. QEMU - A generic and open source machine emulator and virtualizer. https://www.qemu.org/. (2024). Retrieved Mar 14, 2024.

[20] 2024. RFC 5905 - Network Time Protocol Version 4: Protocol and Algorithms Specification. https://datatracker.ietf.org/doc/html/rfc5905. (2024). Retrieved Apr 16, 2024.

[21] 2024. Trace Compass. https://eclipse.dev/tracecompass/. (2024). Retrieved Apr 28, 2024.

[22] 2024. Verilator – the fastest Verilog HDL simulator. https://www.veripool.org/verilator/. (2024). Retrieved Mar 14, 2024.

[23] 2024. Zero-copy networking [LWN.net]. https://lwn.net/Articles/726917/. (2024). Retrieved Feb 15, 2024.

[24] Mina Tahmasbi Arashloo, Alexey Lavrov, Manya Ghobadi, Jennifer Rexford, David Walker, and David Wentzlaff. 2020. Enabling Programmable Transport Protocols in High-Speed NICs. In *17th USENIX Symposium on Networked Systems Design and Implementation (NSDI 20)*. USENIX Association, Santa Clara, CA, 93–109. https://www.usenix.org/conference/nsdi20/presentation/arashloo

[25] Eduardo Argollo, Ayose Falcón, Paolo Faraboschi, and Daniel Ortega. 2010. *Toward the Datacenter: Scaling Simulation Up and Out*. Springer US, Boston, MA, 47–63. DOI:http://dx.doi.org/10.1007/978-1-4419-6175-4_4

[26] Remzi H. Arpaci-Dusseau and Andrea C. Arpaci-Dusseau. 2018. *Operating Systems: Three Easy Pieces* (1.00 ed.). Arpaci-Dusseau Books.

[27] Remzi H. Arpaci-Dusseau and Andrea C. Arpaci-Dusseau. 2023. *Operating Systems: Three Easy Pieces* (1.10 ed.). Arpaci-Dusseau Books.

[28] Paul Barham, Rebecca Isaacs, Richard Mortier, and Dushyanth Narayanan. 2003. Magpie: Online Modelling and Performance-aware Systems. In *9th Workshop on Hot Topics in Operating Systems (HotOS IX)*. USENIX Association, Lihue, HI. https://www.usenix.org/conference/hotos-ix/magpie-online-modelling-and-performance-aware-systems

[29] Nathan Binkert, Bradford Beckmann, Gabriel Black, Steven K. Reinhardt, Ali Saidi, Arkaprava Basu, Joel Hestness, Derek R. Hower, Tushar Krishna, Somayeh Sardashti, Rathijit Sen, Korey Sewell, Muhammad Shoaib, Nilay Vaish, Mark D. Hill, and David A. Wood. 2011. The Gem5 Simulator. *SIGARCH Comput. Archit. News* 39, 2 (aug 2011), 1–7. DOI:http://dx.doi.org/10.1145/2024716.2024718

[30] Gabriel Black, Nathan Binkert, Steven K. Reinhardt, and Ali Saidi. 2010. *Modular ISA-Independent Full-System Simulation*. Springer US, Boston, MA, 65–83. DOI:http://dx.doi.org/10.1007/978-1-4419-6175-4_5

[31] Qizhe Cai, Shubham Chaudhary, Midhul Vuppalapati, Jaehyun Hwang, and Rachit Agarwal. 2021. Understanding host network stack overheads. In *Proceedings of the 2021 ACM SIGCOMM 2021 Conference (SIGCOMM '21)*. Association for Computing Machinery, New York, NY, USA, 65–77. DOI:http://dx.doi.org/10.1145/3452296.3472888

[32] Neal Cardwell, Yuchung Cheng, Lawrence Brakmo, Matt Mathis, Barath Raghavan, Nandita Dukkipati, Hsiao keng Jerry Chu, Andreas Terzis, and Tom Herbert. 2013. packetdrill: Scriptable Network Stack Testing, from Sockets to Packets. In *2013 USENIX Annual Technical Conference (USENIX ATC 13)*. USENIX Association, San Jose, CA, 213–218. https://www.usenix.org/conference/atc13/technical-sessions/presentation/cardwell

[33] Adrian M. Caulfield, Eric S. Chung, Andrew Putnam, Hari Angepat, Jeremy Fowers, Michael Haselman, Stephen Heil, Matt Humphrey, Puneet Kaur, Joo-Young Kim, Daniel Lo, Todd Massengill, Kalin Ovtcharov, Michael Papamichael, Lisa Woods, Sitaram Lanka, Derek Chiou, and Doug Burger. 2016. A cloud-scale acceleration architecture. In *2016 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. 1–13. DOI:http://dx.doi.org/10.1109/MICRO.2016.7783710

[34] M.Y. Chen, E. Kiciman, E. Fratkin, A. Fox, and E. Brewer. 2002. Pinpoint: problem determination in large, dynamic Internet services. In *Proceedings International Conference on Dependable Systems and Networks*. 595–604. DOI:http://dx.doi.org/10.1109/DSN.2002.1029005

[35] Jakob Engblom, Daniel Aarno, and Bengt Werner. 2010. *Full-System Simulation from Embedded to High-Performance Systems*. Springer US, Boston, MA, 25–45. DOI:http://dx.doi.org/10.1007/978-1-4419-6175-4_3

[36] Rodrigo Fonseca, George Porter, Randy H. Katz, and Scott Shenker. 2007. X-Trace: A Pervasive Network Tracing Framework. In *4th USENIX Symposium on Networked Systems Design & Implementation (NSDI 07)*. USENIX Association, Cambridge, MA. https://www.usenix.org/conference/nsdi-07/x-trace-pervasive-network-tracing-framework

[37] Björn Gottschall, Silvio Campelo de Santana, and Magnus Jahre. 2023a. Balancing Accuracy and Evaluation Overhead in Simulation Point Selection. In *2023 IEEE International Symposium on Workload Characterization (IISWC)*. 43–53. DOI:http://dx.doi.org/10.1109/IISWC59245.2023.00019

[38] Björn Gottschall, Lieven Eeckhout, and Magnus Jahre. 2023b. TEA: Time-Proportional Event Analysis. In *Proceedings of the 50th Annual International Symposium on Computer Architecture (ISCA '23)*. Association for Computing Machinery, New York, NY, USA, Article 23, 13 pages. DOI:http://dx.doi.org/10.1145/3579371.3589058

[39] EunYoung Jeong, Shinae Wood, Muhammad Jamshed, Haewon Jeong, Sunghwan Ihm, Dongsu Han, and KyoungSoo Park. 2014. mTCP: a Highly Scalable User-level TCP Stack for Multicore Systems. In *11th USENIX Symposium on Networked Systems Design and Implementation (NSDI 14)*. USENIX Association, Seattle, WA, 489–502. https://www.usenix.org/conference/nsdi14/technical-sessions/presentation/jeong

[40] Jonathan Kaldor, Jonathan Mace, Michał Bejda, Edison Gao, Wiktor Kuropatwa, Joe O'Neill, Kian Win Ong, Bill Schaller, Pingjia Shan, Brendan Viscomi, Vinod Venkataraman, Kaushik Veeraraghavan, and Yee Jiun Song. 2017. Canopy: An End-to-End Performance Tracing And Analysis System. In *Proceedings of the 26th Symposium on Operating Systems Principles (SOSP '17)*. Association for Computing Machinery, New York, NY, USA, 34–50. DOI:http://dx.doi.org/10.1145/3132747.3132749

[41] Sagar Karandikar, Howard Mao, Donggyu Kim, David Biancolin, Alon Amid, Dayeol Lee, Nathan Pemberton, Emmanuel Amaro, Colin Schmidt, Aditya Chopra, Qijing Huang, Kyle Kovacs, Borivoje Nikolic, Randy Katz, Jonathan Bachrach, and Krste Asanovic. 2018. FireSim: FPGA-Accelerated Cycle-Exact Scale-Out System Simulation in the Public Cloud. In *2018 ACM/IEEE 45th Annual International Symposium on Computer Architecture (ISCA)*. 29–42. DOI:http://dx.doi.org/10.1109/ISCA.2018.00014

[42] Pedro Las-Casas, Giorgi Papakerashvili, Vaastav Anand, and Jonathan Mace. 2019. Sifter: Scalable Sampling for Distributed Traces, without Feature Engineering. In *Proceedings of the ACM Symposium on Cloud Computing (SoCC '19)*. Association for Computing Machinery, New York, NY, USA, 312–324. DOI:http://dx.doi.org/10.1145/3357223.3362736

[43] Rainer Leupers and Olivier Temam. 2010. *Introduction*. Springer US, Boston, MA, 1–4. DOI:http://dx.doi.org/10.1007/978-1-4419-6175-4_1

[44] Hejing Li, Jialin Li, and Antoine Kaufmann. 2022. SimBricks: End-to-End Network System Evaluation with Modular Simulation. In *Proceedings of the ACM SIGCOMM 2022 Conference (SIGCOMM '22)*. Association for Computing Machinery, New York, NY, USA, 380–396. DOI:http://dx.doi.org/10.1145/3544216.3544253

[45] Xiaofeng Lin, Yu Chen, Xiaodong Li, Junjie Mao, Jiaquan He, Wei Xu, and Yuanchun Shi. 2016. Scalable Kernel TCP Design and Implementation for Short-Lived Connections. In *Proceedings of the Twenty-First International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '16)*. Association for Computing Machinery, New York, NY, USA, 339–352. DOI:http://dx.doi.org/10.1145/2872362.2872391

[46] Jonathan Mace. 2018. A Universal Architecture for Cross-Cutting Tools in Distributed Systems. https://api.semanticscholar.org/CorpusID:209087683

[47] P.S. Magnusson, M. Christensson, J. Eskilson, D. Forsgren, G. Hallberg, J. Hogberg, F. Larsson, A. Moestedt, and B. Werner. 2002. Simics: A full system simulation platform. *Computer* 35, 2 (2002), 50–58. DOI:http://dx.doi.org/10.1109/2.982916

[48] Radhika Mittal, Alexander Shpiner, Aurojit Panda, Eitan Zahavi, Arvind Krishnamurthy, Sylvia Ratnasamy, and Scott Shenker. 2018. Revisiting network support for RDMA. In *Proceedings of the 2018 Conference of the ACM Special Interest Group on Data Communication (SIGCOMM '18)*. Association for Computing Machinery, New York, NY, USA, 313–326. DOI:http://dx.doi.org/10.1145/3230543.3230557

[49] Simon Peter, Jialin Li, Irene Zhang, Dan R. K. Ports, Doug Woos, Arvind Krishnamurthy, Thomas Anderson, and Timothy Roscoe. 2014. Arrakis: The Operating System is the Control Plane. In *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)*. USENIX Association, Broomfield, CO, 1–16. https://www.usenix.org/conference/osdi14/technical-sessions/presentation/peter

[50] Deepti Raghavan, Shreya Ravi, Gina Yuan, Pratiksha Thaker, Sanjari Srivastava, Micah Murray, Pedro Henrique Penna, Amy Ousterhout, Philip Levis, Matei Zaharia, and Irene Zhang. 2023. Cornflakes: Zero-Copy Serialization for Microsecond-Scale Networking. In *Proceedings of the 29th Symposium on Operating Systems Principles (SOSP '23)*. Association for Computing Machinery, New York, NY, USA, 200–215. DOI:http://dx.doi.org/10.1145/3600006.3613137

[51] Marc Rittinghaus, Thorsten Groeninger, and Frank Bellosa. 2015. *Simutrace: A Toolkit for Full System Memory Tracing.* White paper. Karlsruhe Institute of Technology (KIT), Operating Systems Group.

[52] Benjamin H. Sigelman, Luiz André Barroso, Mike Burrows, Pat Stephenson, Manoj Plakal, Donald Beaver, Saul Jaspan, and Chandan Shanbhag. 2010. *Dapper, a Large-Scale Distributed Systems Tracing Infrastructure.* Technical Report. Google, Inc. https://research.google.com/archive/papers/dapper-2010-1.pdf

[53] András Varga and Rudolf Hornig. 2008. An overview of the OMNeT++ simulation environment. In *Proceedings of the 1st International Conference on Simulation Tools and Techniques for Communications, Networks and Systems & Workshops (Simutools '08)*. ICST (Institute for Computer Sciences, Social-Informatics and Telecommunications Engineering), Brussels, BEL, Article 60, 10 pages.

[54] Zhuolong Yu, Bowen Su, Wei Bai, Shachar Raindel, Vladimir Braverman, and Xin Jin. 2023. Understanding the Micro-Behaviors of Hardware Offloaded Network Stacks with Lumina. In *Proceedings of the ACM SIGCOMM 2023 Conference (ACM SIGCOMM '23)*. Association for Computing Machinery, New York, NY, USA, 1074–1087. DOI: http://dx.doi.org/10.1145/3603269.3604837

Appendix A
Implementation Details

A.1 Gem5 debug flags used within experiments

The following debug flags were set and enabled when gem5's opt version was used in experiments within this work to gather log output used for trace creation:

```
1 —debug-flags="SimBricksAll,SyscallAll,EthernetAll,PciDevice,PciHost,
      ExecEnable,ExecOpClass,ExecThread,ExecEffAddr,ExecResult,ExecMicro,
      ExecMacro,ExecUser,ExecKernel,ExecOpClass,ExecRegDelta,ExecFaulting,
      ExecAsid,ExecFlags,ExecCPSeq,ExecFaulting,ExecFetchSeq"
```

More Information on gem5 debugging can be found on the gem5 projects website [12].

A.2 Chrony Configurations

The two following chrony configurations listed, one for a server and one for a client, were used during the experiments further described in Section 5.2.

Server Config:

```
1 local stratum 1
2 allow 192.168.64.0/24
3 ratelimit interval -2
```

Client Config:

```
1 server {self.server_ip} iburst
```

A.3 Tracing Usage Experiment - Bash Script

The following bash script was used in the experiment described in Section 5.2 to periodically extract timing information from the simulated client and server hosts:

```
for i in {{1..16}}
do
    date +%s%N
    m5 dumpstats
    chronyc -n tracking
    sleep 40
done
```

A.4 Example Tracing Script

Example Tracing Script (see Section 4.7.1) to create traces from two Gem5 instances, two NIC i40e behavioral models and an NS3 instance using multiple filters in each respective pipeline.

```

int main(int argc, char *argv[]) {
  ... // do command line parsing to receive filenames etc.
  const TraceEnvConfig trace_env_config
      = TraceEnvConfig::CreateFromYaml(result["trace-env-config"]
            .as<std::string>());
  TraceEnvironment trace_environment{trace_env_config};
  spdlog::set_level(trace_env_config.GetLogLevel());
  auto exporter = create_shared<simbricks::trace::OtlpSpanExporter>(
      TraceException::kSpanExporterNull,
      trace_environment,
      trace_env_config.GetJaegerUrl(),
      false,
      "trace");
  Tracer tracer{trace_environment, std::move(exporter)};

  constexpr size_t kLineBufferSizePages = 16;
  constexpr bool kNamedPipes = true;
  const size_t event_buffer_size = trace_env_config.GetEventBufferSize();
  const std::set<std::string> blacklist_functions{
    trace_env_config.BeginBlacklistFuncIndicator(),
    trace_env_config.EndBlacklistFuncIndicator()};

  using QueueT = CoroUnBoundedChannel<std::shared_ptr<Context>>;
  auto server_hn = create_shared<QueueT>(
      TraceException::kChannelIsNull);
  auto server_nh = create_shared<QueueT>(
      TraceException::kChannelIsNull);
  auto client_hn = create_shared<QueueT>(
```

```
30      TraceException::kChannelIsNull);
31  auto client_nh = create_shared<QueueT>(
32      TraceException::kChannelIsNull);
33  auto nic_c_to_network = create_shared<QueueT>(
34      TraceException::kChannelIsNull);
35  auto nic_s_to_network = create_shared<QueueT>(
36      TraceException::kChannelIsNull);
37  auto nic_s_from_network = create_shared<QueueT>(
38      TraceException::kChannelIsNull);
39  auto nic_c_from_network = create_shared<QueueT>(
40      TraceException::kChannelIsNull);
41  auto server_n_h_receive = create_shared<QueueT>(
42      TraceException::kChannelIsNull);
43  auto client_n_h_receive = create_shared<QueueT>(
44      TraceException::kChannelIsNull);
45  using SinkT = CoroChannelSink<std::shared_ptr<Context>>;
46  auto sink_chan = create_shared<SinkT>(
47      TraceException::kChannelIsNull);
48
49  std::vector<EventTimeBoundary> timestamp_bounds{EventTimeBoundary{
        lower_bound, upper_bound}};
50
51  auto spanner_h_s = create_shared<HostSpanner>(
52      TraceException::kSpannerIsNull,
53      trace_environment,
54      "Server-Host",
55      tracer,
56      server_hn,
57      server_nh,
58      server_n_h_receive);
59
60  auto spanner_h_c = create_shared<HostSpanner>(
61      TraceException::kSpannerIsNull,
62      trace_environment,
63      "Client-Host",
64      tracer,
65      client_hn,
66      client_nh,
67      client_n_h_receive);
68
69  auto spanner_n_s = create_shared<NicSpanner>(
70      TraceException::kSpannerIsNull,
71      trace_environment,
72      "NIC-Server",
73      tracer,
74      nic_s_to_network,
75      nic_s_from_network,
76      server_nh,
77      server_hn,
78      server_n_h_receive);
79
```

```
80  auto spanner_n_c = create_shared<NicSpanner>(
81      TraceException::kSpannerIsNull,
82      trace_environment,
83      "Client-NIC",
84      tracer,
85      nic_c_to_network,
86      nic_c_from_network,
87      client_nh,
88      client_hn,
89      client_n_h_receive);
90
91  NodeDeviceToChannelMap to_host_map;
92  to_host_map.AddMapping(0, 2, nic_s_from_network);
93  to_host_map.AddMapping(1, 2, nic_c_from_network);
94  to_host_map.AddMapping(0, 3, sink_chan);
95  to_host_map.AddMapping(1, 3, sink_chan);
96  NodeDeviceToChannelMap from_host_map;
97  from_host_map.AddMapping(0, 2, nic_s_to_network);
98  from_host_map.AddMapping(1, 2, nic_c_to_network);
99  from_host_map.AddMapping(0, 3, sink_chan);
100 from_host_map.AddMapping(1, 3, sink_chan);
101 NodeDeviceFilter node_device_filter;
102 node_device_filter.AddNodeDevice(0, 2);
103 node_device_filter.AddNodeDevice(1, 2);
104 node_device_filter.AddNodeDevice(0, 1);
105 node_device_filter.AddNodeDevice(1, 1);
106
107 auto spanner_ns3 = create_shared<NetworkSpanner>(
108     TraceException::kSpannerIsNull,
109     trace_environment,
110     "NS3",
111     tracer,
112     from_host_map,
113     to_host_map,
114     node_device_filter);
115
116 const std::set<EventType> to_filter{trace_env_config.BeginTypesToFilter()
        , trace_env_config.EndTypesToFilter()};
117
118 auto event_filter_h_s = create_shared<EventTypeFilter>(
119     TraceException::kActorIsNull,
120     trace_environment,
121     to_filter,
122     true);
123 auto timestamp_filter_h_s = create_shared<EventTimestampFilter>(
124     TraceException::kActorIsNull,
125     trace_environment,
126     timestamp_bounds);
127 const ComponentFilter comp_filter_server("ComponentFilter-Server");
128 auto gem5_server_par = create_shared<Gem5Parser>(
```

```
129         "parser is null", trace_environment, "Gem5ServerParser",
                comp_filter_server);
130     auto gem5_ser_buf_pro = create_shared<BufferedEventProvider<kNamedPipes,
            kLineBufferSizePages>>(
131         TraceException::kBufferedEventProviderIsNull,
132         trace_environment,
133         "Gem5ServerEventProvider",
134         result["gem5-log-server"].as<std::string>(),
135         gem5_server_par
136     );
137     std::ofstream out_h_s;
138     auto printer_h_s = createPrinter(out_h_s, result,
139                                      "gem5-server-events", true);
140     auto func_filter_h_s = create_shared<HostCallFuncFilter>(
141         TraceException::kActorIsNull,
142         trace_environment,
143         blacklist_functions,
144         true);
145     auto handler_server_host_pipeline =
146         create_shared<std::vector<std::shared_ptr<Handler<std::shared_ptr<
                Event>>>>>("vector null");
147     handler_server_host_pipeline->emplace_back(timestamp_filter_h_s);
148     handler_server_host_pipeline->emplace_back(event_filter_h_s);
149     handler_server_host_pipeline->emplace_back(func_filter_h_s);
150     handler_server_host_pipeline->emplace_back(printer_h_s);
151     auto server_host_pipeline = create_shared<Pipeline<std::shared_ptr<Event
            >>>(
152         TraceException::kPipelineNull,
153         gem5_ser_buf_pro,
154         handler_server_host_pipeline,
155         spanner_h_s);
156
157     auto event_filter_h_c = create_shared<EventTypeFilter>(
158         TraceException::kActorIsNull,
159         trace_environment,
160         to_filter,
161         true);
162     auto timestamp_filter_h_c = create_shared<EventTimestampFilter>(
163         TraceException::kActorIsNull,
164         trace_environment,
165         timestamp_bounds);
166     const ComponentFilter comp_filter_client("ComponentFilter-Server");
167     auto gem5_client_par = create_shared<Gem5Parser>(
168         "parser null", trace_environment, "Gem5ClientParser",
169         comp_filter_client);
170     auto gem5_client_buf_pro = create_shared<BufferedEventProvider<
            kNamedPipes, kLineBufferSizePages>>(
171         TraceException::kBufferedEventProviderIsNull,
172         trace_environment,
173         "Gem5ClientEventProvider",
174         result["gem5-log-client"].as<std::string>(),
```

```
175         gem5_client_par
176     );
177     std::ofstream out_h_c;
178     auto printer_h_c = createPrinter(out_h_c, result,
179                                      "gem5-client-events", true);
180     auto func_filter_h_c = create_shared<HostCallFuncFilter>(
181         TraceException::kActorIsNull,
182         trace_environment,
183         blacklist_functions,
184         true);
185     auto handler_client_host_pipeline =
186         create_shared<std::vector<std::shared_ptr<Handler<std::shared_ptr<
                Event>>>>>("vector null");
187     handler_client_host_pipeline->emplace_back(timestamp_filter_h_c);
188     handler_client_host_pipeline->emplace_back(event_filter_h_c);
189     handler_client_host_pipeline->emplace_back(func_filter_h_c);
190     handler_client_host_pipeline->emplace_back(printer_h_c);
191     auto client_host_pipeline = create_shared<Pipeline<std::shared_ptr<Event
            >>>(
192         TraceException::kPipelineNull,
193         gem5_client_buf_pro,
194         handler_client_host_pipeline,
195         spanner_h_c);
196
197     auto event_filter_n_s = create_shared<EventTypeFilter>(
198         TraceException::kActorIsNull,
199         trace_environment,
200         to_filter,
201         true);
202     auto timestamp_filter_n_s = create_shared<EventTimestampFilter>(
203         TraceException::kActorIsNull,
204         trace_environment,
205         timestamp_bounds);
206     auto nicbm_ser_par = create_shared<NicBmParser>(
207         "parser null", trace_environment, "NicbmServerParser");
208     auto nicbm_ser_buf_pro = create_shared<BufferedEventProvider<kNamedPipes,
            kLineBufferSizePages>>(
209         TraceException::kBufferedEventProviderIsNull,
210         trace_environment,
211         "NicbmServerEventProvider",
212         result["nicbm-log-server"].as<std::string>(),
213         nicbm_ser_par
214     );
215     std::ofstream out_n_s;
216     auto printer_n_s = createPrinter(out_n_s, result,
217                                      "nicbm-server-events", true);
218     auto handler_server_nic_pipeline =
219         create_shared<std::vector<std::shared_ptr<Handler<std::shared_ptr<
                Event>>>>>("vector null");
220     handler_server_nic_pipeline->emplace_back(timestamp_filter_n_s);
221     handler_server_nic_pipeline->emplace_back(event_filter_n_s);
```

```
222   handler_server_nic_pipeline->emplace_back(printer_n_s);
223   auto server_nic_pipeline = create_shared<Pipeline<std::shared_ptr<Event
          >>>(
224       TraceException::kPipelineNull, nicbm_ser_buf_pro,
              handler_server_nic_pipeline,
225       spanner_n_s);
226
227   auto event_filter_n_c = create_shared<EventTypeFilter>(
228       TraceException::kActorIsNull,
229       trace_environment,
230       to_filter,
231       true);
232   auto timestamp_filter_n_c = create_shared<EventTimestampFilter>(
233       TraceException::kActorIsNull,
234       trace_environment,
235       timestamp_bounds);
236   auto nicbm_client_par = create_shared<NicBmParser>("parser null",
          trace_environment,
237                                                      "NicbmClientParser");
238   auto nicbm_client_buf_pro = create_shared<BufferedEventProvider<
          kNamedPipes, kLineBufferSizePages>>(
239       TraceException::kBufferedEventProviderIsNull,
240       trace_environment,
241       "NicbmClientEventProvider",
242       result["nicbm-log-client"].as<std::string>(),
243       nicbm_client_par
244   );
245   std::ofstream out_n_c;
246   auto printer_n_c = createPrinter(out_n_c, result, "nicbm-client-events",
          true);
247   auto handler_client_nic_pipeline =
248       create_shared<std::vector<std::shared_ptr<Handler<std::shared_ptr<
              Event>>>>>("vector null");
249   handler_client_nic_pipeline->emplace_back(timestamp_filter_n_c);
250   handler_client_nic_pipeline->emplace_back(event_filter_n_c);
251   handler_client_nic_pipeline->emplace_back(printer_n_c);
252   auto client_nic_pipeline = create_shared<Pipeline<std::shared_ptr<Event
          >>>(
253       TraceException::kPipelineNull, nicbm_client_buf_pro,
              handler_client_nic_pipeline,
254       spanner_n_c);
255
256   auto event_filter_ns3 = create_shared<EventTypeFilter>(
257       TraceException::kActorIsNull,
258       trace_environment,
259       to_filter,
260       true);
261   auto timestamp_filter_ns3 = create_shared<EventTimestampFilter>(
262       TraceException::kActorIsNull,
263       trace_environment,
264       timestamp_bounds);
```

```
265   auto ns3_parser = create_shared<NS3Parser>("parser null",
          trace_environment, "NicbmClientParser");
266   auto ns3_buf_pro = create_shared<BufferedEventProvider<kNamedPipes,
          kLineBufferSizePages>>(
267       TraceException::kBufferedEventProviderIsNull,
268       trace_environment,
269       "Ns3EventProvider",
270       result["ns3-log"].as<std::string>(),
271       ns3_parser
272   );
273   std::ofstream out_ns3;
274   auto printer_ns3 = createPrinter(out_ns3, result, "ns3-events", true);
275   if (not printer_n_c) {
276     throw_just(source_loc::current(), "could not create printer");
277   }
278   auto ns3_event_filter = create_shared<NS3EventFilter>(
279       TraceException::kActorIsNull,
280       trace_environment,
281       node_device_filter);
282   auto handler_ns3_pipeline =
283       create_shared<std::vector<std::shared_ptr<Handler<std::shared_ptr<
              Event>>>>>("vector null");
284   handler_ns3_pipeline->emplace_back(timestamp_filter_ns3);
285   handler_ns3_pipeline->emplace_back(event_filter_ns3);
286   handler_ns3_pipeline->emplace_back(ns3_event_filter);
287   handler_ns3_pipeline->emplace_back(printer_ns3);
288   auto ns3_pipeline = create_shared<Pipeline<std::shared_ptr<Event>>>(
289       TraceException::kPipelineNull,
290       ns3_buf_pro,
291       handler_ns3_pipeline,
292       spanner_ns3);
293
294   auto pipelines = create_shared<std::vector<std::shared_ptr<Pipeline<std::
          shared_ptr<Event>>>>>("vector is null");
295   pipelines->emplace_back(client_host_pipeline);
296   pipelines->emplace_back(server_host_pipeline);
297   pipelines->emplace_back(client_nic_pipeline);
298   pipelines->emplace_back(server_nic_pipeline);
299   pipelines->emplace_back(ns3_pipeline);
300   spdlog::info("START TRACING PIPELINE FROM RAW SIMULATOR OUTPUT");
301   RunPipelines<std::shared_ptr<Event>>(trace_environment.GetPoolExecutor(),
          pipelines);
302   tracer.FinishExport();
303   spdlog::info("FINISHED PIPELINE");
304
305   exit(EXIT_SUCCESS);
306 }
```