CONTROLLED VIRTUAL TIME ADVANCEMENT IN CONJOINED EMULATION AND NETWORK SIMULATION

BY

VIGNESH BABU

THESIS

Submitted in partial fulfillment of the requirements
for the degree of Master of Science in Electrical and Computer Engineering
in the Graduate College of the
University of Illinois at Urbana-Champaign, 2016

Urbana, Illinois

Adviser:

Professor David M. Nicol

ABSTRACT

Emulations of network services are more accurate than simulated models. However this is achieved at an increased computational cost. Combining emulation with simulation allows more accurate, controllable and repeatable evaluation of applications but such hybrid systems are generally not scalable. Virtual time systems attempt to provide a feasible solution by defining and controlling a virtual clock to alter an experiment's notion of time.

Previous works have motivated and explored the benefits of virtual time systems in improving the scalability of combined emulation-simulation testbeds. One such endeavor resulted in the development of TimeKeeper, an open source virtual time system for Linux. TimeKeeper has been integrated with popular network simulators ns-3, CORE and S3FNet. In this thesis, we extend it further by integrating TimeKeeper with the Extensible Mobile AdHoc Network Emulator (EMANE). We also demonstrate the broad applicability of TimeKeeper by implementing a Programmable Logic Controller (PLC) network emulation tool which can be used to emulate industrial Supervisory Control and Data Acquisition (SCADA) systems. Over the course of the design of these two case studies, we unearthed and fixed a subtle design flaw in TimeKeeper's scheduling mechanism which could potentially starve some processes of CPU time during execution. The purpose of this thesis is twofold (1) to describe improvements to TimeKeeper's design including the logic to ensure fair scheduling and (2) to describe two case studies which demonstrate the scalability and fidelity benefits of running emulations/simulations in close virtual synchrony under the control of TimeKeeper.

To my family and friends, for their love and support.

TABLE OF CONTENTS

LIST OF ABBREVIATIONS

API	Application Programming Interface
CBE	Concurrent Best Effort
CDF	Cumulative Distribution Function
CORE	Common Open Research Emulator
CPU	Central Processing Unit
CTS	Clear To Send
CS	Composite Synchronization
EMANE	Extensible Mobile AdHoc Network Emulator
GTDF	Global Time Dilation Factor
HMI	Human Machine Interface
ICS	Industrial Control System
LXC	Linux Container
MTU	Master Terminal Unit
NEM	Network Emulation Module
OLSRD	Optimized Link State Routing Daemon
OS	Operating System
PLC	Programmable Logic Controller
RAM	Random Access Memory
RTS	Request To Send
RTU	Remote Terminal Unit
TaaS	Testing as a Service
TDF	Time Dilation Factor

TCP	Transmission Control Protocol
SCADA	Supervisory Control and Data Acquisition
SSF	Scalable Simulation Framework
UDP	User Datagram Protocol
VT	Virtual Time
VM	Virtual Machine

CHAPTER 1

INTRODUCTION

Large-scale network services are often provided by complex and interconnected computing devices which need to function cohesively and reliably with each other. Planning and design of such services typically require huge monetary investment and long development cycles leading to implementation issues in software, hardware and deployment. Testing, therefore becomes a necessary pre-requisite prior to live deployment of any new technology and application services. There have been several instances in the recent past like the critical outages in NASDAQ and VISA [1], which underline the dangers of inadequate testing and rushed product deployments. Yet, it is still viewed as one of the areas which can be marginalized to cut production costs.

There are several challenges which need to be overcome to perform effective testing of large and complex computing systems and services. To analyse the expected behavior of the system subject to different stress levels, the test environment has to be of a similar scale and be capable of simulating massive and dynamic loads. Furthermore, if the service needs to be bench marked for comparison with other variants, the testbed evaluations must be reproducible and controllable. Building replicas of the system for testing purposes is not cost effective or scalable and it may be difficult to produce repeatable evaluations under similar work conditions. Thus, designing a scalable, cost-effective, flexible and tractable testing framework could aid in effective testing and evaluation of large-scale applications and services prior to deployment.

Virtual time systems, which are a recent foray into this space, aim to combine the benefits of simulation and emulation and introduce a notion of virtual time to build a scalable and cost-effective testing environment. Simulators offer control over experiment conditions which are necessary to perform reproducible evaluations. Parallel discrete event network simulators like ns-3 [2], CORE [3], EMANE [4] and S3Fnet [5] can precisely simulate link characteristics, delays and traffic patterns to re-create specified workloads and exploit available parallelism to reduce execution time. However, writing simulation models of individual components in the system is often difficult and error prone and needs extensive validation. With the advancement of virtualization technologies, this problem can be addressed by emulating components of the system. Unlike simulated models, emulations are more detailed and mimic the exact functionality of their target components. The expanse in the level of detail and improvement in fidelity however comes at a computation cost which limits the scalability of large-scale emulations. Hybrid systems combine the benefits of simulation and emulation by simulating the links between interconnected emulated components to create a controllable and realistic testing environment but they still suffer from lack of scalability.

As shown is subsequent sections, constraints on computational resources available to hybrid systems could affect the fidelity of experimental results. Virtual time systems seek to address the fidelity and scalability issues by running the emulations in virtual time which progresses at a slower rate than real time. By slowing down the advancement of time, events injected into the simulator are spaced out over larger time intervals which in turn gives the simulator more time to process a larger number of events without falling behind. However, slowing down the advancement of time in emulated components, raises the challenge of maintaining synchrony in virtual time among them to avoid causality violations. Thus to maintain causality and reduce the overall execution time, virtual time systems would require an efficient control phase which directs advancement of virtual time in each individual component with minimal overhead. With these objectives in mind, we developed TimeKeeper, a light-weight virtual time system for Linux [6].

The purpose of this thesis is to demonstrate the scalability and fidelity benefits of integrating network simulators/emulators with TimeKeeper. We build on our previous work which integrates TimeKeeper with ns-3, CORE and S3FNet [7], and extend it further by integrating TimeKeeper with the popular Mobile AdHoc Network Emulator EMANE [4]. Further we also substantiate our claim that TimeKeeper is widely applicable under different contexts, by designing a high fidelity industrial SCADA control Testbed which emulates Networks of Programmable Logic controllers. We use an open source PLC Emulator Awlsim [8] and integrate it with the TimeKeeper+S3FNet system in such a way that emulations and simulation are advancing synchronously in virtual time. Over the course of the design of these two case studies, we also unearthed and fixed a subtle design flaw in TimeKeeper's scheduling mechanism which could potentially starve some processes of CPU time during execution.

The thesis is organised as follows: Chapter 2 motivates the need for a tool like TimeKeeper by discussing the impact of computational resource constraints on emulation results in the context of a routing protocol. Chapter 3 gives a brief overview of related works in this area and their implications. Chapter 4 presents a brief overview of the design decisions in TimeKeeper and subsequent modifications that were made. Chapter 5 presents the first case study which describes and analyses the integration of EMANE with TimeKeeper. Chapter 6 presents the second case study which describes the construction of the PLC Network Emulation Testbed and demonstrates the fidelity of the approach in capturing the operating behavior of a PLC network under varied network conditions and stress levels.

CHAPTER 2

MOTIVATION

In this chapter, we demonstrate the effect of computational resource constraints on the fidelity of experiment results and cite a motivating example to support our claim on the benefits of shifting to virtual time systems.

Ideally, one would expect non-relevant external experimental conditions like the speed of the underlying hardware or other background processes in the system to have negligible impact on the observed experimental results. But this may not always be the case. We demonstrate this claim on an emulation of a simple semi-linear topology running a routing protocol by developing an effective stressing technique to interfere with normal routing path computations.

For this experiment, we chose EMANE [4] (Extensible Mobile AdHoc Network Emulator) which is one of the most widely used wireless network emulators. We constructed a static semi-linear topology resembling a chain of nodes where each node is within the range of two neighbors on either side. Each node in the experiment is emulated by a distinct Linux Container (LXC) running EMANE processes, applications and a routing protocol. Individual LXCs have a fully virtualized network stack and every LXC in EMANE is connected to every other LXC. Each LXC can broadcast messages to every other LXC but the receiver EMANE process decides to process or drop the message depending on the distance and other simulated characteristics of the link between the sender and receiver.

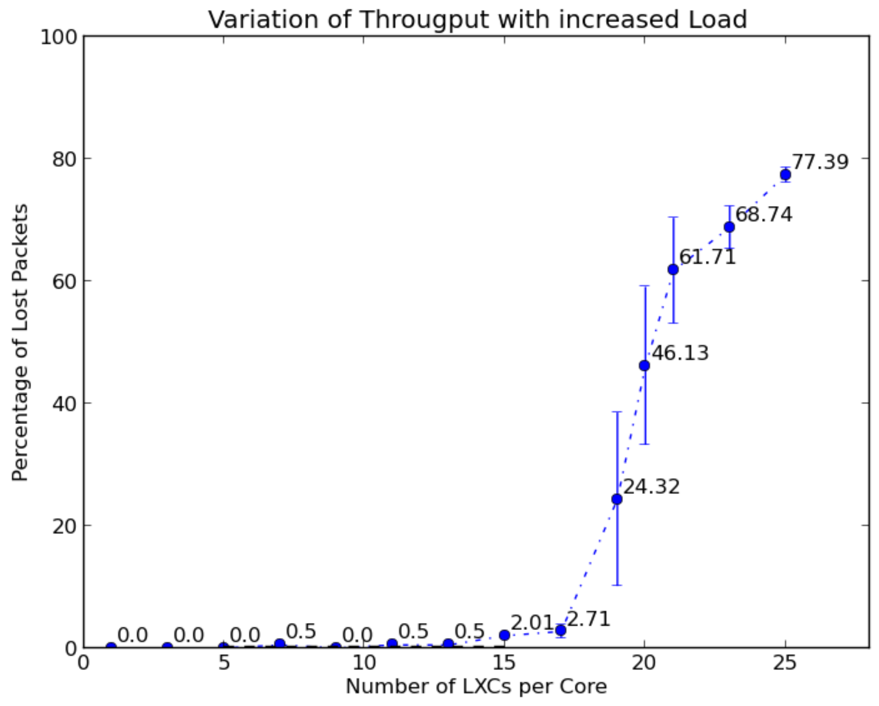
A node at one end of the topology runs a client application which sends UDP packets at a constant rate to the node at the other end running a server application. The server simply echos and logs all received packets locally. EMANE's IEEE 802.11b/g model was used to simulate all wireless links in the topology. All nodes run an instance of OLSRD (Optimized link state routing) protocol [9] to route messages. OLSRD works by periodically exchanging link state updates with neighbors to construct a global view of the network. If a routing update from a particular neighbor does not arrive on time, the neighbor is assumed to have failed and the routing table is altered/flushed. Thus, under the absence of any enforced experimental conditions, the network simulator must ensure that the keep-alive messages get delivered on time to avoid incorrect routing updates.

We impose additional stress on EMANE's ability to deliver these link state messages between nodes by running a high priority background process on alternate nodes. The Linux scheduler allots larger time slices to higher priority processes which can starve other processes on the container. OLSRD processes running on the stress induced LXCs will be starved of CPU time which will interfere with the best effort delivery of link update messages to neighbors. We can furthermore increase the stress by increasing the size of the topology run on a fixed number of cores; this has the same effect of withholding CPU resources from OLSRD processes that need them to keep up with the flow of real time.

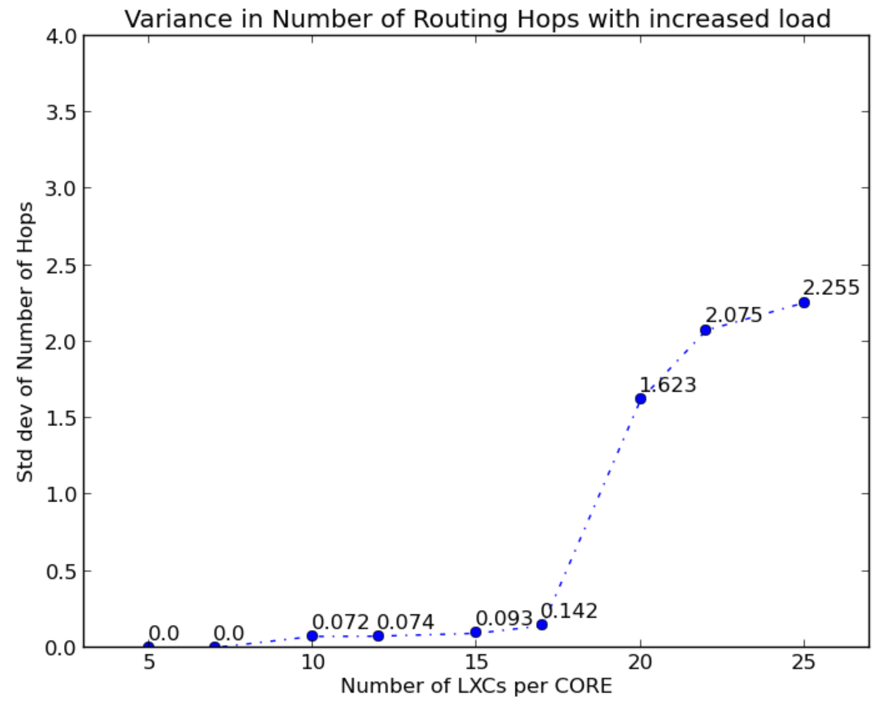
We observed the stressing technique to be very effective. By increasing the number of nodes in the topology, the amount of CPU time available to each LXC per unit real time is reduced. If the LXC is additionally stressed with high priority processes, then OLSRD process running on these LXCs are starved of CPU time and do not send keep alive messages periodically. If a routing update does not arrive within a timeout, the routing daemon could flush its routing tables and all packets which arrive without available routes are dropped. This is observed in the form of increased packet losses in Figure 2.1 where the x-axis is the number of LXCs assigned to each core (of two cores).

The packet transmission rate used in the experiments were low enough so that in a real system packet loss would be very very infrequent. Likewise, under stress we see in Figure 2.1 that there is considerable variance in the number of hops per end-to-end route as the frequency of routing table oscillation increases. In the real system there is no variance.

The observations gathered from the stressing technique confirm our intuition that emulation results can be strongly influenced by irrelevant external factors. To improve the believability of experimental results, we need to de-couple the impact of these extraneous experimental conditions from simulation outcomes. We argue that virtual time systems are a viable solution to this problem. Unlike hybrid simulations advancing in real time, if we can alter a node's perception of time by controlling its clock, we can force the perceived time to progress at a slower rate and make the applications less dependent on the speed of the underlying hardware or other irrelevant factors.



(a) Percentage of Lost Packets



(b) Variance of number of Routing Hops

Figure 2.1: Lost packets and variance in hops in best-effort emulation

CHAPTER 3

BACKGROUND

The notion of virtual time is not new and has been referenced by several works in the past. Lamport [10] first introduced a scheme for assigning virtual timestamps to events in a distributed system so that timestamps of causal events satisfy a well-defined invariant. The scheme defined an artificial clock for each process and advanced it by coarse values at event generation and reception instances. However, such event driven coarse advancement of the virtual clock is only useful when channels are unreliable and delays are arbitrary. In general-purpose simulators, the virtual time advancement needs to be finer because the event send times and receive times are related by the specified channel delay.

Traditionally, there have been two approaches to virtual time (VT) advancement: optimistic and conservative.

3.1 Optimistic Virtual Time Advancement

Jefferson [11] extended Lamport's work and proposed an optimistic virtual time advancement algorithm which uses a rollback mechanism to reorder event execution at a process when a message with a past virtual timestamp arrives at a given virtual time instant. The virtual clock at each process advances in rounds and jumps to the next-lowest virtual receive timestamp in the current round. During such an advancement, the process may generate other events which are added to the receive list of other processes at the end of the current round. Such a virtual time system can be used to emulate the fundamental operation of any discrete event simulator by setting the virtual receive time of an event to its scheduled simulation time.

3.2 Conservative Virtual Time Advancement

The fundamental drawback of hybrid systems built on top of an optimistic virtual time advancement technique is the expensive rollback operation, incurred memory overheads and idempotency requirements.

The only alternative to optimistic rollback mechanisms is to ensure that events are delivered in causal order at all entities. This requires synchronizing the advancement of each emulated entity's virtual clock. It is easy to maintain synchrony in a standalone simulation because the simulated entity's clock is tied to the simulation time and thus the simulator has full control of each entity's clock. However, in hybrid emulation-simulation systems, emulated entities get the notion of time from the system clock. The simulator in such systems, must therefore be fast enough to process and deliver events at appropriate real-time instants consistent with the specified channel delay. This is not always trivial to achieve particularly when the workload is high. If the rate of event injection becomes higher than the maximum event processing rate, the simulator would start to fall behind, leading to causal violations.

In this section, we discuss some of the conservative virtual time techniques that have been studied in the past.

3.2.1 Time dilation

Gupta et al. [12] proposed the notion of time dilation as a potential solution. They define time dilation factor (TDF) as the ratio of rate of progress of real time to virtual time. A TDF of 10 implies virtual time progresses 10 times slower than real time or, in other words, the resources available to the operating system appear to be 10 times faster. An application which is dilated gets more CPU time (real time) to process events per unit virtual time. From the perspective of a hybrid emulation-simulation system, dilating all components of the system would reduce the physical rate at which events are injected into the simulator allowing it to catch up. In this work, the authors give VMs a notion of virtual time. Guest VMs typically synchronize their local clock with the host's clock at periodic timer interrupts using a shared data structure. The Xen hypervisor is modified to change the perceived notion of time of a guest operating system in a VM by scaling the content of the shared data structure and the frequency of timer interrupts with a factor equal to the TDF.

SVEET! [13] is a TCP protocol evaluation testbed built using the time dilation technique discussed above. It stresses the importance of virtual time systems in performance-scalability studies of emerging technologies and demonstrates the cost benefits of time dilation by accurately predicting TCP performance on slower hardware. However, simply changing the OS's perception of time may not be sufficient because some external devices like the disk are not virtualized and drivers for such devices would not perceive scaled response times. To scale the observed Disk I/O throughput with TDF, Diecast [14] which is an extension of [12], employs a disk simulator to compute service time for each I/O request and delays each request by the computed delay in virtual time.

While both SVEET! and Diecast are successful in emphasizing the usefulness of the time dilation primitive, they leave the task of scheduling VMs to the hypervisor. The VMs are usually scheduled with fixed time-slices in a round robin fashion and are not synchronized in virtual time. Each VM's virtual clock experiences interference from other running VMs in the system. The synchronization error would depend on the number of VMs running on host machine. This is not a desirable trait in simulation because the synchronization error needs to be bounded. A control phase is required to schedule entities and precisely direct the advancement of virtual time in each component to keep all components closely synchronized.

3.2.2 Control over the virtual clock

In [15], Zheng and Nicol adopt a different approach to advancing virtual time which is less tied to the advancement of real time. They propose a virtual time system with a simulation control phase which decides how far each container should advance in virtual time. Containers which have advanced too far in virtual time are blocked to allow others to catchup. This keeps all containers closely synchronized in virtual time.

The benefits of such capability are twofold. An application waiting for I/O is suspended and does not contend for CPU resources unlike the previous approach where each VM is guaranteed a virtual time-slice even if the application is blocked. This not only reduces the overall execution time but also allows the simulator to process events without falling behind since it is now able to set the virtual clock of each container when the container resumes and choose to make the virtual time advance faster or slower during the elapsed period. Containers in the experiment are run for a specified virtual time-slice or until they are blocked. The simulation controller can deliver events to containers only during time-slice boundaries and hence the error associated with an event delivery is bounded by the time-slice length. Decreasing the time-slice length will lead to an increased number of expensive context switches and longer execution times.

Zheng and Nicol's approach does not bring the notion of time dilation to the forefront and uses lightweight OpenVZ containers to host applications instead of virtual machines. OpenVZ containers running on a system have isolated process spaces but share the same kernel unlike virtual machines which have their own operating systems and virtualized disks. While this improves scalability, it places a stricter requirement on simulated applications to be lightweight and less sensitive to the view of the underlying hardware. Further, neither of the proposed architectures allows dynamic assignment of TDFs to individual components in the system to reduce execution time.

3.2.3 Variable time-slices

Both of the previously discussed approaches use fixed static time-slices for advancement of entities. This is overly conservative and can lead to context switch overheads if the assigned time-slice is too small. A more effective approach to fully exploit available parallelism would be to allow the simulator to specify virtual time-slice values dynamically. We take a brief digression into synchronization techniques in parallel discrete event simulation to support this claim.

Synchronization techniques are primarily concerned with ensuring that causally incorrect event delivery is avoided. An entity is allowed to advance to a specified time z only after ensuring that no further messages with timestamp less than z will be received at that entity. The most common approaches to tackling this problem are broadly classified into two categories: conservative and asynchronous.

Conservative techniques typically advance each entity by a duration equal to the minimum link delay in the model. Although they are simple and scalable, they make an overly pessimistic assumption that every entity can be affected by every other entity in the model which can reduce the exploitation of parallelism.

Asynchronous techniques on the other hand are less pessimistic and decide each entity's advancement based on only the other entities that can affect it (neighbors). To reduce the overhead associated with asynchronous synchronization and still exploit parallelism, recent works such as [16] by Nicol and Liu propose a composite technique which advances a subset of nodes conservatively and the rest asynchronously. Thus, it is clear that the parallelism benefits offered by asynchronous and composite synchronization techniques can only be leveraged by allowing the simulator to specify the time-slice lengths.

CHAPTER 4

TIMEKEEPER - DESIGN AND IMPROVEMENTS

The discussion presented in the Chapter 3 emphasises the capabilities required to classify a virtual time system as **effective**. Any efficient implementation must have complete control over the scheduling order of emulation entities, advancement of virtual clocks and must support dynamic assignment of TDFs. TimeKeeper [6] is a Linux-based lightweight virtual time system built with these objectives in mind. It differs from other related virtual time systems by allowing a tighter coupling between the advancement of simulation and emulation. TimeKeeper advances emulated entities in rounds. The current round is considered finished when all the emulated entities have advanced by their assigned virtual time-slices. It is able to achieve a tighter coupling between simulation and emulation by allowing the simulator to specify the per round virtual advancement duration for each entity. This is different from other related virtual time systems where the virtual time-slice is fixed prior to the start of the experiment.

In this chapter, we briefly discuss the design choices that were made during TimeKeeper's development and our subsequent contributions to improve the system.

4.1 Design

TimeKeeper's design choices could broadly be divided into three categories: degree of virtualization, control over the scheduling order and running time of processes and control over a process's perceived time. Each requirement is briefly discussed below.

4.1.1 Degree of virtualization

Emulation entities can be virtualized to varying degrees of complexity. In Gupta et al.'s work, VMs ran the emulated applications. VMs are entirely isolated from the host operating system and operate using a virtualized network stack and disks. VMs fall under the category of fully virtualized solutions where the underlying physical system is abstracted out completely. These solutions incur the highest overhead but they allow the ability to run multiple operating systems. Para-virtualization solutions like Xen [17] modify the guest VM's kernel to directly communicate with the host. Lightweight OS-level virtualization solutions like OpenVZ [18] and Linux Containers (LXCs) [19] mandate the guest VMs to share the same host kernel, thereby losing out on the ability to run multiple operating systems. TimeKeeper, uses lightweight Linux containers with fully virtualized network stacks to emulate applications at a large scale. Processes running inside an LXC are isolated from processes running inside other LXCs and during each round, each LXC is advanced by the assigned virtual time-slice.

4.1.2 Control over scheduling order and running time

To control the running time of dilated entities, the control phase must be capable of starting and stopping processes at precise instants of time. The timing errors associated with these actions would be unavoidable because of OS-level overheads but they should be minimal. Kernel-level signalling (SIGSTOP and SIGCONT) is a viable solution currently used by TimeKeeper.

To control the scheduling order of containers, TimeKeeper was designed to maintain a queue of LXCs under its control. During the start of each round, each LXC is assigned a time-slice for which it is required to run (by the simulator) and all LXCs are run one after the other for the duration of their assigned time-slices within that round. A subtle design flaw with this approach is that when an LXC is about to run, TimeKeeper signals all processes inside the LXC to resume at the same time and the scheduling order of these processes is left to the Linux scheduler. This has an unintended outcome because the state of the residual execution times of processes within a container is not retained over successive rounds which could lead to an unfair allocation of CPU execution times to processes within an LXC. We address this issue by redesigning the scheduling mechanism employed by TimeKeeper. This is briefly discussed in Section 4.1.3.

4.1.3 Control over perceived time

Advancing LXCs in bursts requires stopping each LXC's virtual clock when the LXC is not running and scaling the virtual time by a time dilation factor during each time-slice. The control phase would have to set the start time of every entity in the experiment. At the end of each LXC's time-slice, the freeze time must be noted. The amount of physical time elapsed between subsequent burst periods of the LXC also needs to be tracked. The elapsed intermediate physical time must be subtracted from the current real time to determine the dilated running time at the start of the next time-slice. Time-

Keeper performs these necessary operations by modifying the Linux kernel source to add new fields to the process descriptor structure.

When a dilated process gets CPU time, the process must perceive virtual time instead of the actual wall clock time. A process gets its notion of time by querying the operating system using specific system calls. TimeKeeper modifies *gettimeofday* and *sleep* system calls to present the correct virtual time to each dilated process.

4.2 Improvements

4.2.1 Fair process level scheduling

The Linux scheduler meets the requirements for fair scheduling by maintaining multiple queues for runnable processes in the system. A separate queue is allotted for each priority level at each CPU. The scheduler is usually invoked on timer interrupts and it can decide to replace the current running process at a CPU with the next one in its CPU queue or it can pre-empt the current process with a higher priority one. However, in a virtual time system like TimeKeeper, the control phase must be able to circumvent the Linux scheduler's actions because it must be able to precisely control the execution order and running time of dilated processes. Further, over the course of execution, existing processes may spawn new processes which need to be fairly scheduled as well.

The original scheduling mechanism employed by TimeKeeper resulted in unfair execution times to processes within a container. To circumvent this issue, we altered TimeKeeper's scheduling logic to maintain a separate live run queue of processes for each container.

During the start of an LXC's time-slice, the process at the head of the run queue is signalled to run. The execution time quanta assigned to the process is proportional to the relative process priority (i.e. higher priority means a higher positive weight) times the number of threads. However, TimeKeeper also strives to advance all the containers in virtual time more or less uniformly. It moves a window of size W over the virtual time axis, and advances each container in the window by W time before moving the window. For a container with TDF α, the W virtual time allocation is translated into a wallclock αW time allocation. TimeKeeper starts a container running in a window by sending the process at the head of the TimeKeeper queue for that container a Linux SIGCONT signal. It stops that process at the end of its residual service time or at the end of the window (whichever comes first) by sending it a SIGSTOP signal. From a logical point of view, the allocation of execution quanta to processes is orthogonal to the maintenance of temporal synchrony among containers. Algorithms 1 and 2 describe how the LXC's run queue is maintained over successive rounds. Notations cvt and rst denote the current virtual time and residual service time respectively.

A subtle difficulty with this approach is that we cannot always guarantee that a process that is signalled to start at a moment actually starts to run. A process signalled to run is inserted into the run queue maintained by the Linux scheduler and may take while to actually run if there are other processes in the run queue. To ensure that the process immediately starts to run, it must be assigned the highest available priority so that it would preempt any background processes already running.

The solution described above for fair scheduling lets us retain the state of residual execution times and order of execution across successive rounds. However, there is a Linux implementation wrinkle that affects us when em-

Algorithm 1 *run_LXC_one_round*$(LXC\_id, duration\ )$

```
  $LXC\ =\ get\_LXC\_object(LXC\_id)$
  $time\_left\ =\ duration$
  $scan\_add\_new\_processes(LXC, LXC \rightarrow run\_queue)$
  **while** $time\_left\ >\ 0$ **do**
    $head\_process\ =\ LXC \rightarrow run\_queue.pop()$
    **if** $is\_runnable(LXC, head\_process)$ **then**
      $time\_to\_run\ =\ min(head\_process \rightarrow rst, time\_left)$
      $resume\_process(head\_process, time\_to\_run)$
      $pause\_process(head\_process)$
      $head\_process \rightarrow rst = head\_process \rightarrow rst - time\_to\_run$
      **if** $head\_process \rightarrow rst == 0$ **then**
        $reset(head\_process \rightarrow rst)$
      **end if**
      $time\_left\ =\ time\_left\ -\ time\_to\_run$
      $LXC \rightarrow cvt\ +=\ time\_to\_run$
      $update\_current\_virtual\_time(LXC \rightarrow child\_processes, LXC \rightarrow cvt)$
    **end if**
    $requeue(LXC \rightarrow run\_queue, head\_process)$
  **end while**
```

Algorithm 2 *is_runnable*$(LXC, process)$

```
  **if** $process\_invoked\_sleep(process)$ **then**
    **if** $LXC \rightarrow cvt > process \rightarrow wake\_up\_time$ **then**
      $return\ TRUE$
    **end if**
  **end if**
  **if** $process\_invoked\_select(process)$ **then**
    **if** $(LXC \rightarrow cvt > process \rightarrow wake\_up\_time)$
      $OR\ (select\_events\_arrived(process) == TRUE)$ **do**
      $return\ TRUE$
    **end if**
  **end if**
  **if** $process\_invoked\_poll(process)$ **then**
    **if** $(LXC \rightarrow current\_virtual\_time > process \rightarrow wake\_up\_time)$
      $OR\ (poll\_events\_arrived(process) == TRUE)$ **do**
      $return\ TRUE$
    **end if**
  **end if**
  $return\ FALSE$
```

ulated processes use threading.

When Linux starts a process executing, it places all of the process's threads into a separate run-time queue *in a fixed order, independent of any run-time history.* TimeKeeper forces the thread scheduling policy to be "real-time round-robin", and Linux schedules threads with a time quanta that is a function of the process priority. Now when TimeKeeper stops a process with the SIGSTOP signal, all threads are removed from the Linux run-time queue, and (importantly) no memory of the state of the thread queue is retained. When the process is restarted with a SIGCONT, the thread run-time queue is reconstituted in the pre-determined fixed order, giving the threads at the front of that ordering first access to the CPU.

This idiosyncrasy of Linux, coupled with TimeKeeper's reliance on signals to control processes has the potential to starve some threads entirely, and certainly has the potential to give unintended priority to some threads over others. This is illustrated in Figure 4.1 using a simple deterministic simulation of a four-threaded process under various assigned time-slices (cycle durations). The plot clearly indicates how thread 1 always gets more CPU share while thread 4 is starved under these conditions.

The main defense we have against such thread level starvation is a sufficiently large value of αW. But, as we will see, this may conflict with our objective of high temporal accuracy of the integration of emulation and simulation. Fortunately, this is an issue only for containers whose processes have multiple threads, and in Chapter 5 we discuss use of a *Global Time Dilation Factor* that can also address this issue.

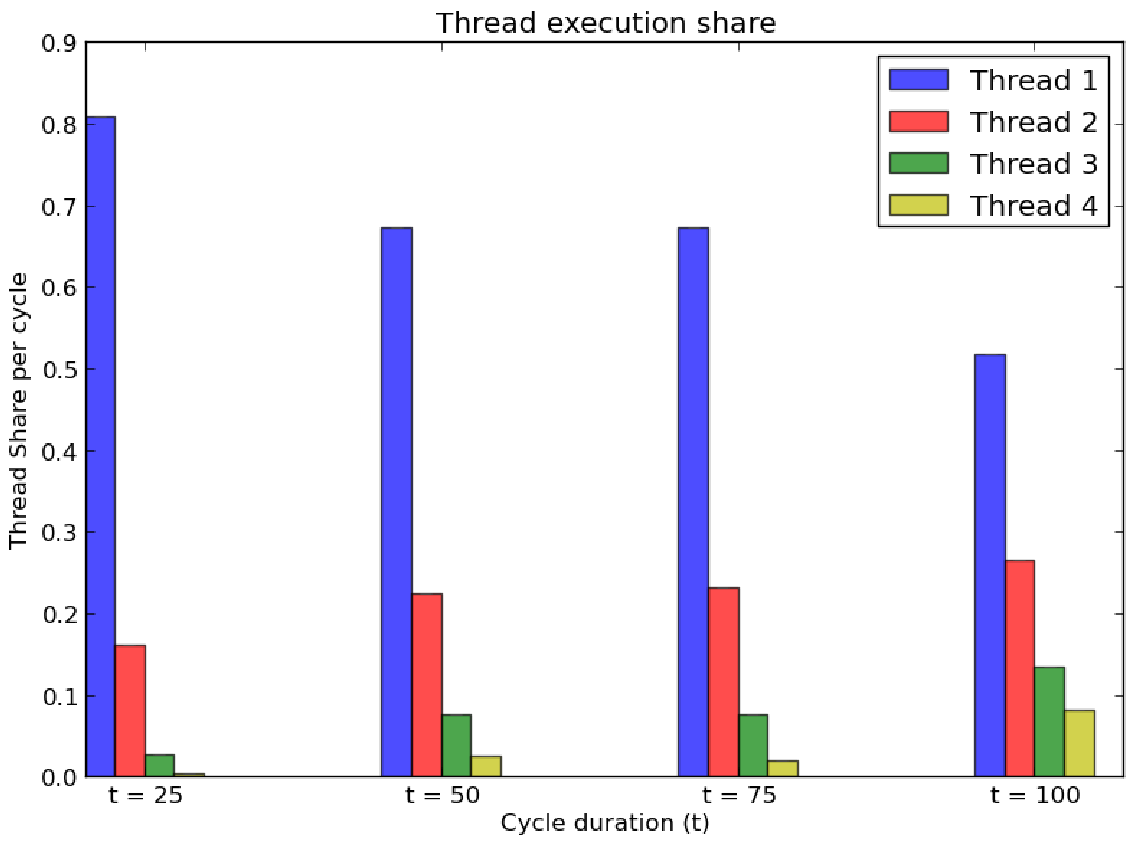


Figure 4.1: Thread execution share for varying cycle durations

4.2.2 Support for additional system calls

We extended the set of system calls supported by TimeKeeper to include two additional system calls: *select* and *poll* which are commonly used for scheduling actions after specific timeout values or after the occurrence of specified events. Poll and select system calls normally block the execution flow by waiting for specific timeouts or events. This is implemented by simply removing the process from the corresponding LXC's active run queue. TimeKeeper checks for the occurrence of events at the start of every round. These blocked processes are inserted back into the run queue after event detection or after the virtual time exceeds the timeout. Algorithm 3 describes how a dilated select system call is handled internally. The algorithm for handling a dilated poll system call is also very similar and hence not included for brevity.

Algorithm 3 $do\_dilated\_select(process, events, timeout)$

```
$S \leftarrow dilated\_processes\_which\_invoked\_select$
if $timeout > 0$ then
    $S \leftarrow S \bigcup (process, events)$
    $LXC = get\_container\_LXC(process)$
    $process \rightarrow wake\_up\_time = LXC \rightarrow cvt + timeout$
    $pause\_process(process)$
else
    $do\_normal\_select(process, events)$
end if
```

CHAPTER 5

CASE STUDY: EMANE WITH TIMEKEEPER

This chapter describes the integration of EMANE with TimeKeeper and demonstrates the scalability and fidelity benefits of such a virtual time driven emulation platform. Further, it also briefly discusses use of a *Global Time Dilation Factor* to suppress the effects of thread-level scheduling artifacts described in Chapter 4.

5.1 EMANE

The Extensible Modile Adhoc Network (EMANE) simulator, developed by [4], is a widely used ad-hoc wireless simulator. EMANE can be used as a standalone package or it can be integrated with CORE and other network simulators to enable robust simulation of wireless links. In this section, we give a brief overview of EMANE and describe the integration of TimeKeeper with the standalone EMANE package.

5.1.1 Overview

EMANE provides Network Emulation Modules (NEMs) to emulate data-link and physical-layer models. EMANE provides data-link layer models such as IEEE 802.11a/b/g and RF-Pipe. The default physical-layer model referred to as the universal physical-layer is responsible for emulating several effects of wireless links such as fading, collisions, interference and noise. In EMANE's implementation, the physical-layer adds a header to each outgoing packet and every outgoing packet is broadcast to every other node in the network. On receiving a packet a node examines the header to determine if the packet should be passed up the stack or dropped. The fields in the header include detail of information such as the sender's location (which can be used to estimate signal strength), sender frequency and the transmit time from which the the propagation delay is calculated.

EMANE uses a transport interconnect to transfer packets between the application and the NEM. The transport interconnect interfaces with NEMs using tun-tap devices. EMANE creates a separate network interface in each LXC to capture application generated packets through the transport interconnect. NEMs and transports function as independent multi-threaded processes. A third independent component called the platform server is responsible for relaying data between NEMs. It collects packets transmitted by the physical-layer and broadcasts it to other NEMs. The goal of integrating EMANE with TimeKeeper forced us to deal with multi-threaded processes, a forcing function with significant ramifications, as we will see.

5.1.2 Deployment configurations

EMANE supports two deployment configurations: centralized and distributed. In a centralized deployment, a separate transport instance is launched inside each LXC, while all NEM instances and a single common platform server instance are launched in the base system outside all LXC process namespaces. All transports interact with NEM instances outside the LXC namespace and messages are relayed by the platform server to different NEM instances using common IPC techniques. In the fully distributed deployment each LXC has its own NEM instance and platform server instance inside its process namespace and messages are multicasted to other LXCs using the ACE wrapper for UDP/IP multicast [20]. We chose to integrate TimeKeeper with the distributed EMANE configuration because it is the most commonly used configuration and offers greater flexibility in allowing the experimenter to assign independent TDFs to each LXC and possibly run experiments on a distributed cluster.

5.1.3 Integration with TimeKeeper

Integration of EMANE with TimeKeeper requires no modifications to EMANE source code. The general architecture of the EMANE-TimeKeeper integration is shown in Figure 5.1. We developed an user interface in python to allow the experimenter to easily specify the dilation factors for each node, model parameters, initial physical locations for every node and specify the applications to run on each LXC. The interfacing code launches EMANE in a distributed deployment configuration which was described earlier. Initial locations specified in the user interface are broadcast to all participating nodes at the start of the experiment. The application is responsible for simulating node movement through sharing of so-called location events. All launched LXCs are added to a synchronized CBE experiment.

Channel delays in EMANE are not explicitly specified. Instead, propagation delays are calculated inside the emulator based on the distance between the transmitting and receiving node. Similarly, the transmission delay is calculated based on the transmission bandwidth and the received packet size. On receiving a packet, EMANE first estimates when to process a packet by computing the transmission and propagation delays. An internal timer is then scheduled to fire after the computed delay elapses and subsequently, the packet is processed and pushed up the stack.

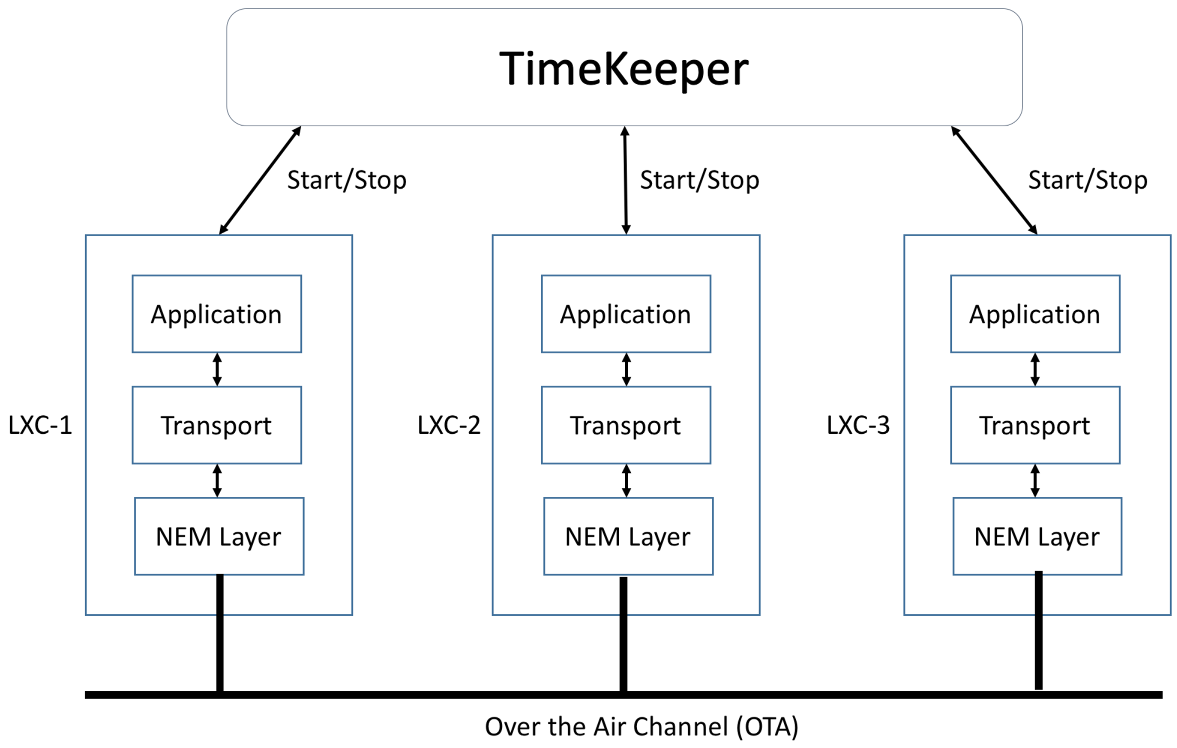


Figure 5.1: EMANE-TimeKeeper integration

The challenge with distributed simulation of EMANE (indeed, a typical challenge for distributed simulation of ad-hoc networks in general) is that generation of message traffic is unpredictable, and the propagation delay of a wireless message is very small. In the TimeKeeper context, temporal accuracy means using a synchronization window for each round that is small enough so that a packet is not both sent and received in the same window. A lower bound on the time between when the first bit of a packet is transmitted and when the last bit is received is the minimum packet size (in bits) divided by the channel bandwidth (in bits per second). As these quantities are buried within EMANE, we do ask the user to provide them. The TimeKeeper window size W is set to this product.

5.1.4 Evaluation

Our intuition is that embedding emulation into virtual time and integrating it with simulation may do "a better job" at reproducing behavior seen in real systems than would simply using best effort. For our evaluation of the EMANE integration with TimeKeeper we use routing behavior to see what the differences may be between EMANE behavior with and without TimeKeeper coordination. We measure the difference between routes selected under the system organizations in terms of the effects the chosen routes have on traffic throughput and latency. For the experiment we used OLSRD [9], a routing module available for use in EMANE.

The EMANE-Timekeeper integration starts an instance of OLSRD in every LXC. OLSRD is a link state based routing protocol in which neighboring nodes exchange periodic keep-alive messages. Keep-alive messages carry link state and the sender's view of the overall topology. Through these messages each node can construct an overall view of the topology and determines the best path to each destination. The protocol implementation allows specification of a periodic keep-alive exchange interval and duration of time for which the link-state information for a particular node is valid. If a node does not get a routing update from a neighbor within the validity period, it considers the link between them to be broken. This of course impacts the routes it subsequently chooses. Thus we see that the behavior of the emulation depends on the network simulation delivering keep-alive messages to the emulation in a timely fashion with respect to the emulation's view of time. Integration with TimeKeeper ensures this is the case.

It was previously shown in Chapter 2 that by engineering a set of stress conditions that affect EMANE's ability to deliver link update messages on time, the behavior of EMANE without TimeKeeper started to diverge significantly

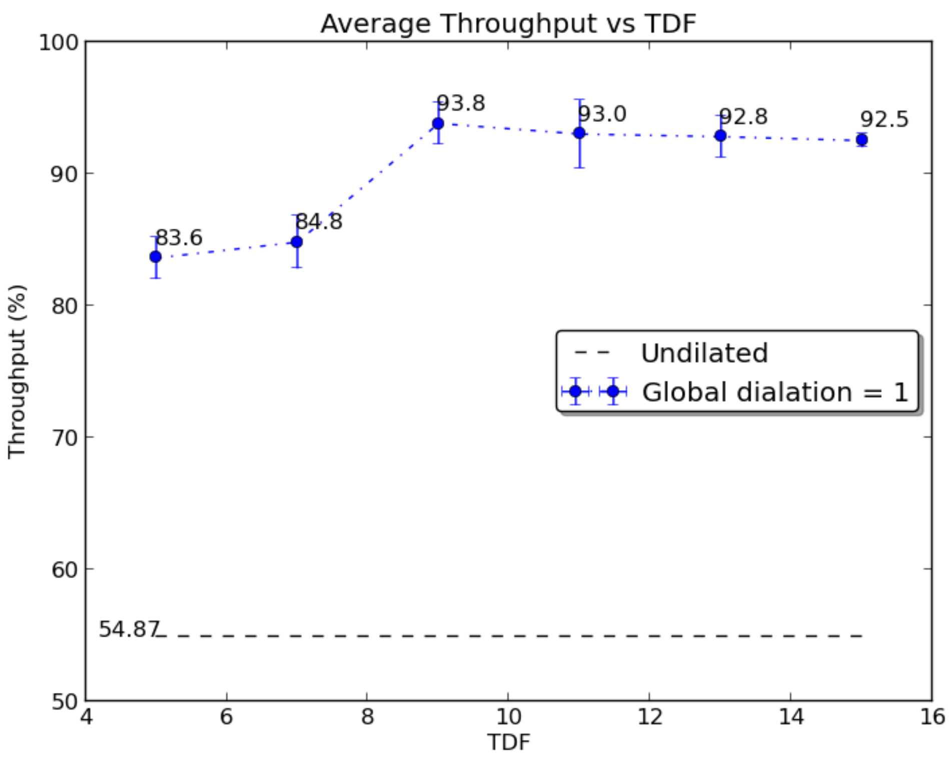


Figure 5.2: Average throughput, GTDF=1

from what is expected. On the contrary, Figure 5.2 shows the throughput of a 43-node chained topology emulated by EMANE with TimeKeeper under the same experimental conditions. From the figure, it is clear that EMANE with TimeKeeper exhibits much lesser deviance in comparison with best effort emulation. However, interestingly the application throughput appears to depend on the TDF of the routers. To explain this dependence, we were forced to understand how Linux manages threads internally and how thread-level scheduling artifacts could have impacted results of the stress test. In Section 5.2, we present our findings and advocate the use of a Global Time Dilation factor to suppress the effects of thread-scheduling artifacts.

5.2 Global Time Dilation

Our experience analyzing the behavior of EMANE managed by TimeKeeper forced us to understand how Linux treats threads. The observations indicated that the application traffic throughput is affected by the assumed TDF of the routers. As TDF increased, we observed an increase in the throughput relative to the theoretical maximum (determined by the application data injection rate), see the GTDF=1 graph at the top of Figure 5.2. The error bars show the standard deviation over 10 experiments. What does this mean? Why are more packets being lost at smaller TDFs?

To explain these trends, we first need to understand how TimeKeeper is able to maintain synchrony between emulated entities. The TimeKeeper synchronization window is the smallest virtual time by which each container can safely advance while maintaining causal packet delivery times. It ensures that the completed receipt of the packet occurs in a different synchronization window than its transmission, which means that TimeKeeper keeps the sender and receiver in close temporal synchrony. Assuming a synchronization window of *virtual* duration W, Timekeeper translates it into a *physical time* duration αW where α is the TDF assigned to the container. Processes belonging to the container are started using the SIGCONT signal, and stopped using the SIGSTOP signal. The Linux scheduler puts the threads of the started process into its thread run-queue in a fixed order, independent of the state of the run-time queue when the process was stopped. Unfortunately, stopping and re-starting a process are not transparent to the behavior of a threaded application, because the threads at the front of the fixed order get preferential service through the start/stop mechanism TimeKeeper employs. We observed that when the time quanta per container per window is small enough some threads are actually starved (Figure 2.1). Even when not starved, withholding enough resources from the thread that provides keep-alive messages impacts routing behavior, as we have seen in earlier experiments. Larger scheduling quanta give the threads the opportunity to run, suspend themselves, and yield the CPU to other threads in need of service.

The TDF of a container is a characteristic of the model which reflects the speed difference between executing code on the emulator's CPU, and executing the code on a device in the field. From a user's point of view, the TDF of a container is a given fixed value. We are faced then with a seeming conflict between the desire for temporal fidelity achieved by small scheduling windows in virtual time, and the desire for functional fidelity which requires larger scheduling windows.

One approach to this dilemma is to yield on the requirement of temporal fidelity. However, depending on the type of application, this approach can impact system metrics of interest. In the context of the OLSRD routing example, the impact is increased variance in the experienced link delays, because looser synchronization allows a packet to be received earlier, or later in the timeline of the recipient than it does under tighter synchrony. The frequency of keep-alive messages is so small that the synchronization window would have to become very large indeed for this variance to impact timely receipt of keep-alive messages, however the latency of application packets will be impacted.

In this section, we describe another approach which is the notion of *global time dilation factor* (GTDF). Increasing the TDF of an LXC changes the view of virtual time of the emulator, *but not the simulator*. We can artificially (and arbitrarily) rescale the virtual time units of both emulator and simulator simultaneously. The effective TDF of a container would be its native TDF, multiplied by the GTDF. In the simulator all simulation durations would multiplied by the GTDF, all bandwidths would be divided by the GTDF, internal processes that occur at rate λ in virtual time now occur at rate $\lambda/GTDF$. From the simulator's point of view, changing the GTDF is equivalent to changing the units of virtual time. Conceptually, by employing GTDF we could achieve the same functional behavior of the system, but using different units of virtual time, slowing both emulation and simulation enough to create large enough *physical time quanta* to deal with the issues created by threading.

It is worth noting that mechanism for rescaling simulation time is related to, but distinct from what is achieved in SVEET [13] and in our treatment of ns-3. In both cases the simulation clock is the product of the wall clock times some scaling factor, and governs when simulation events are executed. In a certain sense the effect is the same as with GTDF, but in another sense not. For SVEET and ns-3, the virtual time units do not change; what changes is the rate at which simulation time advances, and remains the time-scale of the system being modeled. With GTDF the time units actually change, artificially, in both emulation and simulation. Correspondingly the reported results of the experiments have to be rescaled to undo the impact of applying GTDF.

Implementation of GTDF requires support from the simulator, and means of un-scaling results. It is not transparent. Yet, as we will see with preliminary experiments, it can deliver the desired effects. Figure 5.3 shows the application throughput on the routing topology (40 nodes) with a GTDF of 5. Here we see that the benefit of larger real-time scheduling quanta can now be enjoyed on models with small native TDF. Still, under the model conditions we have assumed, we expect for the full 100% of the theoretical throughput to be captured. There may be still undiscovered factors impacting the application performance. This experiment is thus merely a proof-of-concept. There are a number of issues remaining that have to be address if use of

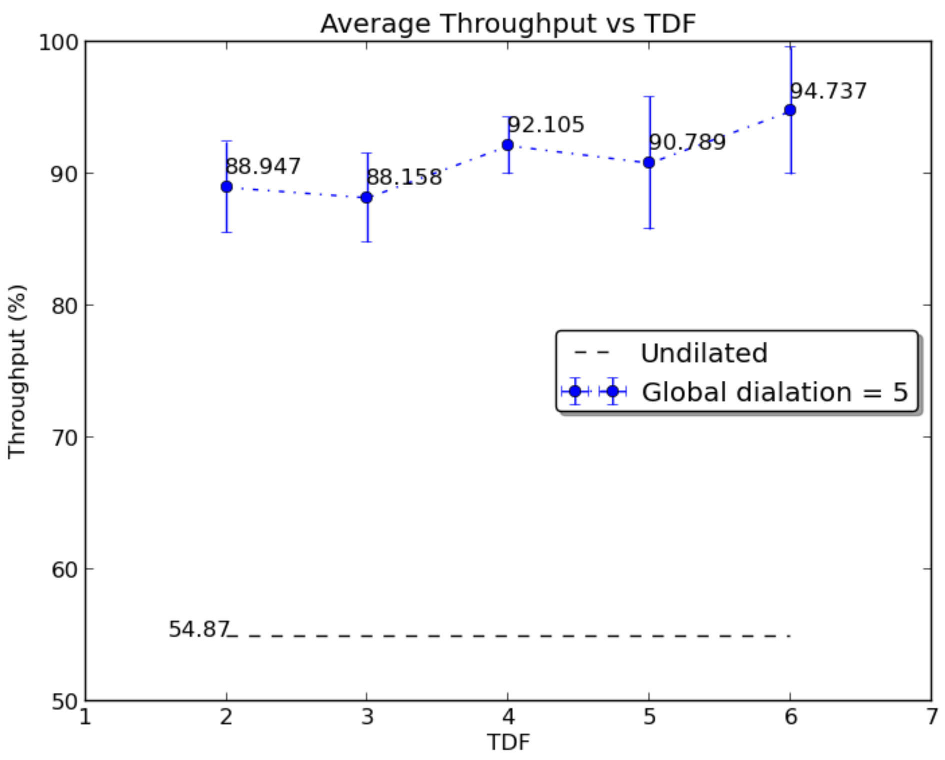


Figure 5.3: Throughput affected by GTDF

GTDF can be made transparent.

While GTDF is a useful concept as a stop-gap measure, we believe that the best possible solution is one where TimeKeeper's technique for starting and stopping processes is transparent to the behavior of the computation. If this were possible the only impact of small scheduling quanta would be the overhead. If we are to have any hope of using threaded processes with the fine-grained synchronization approach we have developed for S3FNet, this will be absolutely necessary. Our future work will investigate our options there.

CHAPTER 6

CASE STUDY: EMULATION OF PLC NETWORKS

In this chapter we substantiate our claim that TimeKeeper is widely applicable under different contexts, by designing a High Fidelity Industrial SCADA control Testbed which emulates Networks of Programmable Logic controllers (PLCs).

PLCs are devices frequently used in industrial control systems with tight real-time constraints on operations. Using emulation and/or simulation to evaluate the behavior of a network of PLCs is difficult because of the lack of tools that accurately mimic the real-time behavior of such networks. This case study addresses this issue by showing how to tightly integrate instances of a PLC emulator Awlsim with the network simulator S3F, in such a way that emulations and simulation are advancing synchronously in virtual time. We demonstrate fidelity of the approach in capturing the operating behavior of a PLC network under varied network conditions and stress levels.

6.1 Introduction

Programmable Logic Controllers (PLC) are an integral part of modern manufacturing plants and critical infrastructures like power stations, marine docking operations, and water treatment units. Their use helps to cut costs, minimize human errors and improve system capability. Networks of PLCs monitor data from different subsystems and send control signals to actuators that impact the behavior of a physical system.

It is difficult to test a network of PLCs without actual deployment. The program of an individual PLCs can be tested via emulation, but evaluation of a *network* of PLCs requires a communication infrastructure, and importantly, a means of ensuring that the temporal aspect of messaging behavior in the evaluation testbed is a good model of what will happen in the field. Effective testing therefore calls for a flexible, scalable and low-cost testbed which can produce realistic behavior. This testbed would facilitate research in cyber-security of industrial control systems using PLCs, where the difficulty associated with gathering generic data has been the main obstacle to development of efficient defensive measures [21]. With the emergence of virtualization technology, parallel simulation/emulation could potentially be an economical solution.

Simulators offer control over experiment conditions, which is necessary to perform reproducible evaluations. In [22], the authors simulate a small virtual factory with four turning machines by writing simulation models for each subsystem in the manufacturing cell. However, writing simulation models of individual components in the Industrial Control System (ICS) is often difficult and error prone and needs extensive validation. Thus it is easier to use emulations of these devices instead.

Previous works on development of testbeds for industrial control systems have typically been either implementation based or emulation based or a combination of both. An example of an implementation-based testbed is [23] and it consists of real PLCs and HMI devices. Data collected from implementation-based testbeds is more accurate and realistic but the costs of expanding the testbed and maintaining it often outweighs the benefits. In [24], the authors propose an open virtual testbed for an ICS which interconnects emulated models of PLCs and Master Terminal Units (MTU) and Remote Terminal Units (RTU) with each other and with a central process simulator which simulates inputs to these virtual devices. However, the testbed does not simulate network links between its components, making the experiments non-repeatable. Best-effort emulation and implementation testbeds such as [25] also exist where physical devices are interconnected with virtual emulated devices. However they are still expensive to maintain and less flexible.

This chapter describes the integration of a PLC emulator Awlsim [8] within the TimeKeeper+S3F system. S3F simulator [5] is a parallel discrete event network simulator built on top of the Scalable Simulation Framework (SSF) API. SSF was redesigned in [26] to improve performance of discrete event simulations by exploiting potential parallelism. S3F relies on standard C++ libraries which attest to its efficiency and simplicity. We chose S3F because of its capability to support creation of complex communication network models using devices like switches and routers operating on top of layered protocols like TCP/IP. In addition, S3F also supports emulation using LXCs and OpenVZ containers which can be conveniently used to run emulations of PLCs. Further, conventional PLC networks support serial (RS-232) capability and hence we augmented S3F with the notion of serial non-IP-speaking communication lines, and with a model of the ModBus communication protocol running on those serial lines. We demonstrate empirically that under TimeKeeper+S3F management, a network of PLC emulations produce expected behavior, but when run under "best-effort" control they do not.

6.2 Motivation

A programmable logic controller (PLC) is a computing device designed for use in industrial control systems. A PLC's role typically is to monitor inputs from sensors, and issue commands to actuators in a physical system as a result of the inputs it has read, and inputs other PLCs have read and communicated to it. Networks of PLCs are in the class of "hard real-time systems", meaning that they are designed to respond to changes in inputs within a set period of time. Failure to do so, for any reason, can mean catastrophic consequences for the controlled system.

To support hard real-time operations, PLC applications run on top of custom operating systems. The applications themselves (and the languages used to describe those applications) are designed around this "read-then-respond" model. A PLC program is described as organization of cycles, where in each cycle one or more inputs are read, a calculation is made, then one or more output messages to actuators or to other PLCs are issued. Determinism in execution time and predictability of program behavior are key in designing PLC networks with predicable execution timing. However, because a PLC's execution behavior can depend in part on communication with other PLCs, the timing of the network supporting inter-PLC communications is also critical to PLC functional behavior and meeting of real-time constraints.

Our penultimate goal is to study cyber-security issues in networks of PLCs. The work reported here is necessary to support that goal, in that our studies will initially involve *simulation* of cyber-attacks and simulation of the impacts that the detection and defensive measures have on the operations of the PLC networks. Therefore, we need a means by which actual PLC programs can be responding to simulated inputs, can communicate with each other over a simulated network, and can send commands to simulated actuators. This integration of emulation and simulation must be as careful as possible to capture the temporal behavior of PLC networks as they operate in the field. As we have previously developed the integrated TimeKeeper/S3F combination to provide fine-grained temporal control over emulations in the context of network simulation, we identified a suitable PLC emulator to integrate with TimeKeeper/S3F. This task required some modifications to the PLC emulator, TimeKeeper and S3FNet. In this chapter we focus on how those modifications deliver much better predictability in temporal behavior than is achieved using "best-effort" emulation. With this basis, our continuing work will focus on studying cyber-security issues in PLC networks.

6.3 Testbed Description

Awlsim [8] is an open source Python-based PLC emulator which supports a large subset of Step 7 programming instructions. It emulates Siemens S7-300 and S7-400 PLC CPUs. PLC programs are written in Step 7 STL language and the source file is simply presented as an input to the emulator. Awlsim also allows the user to simulate sensor inputs and read output values via a GUI interface. We choose Awlsim in part because it is open source, and in part because of its extensive support for real Step 7 programs. However, Awlsim does not itself support the notion of networked PLCs. We provided this by developing additional system blocks/instructions that would allow two Awlsim instances to communicate with each other using the MODBUS application layer protocol.

MODBUS, developed by Modicon in 1979 is a widely used standard for communication over industrial networks still being widely used today [27]. It is a simple master-slave based communication protocol which can run over many physical layers including RS-232, RS-485 and in the TCP/IP mode

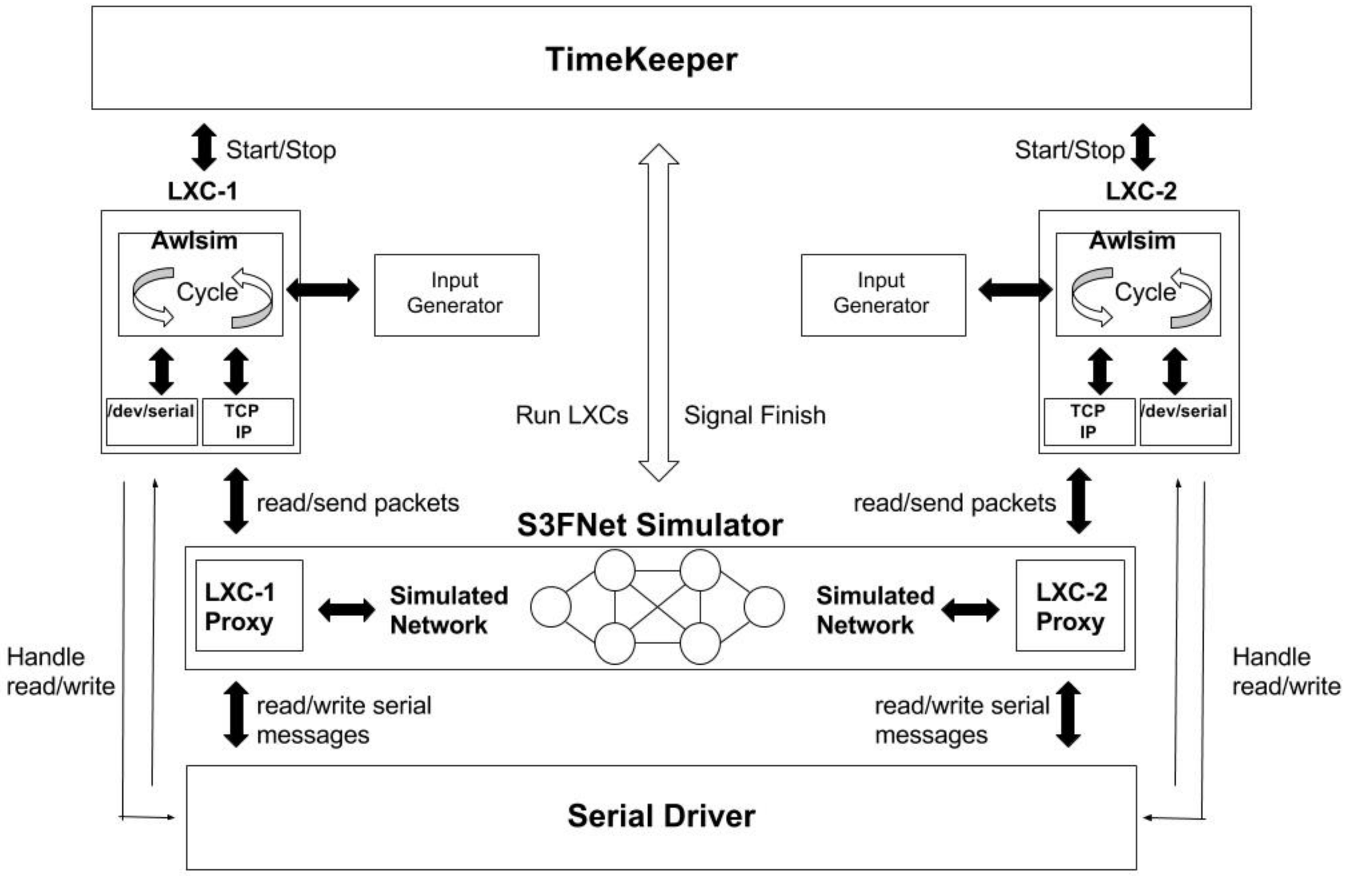


Figure 6.1: General architecture of the PLC network testbed

over ethernet. We implemented a Python-based MODBUS stack which is used by the Awlsim's ModBus instructions we developed to send and receive ModBus messages over simulated IP or serial networks.

6.3.1 General architecture

Figure 6.1, shows the architecture of the TimeKeeper controlled hybird PLC emulation-simulation framework. Emulated PLC instances are run in separate Linux containers whose execution times and scheduling order are controlled by S3F through TimeKeeper. The user is allowed to assign an input generator script for each emulated PLC instance. The input generator script is invoked by the PLC instance at the start of every cycle and the script can be used to model the physical process being monitored by the PLC and alter the inputs to the PLC program. When the assumed connectivity is that of an IP network, packets that are injected by emulated PLCs are captured by the S3F network simulator and injected into the appropriate destination container after the specified link delay. Many legacy PLC systems use point-to-point serial communications. We needed to modify both TimeKeeper and S3F to support serial communication; our solution requires bypassing the Linux network stack completely.

6.3.2 Serial connections

Serial connections were simulated by implementing a serial driver which manages reads and writes to device files. Each Linux container running an emulated PLC instance is assigned a separate device file to which it can read and write data to be sent over a simulated serial connection. The emulated PLC is allowed to maintain multiple serial connections simultaneously by specifying a connection ID along with the read or write requests. We augmented S3F to poll device files of every emulated entity for any available data among all active connections. It then simulates a half-duplex RS-232 connection with handshakes (RTS, CTS) before sending the data over the simulated link. The received data is then written into the destination device file to be later read by the emulated PLC.

6.3.3 Experiment configuration

Within a configuration file read at the beginning of the experiment, a user specifies the network type (IP or Serial) and the Time dilation factor for each emulated PLC. The general topology description and link delays are specified in a separate configuration file. In the IP mode, each PLC is connected to a simulated router by default and different routers are connected with each other in a user specified configuration. In serial mode, the connection is point to point, and thus the user specifies direct links and link delays between different PLCs.

6.3.4 HMI devices and compromised routers

The testbed also supports emulation of Human Machine Interface (HMI) devices to interact with emulated PLCs. HMI devices are frequently used by human operators of industrial control systems to send and receive messages to monitor and update the operator's view of the current system state. HMI devices function as ModBus master devices and can send commands to PLCs functioning as ModBus slaves. The user can leverage the implemented ModBus stack API to write models of HMI devices and send arbitrary commands to slave PLCs.

The testbed also allows the user to experiment with different attack scenarios on the system and study its resilience. The user can designate certain routers in the topology as compromised. Routers which are classified as compromised invoke a specified attack script upon reception of each simulated packet. The user-defined script can passively examine the packet or even perform man in the middle attacks by modifying it. We are using this feature of the testbed to study the effectiveness of intrusion detection algorithms under different active and passive adversarial attack models.

6.4 Evaluation

We now demonstrate how the testbed manages to deliver predictable low-variance behavior as exhibited in a real system, as compared to behavior of the same emulation under best-effort coordination. All of the experiments described here were performed on a single dual core machine with 16 GB of RAM. We consider a hypothetical job routing scenario in a bottling plant. Jobs arrive at an incoming root node which performs some operations on the incoming job before routing the job to one of its child nodes. The topology is organized as a full binary tree of PLCs, with leaves being dispatch units where the finished product leaves the manufacturing plant. For simplicity, we assume that all nodes at a particular level in the binary tree perform the same action (i.e., run the same application) on its incoming jobs. This scenario is similar to automated baggage transfer in modern airports and bottle filling operations in bottling plants.

In this scenario, each PLC controls two conveyor belts connected to two other PLCs (child nodes in the binary tree,) and is also linked to both nodes over a high-speed ethernet connection. An incoming job is routed by the PLC to the child node with the least congestion in its sub-tree. To maintain a real-time view of congestion in each child sub-tree, nodes repeatedly query their children for congestion levels in their sub-trees. The performance of such a dispatch system relies heavily on the frequency of updates and the speed of processing of these messages. Incoming jobs are simulated as sensor inputs at each node. Sensor inputs are scheduled at the chosen child node after the routing decision is made at the current node. Dispatch jobs are also modeled as simulated sensor inputs to the leaves of the tree. All inter-node communication links were assigned the same delay of 4 ms. Figure 6.2 depicts the topology used.

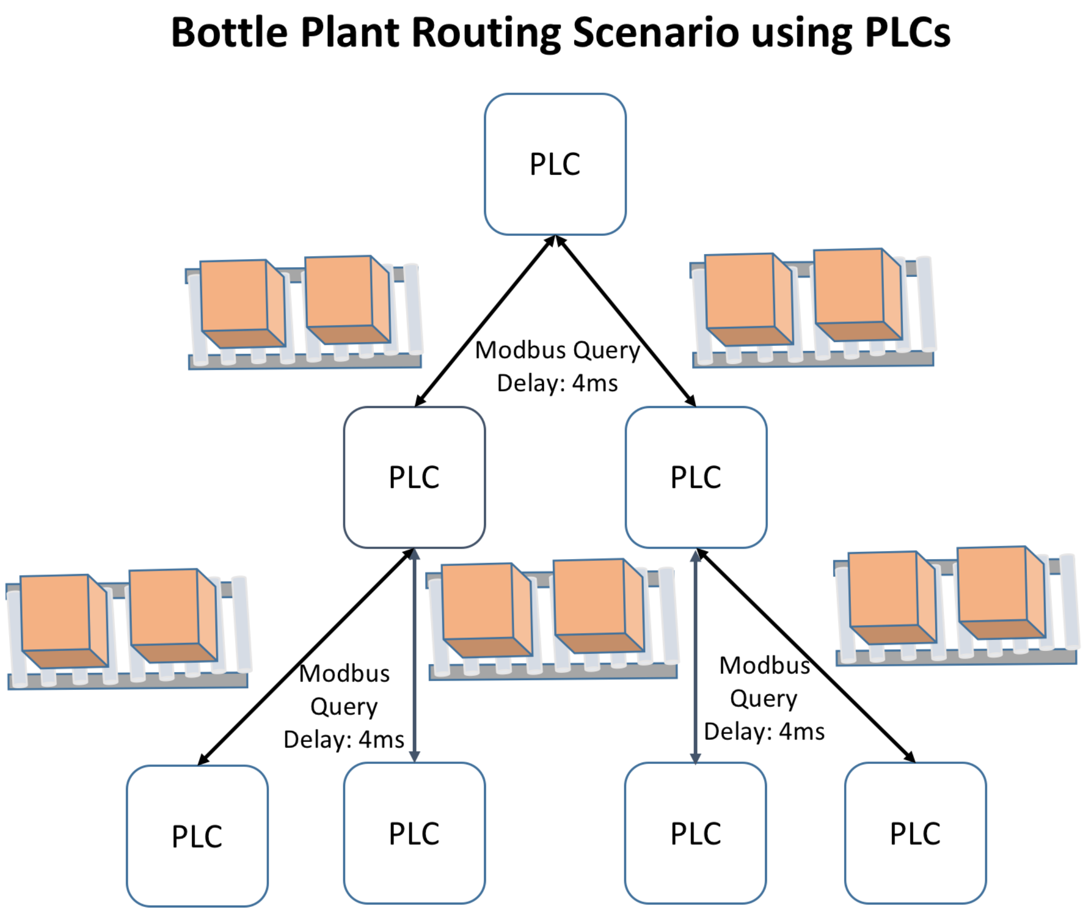


Figure 6.2: Bottle plant routing scenario

We impose stress on the test topology by forcing nodes to send updates at the fastest possible rate. Each node queries both its children, waits for their reply, and then initiates the next query immediately. We define the following metrics to analyse the accuracy of the testbed in simulating the expected behavior of the target system.

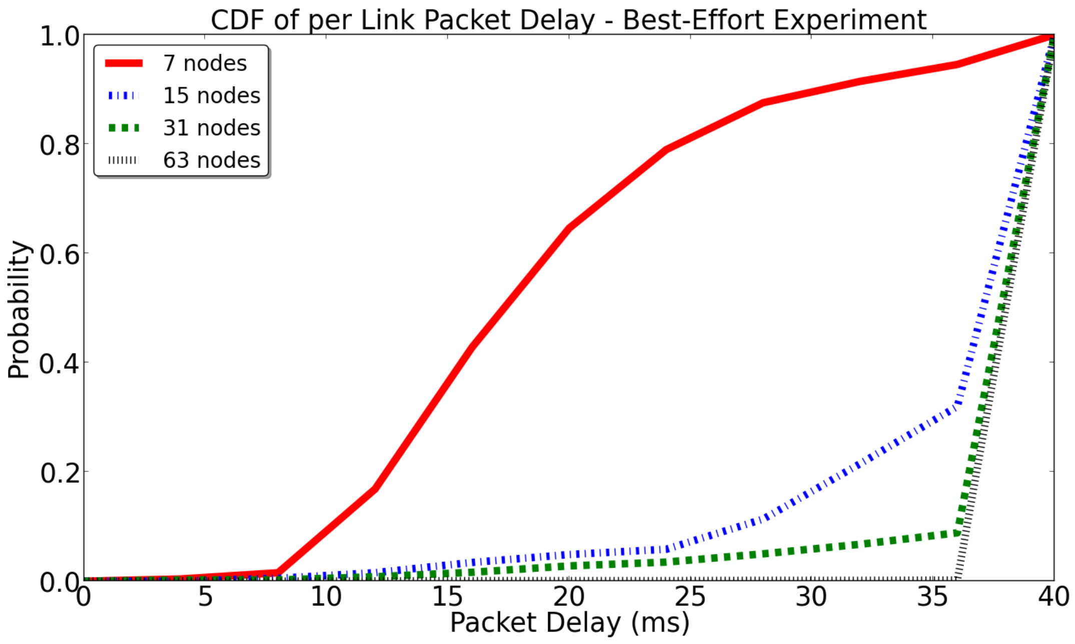
- **Per packet delay:** Isolated industrial control networks typically have static nodes with fairly regular traffic patterns. Hence, the variance in the delay experienced by each packet is expected to be small. A testbed simulating these networks must also be able to guarantee stable packet delays under fixed traffic generation patterns.

- **Throughput of updates:** The number of update messages sent and received over the course of the run time of the system relies heavily on the network delay experienced by each packet and on the speed at which each update is processed. PLCs have stable per cycle execution times which can guarantee bounded information processing delays. The emulation testbed must therefore be able to exhibit low-variance in cycle exhibition times and steady throughput values.

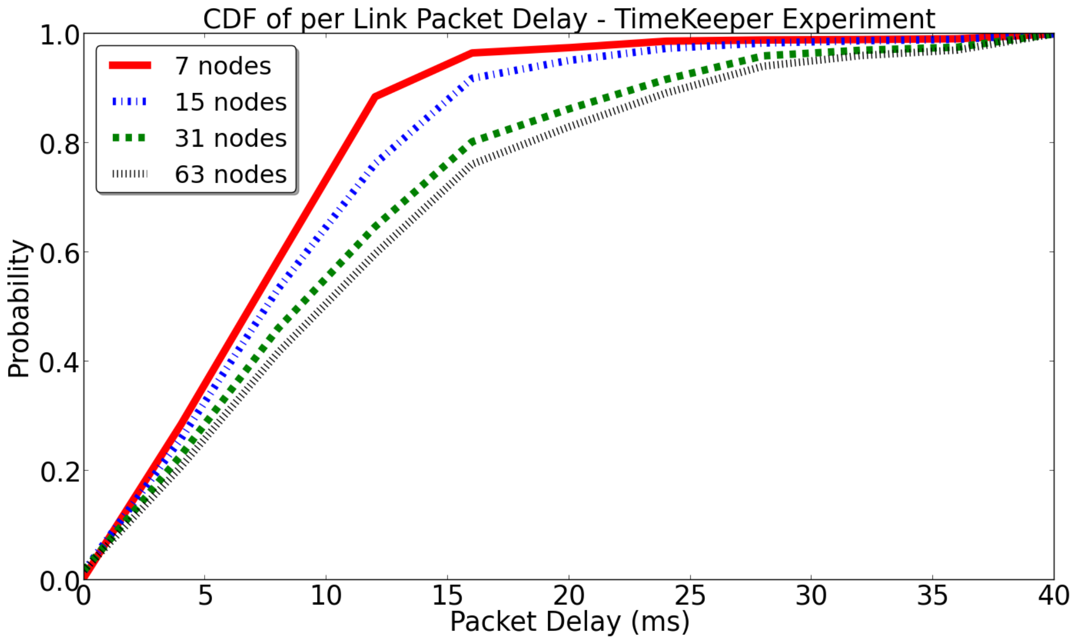
We vary the topology size in our experiments and then study the impact of topology size on the packet delays, cycle execution times and throughput of update messages. In an actual system we expect that packet delay and the cycle execution times will be insensitive to topology size, and we expect the aggregate update throughput to scale linearly with topology size. We compare our observations in *controlled* (under TimeKeeper) and *best-effort* experiments, and by doing so emphasize the benefits of tight coupling between emulation and simulation.

6.4.1 Experiments

For running best-effort experiments, we designed a simple setup where Linux containers running emulated PLCs are directly interconnected with each other. As TimeKeeper and S3F are not involved (and are required for support of serial communication), the emulated PLCs use the IP protocol to carry their Modbus communications. At the point a source constructs a packet, we note the "send time" and have the source delay transmitting the packet for a length of time equal to the modeled transmission delay. The time a packet is received by the target PLC is likewise noted, and the observed packet transmission delay was calculated by logging the receive times and send times of each packet.

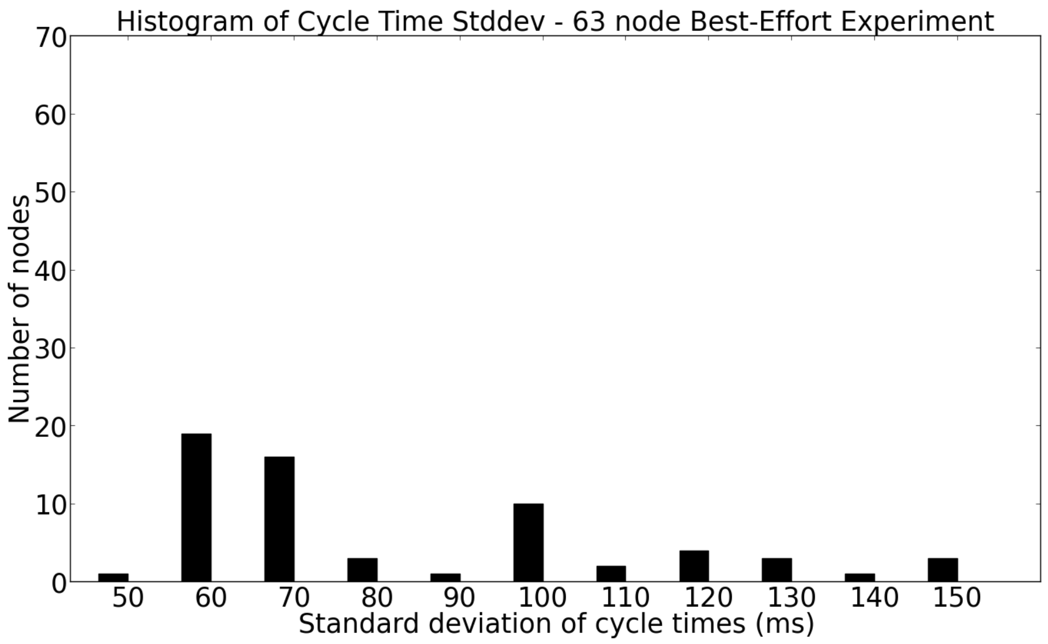


(a) CDF of packet delay for best-effort experiment for different topologies

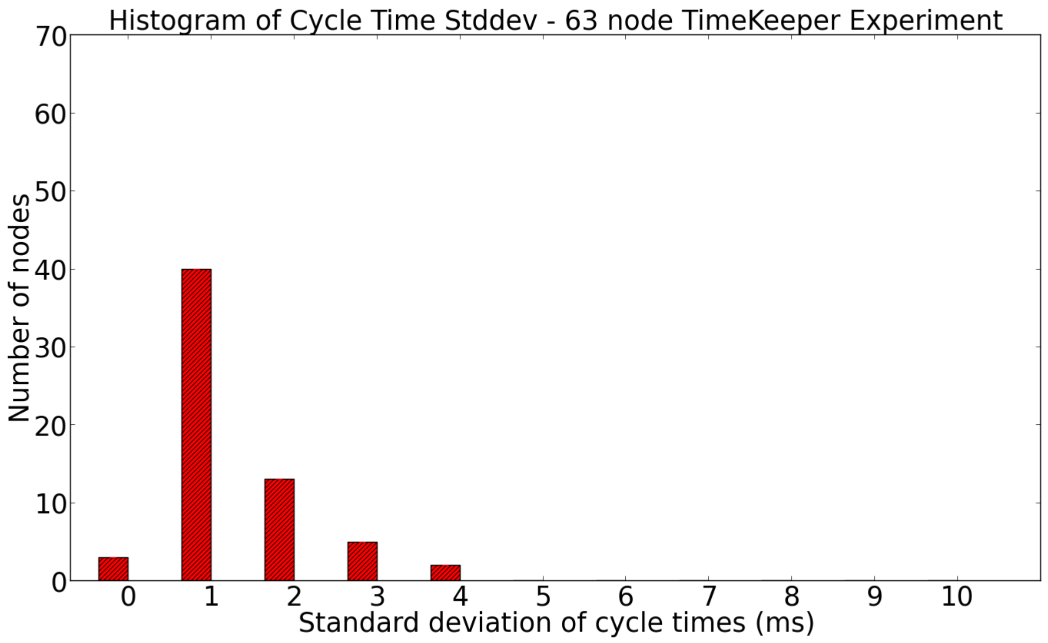


(b) CDF of packet delay for TimeKeeper controlled experiment for different topologies

Figure 6.3: Comparison of CDF of per packet delays for best-effort and TimeKeeper controlled experiments



(a) Histogram of standard deviation of cycle times for undilated 63 node topology.



(b) Histogram of standard deviation of cycle times for dilated 63-node topology.

Figure 6.4: Comparison of standard deviation of cycle times for dilated and undilated 63-node topologies

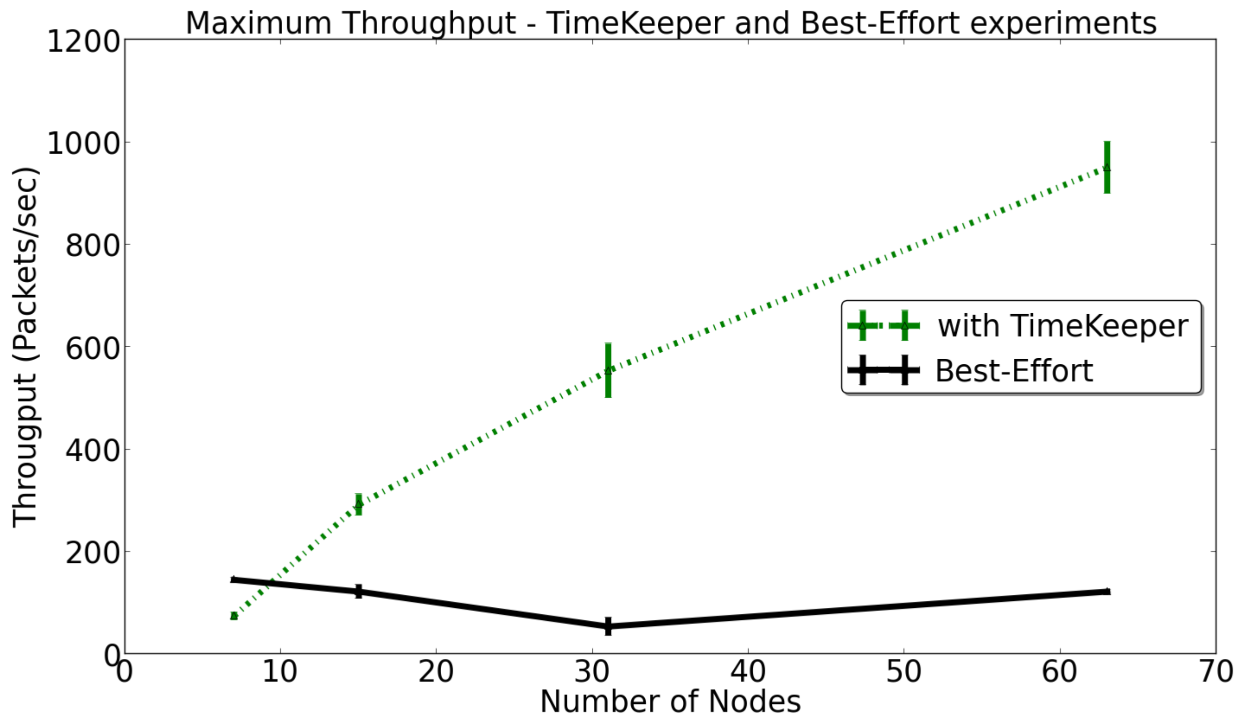


Figure 6.5: Comparison of maximum throughput of update messages for best-effort and controlled experiments.

Figure 6.3a shows the plot of the cumulative density function (CDF) of the observed packet delays for different sizes of full binary trees. At each step, the topology size was nearly doubled but throughout the experiment was run on the same Linux server. Each tree node represents a PLC emulation, so as we increase the size of the topology, the computational resources available to a container decreases; Figure 6.3a shows that this has a clear impact on processing time, increasing topology significantly increases the packet delay. The reason for this is that as these containers are scheduled purely by the Linux kernel, when a container is swapped out in the midst of a transmission (i.e., the send time has been recorded and the wait time before delivery has not yet completed) the time it is swapped out will increase with the number of other processes clamoring for CPU attention. The larger the topology, the longer a container will be swapped out. For the topology with 7 nodes, we measured the mean and 90 percentile delays as 25.26 ms and 32 ms (approximately) respectively. On the other hand, for the topology with 63 nodes, the mean delay equaled 191.52 ms whereas the 90 percentile delay exceeded 200 ms. Whatever the true packet delay might be, we know that it should not change with increasing topology size, and in this system it does.

We repeated the same experiment under a TDF of 15 with S3F and TimeKeeper. Recall that Awlsim is a Python program that interprets Step-7 instructions, and furthermore spawns a number of other processes in support of that interpretation. On a PLC the instructions run almost on bare metal. While a TDF of 15 is admittedly arbitrary, it does reflect that there is considerable overhead in interpreting Step-7 instructions. Fine-tuning a TDF requires instrumenting the modeled PLC to acquire a solid understanding of its native performance, and a study of the cost of interpreting its instructions within Awlsim.

Figure 6.3b, shows the CDF of the observed packet delays in the dilated experiment. Packet delay is considerably smaller; the cumulative distribution functions is largely insensitive to topology size. For the 7 node topology, the mean and 90 percentile delays were approximately 11.42 ms and 16 ms respectively whereas the corresponding values for the 63-node topology equaled 15.57 ms and 26 ms respectively. We attribute the slight increase in the packet delay values to TimeKeeper's timing errors in starting and stopping processes and detecting and adding newly spawned processes to the experiment. It is also important to note that while the specified link delay was 4 ms, the observed mean delays were all greater than 4 ms. This is because each process inside the container is assigned a default fixed virtual time-slice of 1 ms and scheduled in a round robin fashion by TimeKeeper. This can introduce a maximum delay proportional the number of processes running in the container. In our experiments, each container ran a total of 10 processes including the PLC emulator, which can in theory introduce an additional packet reception delay of up to 10 ms. We understand the source of this inaccuracy; to deal with it more directly will require somewhat extensive modifications to the Linux kernel scheduler.

For both sets of experiments, we also compared the standard deviation of each emulated PLC's cycle execution time. Figure 6.4, shows the histogram (taken over all network nodes) of the standard deviation of cycle execution times in the controlled and best-effort executions of the 63-node topology. Each marked point on the x-axis represents a standard deviation interval starting from the specified point until the next one. The y-axis indicates the number of nodes with standard deviation of cycle execution time within the specified interval. From the figures, it is easy to see that the standard deviation of the per cycle execution time is very low in the controlled experiment with the 90 percentile standard deviation below 3 ms whereas it is of the order of hundreds of milliseconds for the best-effort experiment. The best-effort performance can again be attributed to the impact of a container being suspended in the midst of an execution burst, with increasing length of suspension as the number of other containers grows. The fact that the cycle execution times of PLCs in a controlled experiment remain fairly constant at each node further underlines the benefits of the tight coupling between emulation and simulation.

We also measured the maximum number of processed update messages for both classes of experiments. Figure 6.5 plots these measurements for various topology sizes. As expected in the real systems, we observed a near linear increase in the throughput values in the controlled experiment whereas the throughput fell dramatically in the best-effort experiment. The observed variance in the maximum throughput of the controlled experiment was also fairly small, as should be observed in the real system.

CHAPTER 7

CONCLUSIONS AND FUTURE WORK

In this thesis, we argued that conjoining network simulation and emulation with controlled virtual time advancement can improve the scalability and fidelity of emulation results. We described TimeKeeper, a small modification to the Linux kernel which can control the time perceived by a process and advance simulated and emulated entities in close virtual synchrony. We presented two case studies to substantiate these claims.

In the first case study, we studied an EMANE emulation of a wireless link state routing protocol OLSRD which relied on periodic exchange of link update messages. We engineered stress conditions that affected the ability of EMANE to deliver these periodic updates in a timely fashion. We then integrated EMANE with TimeKeeper and observed much greater conformity to expected behavior under stress even for large topologies.

In the second case study, we implemented a cost effective testbed for emulating PLC networks used in Industrial SCADA control. We demonstrated that the testbed delivers expected PLC network behavior in a hypothetical bottling plant job routing scenario by studying the per packet delay and throughput under varied topology sizes. We see that our testbed yields low-variance packet delays and low-variance program cycle times (as expected in a real system) whereas best-effort emulation shows marked increases in both as the size of the network topology increases. Our observations highlight the advantages of tight temporal integration between emulation and simulation and demonstrate the applicability of a tool like TimeKeeper in the design of testbeds for networked applications.

Both case studies have helped us understand subtle implementation level artifacts and yielded interesting problem formulations for the future.

Thread level artifacts: Threads are treated as lightweight processes by the Linux kernel and signalling mechanism used by TimeKeeper to start and stop processes unfortunately stops and resumes all threads associated with the process as well. When a process is resumed, all the threads are inserted into the run queue in a fixed order irrespective of the state of the queue when the process was stopped. This has the potential to starve some threads and needs a fix.

Dilating I/O components: Disks I/O rate is currently not dilated because much of the transfers are handled in firmware and the transfer requests could be arbitrarily batched and served. Disk I/O rate becomes relevant if the application to be tested is sensitive to such discrepancies.

Emulating architecturally different devices: In our current emulation of Step-7 PLC programs, the TimeKeeper/S3F system relies purely on the measured execution time on a Linux platform for an estimate of computational effort expended. But here the actual time spent in processing emulated Step-7 instructions is muddied with the overhead of Python interpretation and helper processes introduced by Awlsim. The future work should focus on annotating the explicit differences between the modeled architectures and the Linux platform, such as the execution times of a Step-7 program's instructions, and somehow communicate those to TimeKeeper and/or the network simulator to allow a more direct (and repeatable) and accurate emulation of architecturally different devices.

Distributed TimeKeeper: The current implementation is only capable of running on a single machine. The broader objective is to have multiple TimeKeeper instances running on different machines and coordinating with each other to advance large scale distributed simulations. A distributed version of TimeKeeper could also open the doors for a new class of cloud-based services called TaaS (Testing as a Service) [28], [29]. A TaaS service provider could test client's applications using a cluster controlled by virtual time systems like TimeKeeper. It raises interesting questions like how to effectively manage cluster resources to handle multiple clients, applications and simultaneously minimize test time.

REFERENCES

[1] Spirent Inc., “White paper on DCN best practices,” http://www.spirent.com, 2016.

[2] The Ns-3 Consortium, “Ns-3,” https://www.nsnam.org/, 2016.

[3] CORE Network Emulator., “Core network emulator.” http://www.nrl.navy.mil/itd/ncs/products/core, 2016.

[4] Naval Research Laboratory, “Extensible mobile adhoc network emulator,” http://www.nrl.navy.mil/itd/ncs/products/core, 2016.

[5] S3F Net, “S3f net,” https://https://s3f.iti.illinois.edu/, 2016.

[6] J. Lamps, D. M. Nicol, and M. Caesar, “Timekeeper: A lightweight virtual time system for linux,” in *Proceedings of the 2nd ACM SIGSIM/PADS Conference on Principles of Advanced Discrete Simulation.* ACM, 2014, pp. 179–186.

[7] J. Lamps, V. Adam, D. M. Nicol, and M. Caesar, “Conjoining emulation and network simulators on Linux multiprocessors,” in *Proceedings of the 3rd ACM Conference on SIGSIM-Principles of Advanced Discrete Simulation.* ACM, 2015, pp. 113–124.

[8] Micheal Busch, “Awlsim,” https://bues.ch/cms/hacking/awlsim.html, 2016.

[9] Open Source Community, “Optimized link state routing protocol,” http://www.olsr.org/mediawiki/index.php/Olsrd2, 2016.

[10] L. Lamport, “Time, clocks, and the ordering of events in a distributed system,” *Communications of the ACM*, vol. 21, no. 7, pp. 558–565, 1978.

[11] D. R. Jefferson, “Virtual time,” *ACM Transactions on Programming Languages and Systems (TOPLAS)*, vol. 7, no. 3, pp. 404–425, 1985.

[12] D. Gupta, K. Yocum, M. McNett, A. C. Snoeren, A. Vahdat, and G. M. Voelker, “To infinity and beyond: Time warped network emulation,” in *Proceedings of the Twentieth ACM Symposium on Operating Systems Principles.* ACM, 2005, pp. 1–2.

[13] M. A. Erazo, Y. Li, and J. Liu, “SVEET! a scalable virtualized evaluation environment for TCP,” in *Testbeds and Research Infrastructures for the Development of Networks & Communities and Workshops, 2009. TridentCom 2009. 5th International Conference on.* IEEE, 2009, pp. 1–10.

[14] D. Gupta, K. V. Vishwanath, M. McNett, A. Vahdat, K. Yocum, A. Snoeren, and G. M. Voelker, “Diecast: Testing distributed systems with an accurate scale model,” *ACM Transactions on Computer Systems (TOCS)*, vol. 29, no. 2, p. 4, 2011.

[15] Y. Zheng and D. M. Nicol, “A virtual time system for openvz-based network emulations,” in *Proceedings of the 2011 IEEE Workshop on Principles of Advanced and Distributed Simulation.* IEEE Computer Society, 2011, pp. 1–10.

[16] D. M. Nicol and J. Liu, “Composite synchronization in parallel discrete-event simulation,” *IEEE Transactions on Parallel and Distributed Systems*, vol. 13, no. 5, pp. 433–446, 2002.

[17] Linux Foundation, “The Xen project,” http://www.xenproject.org/, 2016.

[18] Community PRoject Supported by Odin Inc., “Openvz containers,” https://openvz.org/Main_Page, 2016.

[19] LXC, “Linux containers,” https://linuxcontainers.org/, 2016.

[20] “ACE wrapper for UDP/IP multicast,” http://www.dre.vanderbilt.edu/Doxygen/5.7.9/html/ace/a00617.html, 2015.

[21] R. B. Vaughn Jr and T. Morris, “Addressing critical industrial control system cyber security concerns via high fidelity simulation,” in *Proceedings of the 11th Annual Cyber and Information Security Research Conference.* ACM, 2016, pp. 12–24.

[22] S. Jain, D. Lechevalier, J. Woo, and S.-J. Shin, “Towards a virtual factory prototype,” in *2015 Winter Simulation Conference (WSC).* IEEE, 2015, pp. 2207–2218.

[23] A. Giani, G. Karsai, T. Roosta, A. Shah, B. Sinopoli, and J. Wiley, “A testbed for secure and robust SCADA systems,” *ACM SIGBED Review*, vol. 5, no. 2, p. 4, 2008.

[24] B. Reaves and T. Morris, “An open virtual testbed for industrial control system security research,” *International Journal of Information Security*, vol. 11, no. 4, pp. 215–229, 2012.

[25] T. Morris, A. Srivastava, B. Reaves, W. Gao, K. Pavurapu, and R. Reddi, “A control system testbed to validate critical infrastructure protection concepts,” *International Journal of Critical Infrastructure Protection*, vol. 4, no. 2, pp. 88–103, 2011.

[26] D. M. Nicol, D. Jin, and Y. Zheng, “S3f: The scalable simulation framework revisited,” in *Proceedings of the Winter Simulation Conference.* Winter Simulation Conference, 2011, pp. 3288–3299.

[27] ModBus Protocol, “Modbus protocol,” http://www.modbus.org/, 2016.

[28] J. Gao, X. Bai, and W.-T. Tsai, “Cloud testing-issues, challenges, needs and practice,” *Software Engineering: An International Journal*, vol. 1, no. 1, pp. 9–23, 2011.

[29] K. Incki, I. Ari, and H. Sözer, “A survey of software testing in the cloud,” in *Software Security and Reliability Companion (SERE-C), 2012 IEEE Sixth International Conference on.* IEEE, 2012, pp. 18–23.

[30] “PLCNet: The Awlsim-Timekeeper PLC Network Emulation tool,” https://github.com/Vignesh2208/PLCNet, 2016.

[31] “Timekeeper-2.0,” https://github.com/Vignesh2208/TimeKeeper, 2016.

[32] “EMANE integration with Timekeeper,” https://github.com/Vignesh2208/emane-TimeKeeper, 2016.