Transparent Network Protocol Testing and Evaluation

Xiaoshuang Wang, Sunil Agham, Vikram Munishwar, Vaibhav Nipunage, Shailendra Singh, and Kartik Gopalan
Computer Science, Binghamton University, Binghamton, NY, USA
Contact: kartik@binghamton.edu

***Abstract*—Network protocol developers typically go through a tedious and error-prone process of testing and debugging their protocol implementation for various settings. They perform a number of tasks manually such as configuration of numerous network settings, controlled reproduction of unexpected protocol behavior, and traffic capture and analysis. We present a Protocol Testing and Evaluation System (PTES) to assist developers in transparently testing their protocol implementations. PTES enables a protocol developer to construct and execute various controlled and repeatable testing scenarios. The developer can use simple iptables-like rules to specify various local and distributed network events and actions. During protocol execution, PTES triggers these events and actions in a coordinated manner and records the protocol responses to these events which can later be examined by the developer. We present the design and implementation of three variants of PTES for native, simulated, and emulated platforms for both wired and wireless networks. We demonstrate the utility of PTES by automating the testing of TCP/IP and Optimized Link State Routing (OLSR) protocols.**

I. Introduction

Testing and debugging of network protocol implementations is typically a manual process in which developers test the protocol behavior against various network conditions and configurations. Event-injection is a technique to perform controlled testing of a protocol implementation. Events are injected into the network during protocol execution, either externally or by manually instrumenting the protocol code, and the protocol response to the events is recorded.

A number of challenges make event-based protocol testing and evaluation difficult to execute. First, manual instrumentation of protocol code can make the testing process harder. Second, it is challenging to reproduce and record protocol responses for distributed network events. For instance, in case of wireless protocols such as AODV [1] and OLSR [2], an event on one node can affect the routing decision on another node. Reproducing such distributed dependencies across nodes requires careful co-ordination among multiple nodes, which may be difficult to achieve through manual instrumentation alone. Third, when protocols need to be tested for large deployment scenarios, developers often rely on simulation or emulation for quick testing in order to save on deployment and configuration time. However, again the event-injection and testing process is performed largely manually on a protocol-by-protocol basis. Finally, to evaluate protocol responses to network events, the developer has to again manually instrument the protocol code to capture packet traces at individual nodes.

Existing tools for protocol testing and evaluation that address some of the above concerns tend to be either protocol-specific, testbed-specific or difficult to use due to unfamiliar interfaces that developers need to learn from scratch. In this paper, we present the design and implementation of a Protocol Testing and Evaluation System (PTES) to assist protocol developers in testing their protocol implementation under various network event scenarios. The protocol implementation under test does not need to be modified or instrumented for the sake of testing. Test cases can be specified and executed independent of the protocol implementation.

To assist developers in easily specifying test cases, PTES provides a few basic building blocks. Developers can specify controlled injection of events on various nodes and also specify the corresponding actions to execute in response to those events. PTES provides a familiar `iptables`-like interface [3] for protocol developers to specify Event-Action rules. For example, an event could be receiving certain number of packets of a particular protocol type and an action could be delaying, modifying, or duplicating a packet or rebooting a particular node. PTES also allows distributed Event-Action specification in which an event is monitored on one or more nodes and the corresponding action is executed on a different node. The dependencies and timings among events and actions are transparently managed by PTES during protocol execution. To help developers evaluate the behavior of a protocol in response to a specific event, PTES provides a mechanism to collect an aggregated log of events and actions for analysis.

PTES can work in both native mode on physical machines running Linux and in simulated or emulated mode in the ns-3 [4] platform. The ns-3 platform is a well-known tool for network simulations and emulations. Simulations allow for experiments with a large number of nodes, whereas emulations introduce a certain degree of realism in experiments through the use of real network hardware for packet transmission and reception. Thus developers can use PTES to either test real protocol implementations in the native mode or add PTES hooks to their existing ns-3 scripts, without modifying the underlying ns-3 protocol implementation. It is up to the developers/testers to assess the relative merits of using native implementations vs. simulations/emulations. The role of PTES is primarily to provide a convenient testing framework for each of the environments.

To summarize, our key contributions are as follows. (1) We present the design of a transparent and non-intrusive approach for testing network protocols that is independent of protocol implementations, and works in real, emulated, and simulated platforms. (2) We present a distributed Event-Action specification framework using the familiar `iptables` interface. (3) We present two implementations of the proposed design in native Linux and ns-3 based platforms.

II. High-level Architecture of PTES

PTES enables protocol developers execute test scenarios without modifying their protocol implementation details.

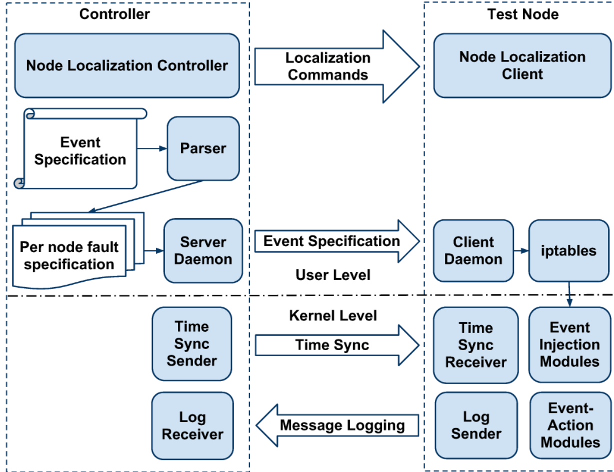

Fig. 1: Components of PTES.

Developers can use a familiar `iptables`-like interface to specify various test configurations. Each configuration consists of a set of rules. Each rule specifies an event to monitor and a corresponding action to execute in response to the event. Once specified, a test configuration is executed in parallel with the protocol execution. PTES records a log of events, actions, and protocol responses during the execution for offline analysis. In this section, we will provide an overview of this Event-Action framework.

A. Event-Action Architecture

Figure 1 shows the high-level architecture of the PTES environment consisting of test nodes and a controller. The controller initiates the experiment by accepting an Event-Action specification from the user. Depending on the specification, the controller assigns each node with the role of either an event node, an action node, or both. When an event occurs, the event nodes inform the corresponding action nodes, which then execute the specified action. This Event-Action coordination could be either local (i.e. the event is the action node) or distributed (i.e. an event could trigger actions on one or more remote action nodes). A kernel module at each node, called the *Event-Action module*, is responsible for coordinating the events and actions between different nodes. To trigger remote actions, the Event-Action module sends an *action message* over the control interface to its peer on the remote node which in turn executes the corresponding action. Protocol testers can also build an Event-Action chain so that a cascade of events and actions can trigger each other in a sequence.

B. Event-Action Specification

Test configurations in PTES are specified in a format similar to the widely used `iptables` rules. Distributed events and actions in PTES are specified as extensions to the existing `iptables` rules. A PTES-specific parser converts each of the extended rules into multiple regular rules that can be executed locally by the `iptables` framework at each node. For example, the following rule specifies that, when event nodes, or `enodes`, 3 and/or 7 receive a TCP packet from node 1, then all action nodes, or `anodes`, should reboot.

```
   iptables -p tcp -s 1 --enodes 3,7
--anodes * -j RESTART
```

The parser converts the above rule into the following per-node rules for event nodes 3 and 7.

```
   iptables -p tcp -s 192.168.1.1 -j
COR --action restart --mac-source
ff:ff:ff:ff:ff:ff
```

In the above conversion, the parser replaces the node IDs with the IP addresses of the corresponding nodes by referring to a mapping table. It also replaces the argument `--anodes *` with the per-node argument `--mac-source` followed by a broadcast address. The additional argument `-j COR` instructs the Event-Action kernel module at nodes 3 and 7 to coordinate with the remote action nodes by sending a RESTART message over the control interface.

C. Time Synchronization

To coordinate effectively, it is critical that all the nodes in PTES maintain the same notion of physical time, preferably at microseconds granularity, to enable meaningful synchronization of distributed events. A kernel module in the controller broadcasts time-sync packets periodically (every 200ms in our current prototype) over its control interface. Each synch packet carries the global time in microseconds since the start of the experiment. In between the arrival of two synchronization packets, each node uses its local clock (at microsecond granularity) to track time more accurately.

III. PTES Modules

PTES modules are protocol-testing components that implement the functionalities of individual events and actions. These modules are implemented as target and match extensions in the `iptables` implementation in the Linux kernel and ns-3. The target modules are used to alter the flow or the contents of network packets or the state of the system. Match modules can be used to decide if a network packet matches an `iptables` rule. Incoming and outgoing packets at each node are intercepted using `Netfilter` hooks, following which one or more modules may examine or process the packet in some manner. Modules currently supported in PTES can be classified as: *1)* Selection modules, *2)* Action modules and, *3)* Logging modules.

A. Selection Modules

In many situations, the simple matching capabilities of `iptables` are not sufficient, such as when one wishes to capture protocol traffic between specific times or would like to match every Nth packet of a connection. Selection modules extend the existing matching capabilities of `iptables` using time range and packet counts. *1) TIME selection module* captures network packets between a start-time and an end-time and sends them to a target modules for further processing. *2) COUNTER selection module* keeps track of the number of packets received and invokes a target module once N packets have been received, where N is specified as a parameter.

B. Action Modules

Action modules allow the injection of various actions on different nodes in response to network events. PTES currently supports the following actions. *1) DELAY module* delays the delivery of an incoming packet by a user-specified time interval. The intercepted packets wait in a kernel queue for

the delay duration after which they are re-injected into to the local protocol stack for further processing. *2) MODIFY module* takes a list of `offset:value` pairs from the user. The value at the specified offset in each intercepted packet is replaced with the provided value. *3) RE-ORDER module* changes the order of packets sent from the node. Module takes a minimum and maximum delay value as inputs. For each intercepted packet, it delays the packet by a random delay value between the minimum and the maximum (by re-using the DELAY module) and then re-inserts packet into the network stack. *4) DUPLICATE module* inserts a duplicate copy of a packet at PREROUTING, POSTROUTING or FORWARD stages of `Netfilter` processing. Duplicate packets are not re-duplicated to prevent infinite loops. *5) REBOOT module* simulates a node crash by rebooting a node, either immediately, or after a specified delay. *6) DROP module* drops matching packets. *7) BROADCAST_STORM module* floods the network to create network congestion.

C. Traffic Logging Modules

Logging modules record packet information for matching packets. The *LOG module* records information from each intercepted packet such as, protocol type, source MAC, destination MAC, IP protocol type, and source/destination IP. The *BANDWIDTH module* calculates and record the bandwidth usage for matching packets in a specified interval. Logs are recorded over the control interface using either *NFS-assisted logging* or *real-time logging*. NFS-assisted logging at each node uses an NFS-mounted directory from the controller to record the system logs (syslog). To reduce the overheads introduced by NFS, PTES also provides a real-time logging mechanism in which a custom light-weight kernel-level protocol between the controller and the nodes transfers the log information to the controller. A logging module at the controller collects and stores all the log data from each node.

IV. Execution Modes of PTES

PTES can be used on various platforms to perform protocol evaluations. Protocol developers can test either wired or wireless network protocols using *1)* native Linux platform on multiple physical machines (also called *native* mode), *2)* ns-3 in simulation mode on a single physical machine, or *3)* ns-3 in emulation mode using multiple physical machines. In addition, PTES can also work in a *hybrid* setup with mix of emulated and simulated ns-3 nodes. It is up to the protocol developers/testers to decide which of the modes will be most accurate for testing their specific protocol implementation. The role played by PTES is mainly to provide a convenient testing framework for each of the above modes.

A. PTES in Native Mode

PTES can work in native mode on physical machines running Linux using the `Netfilter/iptables` framework. The implementation of PTES in native mode is illustrated in Figure 2. `Netfilter` matches the incoming, outgoing, and forwarded packets in the Linux kernel against a set of user-specified rules. PTES modules are loaded into the kernel at each node before the experiment. Nodes in PTES receive per-node Event-Action specifications from the central controller in the form of per-node `iptables` rules that are generated after parsing the user-specified PTES rules. After the PTES

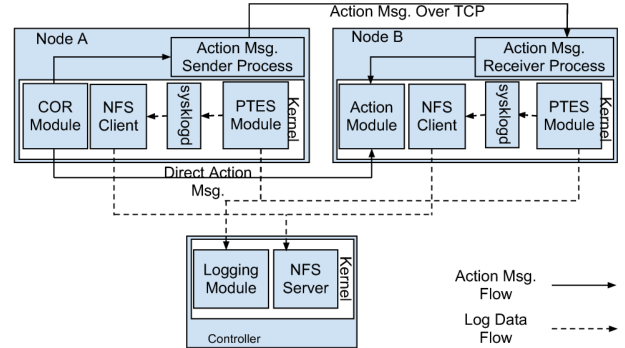

Fig. 2: PTES in Native Mode

injects the per-node `iptables` rules into the kernel, the Event-Action module starts monitoring the network packets for the specified events and triggers the corresponding actions when the rules match.

A COR module (or coordination module) is responsible for the coordination of distributed events and actions. To trigger an action at a remote host in response to an event, the COR module directly sends an Action message (as an Ethernet frame with special protocol field) containing the corresponding action information. The Action messages are acknowledged by the receiver to reduce their likelihood of being lost in transit. When the remote node receives an Action message, it triggers the module that implements the corresponding action. A Logging Module on each node communicates with a corresponding module on the controller and transmits log messages using a low-overhead communication protocol. Alternatively, log events can also be sent to the local *sysklogd* daemon which saves the data to NFS-mounted directories from the controller.

B. PTES on ns-3

We also added support for PTES in ns-3 simulations and distributed emulations. To do so, we first needed to add support in ns-3 for an `Netfilter/iptables`-like packet filtering mechanism. An existing tool, called ns-3-netfilter [5], provides a basic support by adding hooks at level 3 layer of TCP/IP stack in ns-3 by modifying the Ipv4L3Protocol class. However, the extensions provided by the ns-3-netfilter framework are limited to accepting and dropping an incoming packet. To fully integrate PTES with ns-3, we extended ns-3 by adding support for PTES modules at different chains. Specifically, changes and enhancements done to ns-3 and ns-3-netfilter are as follows. *1)* We ported ns-3-netfilter code for the latest ns-3 version. *2)* We extended ns-3-netfilter to accept and parse more complex `iptables` rules, including PTES event-action rules. *3)* We implemented and integrated various PTES modules in the ns-3-netfilter framework. *4)* We implemented a distributed ns-3 Event-Action framework among ns-3 nodes. Our modification to ns-3 adds up to around 1,600 lines of C++ code. The ns-3 Event-Action framework, referred hereafter as ns-3 `Netfilter/iptables` framework, includes both *local* and *distributed* event-injection. We now describe the two modes of PTES in ns-3 – simulation and distributed emulation.

1) **PTES in ns-3 Simulation Mode:** PTES in ns-3 simulation mode monitors every incoming and outgoing packet. For the packet that matches a particular `iptables` rule, the

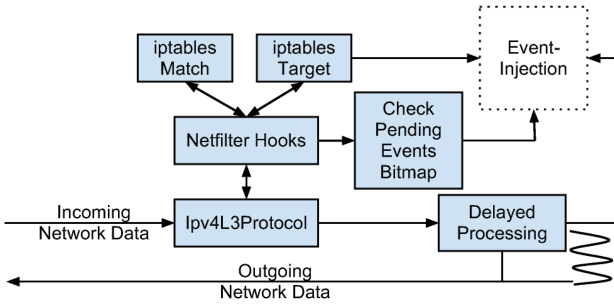

Fig. 3: PTES in ns-3 Simulation Mode

target specified in the iptables rule is launched and the event specified in the rule is processed.

The implementation of PTES in ns-3 simulation mode is illustrated in Figure 3. The ns-3 Netfilter/iptables framework has all the hooks inside the ns-3 class Ipv4L3Protocol, which does most of the Network Layer protocol processing. A new class called Ipv4Netfilter, implementing packet filtering hook is added to class Ipv4L3Protocol.All the five Netfilter hooks, namely PREROUTING, FORWARD, INPUT, OUTPUT, POSTROUTING, are inserted at corresponding places in the network protocol stack. Whenever a packet passes through a Netfilter hook, the Ipv4Netfilter class performs Match() operation on the packet. The Match() operation compares the packet with standard options, such as protocol type, source IP, destination IP, source port, destination port, etc. If a packet matches all the filters in a rule, the Target() operation in the specified target class is executed, which injects the corresponding events and actions into the simulated node.

To trigger actions such as packet delays or packet reordering, we implemented a support for *delayed processing*. Each packet that needs delayed processing is marked by the Netfilter/iptables target and is removed from protocol stack into a sorted queue ordered by packet delay expiration time. A dedicated thread periodically checks if there is any packet ready to be inserted back into the protocol stack.

2) **PTES in ns-3 Distributed Emulation Mode:** PTES in ns-3 distributed emulation mode enables injecting an event on an emulated ns-3 node located on another physical machine. To implement the Event-Action framework in ns-3 distributed emulation, we exploit the emulated node feature and MPI support of ns-3. The EmuNetDevice class from ns-3 allows one or more simulated nodes in ns-3 to transmit and receive real network packets using real network interfaces. MPI helps synchronize parts of an experiments running on various physical machines. MPI is also used for sending Action messages.

Figure 4 shows the implementation of the PTES in ns-3 distributed emulation mode. Each node in the testbed can be identified by a tuple (MID, NID) which consists of a Machine ID (MID), to represent a physical machine and a Node ID (NID) to identify a unique ns-3 node on the given physical machine. When a packet matches an iptables rule on a particular ns-3 node and a remote action needs to be triggered, the coordinator target (COR) sets up an Action message with required information such as MID, NID, action information, remote iptables rules, etc. The COR target then sends the Action message to the destination using MPI [6]. The MPI message arrives at the controller first and gets forwarded to the destination testbed machine.

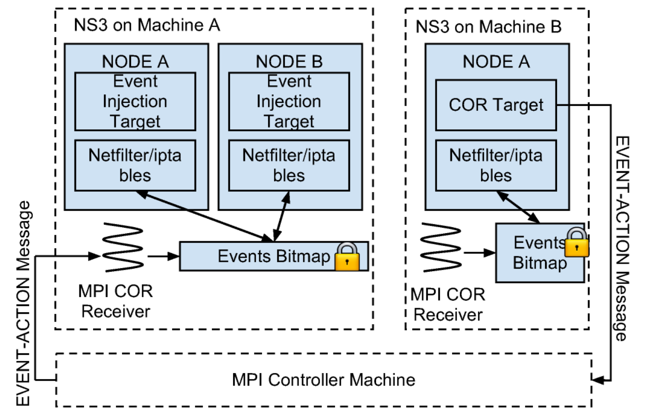

Fig. 4: PTES in ns-3 Distributed Emulation Mode

Each testbed node runs an MPI receiver, which is responsible for receiving all the Action messages from the remote source and injecting the corresponding actions on local ns-3 nodes. MPI receiver maintains a bitmap of events for each node. Once an Action message is received, the MPI receiver thread updates the bitmap. Each time when a Netfilter hook executes, before matching the rules, the Netfilter hook checks the bitmap and executes the pending actions first. In addition, the ns-3 Netfilter/iptables framework periodically checks if any one of the actions in the bitmap is set. If so, the specified action is injected into the experiment. If the Action message contains an iptables rule, the Netfilter framework inserts the new rule to the existing chain.

V. Evaluation

PTES works in different modes – simulation, emulation and native – and with both wired and wireless networks. For wireless network experiments, we use the MiNT-2 testbed [7], where a wireless node is mounted on a iRobot Create robot [8]. Each wireless node is equipped with a Soekris net5501 board, 8G flash card, one miniPCI wireless card, and one wireless USB adapter. The miniPCI wireless card is used for experiments, whereas the wireless USB adapter is used for communicating with the controller and transmitting action messages. Based on the test scenario specification, robots form the desired network topology, on which wireless protocols can be tested. For wired Ethernet network experiments, we run our experiments on machines connected over 1Gbps Ethernet network. We use commonly used OLSR [2] and TCP/IP [9] protocols for evaluation. Figure 5 shows the topology of wireless nodes used for the sections V-A, V-B, V-C, V-D. The solid lines connect the nodes that can communicate with each other. We arrange the nodes so that node 1 can only communicate with node 3 through intermediate nodes 4 or 2 and vice-versa. Node 4 and 2 are configured with 18dB and 12dB transmission power respectively. Thus, when communicating from the source node 1 to the destination node 3, the OLSR protocol selects the node 4 over node 2 to route the packets due to higher signal strength(less packet loss) with node 4.

A. Event-Action Framework Validation

This experiment demonstrates the behavior of OLSR when a DROP event is injected at node 4. Specifically, a DROP action message is transmitted from node 2 to 4 after 80 seconds, which makes node 4 drop all the incoming packets

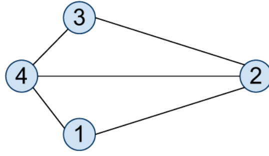

Fig. 5: Topology of the testbed for evaluation

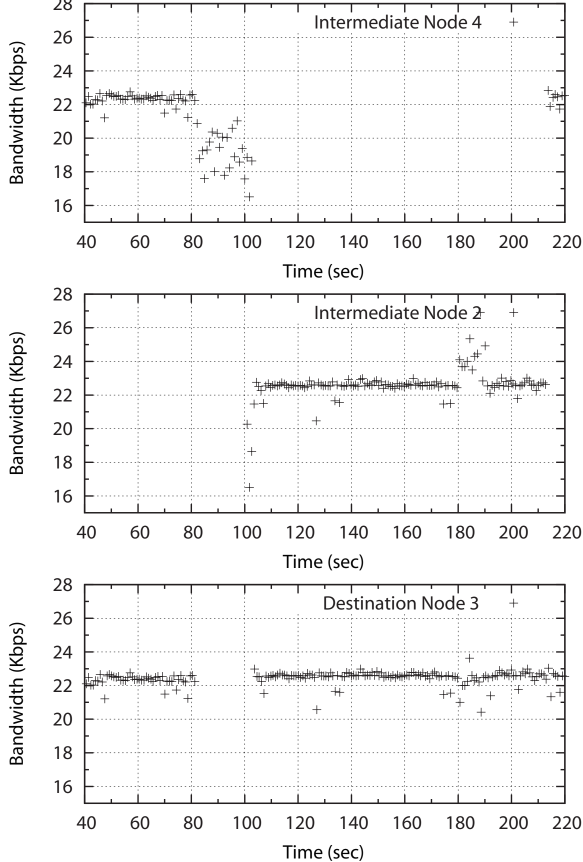

Fig. 6: Effect of DROP module on per-node reception rate

from the experiment interface. After 180 seconds, node 2 sends ACCEPT action message to node 4 so that node 4 can receive network packets from the experiment interface again. The BANDWIDTH module on node 2, 3, and 4 measures incoming bandwidth from node 1 to node 3. In Figure 6, the data points are the ICMP packets captured at the PREROUTING chain of the corresponding node. The first graph is the bandwidth log of node 4. When the DROP action message is received at node 4, it starts to drop all ICMP packets. As a result, node 3 does not receive any packets. OLSR takes some time to recalculate routes after which node 1 starts sending the packets through the intermediate node 2. The second graph shows the ICMP traffic on node 2 after 100 seconds. After 180 seconds, node 4 comes back again and starts forwarding the packets to node 3. However, node 4 takes some time to discover the neighbors. Once the link with node 4 becomes stable, node 1 prefers node 4 as the intermediate node over node 2 because node 4 is configured with a higher transmission power.

B. Effect of Delaying Packets

By default, when two nodes try to communicate with each other, OLSR selects the route with lesser hops. With the same number of hops, OLSR chooses the one with better network link quality. However, the OLSR implementation that

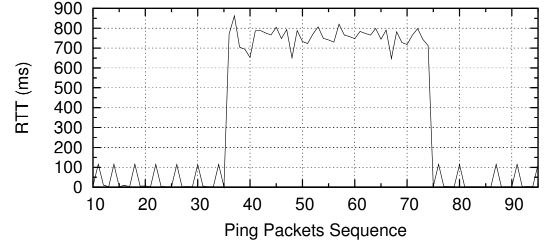

Fig. 7: Effect of DELAY module on round-trip time.

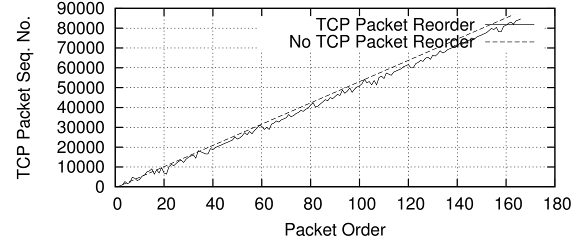

Fig. 8: TCP packet 600ms reorder window random reorder

we use does not consider the latency as a parameter to choose one route over another. To demonstrate this, we use DELAY module to delay the packets on node 4. We insert `iptables` rules on node 4 to enable packet delay of 400ms from time 80s to 120s during the experiment. Node 1 runs regular ping with an interval of 1 second. From figure 7 we can notice that the RTT of ping packets reaches to 800ms (twice the delay time), since each ICMP packet is delayed by 400ms. Despite packets being delayed at node 4, OLSR does not select node 2 as its new intermediate node.

C. Effect of Packet Reordering

The Reorder Module adds a random delay to the transmission time of each matching packet within a specified range (5ms to 60ms). Figure 8 demonstrates the effect of packet reordering on a TCP connection. The X-axis shows the packet reception order, and the Y-axis shows the sequence number of each received packet. The dashed line, which indicates that no packet reordering was used, is a straight line since the packets all arrive in the same order they were sent. The solid line represents actual order of sequence numbers received when the Reorder Module is used. It can be observed that with reordering, some packets with larger TCP sequence numbers are received sooner than those with smaller sequence numbers.

D. Effect of Packet Modification

The OLSR specification [10] mentions that if the *length* field of an OLSR message is invalid, the packet shall be discarded. To verify this behavior of OLSR protocol, we use the MODIFY module to change the length of the packet. With the MODIFY module enabled on node 4 from 60 second to 180 second, the OLSR length field of all incoming UDP packets is overwritten with an invalid value. Therefore, OLSR discards these packets assuming that the data inside the packet is corrupt. The Figure 9 shows the result of this experiment for intermediate nodes 4 and 2, and the destination node 3. We can observe that 60 seconds onward, all the incoming OLSR packets at node 4 are modified at the PREROUTING chain. Since node 4 does not receive any valid OLSR packet for a certain duration of time, corresponding routing table entry expires. Finally node 1 changes its intermediate node from

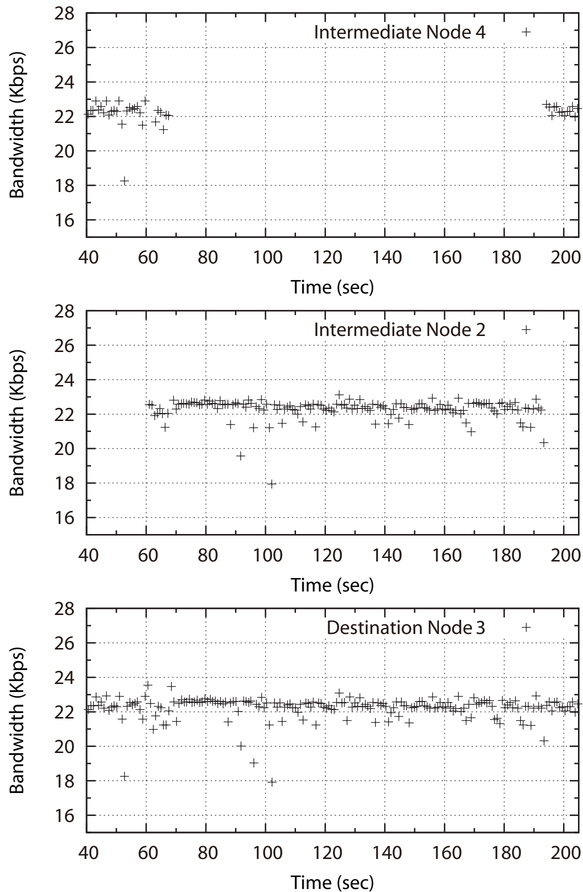

Fig. 9: Effect of MODIFY module on per-node reception rate

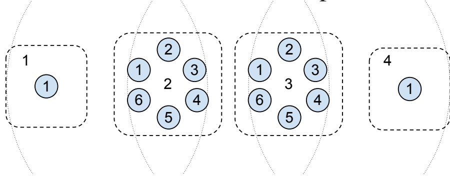

Fig. 10: Topology for evaluating PTES with ns-3.

node 4 to node 2. After disabling the MODIFY module, the node 4 is again selected over node 2 as an intermediate node.

E. PTES with ns-3

This experiment demonstrates the integration of PTES with ns-3 including the ability to work with emulated and simulated nodes. We have four physical machines. Physical machines 1 and 4 each have single emulated ns-3 node. Physical machines 2 and 3 each have 6 emulated ns-3 nodes. For rest of this experiment, node refers to emulated node on a physical machine. Figure 10 illustrates the topology. Numbers in circles show the per node id for emulated nodes. Dotted arcs depicts range of each physical machine. Each physical machine can directly communicate with only its immediate neighbor. Out of the 6 nodes on machine 2, only one is active at a given time, starting with node 1. After an interval of 100s, this node becomes inactive and next node on this machine becomes active. This is automated by setting PTES event-action rules to activate or deactivate a node after a given duration. For reminder of this experiment, a node *b* on machine *a* is represented as *a:b*. Throughout the duration of experiment, node 1:1 transmits ICMP packets to node 4:1. But since machine 1 cannot reach

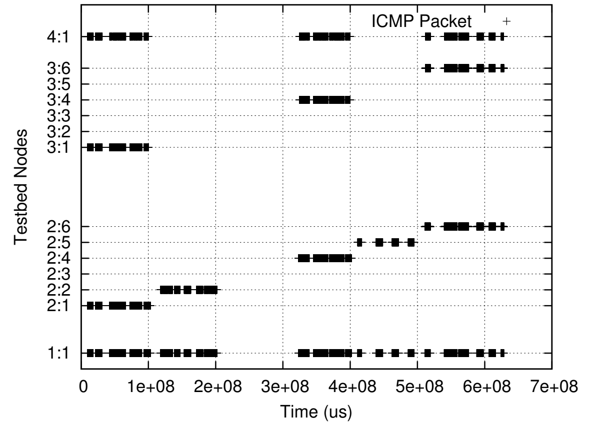

Fig. 11: PTES with ns-3: Observed ICMP packet route.

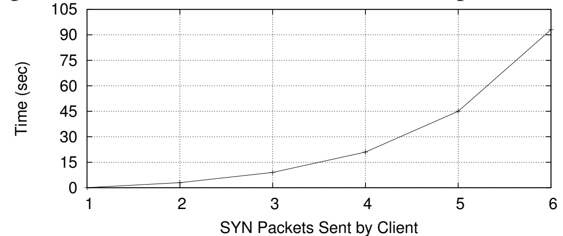

Fig. 12: TCP connection establishment - phase 1

machine 4, the packets hop through machine 2 and 3. During 0 to 100s duration, nodes 2:1 and 3:1 are active. Thus, the route during this period should be 1:1 to 2:1 to 3:1 to 4:1. The observed ICMP packet route is shown in Figure 11. Y-axis represents nodes. Node 1:1 has a total of 72 `iptables` rules. Each 100 second, node 1:1 sends 12 Event-Action messages to all the nodes on machine 2 and 3 so that only one node on each machine accepts network packets. We can see from the figure that in the very beginning, node 1:1 sends ICMP packets through node 2:1 and 3:1. After 100 seconds, the route changes as the nodes 2:1 and 3:1 go silent and 2:2 and 3:2 become active. However, the OLSR routing protocol does not find out the route change immediately. Thus, the node 1:1 sends ICMP packets to the destination only after 300 seconds, when the nodes 2:3 and 3:3 start accepting network packets.

F. TCP Connection Establishment

This experiment uses PTES to verify the TCP connection establishment protocol on a wired network. TCP uses a three-way handshake for connection establishment. In first phase, client sends SYN packet to server. In second phase, server replies with SYN-ACK. Finally, client sends ACK to server and connection is established. If client does not receive SYN-ACK packet, it sends SYN packet again with exponentially increasing intervals. Node 1 acts as a client and tries to establish TCP connection with node 2. A DROP rule drops the incoming SYN-ACK packets at node 1. Figure 12 shows that, since node 1 assumes loss of SYN-ACK packet, it resends successive SYN packets at exponentially increasingly intervals.

G. Effect on TCP Congestion Window Size

This experiment demonstrates the impact of packet loss, caused by the DROP module, on TCP congestion window size. A sender transmits packets to the destination at 1 Mbps for first 50 seconds. The DROP module randomly drops TCP packets between 10 and 22 seconds. Figure 13 shows the change

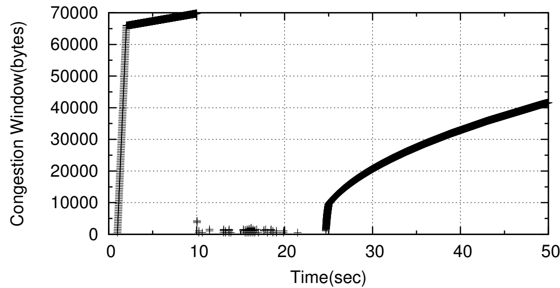

Fig. 13: TCP congestion window with random drop

in TCP congestion window size. TCP congestion avoidance and slow-start maintains two variables for each connection, congestion window size (*cwnd*) and slow-start threshold size (*ssthresh*). The initial value of *cwnd* is small but with each successful transmission it increases by one TCP maximum segment size. Since the TCP packets are never lost for the first 10 seconds, the *cwnd* increases rapidly. After 10 seconds, nearly one third of the TCP packets are dropped randomly by the `Netfilter` framework. As a result, the congestion avoidance algorithm makes *ssthresh* decrease to 2 and *cwnd* decrease to 1. When the random drop `Netfilter` rule expires after 50 seconds, no TCP packet drops are observed. The *cwnd* and *ssthresh* slowly increase back to the old value.

H. Logging Overhead

To measure the overhead of message logging in PTES, we performed periodic bandwidth monitoring at node 4 using a PTES iptables rule as it receives packets from nodes 1 and 3. Due to space constraints, we summarize our findings. As expected, the more frequently the measurement rule is triggered, the higher the CPU usage and network overhead. However, the CPU usage and network overhead are noticeably affected only when the measurement interval is smaller than 50ms and there is no observable overhead above 400ms.

VI. Related Work

Protocol testing has been studied in a number of contexts with focus on specific aspects of protocol behavior. DOCTOR [11] is a software-based tool that injects processor, memory, and communication related events in the HARTS real-time system. ORCHESTRA [12] is a fault probing and injection system that works by inserting a new layer into the protocol stack of the Mach and Solaris operating systems. NFTAPE [13] composes a configurable environment for experiments using existing event injectors. Virtualwire [14] is an event-injection tool that uses a custom declarative scripting language to specify events and actions across multiple nodes. FIAT [15] tests the dependability of real-time distributed systems by injecting events into the source code at the compile-time. NIST Net [16], Dummynet [17], and ComFIRM [18] also rely upon packet capture for event-injection but support neither distributed nor time-based injection capabilities. Cesium [19] is a simulation tool that allows events to be injected into the protocol at runtime. StarBED [20] provides a fault-injection mechanism in a large-scale testbed of 512 physical nodes each having up to 10 virtual nodes. Loki [21] is an fault-injection framework for distributed systems in which faults injected in one node can depend upon the states of the other nodes. Mendosus [22] is an SAN-based platform to test the reliability of network services in the presence of faults. In contrast to the above tools, that require a special GUI or a new programming language, PTES provides uses and familiar `iptables` interface for event-action specification, thus lowering the barriers to usable protocol testing. PTES is also protocol-transparent in that it does not require any protocol modifications and works over native, simulated, and emulated protocol testing platforms.

VII. Conclusion

We presented a Protocol Testing and Evaluation System (PTES) that can transparently test protocol implementations in native, simulated, or emulated settings for wired and wireless networks. PTES enables protocol developers to execute controlled and repeatable testing scenarios using familiar and simple `iptables`-like rules. A distributed Event-Action framework allows the specification and coordination of network-wide events and actions on different nodes, independent of protocol implementation and execution. PTES also records protocol behavior during testing for offline analysis.

Acknowledgment

This work is supported in part by the National Science Foundation through grant CNS-0751161.

References

[1] E. B.-R. C. Perkins and S. Das, "Ad hoc On-Demand Distance Vector (AODV) Routing Protocol," RFC 3561 (Experimental), Oct. 2003.

[2] "Optimized Link State Routing Protocol," (http://www.olsr.org).

[3] "Netfilter/iptables," (http://www.netfilter.org).

[4] "ns-3 Network Simulator," (http://www.nsnam.org/).

[5] "ns-3 Netfilter," (http://code.nsnam.org/adrian/ns-3-netfilter/).

[6] M. Snir, S. W. Otto, D. W. Walker, J. Dongarra, and S. Huss-Lederman, *MPI: The Complete Reference*. MIT Press, 1995.

[7] C. Mitchell, V. Munishwar, S. Singh, X. Wang, K. Gopalan, and N. Abu-Ghazaleh, "Testbed design and localization in mint-2: A miniaturized robotic platform for wireless protocol development and emulation," in *Proc. of COMSNETS*, 2009.

[8] "Roomba Create," (http://www.irobot.com/).

[9] T. Socolofsky and C. Kale, "TCP/IP tutorial," RFC 1180, 1991.

[10] T. Clausen and P. Jacquet, "Optimized Link State Routing Protocol (OLSR)," RFC 3626 (Experimental), Oct. 2003.

[11] S. Han, K. Shin, and H. Rosenberg, "Doctor: Integrated software fault injection environment for distributed real-time systems," in *IPDS*, 1995.

[12] S. Dawson, F. Jahanian, and T. Mitton, "Orchestra: A probing and fault injection environment for testing protocol implementations," in *Proc. of Computer Performance and Dependability Symposium*, Sept. 1996.

[13] D. Stott, B. Floering, D. Burke, Z. Kalbarczpk, and R. Iyer, "NFTAPE: a framework for assessing dependability in distributed systems with lightweight fault injectors," in *Proc. of IPDS*, 2000.

[14] P. De, A. Neogi, and T. cker Chiueh, "Virtualwire: A fault injection and analysis tool for network protocols," in *ICDCS*, 2003.

[15] Z. Segall, D. Vrsalovic, D. Siewiorek, D. Yaskin, J. Kownacki, J. Barton, R. Dancey, A. Robinson, and T. Lin, "FIAT: fault injection based automated testing environment," in *Fault-Tolerant Computing*, 1988.

[16] M. Carson and D. Santay, "Nist net: A linux-based network emulation tool," *Computer Commun. Review*, July 2003.

[17] L. Rizzo, "Dummynet: A simple approach to the evaluation of network protocols," *ACM Computer Communication Review*, January 1997.

[18] R. J. Drebes, G. Jacques-Silva, J. M. F. da Trindade, and T. S. Weber, "A kernel-based communication fault injector for dependability testing of distributed systems." in *Haifa Verification Conference*, Nov. 2005.

[19] G. A. Alvarez and F. Cristian, "Cesium: Testing hard real-time and dependability properties of distributed protocols," in *Object-Oriented Real-Time Dependable Systems*, 1997.

[20] T. Miyachi, Y. Makino, R. Beuran, S. Uda, S. Miwa, and Y. Tan, “Fault injection on a large-scale network testbed,” in *Proc. of Asian Internet Engineering Conference*, 2011.

[21] R. Chandra, R. Lefever, M. Cukier, and W. Sanders, “Loki: a state-driven fault injector for distributed systems,” in *Proc. of Dependable Systems and Networks*, 2000.

[22] X. Li, R. Martin, K. Nagaraja, T. D. Nguyen, and B. Zhang, “Mendosus: A SAN-based fault-injection testbed for construction of highly available network services,” in *Novel Uses of System Area Networks*, 2002.