A Prototyping Environment for Wireless Multihop Networks

Fehmi Ben Abdesslem[1], Luigi Iannone[2], Marcelo Dias de Amorim[1], Katia Obraczka[3], Ignacio Solis[4], and Serge Fdida[1]

[1] Université Pierre et Marie Curie – Paris 6
{fehmi,amorim,sf}@rp.lip6.fr
[2] Université Catholique de Louvain
luigi.iannone@uclouvain.be
[3] University of California at Santa Cruz
katia@cse.ucsc.edu
[4] Palo Alto Research Center
isolis@parc.com

Abstract. Relative to the impressive number of proposals addressing the multitude of challenges raised by IEEE 802.11-based wireless networks, few have known real implementation. In wireless networks, due especially to the unpredictable nature of the wireless channel, bridging theory and practice is far from trivial. In this paper, we advocate including prototyping in the design process of wireless protocols. The goal is to speed up the design process and to help validating novel solutions under real conditions. To this end, we introduce *Prawn*, a tool that allows rapid prototyping of wireless network protocols. The basic idea behind Prawn is to provide a set of basic building blocks that implement common functionalities needed by a wide range of wireless protocols (e.g., neighbor discovery, link quality assessment, message transmission and reception). Besides these ready-to-use blocks, Prawn also provides a standard API that allows protocol designers easy access to the Prawn primitives. Through a number of examples, we showcase Prawn as a simple, yet powerful tool for fast prototyping of wireless network protocols.

1 Introduction

Designing protocols for wireless networks poses countless technical challenges due to a variety of factors such as node mobility, node heterogeneity, and power limitations. Furthermore, the characteristics of the wireless channel are non-deterministic and can be highly variable in space and time. This implies that testing and evaluating such protocols under real operating conditions is crucial to ensure adequate functionality and performance.

In fact, the networking research community has already acknowledged the importance of testing and evaluating wireless protocol proposals under real-world conditions. As a result, over the last few years, a number of testbeds, such as Orbit [1], Roofnet [2], and MiNT-m [3], as well as implementation tools, such

Table 1: Introducing a prototyping step.

	Implementation	*Emulation*	*Simulation*	*Prototyping*
Code	Real	Real	Synthetic	Synthetic
Medium	Real	Synthetic	Synthetic	Real
Examples	Click, Roofnet	Empower	NS-2	**Prawn**

as Click [4] and XORP [5], have been developed to support the deployment and evaluation of wireless protocols under realistic scenarios.

When designing communication systems, and before the final version, there are mainly three evaluation methodologies commonly used, namely mathematical analysis, simulation, and emulation, all of them using a synthetic virtual environment. Evaluation in real environment is done by producing a real implementation of the protocol, with or without the help of implementation tools. Moving to this second step requires non-negligible programming skills and time.

In this paper, we go a step further and advocate including *rapid prototyping* as an integral part of the design process (cf., Table 1). We postulate that what is needed is a tool that makes prototyping as quick, easy, and effortless as possible. To this end, we introduce *Prawn* (PRototyping Architecture for Wireless Networks), a novel software environment for prototyping high-level (i.e., network layer and above) wireless network protocols.[5] On the one hand, rapid prototyping is complementary to current testbeds and tools, which are typically used to produce a beta version of the final implementation. On the other hand, rapid prototyping enables performing correctness verification, functionality, and performance tests under real operating conditions early enough in the design cycle that resulting feedback and insight can be effectively incorporated into the design.

Prototypes implemented with Prawn are not expected to be optimized, offering edge performance. Rather, our focus with Prawn is on obtaining, quickly and with little effort, a complete and fully functional instantiation of the system, in order to gain a first insight into its behavior in real conditions. Prawn makes prototyping as simple as writing network simulation scripts, with the difference that testing is done under realistic conditions. Assessing these conditions is done through the Prawn Engine, which runs as a background process that proactively performs tasks such as neighbor discovery and link quality assessment. This feature allows Prawn to provide accurate and up-to-date feedback from the wireless interface.

As shown by the several case studies presented in this paper, Prawn prototypes can be used for functional assessment as well as both absolute and comparative performance evaluation. Once the prototype has been extensively tested

[5] Currently, Prawn targets IEEE 802.11 networks, although its design can be extended to run atop other wireless network technologies.

and thoroughly validated, and its functional design tuned accordingly, it is then ready for final implementation (which is out of the scope of Prawn).

The remainder of this paper is organized as follows. We put our work on Prawn in perspective by reviewing related work in the next section. Section 3 provides an overview of Prawn, while in Sections 4 and 5 we describe Prawn's two main components in detail. Then we present in Section 6 a number of case studies showing how Prawn makes prototyping rapid and simple. Finally, we present our concluding remarks and directions for future work in Section 7.

2 Related Work

Simulations are perhaps the most widely used methodology for evaluating network protocols. They allow designers to evaluate the system at hand under a wide range of parameters like different mobility models and node heterogeneity, but only under synthetic channel models. Simulation has the advantage of allowing the exploration of the design space by enabling designers to vary individual protocol parameters (e.g., timers) and combinations thereof. Finally, they are instrumental for scalability analysis and they offer reproducibility. Examples of well-known simulation platforms include NS-2 [6], GloMoSim [7], and OPNET [8].

Emulation tries to subject the system under consideration to real inputs and/or outputs. Environments like EMPOWER [9] or Seawind [10] emulate the wireless medium by introducing packet error rates and delays. Other emulators like m-ORBIT [11] also emulate node mobility by space switching over a testbed of fixed nodes. A key advantage of emulation in the context of wireless/mobile networks is to facilitate testing by avoiding, for example, geographic and mobility constraints required for deployment.

More recently, a number of projects have pioneered the field of wireless protocol evaluation under real conditions. They include testbeds such as Orbit [1], and Roofnet [2] as well as tools that support protocol implementation like Click [4] and XORP [5]). As previously pointed out, such tools and Prawn have different goals, address different phases of the design process, and are therefore complementary. While tools like Click and XORP targets the implementation at the final stages of protocol design, Prawn focuses on prototyping a research proposal at the very early stages of the design process. Therefore, through Prawn, protocol designers can very quickly and easily generate a fully functional, although non-optimized, implementation for live testing in real scenarios.

3 Basic design

Prawn targets prototyping protocols and services at the network layer and above. Simplicity was a major goal we had in mind when designing Prawn; we wanted to ensure that learning how to use Prawn would be as intuitive and immediate as possible requiring only basic programming expertise. Our focus was to provide:

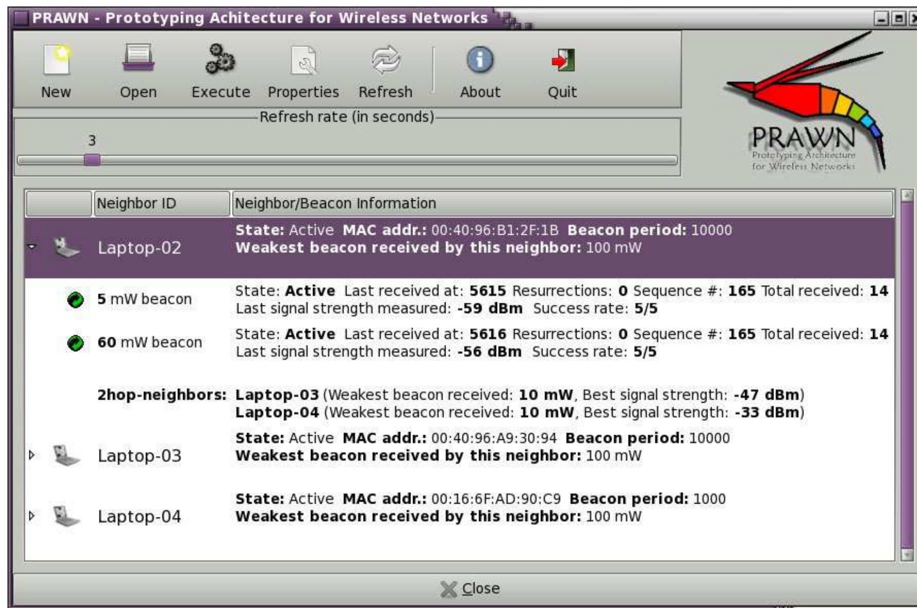

Fig. 1: Prawn graphic user interface.

(1) a concise, yet complete set of functions to realize high-level protocols and (2) a simple, easy-to-use interface to provide access to Prawn's functionalities.

Architecture. Prawn consists of two main components: (i) the Prawn Library (cf., Section 4), which provides high-level primitives to send and receive messages, retrieve information from the network, etc; and (ii) the Prawn Engine (cf., Section 5), which implements the primitives provided by the Prawn Library.

Prawn is distributed under GPL license. The current implementation runs on Linux atop IP for backward compatibility with the global Internet. The interaction between the Prawn Engine and the physical wireless device relies on the Wireless Tools [12]. This set of tools allows retrieving information from most wireless devices as well as setting low-level parameters. Furthermore, it is available with most Linux distributions.

Prawn's functionalities are accessible through the Prawn Library. Messages and requests received from the library are then processed by the Prawn Engine. The Prawn Library and the Prawn Engine communicate with each other through the loop-back interface using a simple request/reply mechanism. This choice simplifies modularity and portability.

Using Prawn. Running Prawn requires only few basic steps. First, it needs to be configured and installed on the machines that will be used in the experiments. In particular, in the Prawn configuration file it is necessary to set the names of the wireless interface and network (e.g., the ESSID). Optionally an IP address

can be specified. Otherwise, Prawn will randomly generate an IP address in a default subnetwork.

Using Prawn itself only requires two operations: executing the Prawn Engine and including the Prawn Library in the prototype code. The Prawn Engine (described in detail in Section 5) is launched as a command line program on machines connected in "ad hoc" mode. Prawn is supposed to run in daemon mode, but can run in console mode for debugging purposes. As stated before, Prawn provides a number of options that can be set/configured at the execution of the Prawn Engine. They are listed in Table 2. Other options are tunable in the `prawn.cfg` configuration file.

The Prawn Library (described in detail in Section 4) is composed of a set of primitives that are linked to the prototype through standard include files. Currently, prototypes can be developed either in C, Perl, or Java. For C development, the file `prawn.h` should be included in the header of the prototype code. Similarly, the file `prawn.pl` is to be included for prototypes developed in Perl, and the Prawn class methods can be used for prototypes developed in Java.

A graphic user interface is also available to monitor the neighborhood and to edit/run Perl scripts of Prawn prototypes (Figure 1).

Example. To illustrate the use of Prawn, we describe how to implement a simple "hello world" prototype using Prawn's Perl library. In this example we send a message from Bob to Alice.

1. Launch Prawn with "`prawn -d -N Bob`" in the first machine and "`prawn -d -N Alice`" in the second machine.
2. Get the first machine ready to receive messages by executing the following Perl script:

```
require "prawn.pl";
while(!@Message){
@Message=Prawn_Receive();
}
print 'Received : '.$Message[4].' from '.$Message[2]."\n";
```

3. On the other machine launch the following Perl script:

```
require "prawn.pl";
Prawn_Send("Hello World!","Bob");
```

The result is trivial: `Alice` sends a "Hello World" message to `Bob`, and `Bob` prints "`Received: Hello World from Alice`" on the screen. This simple example aims at showing the level of abstraction provided by Prawn, where low-level system knowledge (e.g., sockets, addressing) is not required. More elaborated examples, using the advanced features of Prawn, will be presented in Section 6.

4 The Prawn Library

The Prawn Library, currently implemented in C, Perl and Java, provides a set of high-level communication-oriented functions. They hide from protocol designers

Table 2: Prawn's command line options.

Option	Parameter	Default
-N name	node ID	hostname
-b period	beacon period in ms	10000
-h	help	–
-d	daemon mode	–
-v	verbose mode	–
-vv	more verbose	–
-p port	neighbor port	3010
-c port	client port	3020
-i I	uses wireless interface I	ath0
-P	set transmit power level	–
-n	no power control features	–
-W window	window size for PER	5
-V	version	–

lower-level features such as addressing, communication set-up, etc. Their syntax is quite simple and intuitive. Prawn's current set of primitives addresses basic functions required when prototyping a high-level communication protocol. Nevertheless, Prawn was designed to be easily extensible allowing new primitives to be implemented and integrated. The primitives currently available are:

- `Prawn_Info`(): Returns information on the configuration of the local Prawn Engine. Basically, it consists of the list of settings chosen when launching the daemon (cf., Table 2). Some examples are the node's ID, interface port number, and beacon period.
- `Prawn_Neighbors`(): Returns the list of the node's one-hop and two-hop neighbors as well as statistics concerning the quality of the respective links. In Section 5, a thorough explanation of the information returned by the Prawn Engine will be given.
- `Prawn_Send`(`Message`, `ID`, `TX_Pwr`): Sends `Message` to node `ID`; the optional argument `TX_Pwr` can be used to explicitly set the transmit power to be used during the transmit. `Message` can be a string, a number, a data structure, or any other data or control message, depending on the prototyped protocol
- `Prawn_Send_Broadcast`(`Message`, `TX_Pwr`): Sends a broadcast message containing `Message`; in a similar way to `Prawn_Send`(), the optional argument `TX_Pwr` allows to set the transmit power.
- `Prawn_Receive`(): Checks if a message has been received; if so, the message is returned. This primitive is non-blocking: if no message has been received, it just returns zero.

5 The Prawn Engine

The Prawn Engine is event-driven, i.e., its main process remains asleep waiting for an event to occur. An event can be triggered by a request from the Prawn Library (coming through the loop-back interface) or by a message received on the wireless interface. Meanwhile, local neighborhood discovery is performed through beacon and feedback packets.

5.1 Beaconing

To build and maintain the list of neighbors, each node running Prawn broadcasts 24-byte beacons periodically. The beacon period is configurable depending on the requirements of the prototype under development. By default, the Prawn Engine is configured to test connectivity under different power levels (useful for instance to prototype topology control algorithms based on power control [13, 14]). The Prawn Engine applies a round-robin policy to continuously change the transmit power. A beacon is first broadcast with the lowest power value. The transmit power level is successively increased for each beacon; up to the maximum transmit power. We call this sequence of beacons a *cycle*. The different values of the transmit power are either obtained from the interface or set by the user. This cycle is then repeated at every beacon period. This way, the time elapsed between two beacons sent with the same transmit power is equal to the beacon period.

The power control feature is optional, depending on the designer's needs. If this feature is disabled, each cycle is then composed of only one beacon, sent at the default transmit power level. The number of transmit power levels and their values are customizable, depending on the power control features provided by the wireless interface under utilization.

The beaconing packet format, which is illustrated in Figure 2, includes the following fields:

- `Type`: This field is set to '`1`'.
- `Transmit Power`: Transmit power used to send the beacon.
- `Transmitter ID`: Sender identifier.
- `Beacon Period`: Time period between two beacons transmitted with the same power level.
- `MAC Address`: MAC address of the transmitter.
- `Sequence Number`: Sequence number of the beacon.

Upon the reception of a beacon (or sequence of beacons if different transmit powers are used), various statistics can be derived. For instance, a node `A` can determine, at a given point in time, the minimum transmit power that `B` should use to send messages to `A`. This value corresponds to the lowest transmit power among all the beacons received by `A` from `B`. Of course, the minimum transmit power may change over time, and will be updated along the successive cycles.

Configuring Prawn is important to achieve an adequate balance between performance and overhead. For example, sending beacons too frequently would

```
 0              15 16            31
+--------+--------+--------+--------+
|  Type  | Power  |                 |
+--------+--------+                 +
|                                   |
+          Transmitter ID           +
|                                   |
+                 +--------+--------+
|                 |  Beacon Period  |
+--------+--------+--------+--------+
|             MAC Address           |
+                 +--------+--------+
|                 | Sequence Number |
+--------+--------+--------+--------+
```

Fig. 2: Prawn beacon packet format.

```
 0              15 16            31
+--------+--------+--------+--------+
|  Type  | Unused |                 |
+--------+--------+                 +
|                                   |
+           Destination ID          +
|                                   |
+                 +--------+--------+
|                 |Min T.P.|Max RSSI|
+--------+--------+--------+--------+
```

Fig. 3: Prawn feedback packet format.

generate high overhead. On the other hand, limiting the number of beacons is likely to result in out-of-date measures. For these reasons, the beaconing period is one of Prawn's customizable parameters and its value is carried in the header of each beacon sent. A beacon is considered lost when the beaconing period (included in previous received beacons) times out. By default, a neighbor is removed from a node's neighbor table when three consecutive beacons from this neighbor have been lost (or when three consecutive beacons for every transmit power have been lost).

5.2 Replying to Beacons

Nodes reply to beacons using 16-byte feedback packets, as shown in Figure 3. Feedback packets summarize neighborhood– and link quality information as perceived by the receiver of the beacons. This feature allows verifying the bidirectionality of links. Feedback packets are sent to every neighbor after a complete cycle.[6] Prawn keeps sending feedback packets also in the case where a neighbor is considered lost (a unidirectional link may still exist between the two nodes). Feedback packets contain the following fields:

– `Type`: This field is set to '`3`'.
– `Destination ID`: Identifier of the neighbor concerned by the feedback.
– `Minimum Received Transmit Power`: Is the transmit power of the beacon received with the weakest signal strength from that particular neighbor.
– `Maximum Received Power Strength` (in dBm): Is the maximum signal strength measured when receiving beacons from that particular neighbor.

The rationale for reporting the transmit power of the weakest beacon received from a particular neighbor is that it allows to roughly characterize the quality of the corresponding link. This estimation is also confirmed using the maximum received signal strength measured within a cycle.

[6] Note that if the power control feature is disabled, then the cycle is unitary.

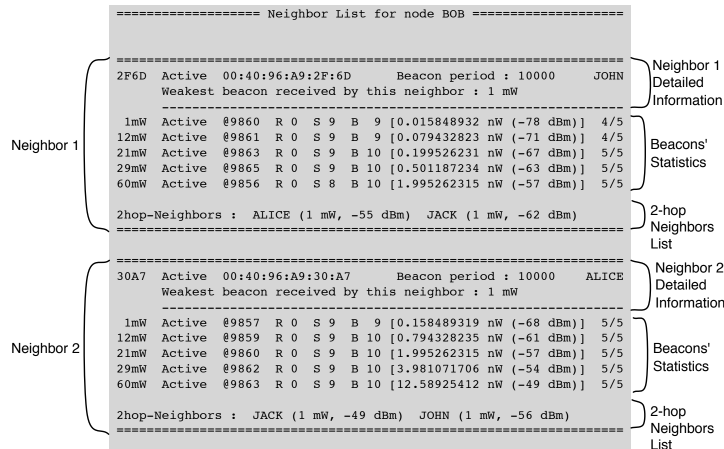

Fig. 4: Information provided by Prawn for a node whose ID is “BOB”.

Although destined to a single neighbor, feedback packets are broadcast and thus overheard by all one-hop neighbors. This way, nodes can obtain information on two-hop neighborhood (cf., Figure 4).

5.3 Getting Information from Prawn

When a node calls the Prawn_Neighbors() primitive, the Prawn Engine returns a data structure with information about the node's neighborhood. This information can be also obtained by running Prawn in console mode, e.g., for debugging purposes. Figure 4 shows a snapshot of the information returned by the Prawn Engine running in console mode on a node named “Bob”. This snapshot shows a list of Bob's neighbors, along with statistics on last beacons received by each neighbor for every transmit power. Basically, Bob has two active neighbors, John and Alice. The link between Bob and Alice has, on average, better quality than the one between Bob and John; indeed for beacons sent at 1 mW and 12 mW, only 4/5 of them have been received.

As previously described, neighborhood information is obtained through beacons and feedback packets. More specifically, broadcast beacons are used to build the list of direct neighbors. This list is established by gathering the transmitter ID of each received beacon. Moreover, data included in beacons and feedback packets inform each node what is the minimum transmit power required to reach a neighbor. Such information is of primary importance in assessing link quality. Another prominent link characteristic is the error rate, which is determined according to the beacon period included in each beacon transmitted. The Prawn Engine considers a beacon as lost when it is not received within the beacon period indicated by the corresponding neighbor. The size of the receiving window used to compute the error rate is customizable. For instance, in Figure 4, the

error rate for John's packets transmitted at 12 mW is 1/5, because over the 5 most recent 12 mW beacons transmitted by John, only 4 have been received.

When receiving a beacon, the Prawn Engine stores the received signal strength. Along with the transmit power of the beacon (which is also included in the beacon), the received signal strength returned by the Prawn Engine helps to evaluate the signal attenuation. The difference between the transmitted power level indicated in the beacon and the signal strength measured when the beacon is received can also be used by a protocol to characterize link quality.

5.4 Sending and Receiving Messages

Two other key functions performed by the Prawn Engine are transmission and reception of data (triggered by the `Prawn_Send`() and `Prawn_Receive`() primitives, respectively). The Prawn Engine is in charge of the communication set up, namely opening sockets, converting the receiver identifier to a valid IP address, encapsulating/decapsulating packets, and adjusting the transmit power before transmission. Figure 5 shows the structure of the data packets, which contain the following fields.

- `Type`: This field is set to '2'.
- `Transmit Power`: Power used to send the packet.
- `Payload Size`: Size of the payload field.
- `Payload`: Data being sent.

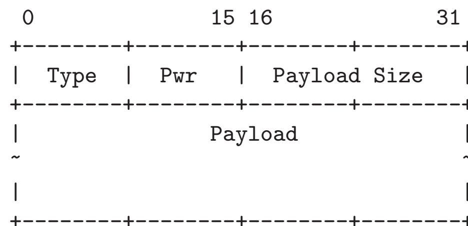

Fig. 5: Prawn data packet.

```
require "prawn.pl";
while(1){
    while(!@Message){
        $Message = Prawn_Receive();
    }
    $msgID = unpack("N",$Message[4]);
    if (!grep(/$msgID/,@ID_list)){
        push(@ID_list,$msgID);
        Prawn_Send_Broadcast($Message[4]);
    }
    @Message = ();
}
```

Fig. 6: Perl code of a flooding prototype.

Data packets are sent using UDP to the corresponding IP address. This explains why their header does not need to include the destination ID.[7] On the receiver side, the Prawn Engine listens on an open socket for any incoming packets. Packets are then decapsulated and sent to the prototype, which retrieves them by using the `Prawn_Receive`() primitive.

[7] Note that the same method cannot be used for beacons, since beacons are always sent broadcast at IP level and thus contain the broadcast address.

6 Prototyping with Prawn

Prawn is intended to be a tool for prototyping a wide range of communication algorithms for heterogeneous wireless networks. In this section, we first illustrate the use of Prawn through a number of case studies, highlighting its range of applicability and ease of use as well as how it can be employed to evaluate and test protocols.

6.1 Example 1: Flooding

Flooding is the simplest possible routing algorithm. Its basic operation is as follows: upon receiving a packet, each node sends it once to all its neighbors. [8] Thus the only requirement to implement this algorithm is to be able to receive and broadcast packets.

Prawn makes this algorithm easier to implement even for inexperienced programmers, since they do not need to know lower-level functions like sockets, ports, addressing, etc. Flooding can be implemented simply by using the `Prawn_Receive`() and `Prawn_Send_Broadcast`() functions.

Figure 6 shows how short and simple the flooding prototype using the Prawn Library is. This 12-line piece of code has been running successfully on our testbed. The behavior of the flooding algorithm is very different from simulations. Even more, Cavin et al. [15] tried to simulate the flooding algorithm using three different simulators namely, NS-2, OPNET, and GloMoSim, with exactly the same parameters and scenarios. Surprisingly, the results were considerably different, depending on the simulator used.

6.2 Example 2: Network Coding

While the previous section illustrates the use of Prawn to prototype one of the simplest protocols, we show, in this section, that Prawn can also be used to prototype more complex protocols. In particular, we show case the use of Prawn to prototype the COPE [16] network coding algorithm. Our goal here is to show that some evaluation of network coding proposals could be easily done without requiring a fully functional implementation of the algorithm.

For clarity, we briefly explain the essence of network coding through a very simple example. In traditional forwarding, when a node `A` and a node `B` want to exchange data via a third node `C`, both send their packets to `C`, and then `C` forwards the packets to `A` and to `B`. Exchanging a pair of packets requires 4 transmissions. Using network coding, instead of sending separate packets to `A` and `B`, node `C` combines (using the XOR function) both packets received from `A` and `B`, and broadcasts the encoded packet. Since `A` knows the packet it has sent, it can decode the packet sent by `B` (applying again the XOR function) from the encoded packet received from `C`. Similarly, `B` can decode the packet

[8] Of course, more elaborated variations of flooding exist, but here we consider it in its simplest form.

```
require "prawn.pl";
my @Stdby=();
my @Msg=();

while(!@Stdby) {@Stdby = Prawn_Receive();}
while(1){
 @Msg = Prawn_Receive();
 if (@Msg){
  if ($Msg[2] ne $Stdby[2]){
   $xored="";
   for ($i=0;$i<=1400;$i++){
    substr($xored,$i,1,substr($Msg[4],$i,1)^substr($Stdby[4],$i,1));
   }
   Prawn_Send_Broadcast($xored);
   @Stdby=();
   while(!@Stdby) {@Stdby = Prawn_Receive();}
  }
  else{
   if ($Stdby[2] eq "NodeA") {Prawn_Send($Stdby[4], "NodeB");}
   else {Prawn_Send($Stdby[4],"NodeA");}
   @Stdby=@Msg;
  }
 @Msg=();
 }
}
```

Fig. 7: Perl code of a network coding algorithm.

sent by A from the same packet received from C. Thus, with this method, only 3 transmissions, instead of 4, are required.

Using Prawn, we implemented a prototype of the algorithm described above. As shown in the Perl code running on node C (Figure 7), the first received packet is stored in a standby variable ($Stdby), then the next packet is stored as $Msg. If the two stored packets are not received from the same node, then they are XORed and broadcast. If, instead, both packets are from the same node, it does not make sense to XOR them. In this case, the packet stored in standby is sent as a normal unicast packet, and the latest packet goes to the standby queue.

We also implemented a prototype of a traditional forwarding algorithm. We compare both implementations to measure the performance gains achieved by network coding when A sends 10,000 packets of 1,400 bytes each to B and vice-versa. Without network coding, the amount of data transmitted was 54 MB on both links. With network coding, only 44 MB were sent. With this code as a starting point, network coding protocol designers can test and tune their algorithms on real platforms under real conditions.

6.3 Example 3: Topology Control

Topology control algorithms require updated information about neighbors. Selecting good neighbors is often beneficial for the whole network. Prawn supports varied neighbors selection criteria relying on cross-layer information. For instance, in order to save energy and reduce interference, neighbors with lowest

```
require "prawn.pl";
$Neighbor = Prawn_Neighbors();
for ($i=1; $i<=$Neighbor[0]; $i++){
    push(@rx_power_list, [$i,$Neighbor[$i]{MAX_POW}]);
}
@sorted_list = sort {($b)->[1]<=>($a)->[1]} @rx_power_list;
@Selected_Neighbors=@sorted_list[0..1];
```

Fig. 8: Perl code of a topology control prototype.

required transmit power can be selected. Conversely, neighbors with the highest signal strength received could be chosen. Many recent research efforts relying on cross-layer approaches would benefit from Prawn's lower layer information.

The code in Figure 8 shows how to get in 7 lines a list of neighbors sorted according to their receive signal strength. This code is running successfully on our testbed consisting of heterogeneous nodes. An important point here is that the received signal strength value retrieved from the wireless driver can be different depending on the wireless device model. If the neighbors do not have all the same wireless cards, the selection could be biased. This is an example of practical issue that cannot be taken into account from simulations. Using Prawn, designers can evaluate their proposal taking into account the features and performance of off-the-shelf hardware and drivers.

7 Summary and outlook

In this paper we proposed Prawn, a novel prototyping tool for high-level network protocols and applications. Prawn's main goal is to facilitate the prototyping of wireless protocols so that prototyping becomes an integral part of the design process of wireless systems.

Prawn is not an alternative to simulation or any other evaluation method. Instead, it stands as a complementary approach that goes beyond simulation by taking into account real-world properties. Prawn surfs the wave of recent research efforts toward making implementation easier (e.g., Click and XORP), but as a preliminary phase in this process. The designer has to keep in mind, however, that the performance of a prototype does not always match exactly with the performance of a final and optimized implementation. Specifically, Prawn performs operations that may not appear in the final implementation (e.g., beacons, feedbacks, data encapsulation). However, first measures of this overhead incurred by Prawn are encouraging. For instance, using a Pentium M 733 (1.1 GHz) laptop, the additional delay to send a packet when using Prawn is only 140 μs (averaged over 10,000 packets), whereas the bandwidth loss is around 1.8 percent.

Through several case studies, we showcased the use of Prawn in the context of a wide range of network protocols. But the possibilities of Prawn are not restricted to the examples given in this paper. Other experiments where Prawn can

be useful include: evaluating existing protocols for wired networks in the wireless context, implementing new routing protocols, testing overlay approaches in wireless multi-hop networks, evaluating distributed security algorithms, testing new naming mechanisms over IP, testing incentive mechanisms for communities, implementing localization algorithms, measuring wireless connectivity in both indoor and outdoor scenarios, evaluating peer-to-peer algorithms, testing opportunistic forwarding mechanisms.

We hope our work will provide a starting point for an improved design methodology as prototyping provides both easy and accurate evaluation of wireless protocols and services under real conditions. This paper has demonstrated that this is feasible – Prawn is a fully functional tool that responds to the needs of early protocol evaluation. Finally, we expect that Prawn's simplicity will allow researchers to adopt it. To help this becoming true, ongoing work includes adding new prototyping facilities (TCP data packets, automatic update of the OS routing table, more physical values retrieved from the driver, etc) and porting Prawn to other operating systems such as FreeBSD and Microsoft Windows.

References

1. D. Raychaudhuri, M. Ott, and I. Seskar, "Orbit radio grid tested for evaluation of next-generation wireless network protocols." in *Proceedings of Tridentcom*, 2005, pp. 308–309.
2. MIT Computer Science and Artificial Intelligence Laboratory (CSAIL). MIT roofnet. [Online]. Available: http://pdos.csail.mit.edu/roofnet/doku.php
3. P. De, A. Raniwala, R. Krishnan, K. Tatavarthi, J. Modi, N. A. Syed, S. Sharma, and T. cker Chiueh, "Mint-m: an autonomous mobile wireless experimentation platform." in *Proceedings of ACM/USENIX Mobisys*, 2006, pp. 124–137.
4. E. Kohler, R. Morris, B. Chen, J. Jannotti, and M. F. Kaashoek, "The Click modular router," *ACM Trans. Comput. Syst.*, vol. 18, no. 3, pp. 263–297, 2000.
5. M. Handley, O. Hodson, and E. Kohler, "XORP: an open platform for network research," *Computer Communications Review*, vol. 33, no. 1, pp. 53–57, 2003.
6. The Network Simulator NS-2. [Online]. Available: http://www.isi.edu/nsnam/ns/
7. X. Zeng, R. Bagrodia, and M. Gerla, "GloMoSim: a library for parallel simulation of large-scale wireless networks," in *Proceedings of Workshop on Parallel and Distributed Simulation*, Banff, Canada, May 1998.
8. OPNET Modeler. [Online]. Available: http://www.opnet.com/products/modeler/
9. P. Zheng and L. Ni, "EMPOWER: A network emulator for wireless and wireline networks," in *Proceedings of IEEE Infocom*, San Francisco, CA, Apr. 2003.
10. M. Kojo, A. Gurtov, J. Manner, P. Sarolahti, T. O. Alanko, and K. E. E. Raatikainen, "Seawind: a wireless network emulator," in *GI/ITG Conference on Measuring, Modelling and Evaluation of Computer and Communication Systems*, Aachen, Germany, Sep. 2001.
11. K. Ramachandran, S. Kaul, S. Mathur, M. Gruteser, and I. Seskar, "Towards large-scale mobile network emulation through spatial switching on a wireless grid," in *Proceedings of ACM Sigcomm*, Philadelphia, PA, Aug. 2005.
12. Wireless Tools for Linux. [Online]. Available: http://hpl.hp.com/personal/Jean_Tourrilhes/ Linux/Tools.html

13. D. M. Blough, M. Leoncini, G. Resta, and P. Santi, "The k-neighbors approach to interference bounded and symmetric topology control in ad hoc networks," *IEEE Transactions on Mobile Computing*, vol. 5, pp. 1267–1282, Sep. 2006.
14. N. Li and J. Hou, "Localized topology control algorithms for heterogeneous wireless networks," *IEEE/ACM Transactions on Networking*, vol. 13, pp. 1313–1324, Dec. 2005.
15. D. Cavin, Y. Sasson, and A. Schiper, "On the accuracy of manet simulators," in *In proceedings of POMC'02*, Toulouse, France, Oct. 2002.
16. S. Katti, H. Rahul, W. Hu, D. Katabi, M. Médard, and J. Crowcroft, "XORs in the air: practical wireless network coding," in *Proceedings of ACM Sigcomm*, Pisa, Italy, 2006.