Automated and Distributed Protocol Testing and Debugging for Wireless Ad Hoc Networks

Xiaoshuang Wang, Vaibhav Nipunage, Umesh Deshpande, and Kartik Gopalan
Computer Science, Binghamton University
Binghamton, NY, USA
{xwang2,vnipuna1,udeshpa1,kartik}@binghamton.edu

Categories and Subject Descriptors

D.2.5 [**Software Engineering**]: Testing and Debugging

General Terms

Design

Keywords

fault injection, fault analysis, iptables, network protocol

1. INTRODUCTION

Testing and debugging of new wireless network protocols is typically a tedious and error-prone process. Testing complex protocols requires manual configuration of numerous network testbed settings, controlled reproduction of anomalous network and protocol settings, manual traffic capture and careful analysis of protocol behavior during the test. The complexity of protocol testing grows with the length of protocol specifications and the size of the network under test. In this paper, we present the design and implementation of an automated protocol testing and analysis (APTA) tool to assist developers and protocol testers to test their protocol implementations. The central aim of our tool is to speed up the process of finding hard to reach situations where a protocol implementation may be incorrect. Our tool allows the protocol developer to specify faults using simple Netfilter/iptables-based [1] rules at run time and record the protocol responses against these faults so that the user can verify whether the protocol works as expected. Our protocol testing and analysis system allows the protocol developer to specify network events and actions in terms of distributed event-action pairs using iptables rules. For example, receiving certain number of packets of a particular protocol type can be considered as an event and delaying, modifying, or duplicating the packet or rebooting a particular node could be an action. Besides these primitive events and actions, our system also supports distributed event-action specification, which means that APTA can monitor an event on one or multiple nodes and execute the corresponding action on a different node. Verifying the behavior of a protocol in response to a specific event is also a tedious task. To make such analysis easier, we developed a mechanism that allows protocol responses to be logged and sent to a central controller for further merging and analysis. Since APTA uses Netfilter/iptables architecture, it does not need to modify the protocol implementation under test. As a result APTA can be used to test most of the network protocols on any platform as long as test nodes are running Linux. We demonstrate the utility of our tool by automating the testing of a real world wireless ad-hoc routing protocol called Optimized Link State Routing (OLSR).

The fault injection tools can be divided in two categories: hardware implemented fault injection and software implemented fault injection (HWIFI & SWIFI). MESSALINE [3] is one of the HWIFI tools that is used to inject hardware faults by manipulating the pin voltages in circuits. Since the SWIFI tools are more flexible, inexpensive, they are more popular compared to HWIFI tools. In SWIFI tools, DOCTOR [8] was developed to inject processor, memory, and communication related faults in real-time system but it was not sufficient for large distributed systems. ORCHESTRA [5] employs script-driven fault probing and fault injection by inserting a new layer into the protocol stack to test the distributed protocol. Virtualwire [6] is a protocol transparent fault injection tool that also uses a declarative scripting language to specify the faults. Further, NFTAPE [12] composes a configurable environment for experiments using existing fault injectors. However, in contrast to the aforementioned script based fault injection tools, APTA uses a simple fault specification technique using iptables rules. Moreover, unlike APTA, above systems mainly consist of stand-alone nodes, and don't provide more complex features such as propagation of an action from the one node to the other nodes in the system upon an occurrence of a desired event. FIAT [11] is one of the first SWIFI tool. It tests dependability in real-time distributed systems by injecting faults into the source code at the compile-time, whereas APTA provides more user-friendly, run-time fault injection capability. Other systems like NIST Net [4], Dummynet [10] and ComFIRM [7] also use a similar architecture but they all have limited fault injection features with neither distributed nor time based fault injection mechanism.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
MobiWac'11, October 31–November 4, 2011, Miami, Florida, USA.
Copyright 2011 ACM 978-1-4503-0901-1/11/10 ...$10.00.

2. DESIGN

Figure 1 shows the high level architecture of the APTA environment. In this design we have two communication in-

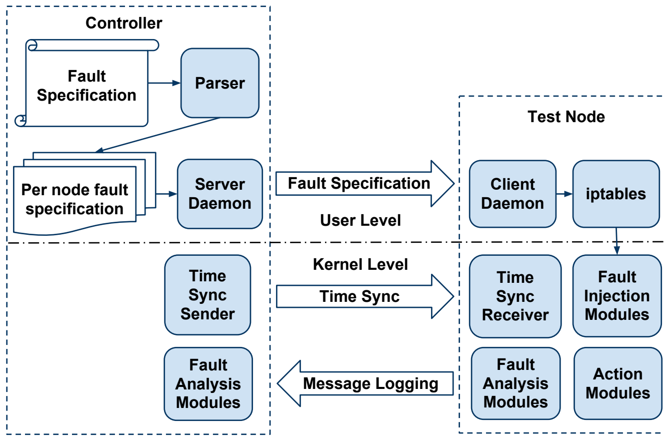

Figure 1: Architecture of the APTA environment.

terfaces. The experiment interface transmits the actual experiment data and the control interface communicates with the test nodes from the controller. The controller is responsible of initializing the experiment and controlling the time synchronization of the test environment. After the experiment terminates, the controller collects the data log from the individual test nodes. Role of each test node is to execute the fault specification rules and save the results for later analysis.

2.1 Processing & Inserting Fault Specification

Parser is a component on the controller that parses the fault specifications from the user. The fault specification is similar to the iptables rules. The difference is that we have our event-action pair extension to it so that each iptables rule can be converted into multiple rules on different test nodes. Following format is used to specify the faults.

iptables [table] [chain] [command] [match] [enodes] [anodes] [target/jump]

'enodes' are the event nodes and 'anodes' are the action nodes. When the user specifies these options, the parser generates an iptables rule for each corresponding event and action nodes. The event node sends an action packet to the action node, that is executed on the action node on reception.

Insertion of the following rule initiates the monitoring of an event of an incoming TCP packet from source node 1 on nodes 3 and 7. On reception, the receiving nodes send the reboot action packets to all the nodes.

iptables -p tcp -s 1 –enodes 3,7 –anodes * -j reboot

The parser creates a rule for each event node and stores it in their temporary rule file when it receives the event-action rule. It will also parse the node ID to its ip address. The daemon running on the controller sends these files to the respective test nodes. For enodes 3 and 7, the same rule are generated.

iptables -p tcp -s 192.168.1.1 -j COR –action reboot –mac-source 255:255:255:255

3. IMPLEMENTATION

3.1 APTA Modules

The APTA modules play a significant role in assisting the users to analyze the protocol behavior. The APTA modules consist of iptables target and match extensions. These modules can inject various kinds of faults into the network and record the faults or network packets information for further analysis.

One of the main feature of our APTA is to provide distributed fault injection functionality. To achieve this goal we have implemented an iptables co-ordination target module (COR), which sends a co-ordination packet with an action value to the specified node. Upon receiving the co-ordination type packets, receiver side module (COR_RECEIVE) executes the action contained in the co-ordination packet. To inject a co-ordinated fault, the user needs to specify the target node's address and the action type to the iptables module. The kernel module constructs a packet of our own custom protocol and fills the packet with an action message and sends it to the destination node.

There are two ways to receive an action message at the receiver side. First, the COR_RECEIVE module registers a packet handler for our custom protocol which is invoked on the arrival of such packets. In another approach, the module creates a proc entry to allow user-level process to send the action message to this module.

The DELAY module takes a delay time (in ms) as an argument from the user. This information is then used by DELAY module to delay the incoming packets. The packets intercepted by the Netfilter hook wait in a queue until their expiry time. The expiry time is calculated based on the arrival time of the packet and the delay interval. Delayed packets are re-injected to the local network stack for further processing when they reach the assigned expiry time.

The MODIFY module enables user to modify the packets on-the-fly. The user can change packet fields to observe the behavior of the protocol. For modifying the packets, the module takes a list of offset:value pairs from the user. When packet is intercepted in the hook, the value at the specified offset is replaced with the provided value.

To block unwanted packets for a particular period of time, we have a module called EXPERIMENT_TIME. It accepts incoming packets only if the current time is within the specified time boundaries.

The LOG module extracts information such as, protocol type, source MAC, destination MAC, IP protocol type, source/destination IP from each intercepted packet. This information is later transferred to the controller for analysis. The BANDWIDTH module saves the number of bytes received for specific time interval on a particular iptables rule. The BROADCAST_STORM module is used to flood the network with packets and test the protocol for packet congestion. The DUPLICATE module can be used to create duplicate packets and send those packets to the network and observe the behavior of the protocol. Further, we enable users to change the sequence of the delivery of packets with the REORDER module. Finally, the RESTART module allows node restart at specified time.

3.2 Action Message & Collecting Logs

Action message is a messages sent from the event node to the action node on occurrence of a particular event. There

are two ways to transmit the action message, Direct-Action-Message (DAM) and Action-Message-Over-TCP (AMoT).

The DAM is sent from event node COR module directly to the action node over a custom unreliable protocol. The protocol developers can use this protocol for time-sensitive experiments when a reliable control interface is available to them, otherwise AMoT can be used. In the AMoT, all action messages are transmitted through TCP connections from the event node to the action node. As we know the TCP protocol is reliable, all the action messages will arrive to the destination.

The controller collects the log data from all the test nodes. The log message can be saved in real-time or when the experiment has finished, which is referred to as Collecting Log Offline (CLO). By default, all the logs are saved locally. The controller is responsible for collecting all the logs after the experiment. The following two mechanisms are used for real-time logging, NFS Assisted Logging (NFSAL) and Real-time Logging (RTL). The NFSAL uses a NFS directory exported from the controller to write the system logs (syslog). All the logging messages are sent to the controller by the NFS. The RTL is also implemented using unreliable protocol to speed up the logging as well as to reduce the resource utilization at the node. The logging module from the controller side receives all the log data from each node, which is stored on its local storage.

4. EVALUATION

We build our experiment environment on MiNT-2 [9] testbed. Each node is equipped with a Soekris net5501 board, 8G Compact Flash card. Each node has one miniPCI wireless card and one wireless USB Adapter. The miniPCI wireless card is used for experiment interface. The wireless USB adapter is used for the control interface to communicate with the controller and to communicate action messages. Finally, we choose the OLSR [2] protocol as our test routing protocol. For the experiment we designed the topology such that node 1 can only communicate to node 3 through intermediate node 4 or 5 and vice-versa.

4.1 Event-action Evaluation

In this experiment, node 1 executes ping to node 3 with an interval of 0.03 second. A DROP action message is transmitted from node 5 to 4 after 80 seconds which makes node 4 drop all the incoming packets from the experiment interface. After 180 seconds, node 5 sends another ACCEPT action message to node 4 so that node 4 can receive network packets from the experiment interface again. We insert our BANDWIDTH module on node 3, 4 and 5 to monitor the incoming ICMP data flow from node 1 to node 3.

In Figure 2 the data points are the ICMP packets captured at the PREROUTING chain of the corresponding node. The X-axis represents the time in the experiment while the Y-axis is the ICMP protocol bandwidth calculated per 800 ms.

The first graph in Figure 2 is the bandwidth log data of node 4. We can see that when the DROP action message comes, node 4 starts to drop all the ICMP packets. As a result, node 3 doesn't receive any of the ping packets. The OLSR takes some time to re-discover the change in network and node 1 still keeps on sending packets through node 4. After the information about new route is updated by the OLSR on all the nodes, node 1 starts sending the packets through the intermediate node 5. From Figure 2 we can

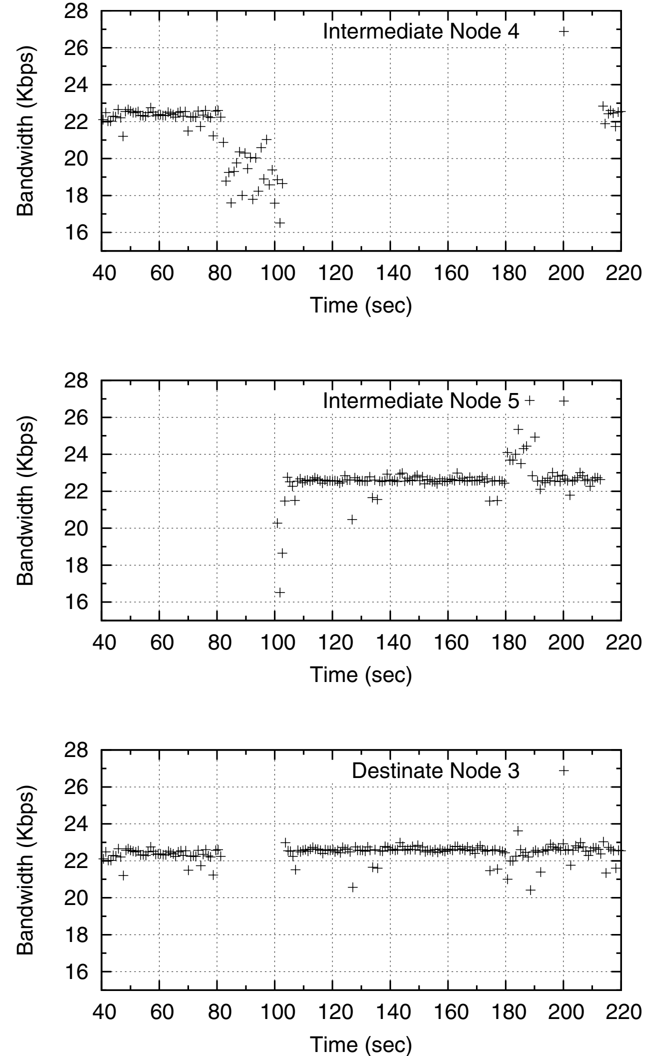

Figure 2: Event-action test case result.

see the ICMP traffic on node 5 after 100 second. After 180 second, node 4 comes back again and starts forwarding the packets to node 3. However, after coming up it takes some time discover the neighbors and become operative again. When the links between node 4 and other nodes become stable, Since node 4 is configured with higher transmission power, node 1 prefers node 4 as an intermediate node over node 5.

4.2 Packet Delay

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 we use, does not consider the latency as a parameter. To demonstrate this, we use DELAY module to delay the packets on node 4. We insert iptables rules to enable packet delay of 400ms from 80 to 120 second interval during the experiment. Node 1 runs regular ping with an interval of 1

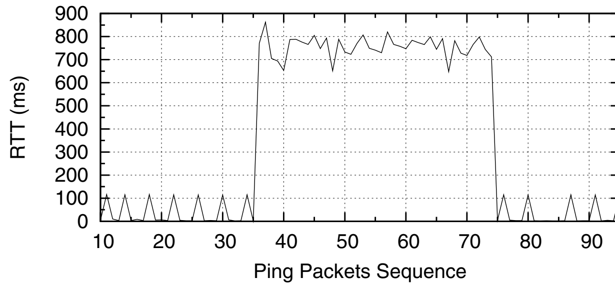

Figure 3: Packet delay experiment for ICMP from node 1 to node 3

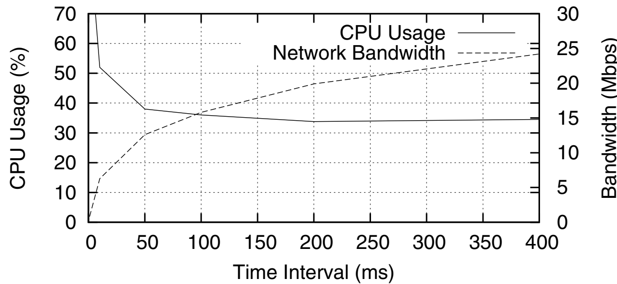

Figure 4: CPU usage and network bandwidth overhead.

second. From figure 3 we can notice that the RTT of the ping packets reaches to 800ms (twice of the delay time), because on each journey the ICMP packets gets delayed for 400ms so the round trip delay becomes 800ms. In spite of packets being delayed at node 4, OLSR does not select node 5 as its new intermediate node.

4.3 APTA Overhead

For this experiment, we use only two nodes 1 and 4. We perform the bandwidth monitoring on node 4, while it receives packets from node 1. The BANDWIDTH module is used to calculate and print the bandwidth for the specified interval. Each packet matching iptables rule increases the counter inside the BANDWIDTH module.

Because the code is re-entrant, we use spin lock to avoid racing condition. However, using spin lock incurs additional overhead, especially when it is frequently accessed. We measure the overhead of our implementation by varying the BANDWIDTH module calculation intervals.

From Figure 4 we can observe that the CPU usage decreases and bandwidth increases as the BANDWIDTH module interval changes from 1ms to 400ms. However, with the BANDWIDTH module the CPU performance is noticeably affected only when the interval is smaller than 50ms. Moreover, we realized that even if without any modules inserted, the bandwidth and CPU usage performance remains the same as when the bandwidth monitoring interval is 400ms. Therefore, the overhead the module depends on how frequent an user wants to log the information.

5. CONCLUSION

In this paper we present the design and implementation of our APTA tool using Netfilter/iptables framework. This tool can be used to test the different characteristics of the network protocols. Using extended iptables rule format and different modules we can inject wide variety of faults across the network while keeping the fault specification simple for the user. Our tool helps in speed-up the process of finding faults in the protocol implementation in comparison to the code instrumentation technique. It is also capable of testing most of the protocol without any modification in the protocol code. It will also help them to verify whether the protocol is working as per the specifications.

6. REFERENCES

[1] Netfilter/iptables http://www.netfilter.org.

[2] Optimized Link State Routing Protocol http://www.olsr.org.

[3] J. Arlat, M. Aguera, L. Amat, Y. Crouzet, J.-C. Fabre, J.-C. Laprie, E. Martins, and D. Powell. Fault injection for dependability validation: a methodology and some applications. *IEEE Transactions on Software Engineering*, February 1990.

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

[5] 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*, September 1996.

[6] P. De, A. Neogi, and T. cker Chiueh. Virtualwire: A fault injection and analysis tool for network protocols. In *Proc. of International Conference on Distributed Computing Systems*, May 2003.

[7] 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 S. Ur, E. Bin, and Y. Wolfsthal, editors, *Haifa Verification Conference*, November 2005.

[8] S. Han, K. Shin, and H. Rosenberg. Doctor: an integrated software fault injection environment for distributed real-time systems. In *Computer Performance and Dependability Symposium, 1995. Proceedings., International*, April 1995.

[9] V. Munishwar, S. Singh, C. Mitchell, X. Wang, K. Gopalan, and N. Abu-Ghazaleh. Rfid based localization for a miniaturized robotic platform for wireless protocols evaluation. In *Proc. of Conference on Pervasive Computing and Communications*, March 2009.

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

[11] 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 *Proc. of International Symposium on Fault-Tolerant Computing*, June 1988.

[12] 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 Computer Performance and Dependability Symposium*, March 2000.