Wireless Sensor Network Simulation with Xen

Paul Harvey and Joseph Sventek
School of Computing Science
University of Glasgow
Glasgow, G12 8QQ
p.harvey.1@research.gla.ac.uk and joseph.sventek@glasgow.ac.uk

Keywords: Xen, wireless sensor network, TinyOS, Contiki, InceOS

Abstract

The large-scale and inaccessibility of deployed wireless sensor networks mandate that the code installed in sensor nodes be rigorously tested prior to deployment. Such testing is primarily done using software simulators. Discrete event simulators, by their very nature, mask race conditions in the code since simulated interrupts never interrupt running code; an additional limitation of most such simulators is that all simulated nodes execute the same application code, at variance with common practice in actual deployments. Java-based simulators often suffer from efficiency issues, which limits the scale and performance of the simulation. Since all of these problems reduce confidence in the deployed system, the focus of this work is to eliminate these problems via complete emulation of wireless sensor networks using virtualisation techniques in a scalable manner. This work describes the virtualisation of the Contiki, TinyOS, and InceOS wireless sensor network operating systems to execute as guest domains over the Xen hypervisor. In each case, the hardware virtualisation is performed at the lowest possible layer, maximising the amount of OS and application code which is executed during the emulation. This work also introduces a novel Xen-specific radio model mechanism, easing the introduction of different radio models for use during emulations.

1. INTRODUCTION

Wireless Sensor Networks (WSNs) are a collection of low cost, resource-constrained, battery-powered sensor motes which interact via radio communication; each sensor node will typically measure or effect a change in its surrounding environment, and communicates with other nodes to facilitate the transfer of measurements or control information to/from a control point. Due to the short range of these radios, most nodes will also act as routers in the network.

Such networks are usually widely distributed, often in inaccessible places [6]. Such inaccessibility of deployed WSNs makes it difficult (or impossible if the nodes are installed in an area hazardous to humans) to access the nodes *in situ*, making it essential that the nodes be rigorously tested prior to their deployment.

One approach to WSN testing is to use a testbed. A testbed is a deployment of motes in a controlled environment, where each mote runs the actual code to be deployed. The resource-constrained and embedded nature of sensor nodes means that they lack detailed output capabilities, thus making the collection of debugging information difficult. The SenseLab project [11] attempts to mitigate these issues by offering a generic testbed of 1000 nodes deployed across four seperate regions of France. While this enables the collection of accurate debugging information, it does not take into account the environmental conditions of a specific deployment. This includes deployments which require more than 1000 nodes, or field conditions which may affect the inter-node radio communication. As a result of these issues, testbeds are usually difficult to setup and collect information from, and unlikely to give an accurate representation of the actual deployment.

To overcome these difficulties, sensor network testing is typically done using software simulation [4, 8, 15]. By simulating all nodes within a single running program on a commodity PC, simulators overcome the lack of detailed output capabilities, or the single radio medium available in a hardware testbed. With a sufficiently precise model of radio communications, simulators are able to model the geographical spread of large-scale networks, and to emulate adverse/challenging field conditions.

Discrete event simulation is the most common approach used to simulate WSNs. By its very nature, discrete event simulation models the activity of network nodes and the software components on those nodes as discrete events in time.

Due to this event-driven approach, an interrupt which can cause incorrect behaviour in a physical node will not cause the same error in simulation. This situation limits discrete-event simulation as a reliable tool to provide strong guarantees about the suitability of software for deployment in the field. For example, an interrupt may modify a variable while normally executing code was in the middle of reading the variable. For the same reason, any code that performs an infinite loop awaiting an interrupt will never terminate in the simulator. This is the primary limitation that this work is intended to overcome.

Simulators implemented in Java often suffer from efficiency issues [14]. Consequently, as a simulation scales, the performance and usability of the simulation decreases, limiting the ability to simulate large WSN deployments.

An alternative to software testing is the ability to dynamically re-program deployed sensor nodes at runtime. While over-the-air programming is a more dynamic approach than simulation, it requires software to be present on the resource constrained motes to enable updates, and power to be consumed on the motes in the network who route the update commands and data to their destination.

The remainder of this paper is organised as follows: Section 2. discusses existing WSN simulators, Section 3. describes Xen, Sections 4. and 5. describe how Xen is used as a simulation platform, including node emulation, communication and placement. Section 6. describes the performance of Xen as a WSN simulator, and Section 7. discusses future work, and summarises the findings of the work.

2. SIMULATION ENVIRONMENTS

There are many different simulators used to test WSNs, the majority of which are discrete event simulators. There are also trace-driven simulators, however these are less common because of the lack of standard trace data, due to a lack of a standard model of a sensor network application. Some simulators are general purpose simulators which have been extended to include WSN simulation [13, 2], and others are purpose-built WSN simulators [15, 10]. The two most prominent purpose-built simulators are TOSSIM [8] and Cooja [9].

TOSSIM is a discreet event simulator designed specifically to simulate WSNs with each node in the network running an application of TinyOS [7]. TOSSIM is integrated directly into the TinyOS build tree, such that a user may invoke `make sim` to compile a TinyOS application to work with TOSSIM. At compile-time, certain TinyOS components are replaced with *simulation* implementations. Users may also view the simulation via a GUI known as TinyVIZ.

There are three main drawbacks to TOSSIM. Firstly, as TOSSIM models all interrupts as events, it does not model pre-emption and the resulting potential data-races that may occur. Consequently, it does not give a completely accurate representation of the deployed system, reducing confidence that the deployed system will work correctly. Secondly, TOSSIM can only support applications running on TinyOS. Finally, each mote in a given simulation must execute identical source code.

Cooja is a Java-based network simulator, and is a hardware emulator when combined with MSPSim [4]. MSPSim is also a discrete-event simulator, however by simulating at the hardware instruction level, it offers the same guarantees as real hardware. Cooja has three primary simulation options. It can perform application-level and OS-level simulation of Contiki code. Additionally, it can perform simulations which simultaneously include emulation of WSN platform binaries, and simulation of application code. Cooja has a versatile and complete user interface that enables many aspects of the simulation to be examined and probed. The main disadvantage of the simulator is that it is inefficient at scale. Given the setup described in Section 6., a simulation involving 1000 emulated nodes requires 3 hours to yield 34 seconds of emulation. In general, large-scale emulations are heap limited, as an emulation of 1000 nodes with 5 GB of heap space eventually failed.

3. XEN

Xen [1] is a virtual machine monitor (hypervisor) which enables multiple operating system (OS) instances to run simultaneously on a single machine. Each OS instance is known as a domain. Xen regulates access to the physical resources between the domains, and ensures that domains are appropriately isolated from each other - e.g. a domain may not access the memory of another.

3.1. Paravirtualisation

Full virtualization provides complete emulation of the hardware, including privileged instructions; thus, a guest operating system instance is unaware that any abstraction from the hardware is occurring. Xen uses paravirtualization, in which Xen exports its own set of hardware abstractions, and the source code for each guest operating system must be modified to interact with those abstractions. Porting an OS to Xen is similar to porting to a new hardware platform.

The hypervisor provides a software ABI (the hypercalls) which replaces manipulation of the actual hardware; this includes, for example, privileged instructions such as updating page tables and requesting access to hardware resources. Although this requires development time to be spent in porting to the hypervisor, the technique yields considerable increases in performance over full virtualization [1]. It should be noted that only the OS kernel requires modification, and that user applications remain unchanged.

3.2. Domain Management

A single, specially privileged domain (Dom0) is the first domain that the hypervisor loads when booting; this is typically a modified Linux kernel. This domain has access to available hardware such as hard disks and network interfaces. Other domains (termed DomU or guest domains) typically do not have direct access to these resources and must request access via Dom0; this distinction is transparent to all but the lowest layers of the guest operating system.

Dom0 privileges include the ability to create and destroy domains. This is typically achieved by executing the relevant `xm` subcommand, providing configuration parameters in a file or from the command line. For example, `xm create` requires the new domain's name, memory allocation, and the virtual network bridge to which its virtual network card will be connected.

3.3. Split Drivers

As guest domains typically do not have access to the physical hardware, their device drivers are replaced with a split driver implementation. The backend portion of the driver resides in Dom0 and accesses the hardware resource. The matching frontend in a DomU communicates with the backend to obtain the relevant service; it is for this reason that the guest OS's drivers must be modified.

4. XEN AS A SIMULATOR

To explore the efficacy of Xen as a platform for supporting WSN simulation, the TinyOS [7], Contiki [3], and InceOS [5] WSN operating systems have been ported to run on the Xen platform as guest domains. These OSs were chosen as they represent the different equivalence classes present in WSNs: event-based, protothread-based, and actor-based.

As with other guest OSs, it was necessary to port all of these systems to use the Xen hypercalls instead of directly manipulating the hardware of motes. When porting these systems to Xen there were two goals:

1. Modifications to the OS should be done at the lowest possible level, so as to ensure that as much of the OS and application code as possible is used in the simulation.

2. Any changes should be integrated into the build process of the system, such that building for Xen is no more difficult than any other platform.

4.1. Mini-OS

The Xen hypervisor is delivered with an OS called Mini-OS. Mini-OS is designed to serve as a reference OS for those wishing to port their own OSs to run as Xen domains. It shows how a domain can achieve memory allocation, networking, concurrency, timing, backing storage, and how to interact with the XenStore, which will be discussed later.

Within this work, Mini-OS was used as more than just a reference OS. Instead, Mini-OS was used as the base OS for the domain, and the relevant WSN OS is run as a thread ontop of Mini-OS. No excess processing is required as, after Mini-OS is initialised, the WSN OS thread is the only active thread in the domain.

Using Mini-OS simplified the process of porting each OS to Xen, as the low level implementation of interacting with Xen is already provided. In this way Mini-OS aided in the understanding of how to use the hypercalls, as well as reducing the development time required in porting the systems. Figure 1 shows the architecture of the Xen simulation environment.

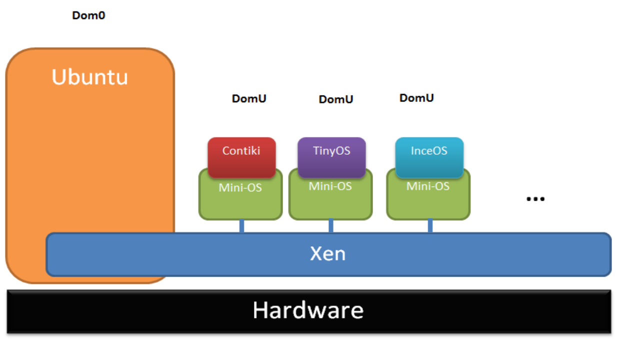


Figure 1. The Architecture of the Xen WSN Simulator

In order to produce an image to be instantiated as a guest domain, three steps were required to combine the WSN OS and application with Mini-OS:

1. Each OS was built for the Xen platform. Normally TinyOS generates a single .c file which is then compiled to the relevant binary, whereas InceOS and Contiki generates multiple object files which are linked together before generating the relevant binary. In each case the binary generation phase is skipped.

2. For all systems, the `main()` function was renamed to `app_main()` before generating an object file to be linked against Mini-OS. `app_main()` is the function called by Mini-OS after it has completed initialisation.

3. The object files are linked with Mini-OS to create a Xen domain image.

4.2. Xen-Timers

Each OS requires the use of timers for different activities, and each system provides a software interface for requesting time-outs at certain intervals. Behind this software API, the timer module(s) maintain accounting information for the outstanding timers, and code to interact with the hardware to request, and be notified of the expiration of a timeout.

Xen does not permit access directly to the underlying hardware for guest domains. Instead, Xen provides a set of *hypercalls* to mediate access to the hardware. Specifically for timeouts, Xen provides the `set_timer_op(timeout)` hypercall, which causes a virtual timer interrupt to be delivered after `timeout` nanoseconds. Each domain may only have one outstanding timer request at any one time, therefore any new timer request will replace the outstanding one.

Mote hardware, such as the MSP430, or Atmel microprocessor, will usually have a number of timers which will run at (configurable) rates. Given that Xen only permits a single timer to be outstanding at any one time, the `XenTimer`

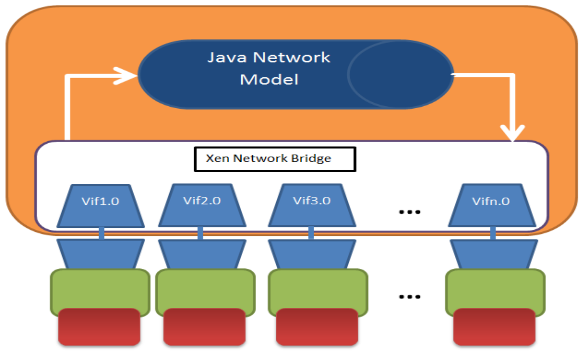


Figure 2. Xen Simulator Networking Layout

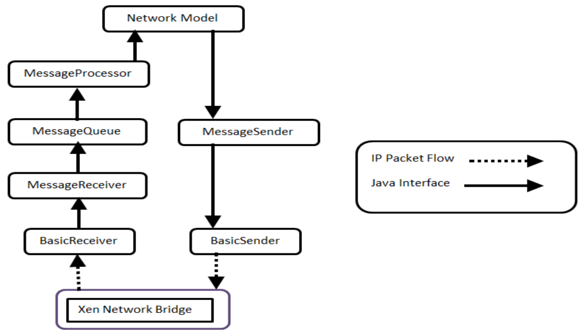


Figure 3. Dom0 Java Network Application Architecture

driver was introduced to merge the requests from multiple timers into a single request queue.

4.3. Xen-Radio

On real mote platforms, the radio is used to facilitate inter-node communication through the surrounding medium. Similarly to the timers above, a software API is provided to simplify interaction with the underlying hardware. In Xen, domains communicate with each other using virtual interfaces (vifs). Vifs are virtualised network interface cards, with their own IP and MAC addresses, thus mimicking real network interface cards. The network architecture and Java network model can be seen in Figure 2.

In each OS, it is the responsibility of the radio driver to abstract the interaction with the radio hardware, such that the driver is given and returns packet payloads and addresses. For the Xen simulation platform this is no different.

In order for each system to communicate across Xen, the XenRadio driver was introduced to each system. It is the responsibility of this driver to convert to and from the UDP packets which are sent and received from Xen. UDP packets are used as they mimic the guarantees present in a WSN network. As a consequence of this, all current and future WSN domains are interoperable, it is only then necessary to be running the same network protocols in each OS to transmit and receive from other domains.

4.4. Xen Simulation Networking

In the field, radio communication between WSN motes exhibit various properties that must be simulated: packet loss, radio noise, and collisions. Signal power is reduced as the distance between the nodes increases, often resulting in increased bit-errors and loss.

The goal of this work was to provide a framework in which various radio models, of different foci/accuracy can be easily implemented and inserted. Of particular importance was the desire that network model developers not have to be expert programmers; the framework needed to provide clear interfaces for accessing the underlying functionality.

The key functionality provided by these interfaces are:

- notification of all incoming frames from the mote
- access to the topology in which the motes are deployed
- the ability to transmit frames to motes (as dictated by the radio model)

A centralized approach has been adopted, with the network model implementation active on Dom0. When a mote transmits a packet, it is forwarded to the network model; the model determines which other motes, if any, should receive that packet, and then forwards the packet to those motes.

The radio communication emulation uses the *vif* provided by Mini-OS. Each 802.15.4 packet is encapsulated into a UDP packet and relayed to Dom0 via the frontend-backend communication channel. This is achieved by associating a unique IP address to each mote identifier and binding it to the domain's virtual MAC address by loading the ARP cache. The radio model running in Dom0 intercepts the packets after they have been processed by the IP networking stack.

To support the straightforward creation of different network models, the Java framework shown in Figure 3 is provided. The `NetworkModel` class is abstract and defines a constructor which takes as parameters a Topology and a class which implements the `MessageSend` interface. The Topology class, discussed in the next section, can be used by the network model to determine distances between nodes.

An example network model implemented in the distribution, `PerfectRadioModel`, performs simple distance-based filtering of packets. The bytes of packets are unmodified (no noise), and packets are relayed if the distance from the sender is less than a particular bound. While this is clearly too simplistic for a real simulation, it provides a starting point for future development of more accurate replacements.

4.5. The XenStore

Each system requires information at boot-time. Some information is implicit, such as hardware details, and some is user-specified, such as the unique `TOS_NODE_ID` value which is specified for TinyOS at compile-time.

Xen provides a useful method of transferring start-of-day information to domains via the *XenStore*. The XenStore is a tree-structured database in which each domain has its own directory. Dom0 can write key-value pairs into any of the domains' directories, and each guest domain can read/write key-value pairs into its own directory.

During the boot sequence of each OS, an initialisation function has been added to to read values from the XenStore. Values to be read include the `TOS_NODE_ID` for TinyOS, and more generally the IP and MAC address to be used in the XenRadio.

4.6. Microcontroller Sleep

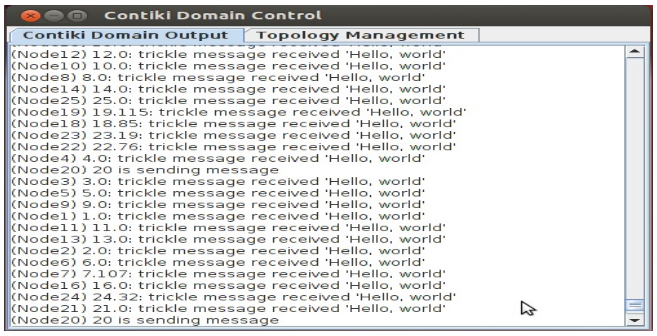
WSN motes are small battery-powered computers. In order to preserve power for as long as possible, motes often have a number of low power states. Each state is responsible for enabling or disabling certain hardware elements, such as clocks and the radio. In each OS, when there is no more work to be done, the scheduler will place the mote into a *sleep* mode to conserve power, rather than having an idle task.

Within the Xen simulation, a domain may also go to sleep until awoken by an interrupt from the hypervisor. The `block_domain(duration)` hypercall will block a domain from executing until `duration` nanoseconds have passed, or an interrupt is received, thus mimicking the operation of hardware. In each system, `duration` is set to FOREVER, meaning that the domain will only wake up when a timer fires, or radio packet is received. This also has the advantage of releasing CPU cycles for other work in the simulation.

4.7. Debug Output

A benefit of simulators is the ability to print debugging output, and is also a requirement for the Xen simulator. Within Mini-OS `printk()` is used to print information to a console. This (virtual) console can be connected to/from Dom0.

To avoid having to open multiple terminals in Dom0, the Java control program collects the output from the guest domains and displays them in a single window, Figure 4.



```
(Node12) 12.0: trickle message received 'Hello, world'
(Node10) 10.0: trickle message received 'Hello, world'
(Node8) 8.0: trickle message received 'Hello, world'
(Node14) 14.0: trickle message received 'Hello, world'
(Node25) 25.0: trickle message received 'Hello, world'
(Node19) 19.115: trickle message received 'Hello, world'
(Node18) 18.85: trickle message received 'Hello, world'
(Node23) 23.19: trickle message received 'Hello, world'
(Node22) 22.76: trickle message received 'Hello, world'
(Node4) 4.0: trickle message received 'Hello, world'
(Node20) 20 is sending message
(Node3) 3.0: trickle message received 'Hello, world'
(Node5) 5.0: trickle message received 'Hello, world'
(Node9) 9.0: trickle message received 'Hello, world'
(Node1) 1.0: trickle message received 'Hello, world'
(Node11) 11.0: trickle message received 'Hello, world'
(Node13) 13.0: trickle message received 'Hello, world'
(Node2) 2.0: trickle message received 'Hello, world'
(Node6) 6.0: trickle message received 'Hello, world'
(Node7) 7.107: trickle message received 'Hello, world'
(Node16) 16.0: trickle message received 'Hello, world'
(Node24) 24.32: trickle message received 'Hello, world'
(Node21) 21.0: trickle message received 'Hello, world'
(Node20) 20 is sending message
```

Figure 4. Debug Output From Running Domains

5. TOPOLOGY MANAGEMENT

There are a number of steps required to create a Xen WSN domain. Firstly, the OS and application must be compiled; then the Mini-OS Makefile must be modified to include the object file produced. `make` is then run in the Mini-OS directory to create the absolute binary. Finally, this binary is loaded into a Xen guest domain using the `xm` command in Dom0.

Also, there are a number of per-domain settings that must be specified: the domain config file must contain a unique domain name, the XenStore must have the new domain's start-of-day constants written into it (including the node's geographic coordinates), and the `arp` command must be issued to bind the IP address for each node to its virtual network interface.

The topology of a sensor network is specified in a topology file. Each line of the file contains information for one node in the system as follows:

- node ID
- latitude (+/-,degrees minutes, seconds)
- longitude (+/-, degrees minutes, seconds)
- altitude (meters)
- node application

Note that we have chosen latitude, longitude and altitude as node coordinates to support realistic deployments scenarios. Field deployment of motes is usually driven by knowledge of the terrain and the locations of special interest for sensors; these coordinates are determined using hand-held GPS (Global Positioning System) devices. This leads to a direct mapping between testing and deployment configurations, providing additional assurance that the nodes will operate as expected when deployed. This differs from TOSSIM in which the topologies are defined in terms of radio gain (signal strength) between pairs of nodes.

In order to address the above-mentioned complexities, as well as to provide coordinate information for use by network model implementations, a control program, written in Java, is supplied with the system. This control program:

- parses a topology file, constructing an instance of the Topology class
- instantiates the network model using that Topology class instance

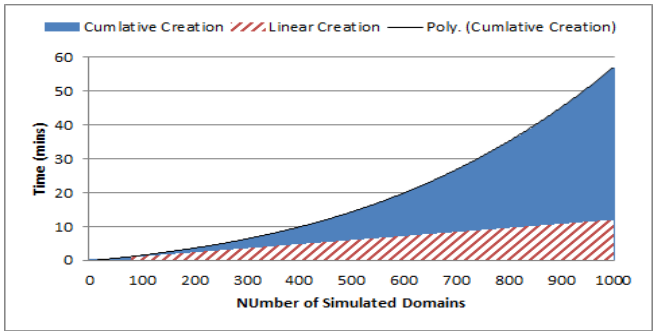


Figure 5. Cumulative Domain Creation and Initialisation Times over 1000 Nodes

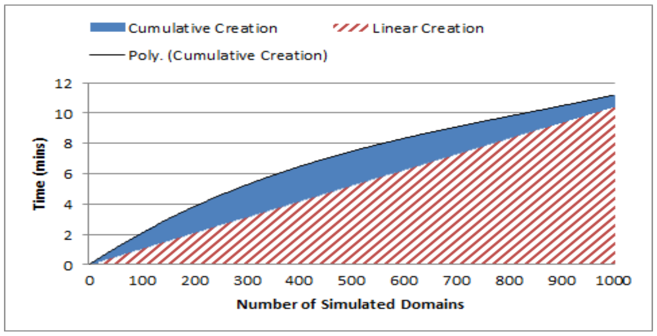


Figure 6. Cumulative Domain Destruction Times over 1000 Nodes

- for each entry in the file, creates a guest domain running the application and pauses the guest domain
- after the last domain has been created, the domains are resumed in rapid succession

When a simulation is in progress, it is possible to manipulate the topology in various ways. Nodes can be added and deleted. Additionally, if supported by the network model, nodes may be moved to new coordinates.

6. EVALUATION

In this section we categorise the costs and limitations of the Xen simulation technique. The two main questions are how many nodes can we simulate, and what is the performance degradation in the simulation as the number of nodes increase?

6.1. Experimental Setup

All tests were carried out using Xen 4.1.2. Dom0 was a 64-bit version of Ubuntu 12.04 with Linux 3.2.0-33-generic. All testes were carried out on an Intel Core i5 3550 quad-core processor running at 3.33 GHz, with 12 GB of RAM.

Dom0 is initially able to execute on any of the CPUs in the system, however, after all the nodes of a simulation have been created, it is pinned to execute on a single CPU before unpausing the nodes. Each guest domain is able to run on any CPU except the one which is allocated for Dom0. This is done to ensure that one CPU will always be available to service Dom0, and by extension, the network traffic. This also prevents scheduling anomalies which would have been caused by scheduling competition between small sensor DomUs and Dom0. The Xen credit-scheduler was used, with all domains receiving the default weight.

6.2. Memory Consumption

Within Xen, the minimum amount of memory which can be allocated to a domain is 4MB, which is approximately two orders of magnitude greater than the RAM available on a typical mote. This is irrespective of the WSN OS being used within the domain. This limit is enforced by Xen due to 4MB alignment of the memory allocation. The Xen balloon driver enables domains to be given more RAM if they require it and if enough RAM is available. This feature was disabled during testing. In addition to the RAM allocated when a DomU is created, experiments show that Xen itself requires a further 0.14 MB per domain.

In contrast to the DomU's, Dom0 was specified to have a fixed memory of 3 GB during testing. Given these values, the theoretical maximum number of WSN OS domains that could be created is 2226.

6.3. Performance

In order to categorise the performance of the simulator, the times required to perform different actions were measured.

6.3.1. Startup Delay

When setting up a simulation, each domain is compiled as described previously, created, and executed. Figure 5 shows the cumulative amount of time for one thousand domains to be created and setup under Xen. In total this requires 56.9 minutes to setup a simulation involving 1000 nodes.

The first domain requires 0.7s to create, and the cost grows quadratically. These values are independent of the domain to be run, and are due to Xen itself. After consultation with the Xen developers, it is not clear why the time required to create a domain grows quadratically. In contrast to the creation times, Figure 6 shows the cumulative amount of time to destroy one thousand domains. In total this took 11.1 minutes, with the longest destruction taking 1.5s.

While the creation time is large, there are two important points to consider. Firstly, each simulated node is an isolated Xen domain, which is fully interruptible at any point and contains its own networking interface. Secondly, the creation time is a *one-off* start up cost, which has no bearing on the simulation time.

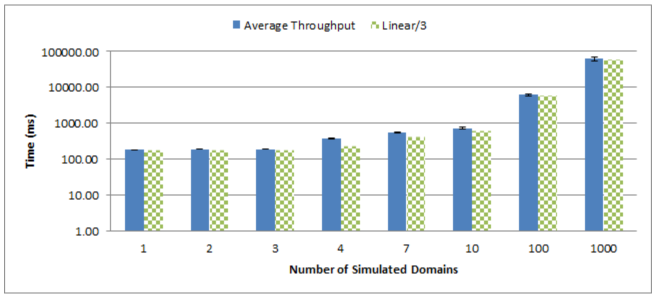


Figure 7. Single Node Throughput

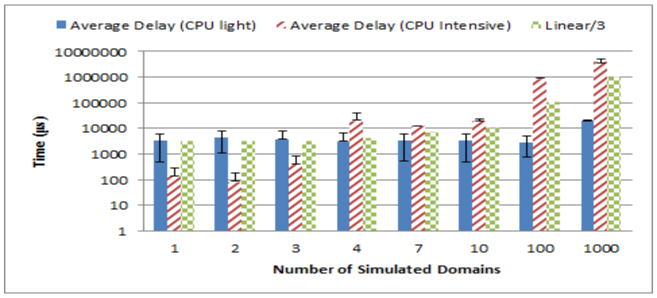


Figure 8. Delay on Requested Timer Interrupt

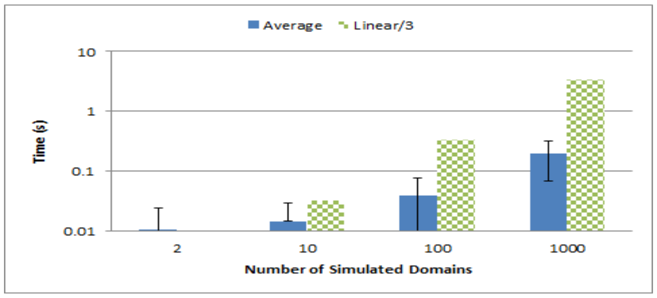


Figure 9. Single Hop Packet Round Trip Time

6.3.2. Scalability

To assess how the simulator copes under scale, three experiments were carried out. The first was to assess the throughput of each domain, the second was to assess the delay on timer interrupts, and the third was to determine the round trip time between two nodes sending packets.

In the first case, each simulated node repeatedly calculates the integer fast Fourier transform (ifft) of a 1 KB array of numbers. In the second, two applications were used, one which calculates the ifft, and one which simply reads from a sensor and sleeps periodically. In the third, the round trip time (RTT) between a single pair of nodes is measured, with every pair of simulated nodes sending packets to each other via the network model in Dom0. This simulation used the `PerfectRadioModel`. The ifft calculation represents the worst-case situation for the simulator, when nodes never go to sleep. In each graph, the time was taken after all nodes were unpaused so as to observe the greatest load on the simulator.

Figure 7 shows the average time required to perform 1000 iterations of the *ifft* as the number of simulated nodes increase. The results show that the throughput scales approximately linearly with the number of nodes divided by three. This is due to the execution of guest domains across three CPUs. For each simulation run, core 3 is reserved for Dom0, with each DomU being allocated amongst the remaining three cores in a round-robin fashion.

There is currently a bug within a particular Xen daemon, which occurs when more than 338 WSN DomUs are created, and prevents connection to the console of any subsequently created domain. The cause of this problem is a *statically* allocated array of open file handles. The Xen developers have been notified, and a patch is under way. This limitation does not prevent the creation or operation of subsequent domains.

Figure 8 shows the delay in μs observed on timer requests made to Xen as the number of nodes grow for both a CPU intensive and non-intensive application. In general, the CPU intensive application has a lower delay when there are fewer nodes, but scales super-linearly. In contrast, the CPU light application performs worse at low number of domains, but scales sub-linearly. This is due to the CPU light domains going to sleep, since approximately 3ms are required to wakeup the domain. However, with larger numbers of domains, the CPU light applications consume very few cycles, leaving plenty of time for Xen to manage timer requests. The CPU intensive applications do not give up cycles, hence the poor performance at scale. The CPU light application more accurately represents typical WSN behaviour.

As a point of comparison, a Cooja simulation of 1000 nodes performing a similar CPU light application without networking took 3 hrs to generate 34s of emulation, including the almost instantaneous the start-up time. The Xen simulation, using the CPU intensive application, took 3.3 hrs to generate 34s of emulation, including start-up time. Focusing on emulation time alone, the Xen simulator produces faster per second emulation at scale.

Finally, Figure 9 shows the single hop RTT between a pair of simulated nodes as the total number of nodes increase, averaged over 100 packets. The figure shows that the RTT scales sub-linearly as the number of nodes increases, even accounting for the standard deviation, shown by the error bars. This is caused by a combination of Dom0 being allocated a dedicated CPU, as well as the radio model being relatively simple. The next step would be to re-run this experiment with a more complicated radio model.

7. SUMMARY AND FUTURE WORK

This paper presents a Xen-based wireless sensor network simulator, upon which real sensor network components operate. This occurs in real-time with the same behaviour as running on actual hardware. In particular, race conditions are able to manifest themselves, heterogeneous sensor networks can be emulated, and a framework supporting different radio models has been provided to enable realistic radio transmissions for use in simulations. The results show that while there is a high cost in creating large simulations, Xen copes well during emulation with large numbers of simulated nodes, scaling linearly, thus enabling the emulation of large scale sensor fields. Three WSN OSs have been ported to the platform, and a wide variety of their applications have been tested using a perfect radio model, and work correctly.

Given the Xen results for domain creation and timer delay under load, it is worth exploring alternative solutions. Consequently, the next step is to explore process level emulation, as a process offers similar isolation guarantees as a domain. Cgroups [12] enables the fine grained control of a process's resources, while maintaining the isolation of the process.

REFERENCES

[1] Paul Barham, Boris Dragovic, Keir Fraser, Steven Hand, Tim Harris, Alex Ho, Rolf Neugebauer, Ian Pratt, and Andrew Warfield. Xen and the art of virtualization. In *Proceedings of the nineteenth ACM symposium on Operating systems principles*, SOSP '03, pages 164–177, New York, NY, USA, 2003. ACM.

[2] OMNeT++ Community. Omnet++ discrete event simulator. Online, October 2012. http://www.omnetpp.org.

[3] Adam Dunkels, Bjorn Gronvall, and Thiemo Voigt. Contiki - a lightweight and flexible operating system for tiny networked sensors. In *Proceedings of the 29th Annual IEEE International Conference on Local Computer Networks*, LCN '04, pages 455–462, Washington, DC, USA, 2004. IEEE Computer Society. Contiki OS.

[4] Joakim Eriksson, Fredrik Österlind, Niclas Finne, Nicolas Tsiftes, Adam Dunkels, Thiemo Voigt, Robert Sauter, and Pedro José Marrón. Cooja/mspsim: interoperability testing for wireless sensor networks. In *Proceedings of the 2nd International Conference on Simulation Tools and Techniques*, Simutools '09, pages 27:1–27:7, ICST, Brussels, Belgium, Belgium, 2009. ICST (Institute for Computer Sciences, Social-Informatics and Telecommunications Engineering).

[5] Paul Harvey, Alan Dearle, Jonathan Lewis, and Joseph S. Sventek. Channel and active component abstractions for wsn programming - a language model with operating system support. In *Proceedings of the 1st International Conference on Sensor Networks*, pages 35–44, 2012.

[6] Andreas Hasler, Igor Talzi, Christian Tschudin, and Stephan Gruber. Wireless sensor networks in permafrost research - concept, requirements, implementation and challenges. In *Proc. 9th International Conf. on Permafrost (NICOP 2008*, 2008.

[7] Jason Hill, Robert Szewczyk, Alec Woo, Seth Hollar, David Culler, and Kristofer Pister. System architecture directions for networked sensors. *SIGOPS Oper. Syst. Rev.*, 34(5):93–104, 2000.

[8] Philip Levis, Nelson Lee, Matt Welsh, and David Culler. Tossim: accurate and scalable simulation of entire tinyos applications. In *Proceedings of the 1st international conference on Embedded networked sensor systems*, SenSys '03, pages 126–137, New York, NY, USA, 2003. ACM.

[9] F. Osterlind, A. Dunkels, J. Eriksson, N. Finne, and T. Voigt. Cross-level sensor network simulation with cooja. *Local Computer Networks, Annual IEEE Conference on*, 0:641–648, 2006.

[10] Jonathan Polley, Dionysys Blazakis, Jonathan Mcgee, Dan Rusk, and John S. Baras. Atemu: A fine-grained sensor network simulator. In *IEEE Communications Society Conference on Sensor and Ad Hoc Communications and Networks*, 2004.

[11] SenseLab. Senselab: Very large scale open wireless sensor network testsbed, November 2012. http://www.senselab.info.

[12] Balbir Singh and Vaidyanathan Srinivasan. Containers: Challenges with the memory resource controller and its performance. In *Proceedings of the Linux Symposium*, pages 101–110, Ottawa, Canada, June 2007.

[13] Clay Stevens, Colin Lyons, Ronny Hendrych, Ricardo Simon Carbajo, Meriel Huggard, and Ciaran Mc Goldrick. Simulating mobility in wsns: Bridging the gap between ns-2 and tossim 2.x. In *Proceedings of the 2009 13th IEEE/ACM International Symposium on Distributed Simulation and Real Time Applications*, DS-RT '09, pages 247–250, Washington, DC, USA, 2009. IEEE Computer Society.

[14] Harsh Sundani, Haoyue Li, Vijay K. Devabhaktuni, Mansoor Alam, and Prabir Bhattacharya. Wireless sensor network simulators a survey and comparisons. *International Journal Of Computer Networks (IJCN)*, 2(5), 2010.

[15] Hejun Wu, Qiong Luo, Pei Zheng, and Lionel M. Ni. Vmnet: Realistic emulation of wireless sensor networks. *IEEE Trans. Parallel Distrib. Syst.*, 18(2):277–288, February 2007.