TinyWifi: Making Network Protocol Evaluation Portable Across Multiple Phy-Link Layers

Muhammad Hamad Alizai[†], Hanno Wirtz[†], Bernhard Kirchen[†],
Tobias Vaegs[†], Omprakash Gnawali[‡], Klaus Wehrle[†]
[†]Communication and Distributed Systems, RWTH Aachen University, 52056 Aachen, Germany
[‡]Department of Computer Science, Stanford University, Stanford, CA 94305, USA
[†]{lastname}@comsys.rwth-aachen.de, [‡]gnawali@cs.stanford.edu

ABSTRACT

Multihop wireless networks, such as sensor-, ad hoc- and mesh-networks, although different share some common characteristics. All these networks exhibit link dynamics. Protocols designed for these wireless networks must overcome the challenge of link dynamics and the resulting churn in network topology. Due to structural and topological similarities, protocols developed for one class of wireless network should also be applicable in the other classes. However, network-layer protocols are usually developed for and tested in only one class of wireless network due to the lack of a platform that allows testing of protocols across different classes of networks. As a result, we unnecessarily constrain the range of settings and scenarios in which we test network protocols.

In this paper, we present TinyWifi, a platform for executing native sensornet protocols on Linux-driven wireless devices. TinyWifi builds on nesC code base that abstracts from TinyOS and enables the execution of nesC-based protocols in Linux. Using this abstraction, we expand the applicability and means of protocol execution from one class of wireless network to another without re-implementation. We demonstrate the generality of TinyWifi by evaluating four well-established protocols on IEEE 802.11 and 802.15.4 based testbeds using a single implementation.

Categories and Subject Descriptors

C.2.4 [**Computer-Communication Networks**]: Distributed Systems—*Network operating systems*; C.2.2 [**Computer-Communication Networks**]: Network Protocols—*Routing protocols*

General Terms

Design, Experimentation, Measurement, Performance

Keywords

TinyOS, Linux Nodes, Wireless Mesh Networks, Network Protocols, Protocol Evaluation

WiNTECH'11, September 19, 2011, Las Vegas, Nevada, USA.

1. INTRODUCTION

Sensor-, ad hoc-, and mesh-nets, represent vastly different classes of wireless networks. They not only use different types of radios and link layers but also OS, hardware platform, programming/runtime environment, and application scenarios. While different, they also share some commonalities: (1) dynamic and bursty links due to radio interference and other physical influences, (2) use of multihop protocols to reach nodes not within radio range, (3) the intended use cases demand a reliable and scalable communication infrastructure, and (4) they are self-organizing in arbitrary and temporary network topologies.

These similarities lead to an important question: How well can the algorithmic concepts, proven methods, and protocols from one class of wireless network be adapted to the other classes of wireless networks? In general, research efforts, such as on link estimation [7, 8], routing [18, 19] and addressing [3, 8], explore the feasibility of these protocols in one class of networks and implicitly assume their applicability in the other, based on the above mentioned similarities. This assumption is rarely validated due to the lack of a common development platform that allows us to test the protocols across the vastly different classes of wireless networks.

To understand the performance of the protocols and their applicability across multiple wireless network classes, a common programming environment and a runtime platform is essential. It is well understood that incompatible application requirements and unequal resource constraints make for a significant diversity among these different classes of networks. However, this diversity, in most cases, only demands appropriate adaptations in operational parameters of the underlying protocols while the core mechanisms still remain the same [2, 7, 21]. For example, to account for the underlying resource availability in different networks, routing protocol configurations may only need to adjust parameters such as routing table sizes and the frequency of routing updates. Nonetheless, the metrics used to select a next hop and establish routing paths - the core and the most complex mechanisms of a routing protocol mechanism - remain the same: ETX (expected transmission count) [5] is the most prevalent routing and link metric both in sensornets and meshnets [21]. Moreover, a common development platform will help determining the impact of lower layer technologies, such as medium access, coding, and modulation schemes, which are different in IEEE 802.11 and 802.15.4 standards, on the performance of these core protocol mechanisms.

As a first step towards such a platform, we introduce TinyWifi, a TinyOS platform supporting Linux driven devices and thereby the IEEE 802.11 based Wi-Fi standard. The utility of TinyWifi is twofold: (1) It is a runtime platform that allows direct execution of protocol libraries in three different network classes. (2) It makes the very rich and mature protocol repository of TinyOS available for broader wireless research. TinyWifi supports a wide variety of Linux kernel derivatives representing all major Linux distributions such as OpenWRT, Debian, Slackware and Ubuntu. We (and a few other research groups) are using TinyWifi[1] to run nesC protocols in meshnets.

We evaluate the correctness of our TinyWifi implementation by comparing two different implementations of the Collection Tree Protocol (CTP) [12], one in nesC and the other in Click [15]. Our comparison proves the equality of these two implementations and demonstrates the utility of TinyWifi as a customary wireless research and runtime platform. We then evaluate three routing protocols - BVR [8], S4 [18], PAD [3] - both in sensornets and meshnets and analyze the respective performance. During our evaluation on an IEEE 802.11 based testbed, we observed that TinyWifi is particulary useful for (1) evaluating prototypes, (2) fine-tuning protocol parameters and (3) establishing multiple performance metrics in different classes of wireless networks without reimplementation.

The rest of this paper is structured as follows: We briefly discuss the overall design and key features of TinyWifi in Section 2. The detailed architecture is discussed in Section 3. Section 4 presents evaluation results. We discuss limitations in Section 5 and related work in Section 6 before we conclude with Section 7.

2. PRELIMINARIES

We first provide necessary background by briefly introducing TinyOS. Then, we present the overall design of TinyWifi. Finally, we highlight the key features of our TinyWifi implementation.

2.1 TinyOS

TinyOS is the de facto standard operating system for sensornets. It has an event driven architecture which enables development of energy-efficient sensornet applications. It has been in active research and development over the past decade and its novel protocol mechanisms, such as in link estimation, routing and addressing, are developed and actively used worldwide. Applications and protocols in TinyOS are written in nesC [10], a modular extension of the C programming language specifically designed to support the execution model and structuring concept of TinyOS. nesC provides *interfaces* to develop *modules* that are wired together to achieve the desired functionality.

The architecture of TinyOS is divided in three abstraction layers [13]: Hardware Presentation Layer (HPL), Hardware Abstraction Layer (HAL), and Hardware Independent Layer (HIL). The modules at HPL are hardware-dependent and present the capabilities of the underlying hardware while hiding its intricacies. In contrast, the modules at HAL and HIL are platform independent and can be used across different hardware platforms. Protocols and applications are built on top of HAL and HIL. TinyOS can be extended to new hardware platforms by providing the corresponding HPL support for that platform.

[1]The source code of TinyWifi is available for download at http://www.comsys.rwth-aachen.de/research/projects/tinywifi/.

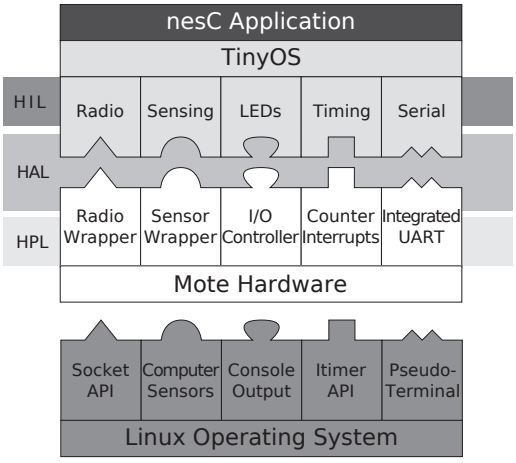


Figure 1: TinyWifi Architecture

The hardware abstraction layer (HAL) translates hardware independent functionality (HIL) to the device specific modules of the hardware presentation layer (HPL). TinyWifi replaces the hardware dependant modules at the HPL layer with its corresponding Linux based implementation of HPL components.

TinyOS owns a very rich protocol repository for IEEE 802.15.4 based networks. Among the most prominent protocols developed for TinyOS are CTP [12], 4BLE [7], BVR [8], S4 [18], BCP [19], and PAD [3]. Supporting flexible networking structures and achieving reliable and energy-efficient multihop communications drives the design philosophy of these protocols.

2.2 Design Overview

TinyWifi enables the execution of protocols developed in nesC (i.e. for TinyOS) on Linux based Wi-Fi devices, i.e., nodes in meshnets. The key idea is to exploit the modularity of the TinyOS hardware abstraction architecture: TinyWifi replaces the existing TinyOS core at HPL to provide the exact same hardware independent functionality and interfaces as a regular sensor node platform (cf. Figure 1). For example, the active messaging interface for IEEE 802.15.4 based CC2420 chips is replaced with a socket based communication interface for Linux networking. Similarly, hardware timers are replaced with Linux timers.

This seamless integration enables TinyWifi to export the resources of typical Linux network devices such as large memory, more processing power, and higher communication bandwidth to the sensornet protocols developed in nesC. However, this transition from mote-class devices to Linux-driven nodes at HPL is not straight forward [1]. Apart from handling hugely different link layers, TinyWifi has to deal with completely different hardware platforms, programming and runtime environments, and computational resources as discussed in Section 3.

TinyWifi runs as a Linux user space process. It is easy to use and provides simple command-line primitives such as *make linux* and *make linux run* for compiling and executing protocols. The TinyWifi specific code integrates seamlessly into the existing TinyOS source tree. Using TinyWifi as a

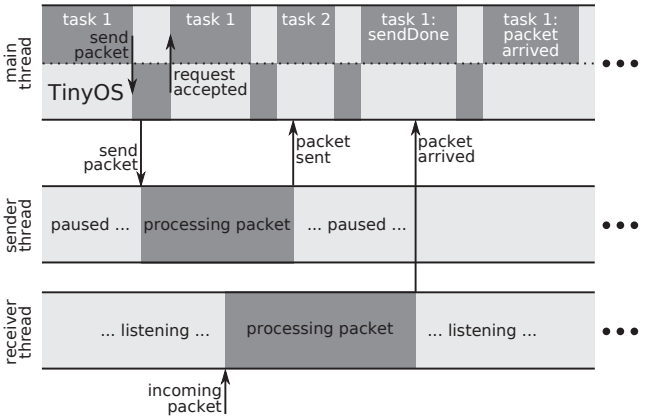


Figure 2: Split-phase operation
Using two parallel threads, e.g. a sender and a receiver in the case of radio communication, we achieve the split-phase functionality of TinyOS in TinyWifi.

development platform, any protocol that is written in C or nesC language can be executed both in IEEE 802.11 and 802.15.4 based networks.

2.3 Key Features

TinyWifi is centered around four design features:
Transparency: Existing sensornet protocols and algorithms developed in nesC shall not break when we run them on Linux based platforms despite the change in the underlying platform characteristics, such as medium access technologies and hardware capabilities.
Versatility: The implementation should be adaptable to the characteristics of the target platform, for example, wether to encapsulate TinyOS packets in UDP datagrams or bypass the network stack and send them directly over the wireless interface.
Usability: No modifications should be necessary for nesC protocols and the target platform (i.e. Linux) to function. In other words, TinyWifi should be directly deployable in any network that supports Linux based nodes.
Adaptability: TinyWifi should expose the additional capabilities, such as larger memory and processing power, of the Linux platform to the TinyOS protocols.

3. DETAILED ARCHITECTURE

We now describe the detailed architecture of each component in TinyWifi.

3.1 Radio Communication

Radio communication is the most vital service and the pivotal difference between sensornets and meshnets at the MAC and PHY layers. TinyOS provides an *active messaging* service [14] on top of a mote's low-power radio chip such as CC1000, CC2420 etc. An active message contains the identification number of the user-level handler, and the data payload is passed as arguments. The network is modeled as a pipeline and there are no additional buffers used to store messages. Therefore, the handler is responsible for accepting the message from the network and processing it quickly to be able to receive the next message. TinyWifi replaces this active messaging layer with its own communication service. It provides two flavors of communication services on top of the IEEE 802.11 based network interface: UDP based overlay and direct MAC access.

In UDP based overlay communication, we encapsulate TinyOS messages in UDP packets using datagram sockets. We broadcast UDP packets but suppress routing by adjusting the TTL-field so that packets are only received by TinyWifi nodes within the radio range. This flavor of communication has four key advantages: (1) It is simple to implement and very useful for initial debugging and testing, (2) it maximizes portability, (3) it minimizes interference with different applications on the network, and (4) it allows direct execution of TinyWifi without negotiating special kernel level privileges.

However, UDP based communication has two main disadvantages: (1) It introduces significant processing overhead in processing each packet at the IP and UDP layers which is irrelevant for TinyWifi, and (2) it does not provide direct access to the wireless interface to utilize important information, such as RSSI and LQI, which might sometimes be essential for higher layer protocols. For this reason, we provide an interface that utilizes *raw sockets* to enable direct access to the underlying wireless interface. In the current TinyWifi implementation, this interface is the default communication device.

3.2 Split-Phase Operation

TinyOS employs split-phase operations [9] for system calls, which is a significant departure from how Linux handles its system calls. The key idea of a split-phase operation is to account for the mote's concurrency and avoid blocking-calls in the system. Many system services, such as sending/receiving a packet, are completed in two phases. A command that starts a system service returns immediately while the completion of that service is signaled later via a *callback* event. This mode of operation allows TinyOS to process multiple services and the main program in parallel using concurrent processing hardware.

TinyWifi supports both blocking system calls and split-phase operations. The support for blocking system-calls in Linux is trivial (i.e., it is built on native blocking calls). However, to mimic the split-phase programming and runtime operation of TinyOS, we use threads to monitor I/O related operations that run in parallel with the CPU, for example on network cards. When an application module needs to perform an I/O operation, the corresponding thread is activated and the application continues with its own execution. The completion of these parallel processing threads is then indicated via a Linux signal, which in turn triggers the main TinyOS thread.

Figure 2 shows the split-phase operation of TinyWifi for radio *send* and *receive* primitives. A sender and a receiver thread are responsible to handle the respective requests from applications and later signal their completion. The provision of both, the blocking system calls and the split-phase operations, in TinyWifi allows developers to choose a mechanism appropriate for their protocols and applications.

3.3 Timers

The accuracy of timer operation is critical for the functioning of protocols and time synchronization mechanisms. On the sensor-motes, protocols can directly access hardware counters and timers but this is generally not done by the protocols on Linux based network devices.

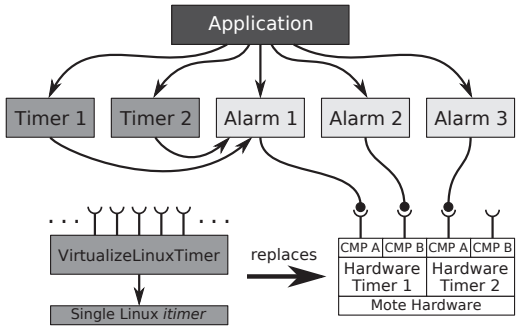


Figure 3: Timers

The TinyWifi timer implementation provides several instances of alarms and timers because Linux only provides a single realtime timer per process.

The TinyOS timing functionality is based on the hardware timers present in microcontrollers. A sensor-node platform provides multiple realtime hardware timers to specific TinyOS components at the HAL layer - such as alarms, counters, and virtualization. Once configured, these timers trigger an interrupt in the future without the need for continuous monitoring.

Although our target devices provide hardware timers as well, user space applications have no access to them. Therefore, we use Linux's *itimer* library. This library only provides a single realtime timer to each process running on a Linux kernel. However, TinyWifi requires multiple timers to cater the needs of multiple protocols running inside one TinyWifi process, such as link estimators and routing. Therefore, we introduce a new *VirtualizeLinuxTimer* component that virtualizes a single *itimer*. This component provides multiple instances of the new *LinuxTimer* module. Figure 3 shows the concept of virtual timers and alarms on top of a single *itimer* that replaces the hardware timers of a mote. This virtualization of a single timer is achieved by maintaining a delta-queue, sorted in the order of time, of the registered timer events. The *itimer* is then rescheduled to the most significant event in the queue, i.e., the event at the front end of the queue. This way, we provide timing functionality analogous to typical mote platforms.

3.4 Miscellaneous Services

Radio communication, split-phase operation and timers make up the major pieces of our design. However, there are certain functionalities, e.g. serial communication and debugging support, peculiar to motes that are used by the majority of sensornet applications. TinyWifi also provides these functionalities to (i) enhance *usability* by enabling full fledged TinyOS support in meshnets, and (ii) to ensure a *transparent* application transition between TinyWifi and TinyOS.

3.4.1 Serial Communication

The majority of TinyOS applications uses the serial communication for mote-to-PC data exchange. In order to provide a similar functionality, i.e., serial active messaging on a TinyWifi device, TinyWifi uses a Linux pseudo terminal. As with typical motes, an unaltered serial forwarder based on the C programming language connected to the pseudo terminal allows for sending and receiving serial data to and from a TinyWifi node.

Testbed	Available Nodes	Node Degree	Radio	Path Stretch
UMIC	35	4	802.11	$\backsim 3$
Indriya	125	18	802.15.4	$\backsim 3$

Table 1: Testbed Characteristics

UMIC is an IEEE 802.11 based meshnet while Indriya is a TinyOS based sensornet. Node Degree refers to the average number of one-hop neighbors. Path Stretch refers to the average number of hops between two non neighboring nodes, derived from the connectivity graphs.

3.4.2 Sensing and Debugging

Since our focus is network protocol testing, sensing is a subordinate issue. Nevertheless, we do supply dummy sensor implementations to allow for TinyWifi to be used out of the box.

In addition to the *printf* library to output debugging information through the serial interface to an attached PC and displayed in a human readable manner, TinyOS provides *dbg* functions to print additional information. In the TinyWifi implementation, we print those messages directly to the standard output. Similarly, to indicate the status of a physical mote to a developer, motes are equipped with LEDs. TinyWifi provides pseudo-LEDs: Messages are sent to standard output similar to the debugging mechanism of the TOSSIM [17] simulator.

4. EVALUATION

Our evaluation of TinyWifi focuses on the correctness of TinyWifi implementation by observing link and network layer behavior. We also show the utility of TinyWifi as a customary wireless evaluation platform by evaluating three point-to-point routing protocols on an IEEE 802.11 testbed and comparing them to the results of evaluation on IEEE 802.15.4 testbed. We note that a single implementation of the protocol was used for these evaluations on IEEE 802.11 and 802.15.4 testbeds. These protocols are only implemented in nesC for TinyOS. Our evaluation aspects aim to demonstrate the correctness and versatility of TinyWifi rather than stress-testing the employed protocols or platforms.

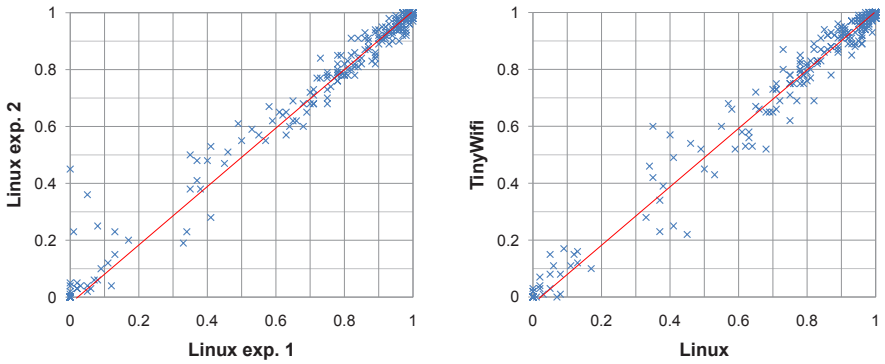
We evaluate TinyWifi on UMIC [22] and Indriya [6] testbeds. UMIC is a Linux based meshnet deployed at RWTH Aachen University. It consists of 51 IEEE 802.11a/b/g based meshrouters[2] located in various rooms at the department of computer science. Each node has a 500 MHz CPU and 256 MB of RAM. Indriya is a sensornet deployed at National University of Singapore. There are 127 nodes on Indriya. Each node on Indriya has an MSP430 CPU with 10 KB of RAM and a low power CC2420 radio, which can run IEEE 802.15.4 protocols. The major characteristics[3] of these testbeds are shown in Table 1.

4.1 Evaluating TinyWifi Implementation

In the following we evaluate the correctness and applicability of TinyWifi both at the link and network layers and through the behavior of native TinyOS protocols in an IEEE 802.11 testbed.

[2]Only 35 were available for our experiments

[3]We refer readers to the respective testbed websites for connectivity graphs and further information: http://www.umic-mesh.net/meshconf/#geographical and http://indriya.comp.nus.edu.sg/motelab/html/index.php



(a) PRRs for back-to-back experiments in Linux

(b) PRRs for TinyWifi vs Linux

Figure 4: Packet Reception Rates

PRR comparison between TinyWifi and Linux on IEEE 802.11. Each link PRR is estimated using a native Linux socket protocol as well as TinyWifi protocol. If both the protocols estimated that a given link is of the same quality, the point would lie on the 45 degree line. There are a total of 1226 points representing the PRR of each link in the network. Overall, TinyWifi and Linux native link estimation agree, hence most of the points are near the 45 degree line.

4.1.1 Link Layer

We first show that the communication service of TinyWifi does not impact the behavior of the underlying link layer when compared to the native platform. To this end, we correlate the Packet Reception Rates (PRR) of nodes using both TinyWifi and the native platform, i.e., Linux. Ideally, in the scatter-plot representation of such a correlation, every single data point should lie on the 45 degree line. However, this is not even achieved in back-to-back experiments on the native Linux platform as shown in Figure 4(a). This is due to the unpredictable and highly dynamic nature of the wireless medium. Figure 4(b) depicts the correlation between PRRs of TinyWifi and Linux platforms. We can clearly observe the strong similarity between Figures 4(a) and 4(b). Hence, we conclude that the (user-space) implementation of TinyWifi does not adversely affect the link level behavior of the underlying radio technology.

4.1.2 Network Layer

To evaluate the correctness of TinyWifi on the network layer, we compare the behavior of a native TinyOS protocol in TinyWifi with the behavior of a Linux-native implementation of the same protocol. We show that the nesC implementation of a protocol for TinyOS, when evaluated using TinyWifi, is commensurate with its native counterpart. To this end, we evaluate the behavior of the Collection Tree Protocol (CTP) in both its nesC and Click [15] implementations. Click is a highly recognized software architecture for building modular and configurable protocols.

Our comparative analysis uses the CTP protocol for several reasons: CTP has become a de-facto standard in collection routing in sensornets. It has also been implemented in various languages to support different OS and simulation platforms, such as Mantis OS, Contiki OS, Sun SPOTs, and Castalia Simulation. It has been thoroughly tested using six different MAC layers. The mechanisms used in CTP have also been incorporated in IETF RPL - the IPv6 protocol for low-power and lossy networks. Recognizing its highly efficient and reliable delivery in networks with lossy links, CTP has been extended for point-to-point communications in meshnets.

Delivery rate is a metric commonly used in evaluating sensornet protocols. It is equivalent to the average end-to-end reliability between sensor nodes and the receiver that receives the sensor data using multihop routing. Our key evaluation metric is the *delivery rate* for two reasons: (1) The current TinyWifi implementation is not optimized for throughput evaluations, and (2) the default operational parameters, such as buffer sizes, of the protocols and the platforms under consideration are different. To establish a fair comparison base for other performance benchmarks, such as throughput and jitter, we need to modify these parameters. However, this is beyond the scope of our contribution in this paper. In our experiment on UMIC testbed, we used one node as the destination in the network. All other nodes send a burst of 100 packets, one at a time to the single destination. The receiver node simply logs the received packets identified by a unique sequence number and a sender ID. To establish a baseline and to enable better understanding of the results, we also compare CTP with OLSR, a standard routing protocol for meshnets. Figure 5 shows the cumulative distribution of the delivery rates for both implementations of CTP and OLSR. Figure 6 displays the pairwise delivery rates for each node pair in detail. These results show that the performance of Linux native CTP (which required reimplementation) is similar to the TinyWifi version of CTP.

Overall, these results conclude that TinyWifi enables the direct and unaltered execution of nesC protocols in IEEE 802.11 based networks. The implementation overhead of TinyWifi modules does not influence protocol performance as shown in the case of CTP. The minor difference in the results (i.e., 1%) could be due to the varying link qualities across the experiments. This means that using TinyWifi, the implementation effort for the Click implementation of CTP [4] (i.e., approx. 7000 lines of codes excluding Click libraries) could be saved.

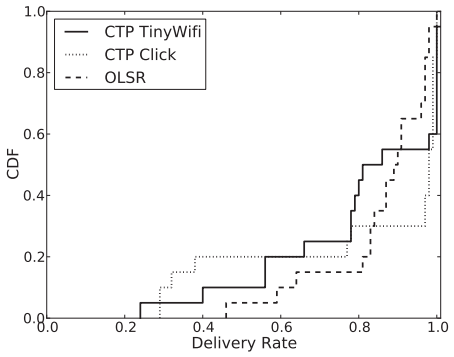


Figure 5: CDF of per node delivery rate
The performance of CTP under TinyWifi and Click is very comparable. The average delivery rate of CTP TinyWifi is 0.81, CTP Click achieves 0.82 and OLSR 0.85.

4.2 Protocol Evaluation

After evaluating TinyWifi implementation, we now demonstrate its utility as an efficient evaluation and runtime platform. To this end, we evaluate three well-established routing protocols - namely BVR [8], PAD [3] and S4 [18] - in two different classes of networks. These protocols are native to sensornets (IEEE 802.15.4) but their mechanisms are believed to be equally relevant for meshnets and ad-hoc networks (IEEE 802.11) [1, 18]. TinyWifi enables evaluation of these protocols in IEEE 802.11 networks and thus provides a deeper insight into their behavior without re-implementation. In the following, we briefly describe these protocols.

Beacon Vector Routing (BVR) is a scalable point-to-point routing protocol for wireless networks. In BVR, a node address reflects the hop count from this node to a small set of designated nodes (landmarks) in the network. At each hop, the best next hop distance-wise is selected.

Small Stretch and Small State Routing (S4) is a cluster based extension of BVR. In S4, a node is identified by the landmark node closest to the destination instead of its virtual coordinates. S4 aims at reducing the routing stretch of BVR at the cost of higher state maintenance, i.e., each node additionally maintains a local cluster of neighboring nodes.

Probabilistic Addressing (PAD) is an extension of BVR for unstable networking conditions. It assigns probabilistic address to nodes instead of sharp virtual coordinates and thus achieves a significant reduction in address updates. The routing mechanism of PAD is similar to BVR.

The key performance metrics of these protocols considered here include: (1) address stability, (2) average hop distance from landmarks, and (3) the number of transmissions required for a packet to reach its destination. Our evaluation for the first two metric compares PAD with BVR only. This is because S4 shares the virtual coordinates establishment with BVR.

Address Stability: Address updates are expensive in wireless networks where nodes have to determine their own addresses based on the underlying connectivity in the network. In such virtual coordinates based protocols, addresses are typically stored in a database. Frequent address changes thus result in a significant overhead due to frequent updates in the address database. Hence, address stability is one of the key performance measures of virtual coordinates based routing protocols. Figures 7(a) and 7(b) show the cumulative distribution of address change rate in IEEE 802.11 (UMIC) and IEEE 802.15.4 (Indriya) networks, respectively. The address change rate is defined as the share of routing update intervals in which the nodes update their addresses. These results clearly show that PAD performs better than BVR in IEEE 802.11 networks. However, the magnitude of improvement is smaller compared to what was observed in IEEE 802.15.4 networks. This is due to the different node degrees in the networks: PAD derives its addresses from multiple paths leading towards a landmark and tolerates link quality changes along a specific path. Hence, in a sparse network, such as UMIC, there is only a limited number of unique paths that can be represented in a PAD address.

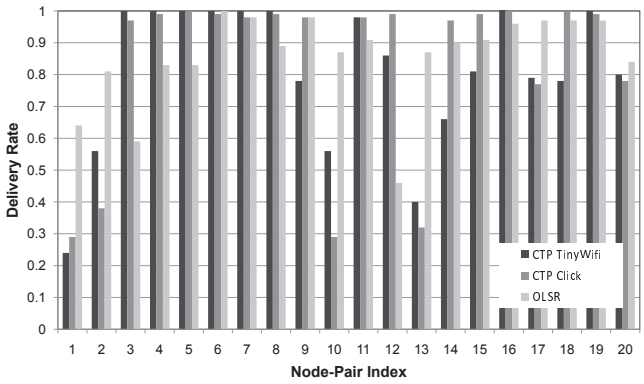


Figure 6: Delivery rates for each sender-receiver pair using OLSR and two implementations of CTP.

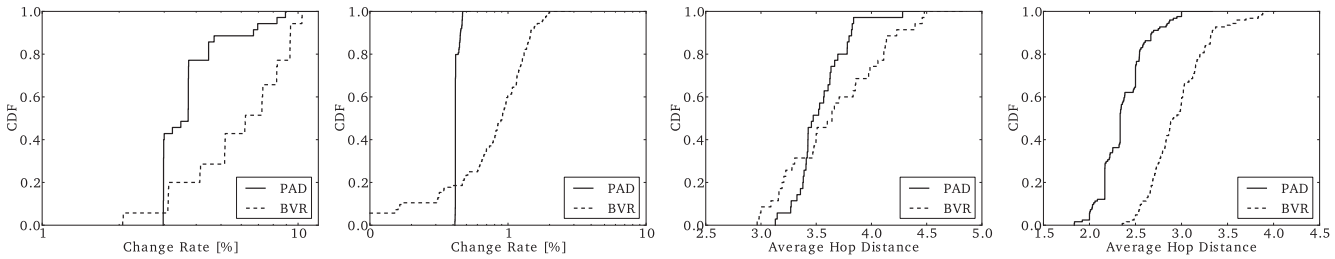
Hop Distance: The hop distance metric determines the number of hops between a node and all landmarks in the network. Figures 7(c) and 7(d) depict the CDF of hop distances averaged over all landmark trees. It can be seen that PAD achieves lower hop distances than BVR in both testbeds. This is because PAD always enables shortest paths to dominate its coordinate distributions [3]. Whereas, BVR only selects good quality paths using PRR based link estimation. Hence, due to the higher node degree in Indriya, the probability of the shortest path being different than the stable path selected by BVR is much higher.

Routing Cost: Finally, we evaluate the routing cost, i.e., the average number of transmissions required for a packet to reach its destination. Figure 8 shows that PAD outperforms both S4 and BVR in the IEEE 802.15.4 network. However, in the IEEE 802.11 network, PAD and BVR achieve similar results while still performing better than S4. These results show that S4's performance is dependent upon dense deployments and a stable network topology.

Overall, these evaluation results indicate that TinyWifi provides important hints about protocol performance on different link layers and in different network types. Hence, the feasibility of a protocol in different classes of wireless network cannot simply be assumed, it rather needs to be validated using platforms such as TinyWifi.

5. LIMITATIONS

In its current implementation, TinyWifi serves as a general *enabling* platform for multiple link layers. Due to this focus and our effort to keep a small code-base, TinyWifi does not export specific link layer services of either the original or the target OS. Currently, this means that TinyOS protocols that rely on a specific link-layer service which is not



(a) Coordinate change rate on 802.11 testbed (b) Coordinate change rate on 802.15.4 testbed (c) Average hop distance from landmarks on 802.11 testbed (d) Average hop distance from landmarks on 802.15.4 testbed

Figure 7: Addressing

Addressing results from IEEE 802.11 and 802.15.4 based testbeds. PAD maintains its superior performance in terms of address stability across multiple wireless network classes.

provided by IEEE 802.11 link layers are not supported. As one example, the MultihopLQI collection protocol (released with TinyOS) heavily relies on *link quality indicator* (LQI) information for establishing routing tables. Hence, MultihopLQI will only be applicable in a meshnet if the corresponding link-layer exports LQI information to higher layer protocols.. The support of specific link layer services means a tradeoff between the implementation complexity and the usefulness and applicability of this service in the target domain.

Similarly, TinyWifi, at the moment, only focusses on enabling TinyOS routing protocols to run in Linux. We see this as a first step towards a general platform for protocol and application experimentation and evaluation. However, some TinyOS characteristics, such as a one-packet outgoing buffer, remain because TinyOS protocols rely on and are designed for them. To make full use of a target platform's (additional) resources, we need a mechanism to allow protocols and applications to capitalize on these resources as well as they can.

It is widely believed that network layer protocols such as geographic routing, graph embedding [20], and AODV can work on a variety of wireless networks. TinyWifi enables the validation of such hypotheses. Our preliminary results suggest that some protocols (e.g., routing) do work on vastly different wireless networks - on radios that support a few tens of Kbps to those that support tens of Mbps data rates. However, we note that not all protocols can work in such widely different platforms. The network protocols running on IEEE 802.15.4 radios can assume constant bit rate while the network protocols running on IEEE 802.11b radios cannot make that assumption. Packet delivery takes a more predictable time on 802.15.4 radios compared to 802.11b radios due to more complex OS kernel, NIC driver, and generally higher level of programming interface. Some assumptions about link layer properties, although ideally avoided, are implicitly embedded in the network protocol design. TinyWifi helps us test the network protocols in vastly different radio platforms and discover such implicit assumptions behind network protocol design.

6. RELATED WORK

EmStar [11] and VIPE [16] are two notable related efforts that enable network protocol execution across heterogenous computation and communication platforms.

EmStar is a software environment for deploying complex applications on heterogenous sensornet designs, incorporating a mixture of mote-class devices and Linux driven micro-servers. The idea is to leverage the additional resources of a distributed micro-server based network to improve robustness and system visibility of sensornet deployments. EmStar provides its own runtime environment, protocol execution on different classes of devices is not a goal of this approach. TinyWifi, on the other hand, aims at migrating the whole protocol to a completely different class of wireless networks.

VIPE evolves the implementation of a protocol from its design up to its deployment without re-implementation of single parts. It provides an encapsulation of minimal core functionality in small building blocks. These blocks may then be used by a protocol on different platforms (i.e. simulation, emulation, testbeds, deployment). TinyWifi is similar to VIPE in a sense that it provides a common service architecture across multiple platforms. However, VIPE assumes a common programming and runtime environment across these platforms. Unlike TinyWifi, it does not address the challenges associated with spanning a wider range of platforms and thus would require existing IEEE 802.15.4 based protocols to be re-implemented using VIPE's interface abstractions.

Building protocols for multiple networking classes is a common trade in today's systems. For example, DHCP operates on multiple link layers (e.g. Ethernet and WiFi) as well as wireless cards from different vendors. However, these link layers are highly standardized (with common interfaces and runtime environments), and span resource-rich platforms ranging from data centers to embedded systems. Hence, existing cross-platform protocols are restricted to very similar platforms. Besides saving re-implementation effort, the distinctive feature of TinyWifi is that it enables protocols to run across *vastly* different link layers (i.e. IEEE 802.11 and 802.15.4) with even wider ranges of device capabilities: Going all the way from 8-bit micro-controllers, with a few KB of memory and low-power radios capable of data rates as low as a few tens of Kbps, to platforms with an order of magnitude higher processing and storage capabilities and equipped with radios that support data rates up to few tens of Mbps. TinyWifi elegantly addresses the associated challenges, such as a different programming and runtime environment, for bridging protocols between such a wider range of platforms.

7. CONCLUSIONS AND FUTURE WORK

We presented TinyWifi, a network protocol evaluation platform for diverse classes of wireless networks. Using TinyWifi, developers can evaluate a single implementation of prototypes across multiple wireless network classes such as

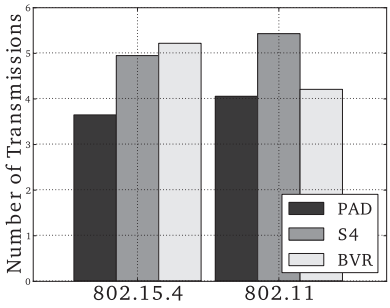


Figure 8: Routing

Routing results from IEEE 802.11 and 802.15.4 based testbeds. The results show a similar trend in both domains: S4's cluster based approach struggles in sparse network environments.

sensornets and meshnets. We demonstrated the utility of TinyWifi by evaluating four well known routing protocols in two testbeds that make use of different radio technologies. Our evaluation shows that TinyWifi allows us to better understand and reason about the performance characteristics of protocols in different networking environments.

We are still in the initial phases of our work. Although our protocol evaluation demonstrates the correctness of our TinyWifi implementation, we still need to stress-test different design components such as timers and split-phase operations. Besides using multi-hop routing protocols, we plan to expand our work to evaluating and supporting dissemination and network time synchronization protocols. Finding a well-balanced set of features that supports a multitude of protocols and applications as well as better providing the resources of the target platform to protocols are further steps in the development of TinyWifi.

Acknowledgments

Many thanks to Jung Il Choi for contributing the Click implementation of CTP. This research was funded in part by the DFG Cluster of Excellence on Ultra High-Speed Mobile Information and Communication (UMIC), German Research Foundation grant DFG EXC 89, and the Stanford Army High Performance Computing Research Center (grant W911NF-07-2-0027).

8. REFERENCES

[1] M. H. Alizai, B. Kirchen, J. A. B. Link, H. Wirtz, and K. Wehrle. Poster abstract: Tinyos meets wireless mesh networks. In *ACM Sensys*, 2010.

[2] M. H. Alizai, O. Landsiedel, J. A. Bitsch Link, S. Goetz, and K. Wehrle. Bursty traffic over bursty links. In *SenSys'09*, Nov. 2009.

[3] M. H. Alizai, T. Vaegs, O. Landsiedel, S. Götz, J. A. Bitsch Link, and K. Wehrle. Probabilistic addressing: Stable addresses in unstable wireless networks. In *ACM/IEEE IPSN*, 2011.

[4] J. I. C. Choi, M. Jain, J. W. Lee, and J. Batiz-Benet, 2011. CLICK CTP, Stanford Information Networking Group: http://sing.stanford.edu/gnawali/ctp/.

[5] D. S. J. De Couto, D. Aguayo, J. Bicket, and R. Morris. A high-throughput path metric for multi-hop wireless routing. In *ACM MobiCom*, 2003.

[6] M. Doddavenkatappa, M. C. Chan, and A. A.L. An experience of building indriya. In *National University of Singapore*, 2009.

[7] R. Fonseca, O. Gnawali, K. Jamieson, and P. Levis. Four-bit wireless link estimation. In *HotNets*, 2007.

[8] R. Fonseca, S. Ratnasamy, J. Zhao, C. T. Ee, D. Culler, S. Shenker, and I. Stoica. Beacon vector routing: Scalable point-to-point routing in wireless sensornets. In *NSDI*, May 2005.

[9] D. Gay, P. Levis, and D. Culler. Software design patterns for tinyos. In *Proceedings of the 2005 ACM SIGPLAN/SIGBED conference on Languages, compilers, and tools for embedded systems*, LCTES '05, pages 40–49, New York, NY, USA, 2005. ACM.

[10] D. Gay, P. Levis, R. von Behren, M. Welsh, E. Brewer, and D. Culler. The nesc language: A holistic approach to networked embedded systems. In *ACM SIGPLAN PLDI*, 2003.

[11] L. Girod, N. Ramanathan, J. Elson, T. Stathopoulos, M. Lukac, and D. Estrin. Emstar: A software environment for developing and deploying heterogeneous sensor-actuator networks. *ACM Trans. Sen. Netw.*, 3, 2007.

[12] O. Gnawali, R. Fonseca, K. Jamieson, D. Moss, and P. Levis. Collection tree protocol. In *SenSys '09*, 2009.

[13] V. Handziski, J. Polastre, J. H. Hauer, C. Sharp, A. Wolisz, and D. Culler. Flexible hardware abstraction for wireless sensor networks. In *In EWSN*, 2005.

[14] J. Hill, R. Szewczyk, A. Woo, S. Hollar, D. Culler, and K. Pister. System architecture directions for networked sensors. *SIGPLAN Not.*, 35:93–104, 2000.

[15] E. Kohler, R. Morris, B. Chen, J. Jannotti, and M. F. Kaashoek. The click modular router. *ACM Trans. Comput. Syst.*, 18:263–297, August 2000.

[16] O. Landsiedel, G. Kunz, S. Götz, and K. Wehrle. A virtual platform for network experimentation. In *ACM VISA '09*, 2009.

[17] P. Levis, N. Lee, M. Welsh, and D. Culler. Tossim: accurate and scalable simulation of entire tinyos applications. In *SenSys*, 2003.

[18] Y. Mao, F. Wang, L. Qiu, S. S. Lam, and J. M. Smith. S4: Small state and small stretch routing protocol for large wireless sensor networks. In *NSDI*, 2007.

[19] S. Moeller, A. Sridharan, B. Krishnamachari, and O. Gnawali. Routing without routes: The backpressure collection protocol. In *IPSN*, 2010.

[20] J. Newsome and D. Song. Gem: Graph embedding for routing and data-centric storage in sensor networks without geographic information. In *ACM SenSys*, 2003.

[21] A. Woo, T. Tong, and D. Culler. Taming the underlying challenges of reliable multihop routing in sensor networks. In *SenSys '03*, 2003.

[22] A. Zimmermann, M. Güneş, M. Wenig, J. Ritzerfeld, and U. Meis. Architecture of the hybrid MCG-mesh testbed. In *ACM WiNTECH'06*, 2006.