VirtualWire: A Fault Injection and Analysis Tool for Network Protocols

Pradipta De Anindya Neogi Tzi-cker Chiueh
Experimental Computer Systems Laboratory
Computer Science Department
State University of New York at Stony Brook, Stony Brook, NY 11794-4400
{prade, neogi, chiueh}@cs.sunysb.edu

Abstract

The prevailing practice for testing protocol implementations is direct code instrumentation to trigger specific states in the code. This leaves very little scope for reuse of the test cases. In this paper, we present the design, implementation, and evaluation of **VirtualWire***, a network fault injection and analysis system designed to facilitate the process of testing network protocol implementations. VirtualWire injects user-specified network faults and matches network events against anticipated responses based on high-level specifications written in a declarative scripting language. With VirtualWire, testing requires no code instrumentation and fault specifications can be reused across versions of a protocol implementation. We illustrate the effectiveness of VirtualWire with examples drawn from testing Linux's TCP implementation and a real-time Ethernet protocol called Rether. In each case,* 10 *to* 20 *lines of script is sufficient to specify the test scenario. VirtualWire is completely transparent to the protocols under test, and additional overhead in protocol processing latency it introduces is below 10% of the normal.*

1. Introduction

Network protocols specify the rules of engagement among multiple parties over the network. Formal verification methods [1] can detect specific design flaws in protocol specifications. Simulation tools, like VINT and REAL [2], [3] can model the protocol behaviors under various system parameters and workload conditions. However, neither of the methods can be used satisfactorily to guarantee network protocol *implementation* correctness. The main challenge in testing a protocol implementation is to guarantee that the protocol behavior conforms to the specifications in the presence of any abnormal network condition generated due to drop, delay, duplication, reordering or modification (corruption) of a packet. In the absence of a protocol testing infrastructure, developers are forced to instrument the network protocol implementations to "simulate" the intended abnormalities, and analyze the resulting outputs against desired responses as specified by the protocol specifications. Direct code instrumentation is not only a slow process but also entails the danger of introducing or hiding subtle bugs. Worse yet, each new release of the same protocol often requires recreating the test cases afresh.

We experienced the difficulties and effort involved in testing a network protocol implementation, while verifying the implementation of *Rether* [4], [5], a software-based real time Ethernet protocol. Rether is a software-based token passing protocol that regulates network nodes' access to a shared bus or a wireless channel. There is a control token circulating among the network nodes, and a node on the shared medium is allowed to transmit data only when it is in possession of the control token. Since a node or link failure may result in the network having either no token or multiple tokens, Rether incorporates elaborate mechanisms to keep a single token in circulation in spite of packet drops and node failures. To verify the correctness of the Rether implementation, one needs to enumerate all possible combinations of node/link failures, and check Rether implementation's reactions under each of these failure scenarios. However, arriving at these failure scenarios requires modifications to the kernel, because Rether itself is implemented as a layer between the Ethernet driver and the IP stack. This means that testing each failure scenario requires kernel instrumentation and recompilation, a highly time-consuming and error-prone process. Analysis of the packets that Rether generates under specific failure scenario was also equally tedious, because it involved collecting $tcpdump$ traces and inspecting them manually or through some simple test-case specific filter programs. With such a scheme it was not unusual to take one or multiple days to complete just one test case. This experience motivated us to develop an effective protocol testing infrastructure which can provide the user an implementation independent way to specify the scenarios to be verified. The result of this effort is the network fault injection and response analysis system called **VirtualWire**.

VirtualWire allows developers to specify *which* network faults should happen *when*, using an event-based scripting language, and injects these faults into actual protocol test runs according to the high-level specifications. This obviates any code instrumentation inside the kernel by the user. The same scripting language can also be used to specify filters to determine in real time from the network packet trace whether the protocol implementation under test generates appropriate responses for given network faults. Essentially VirtualWire provides

Proceedings of the 23rd International Conference on Distributed Computing Systems (ICDCS'03)
1063-6927/03 $17.00 © 2003 IEEE

a virtual network abstraction on top of the physical network used in a testbed, which could emulate any collection of network faults according to a user-defined script in a way that is independent of the protocols being tested and the operating system of the testbed nodes. As a result, network protocol developers/testers can insert protocol-specific network faults into a testbed without additional code instrumentation effort.

The goals achieved by VirtualWire are:

- **Test the *implementations* of network protocols:** VirtualWire tests the actual implementation of network protocols on real network testbeds without requiring any code instrumentation.
- **Generate realistic network faults:** VirtualWire faithfully injects a wide variety of network faults and misbehaviors according to the user's specifications. The faults may be injected at multiple points within the network and their occurrences may be ordered through explicitly stated dependencies.
- **Simplify network fault injection:** VirtualWire provides a declarative scripting language for developers to specify network faults to be injected at run time. The specifications are independent of the implementations, hence can be reused for testing multiple versions of the same protocol implementation.
- **Automate packet trace interpretation:** The scripting language in VirtualWire allows developers to specify the desired response for a given test scenario, which is matched against the network packet trace resulting from actual protocol test runs. This trace filtering capability makes it possible to run through a large number of test cases without human intervention, a particularly important feature for regression testing.

The rest of the paper is organized as follows. We compare VirtualWire with related research efforts in Section 2. The architecture of VirtualWire is described in Section 3. We describe the Fault Specification Language in Section 4. In Section 5 we discuss the implementation aspects of VirtualWire. Section 6 illustrates the utility of VirtualWire with script examples. Section 7 gives the run time performance overheads of VirtualWire. We conclude in Section 8.

2. Related Work

In this section we will survey the research work leading to the different methodologies aimed at uncovering faults in systems. The idea of injecting faults into a system to validate its capabilities under exceptional conditions has been a well-known technique in hardware testing. This idea is applied to test software systems by injecting errors into it in a program-driven manner and observing the response of the system to such exceptions. This is called Software Implemented Fault Injection (SWIFI). Some of the earlier works in SWIFI are FIAT [6], Xception [7], DOCTOR [8], NFTAPE [9]. These systems focus on testing a wide range of faulty behaviors. We believe that restricting the fault injection system to test a specific property makes it compact, flexible and easy-to-learn. VirtualWire is aimed at detecting faults in communication protocol implementations.

We can categorize the approaches that are aimed at detecting faults in implementations as follows, (i) static analysis of the code using compiler extensions, (ii) static analysis of the outputs as a result of interaction among nodes running the protocol, and (iii) active probing. Engler et al. uses static code analysis to catch code invariants, e.g. permanently disabled interrupts [10]. This will, however, fail to cover the bugs that may surface at run time and cannot be expressed using static code patterns. Analyzing the outputs from a protocol's interactions to infer about the behavior, as used in Tcpanaly [11], requires collecting large traces. Instead we flag the error at run time by monitoring the protocol invariant that the user specifies.

Active probing idea was introduced by Comer and Lin [12]. Orchestra [13] extends it with the flexibility for manipulating messages. Orchestra uses a fault injection layer between the layer under test and the layer below on the network stack. The user has to program the fault injection experiment using Tcl and C. When generating test cases from the protocol specifications, it is more intuitive to use a declarative language for generating the fault injection experiments because like the design specifications, the fault scenario description can be kept separate from the specifics of implementation. Therefore, we present a domain specific language with a set of primitives that can represent all network faults, as well as, perform analysis at run time. The Piranha project [14] and Tsai [15] also uses active probing to corrupt/modify packets to test system behavior. But they are limited in the types of faults that can be tested.

Emulating network interactions is also a widely used technique to observe the protocol behavior. Delayline [16] provides a configurable environment for emulating wide-area network over local-area network by specifying topologies with different link delays. Dummynet [17] can simulate different features of the network, like finite queue size, limited bandwidth and delays. It can be used to test real protocol implementations. However, they lack the ability for packet manipulation.

3. System Overview

VirtualWire is a tool capable of testing network protocol implementations that are running in a distributed manner across multiple nodes. In this section, we first discuss the setup required for using VirtualWire as the testing tool. We will then introduce the main components of VirtualWire, which includes the programming front-end that parses the input specifications and initiates all the nodes in the testbed to start the test runs, a Fault Injection Engine (FIE) that injects faults into the network sessions in progress, and a Fault Analysis Engine (FAE) that analyzes responses from the protocol

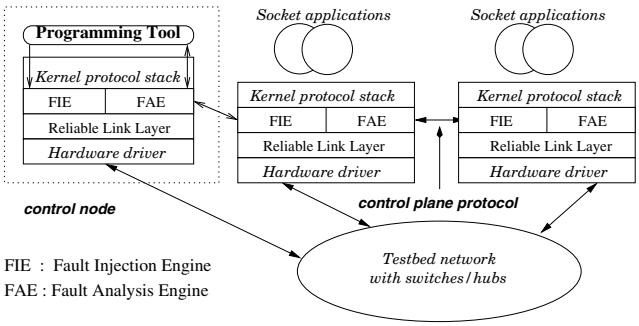

Fig. 1. **System Architecture: VirtualWire can be used to test kernel stacks or user-level socket applications. Fault injection and analysis layers reside on each node. The control node hosts the programming tool that parses the user-defined fault injection and analysis scripts and initializes the FIEs and FAEs for each test case scenario through a control protocol. The control plane enables the components in the system to communicate during initialization, test case execution and error reporting.**

implementation under test. As shown in Figure 1, each testbed node includes a FIE and FAE engine, and one of the testbed nodes serves as the *control node* that hosts the programming front-end.

3.1. Testbed Environment

VirtualWire is designed to facilitate the testing of any network protocol operating within a local area network, e.g. a web server cluster. Thus a realistic testbed that prototypes the actual system may consist of hubs and switches connecting a set of host machines using a shared bus or point-to-point links. It is assumed that VirtualWire components can only be installed on host machines and not on switches. Thus the latter are invisible to VirtualWire as far as fault injection is concerned. Since, we cannot install any software on switches, additional nodes may need to be placed to emulate faults on links whose adjacent nodes are switches. We may also need to test a protocol in an environment where not all the host systems are under the control of the tester. In that case, the script has to be written in a way so that we do not need to run the FIE/FAE on that node. This is possible because the network activity can be monitored completely either on the sender or the receiver node.

3.2. Programming Front-end

The programming front-end in VirtualWire is intended to make the specification of the faulty behavior easier for the user. For this purpose a scripting language has been designed, which is discussed in Section 4. The language provides primitives to enable different kind of faults in the network, like drop, delay, reorder, modification or duplication of the packets. The language constructs can also be used to inject program-driven faults in a distributed fashion across multiple nodes. The idea is to provide the flexibility of ordering the network events, i.e. exchange of packets, and faults in the network in a deterministic or non-deterministic fashion to specify all possible test cases. A central node interprets the script and initializes the test nodes with the relevant data structures. In Figure 1 we refer the node hosting the interpreter as the *control node*.

3.3. Fault Injection and Analysis Engine

The Fault Injection and Analysis Engine (FIE/FAE) in VirtualWire is based on the concept of probing the network traffic that is in progress and taking actions on the packets based on user-defined specifications. An important design criteria for the FIE/FAE is that insertion of the layer should not require changes to the host operating system. The FIE/FAE works by monitoring and counting packets of interest that are exchanged among the participating nodes. The FIE and FAE modules are inserted between the network interface card's device driver and the IP protocol stack, and therefore can intercept all incoming/outgoing packets through the hosts. The user-specified script dictates when the FIE should trigger an action, which leads to a fault being injected or a counter update. Similarly FAE tracks user-defined invariants on some counter(s) and flags an error on any violation.

Since VirtualWire supports distributed evaluation of conditions, and execution of actions spread across multiple nodes, it requires a *control protocol* to exchange state information (states of counters and terms) across the hosts. An important consideration in the design of a fault injection tool such as VirtualWire is that there should not be any faults that it cannot account for. Otherwise, it is difficult to present a truly "controlled" environment for protocol testing. Toward this end, VirtualWire implements a *Reliable Link Layer* (RLL) to prevent MAC layer bit errors from causing a packet drop when the FIE/FAE is unaware of the packet loss. The RLL guarantees reliable delivery of packets handed over to it by the VirtualWire layer, and is based on a simple sliding window protocol.

4. Fault Specification Language

In this section we will briefly introduce the Fault Specification Language (FSL) which enables the user to write test case scenarios for fault injection and fault analysis. Each scenario is an unordered set of rules, which are $\{condition >> action\}$ pairs. An action is triggered whenever a condition is satisfied.

The goal of FSL is to define different network characteristics. The data types are specific to this purpose. The three data types are, packet definition, node definition, and counter definition.

Packet definition is used to specify the packet types that will be monitored by the VirtualWire FIE and FAE. A packet definition is a logical AND-ing of all the tuples, where each tuple consists of the starting offset of the bytes to match, number of bytes to match, an optional bit mask, and the hex pattern to look for. Example of packet definitions are shown as part of the $Filter\ Table$ in Figure 2.

```
VAR SeqNoData, SeqNoAck;

FILTER_TABLE
TCP_data_rt1 : (34 2 0x6000),  (36 2 0x4000),
               (38 4 SeqNoData),  (47 1 0x10 0x10)
TCP_ack_rt1 : (34 2 0x4000),  (36 2 0x6000),
              (42 4 SeqNoAck),  (47 1 0x10 0x10)
TCP_syn : (34 2 0x6000),  (36 2 0x4000),
          (47 1 0x02 0x02)
TCP_synack : (34 2 0x4000),  (36 2 0x6000),
             (47 1 0x12 0x12)
TCP_data : (34 2 0x6000),  (36 2 0x4000),
           (47 1 0x10 0x10)
TCP_ack : (34 2 0x4000),  (36 2 0x6000),
          (47 1 0x10 0x10)
END

NODE_TABLE
node0  00:46:61:af:fe:23  192.168.1.1
node1  00:23:31:df:af:12  192.168.1.2
END
```

Fig. 2. **Filter Table and Node Table: Examples of Packet Definitions and Node Definitions. The packet definitions are used to distinguish different packets in TCP protocol. The node definitions comprise the hardware address and the IP-address.**

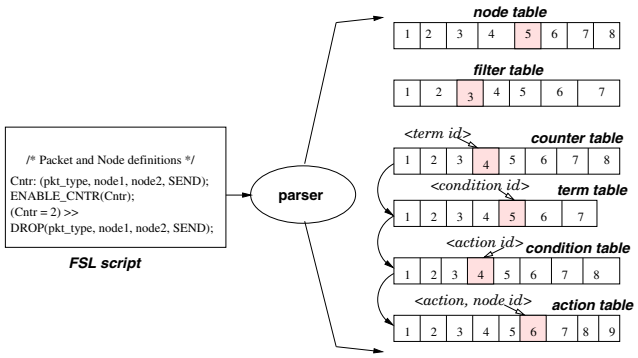

Fig. 3. **Maintenance of Execution States in Fault Injection Engine: The FSL parser generates six tables from the FSL script. The tables are sent to all the participating nodes. The packet and node definitions in the script set up the filter table and node table respectively. In the figure, a matched packet has affected counter 4 in the counter table. Counter 4 uses the term-id it maintains to index into the 5th entry in the term table. Similarly, 5th term entry indexes using the condition-id stored in it to trigger evaluation of the 4th condition. If 4th condition evaluates to true, it will use the action index to trigger the 6th action in the action table.**

Node definition gives the mapping from hostname to its MAC-address and IP-address. The list of node identifiers is referred to as the $Node\ Table$. An example of Node Table is in Figure 2.

A counter definition in FSL is used to count the events, which in essence is the send/receive event of a specific packet type. It can also be used as a local variable on a node. In this case, the counter has to be explicitly controlled by the user-specified instructions. The details of the syntax for counter definition can be found in [18].

The semantics of FSL involves counters, terms and conditions. A *term* in FSL is a boolean relation between two counter values, or between a counter value and an integer constant. FSL supports most of the C-like relational operators, viz. $>$, $<$, $\geq$, $\leq$, $=$, $\neq$. A *condition* is a logical expression of *terms*. The terms can be combined using relational operators, like AND, OR, NOT to represent complex conditions. When a packet of a particular type is encountered, VirtualWire could trigger a counter update, which in turn could trigger a term computation, leading to a condition evaluation and eventually executing an action that can either be an injected fault or another counter update.

The set of primitives for manipulating the counters and specifying the actions can be found in Table I and Table II respectively.

5. System Implementation

In this section we describe the implementation of the programming front-end and the Fault Injection and Analysis Engine (FIE/FAE) in VirtualWire. The FAE implementation is similar to the FIE because the basic mechanism of flagging errors is based on the same idea of counting events based on packets transferred.

5.1. FSL Interpreter

The programming tool is a user level process active on the control node. The user writes a script using the specification language and submits it to the FSL parser through a command line interface. The interpreter parses the script to generate a set of six tables which are used to initialize each FIE and FAE involved in the test scenario. For simplicity, all FIEs and FAEs are sent the entire set of tables even though each node may touch only a subset of the entries in each table.

The $filter\ table$ and the $node\ table$ are used for classification of each packet. Hence these are static tables, unless there is a variable in the filter table which is defined at run time. The rest of the tables are used to maintain VirtualWire's execution states across all the testbed nodes. These four tables are,

- *counter table*: A $counter\ \ table$ contains the list of counters used in the scenario script. For each counter entry, the parser generates pairs of $\{term\_id,\ \ condition\_id\}$ that are dependent on the counter's value, as well as, the nodes which need to be reached. A counter may appear in multiple terms and a term may appear in multiple conditions. Whenever a counter value changes we need to update the term as well as, reevaluate the conditions. Hence, it helps to tag which term and condition will get affected by a particular counter.
- *term table*: A $term\ table$ is indexed by $term_id$s, with each entry storing term expression as a tuple comprising $counter_id$s or integer constant and relational operator connecting them. A term expression is evaluated and stored when the corresponding counter value changes.

ASSIGN_CNTR(counter id)
ENABLE_CNTR(counter id)
DISABLE_CNTR(counter id)
INCR_CNTR(counter id, value)
DECR_CNTR(counter id, value)
RESET_CNTR(counter id)
SET_CURTIME(counter id)
ELAPSED_TIME(counter id)

TABLE I

Counter-Manipulation Primitives and Syntax

DROP(pkt_type, node id, node id, SEND/RECV)
DELAY(pkt_type, node id, node id, SEND/RECV, duration)
REORDER(pkt_type, node id, node id, SEND/RECV, #pkts, order)
DUP(pkt_type, node id, node id, SEND/RECV)
MODIFY(pkt_type, node id, node id, SEND/RECV, pattern)
FAIL(node id)
STOP
FLAG_ERR

TABLE II

Action-Specification Primitives and Syntax

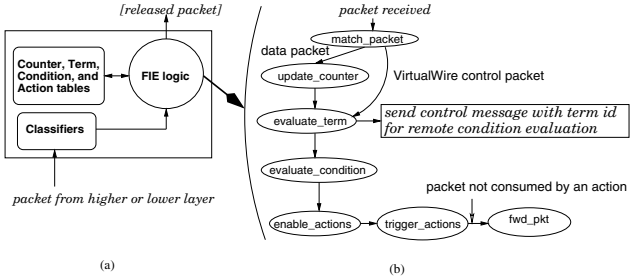

Fig. 4. **The Control Flow of Fault Injection Engine: (a) The software architecture of the Fault Injection/Analysis Engine. The "classifiers" denote the Filter Table and the Node Table used for trapping the packets to monitor. The rest of the tables maintain state information for the FIE/FAE. (b) The FIE control flow for every packet that matches a packet definition in the Filter Table. A matching packet denotes an event that will affect at least the counter table. New counter value can turn a term true, which will lead to a condition evaluation. If condition is satisfied it triggers an action. A fault type action, like drop will consume the packet, but a counter manipulation action will release the packet.**

- *condition table*: A $condition\ \ table$ is indexed by $condition\_id$. It stores the condition expression in terms of $term_id$s and logical operators connecting them. It also maintains a list of $\{node\_id,\ \ action\_id\}$ pairs so that whenever a condition is satisfied the action can be triggered.
- *action table*: An $action\ \ table$ is indexed by $action_id$s with each entry storing the action to be performed and the corresponding node identifier.

The interactions among these tables are shown in Figure 3. In FSL, one can specify a counter on a packet type on one node that can trigger the computation of a term maintained on a remote node. Similarly, a condition that is found to be satisfied on one node can trigger an action on another node.

5.2. Distributed Run-Time Engine

The main issue in the implementation of the Fault Injection and the Analysis Engine (FIE/FAE) is to keep the host Operating System untouched. We choose Linux 2.4 as the platform for this prototype implementation and used the module infrastructure. The FIE/FAE engines use the Netfilter facility provided in Linux 2.4 kernels to register hooks to redirect the packets for classification. The software architecture, therefore is as shown in Figure 4(a).

Once a packet is identified to be belonging to the classifier tables (Filter and Node Tables), it goes through the logic as shown in Figure 4(b). An *update_counter* routine sets the value for the counter defined for that packet type. An update of a counter must trigger evaluation of the terms present on that node. A change in term state leads to evaluation of the condition. The condition may be local to that node, or may be composed of terms being evaluated on different nodes. In the latter case the changed term status is reported to the nodes where the condition is evaluated. We chose to evaluate the condition at the nodes, where an action dependent on that condition, might have to be triggered. This is followed by the triggering of actions, which can be a fault type or a counter-update operation.

The different network faults supported in VirtualWire are drop, delay, reorder, duplicate or modify (corrupt) a packet. The implementation of the delay primitive uses the software timer facility in Linux kernel. Hence the granularity of delay can be no less than a jiffy, i.e. 10 ms. The reorder and duplicate primitives are implemented by queueing the specified set of packets and then releasing them in burst when the bottom half is scheduled next. Modification for the packet uses random perturbation of the bytes. In this case, if a user specifically wants to set the bytes, it is possible to specify it. The checksum in such a case must be set correctly by the user.

The distributed evaluation and execution in VirtualWire is supported by a *control plane protocol* that coordinates among the FIEs across multiple hosts. The control plane messages are implemented as payloads of raw Ethernet frames. Control messages are exchanged to communicate changes in counter values and term state to the appropriate nodes. For example, if a term has the second operand as another counter and it is maintained at a different node from the first operand counter, then every time the counter value changes it needs to be sent to the other node. Otherwise, if one of the operands is an integer, then the term can be evaluated locally and a term status is conveyed only in case of a change in its status.

6. Illustrative Examples

In this section, we illustrate the usage of VirtualWire using examples from TCP and Rether implemen-

```
1.  SCENARIO TCP_SS_CA_algo
2.  SYNACK : (TCP_synack, node2, node1, RECV)
3.  SA_ACK : (TCP_data, node1, node2, SEND)
4.  DATA : (TCP_data, node1, node2, SEND)
5.  ACK : (TCP_ack, node2, node1, RECV)
6.  CWND: (node1)
7.  CanTx : (node1)
8.  CCNT : (node1)
9.  SSTHRESH : (node1)

10. (TRUE)  >>  ENABLE_CNTR( SYNACK ) ;
11.         ENABLE_CNTR( SA_ACK );
12.         ENABLE_CNTR( ACK );
13.         ASSIGN_CNTR( CWND, 1 );
14.         ASSIGN_CNTR( CanTx );
15.         ENABLE_CNTR( CCNT );
16.         ASSIGN_CNTR( SSTHRESH, 2 );
    /* Fault Injection: Drop SynAck at Receiver node */
17. ((SYNACK > 0) && (SYNACK < 2))  >>
            DROP TCP_synack, node2, node1, RECV ;

       /*** ANALYSIS SCRIPT ***/
    /* ACK in response to SYNACK matches tcp_data
18. ((SA_ACK = 1)) >>  ENABLE_CNTR( DATA );
19.         DISABLE_CNTR( SA_ACK );
20. ((DATA = 1 ))  >>  RESET_CNTR( DATA );
21.         DECR_CNTR( CanTx , 1 );
    /* slow-start */
21. ((CWND <= SSTHRESH) && (ACK = 1)) >>
            RESET_CNTR( ACK );
22.         INCR_CNTR( CWND, 1);
23.         INCR_CNTR( CanTx, 1);
    /* congestion avoidance */
24. ((CWND > SSTHRESH) && (ACK = 1)) >>
        RESET_CNTR( ACK );
25.         INCR_CNTR( CanTx, 1 );
26.         INCR_CNTR( CCNT, 1 );
27. ((CWND > SSTHRESH) && (CCNT > CWND)) >>
            RESET_CNTR( CCNT );
28.         INCR_CNTR(CWND, 1);
29.         INCR_CNTR(CanTx, 1);
    /* Number of data packets that can be sent out
       is never negative */
30. ((CanTx < 0 )) >> FLAG_ERROR ;
31. END
```

Fig. 5. **Script to test the implementation of the switch from slow-start to congestion avoidance algorithm in TCP. This script verifies whether the the implementation can detect the crossing of the ssthresh value and trigger the congestion avoidance.**

tations in Linux 2.4.17 kernel. VirtualWire module is itself implemented for Linux 2.4 kernels. The test cases demonstrate that simple and short fault injection and analysis scripts can be used to test complex protocol behaviors without any instrumentation of the protocol code.

6.1. TCP Congestion Control Implementation in Linux

In the following experiments, we setup a testbed with two Pentium-4 machines running Linux 2.4.17 as the host Operating System and inserted the VirtualWire module using the Netfilter hook. We established a TCP connection from port 24576 ($0x6000$) of the sender on $node1$ to the port 16384 ($0x4000$) of the receiver on $node2$. The packet types and the host nodes in the experiments are defined in Figure 2. The priority of the filter rules is in descending order of occurrence. If a match is found with one rule then there is no need to match the subsequent rules.

Two main algorithms in TCPs congestion control mechanism are the slow-start and congestion avoidance, as specified in [19]. When a connection is established, TCP initializes 2 variables on the sender, congestion window ($cwnd$) and threshold value ($ssthresh$). Initially, $cwnd$ can be set to 1, 2 or 4 times the TCP Maximum Segment Size (MSS), and $ssthresh$ is 64KB. With each successful transmission, the $cwnd$ is incremented by 1 according to the slow-start algorithm till the number of bytes transmitted reaches the $ssthresh$ limit. Once this limit is reached, the congestion avoidance algorithm is triggered. If there is retransmission of any packet, then $cwnd$ is reset to 1, and $ssthresh$ drops to half the size of $cwnd$ but not less than 2 MSS. The transmission reverts to slow-start.

This experiment was aimed to test the transition of TCP data transmission from slow-start algorithm to the congestion avoidance algorithm once the $cwnd$ goes above $ssthresh$. The $ssthresh$ value was manipulated to 2 by dropping one $SYNACK$ at the receiver during *connection establishment*, as shown in Line 17 of the script in Figure 5. It caused a retransmission of the SYN packet. Hence $ssthresh$ is reset to 2 and $cwnd$ to 1. In a correct implementation, in the next try when the connection is established, the $cwnd$ should exceed $ssthresh$ once 2 ACKs come back, and must trigger congestion avoidance. We keep track of each of these packets in the analysis section of the script and flag error if it does not conform to the expected result. The TCP implementation in Linux 2.4.17 behaved correctly by switching to congestion avoidance algorithm.

6.2. Token Passing in Rether

In this section, we will demonstrate a test scenario using Rether protocol, that involves distributed rule execution, i.e. counter update is done at a node different from where the action, dependent on that counter, is executed. The Rether protocol has been briefly explained in Section 1. For the purpose of this experiment, we have 4 nodes that are exchanging tokens in a circular fashion. We will let $node3$ "fail" using the fault injection script, and observe through the analysis script, if Rether detects the failure and the token cycle is reconstructed among the remaining nodes, so that the real time data transport remains unaffected.

The packet types that need to be identified for this test are defined in the Filter Table in Figure 6, where 0x9900 is the protocol identifier of a rether control packet. In best-effort mode, the token circulates among the nodes in a fixed round-robin order. Our Rether testbed consists of four nodes in the following round-robin order: $node1$, $node2$, $node3$, and $node4$. $node1$ and $node4$ have a real

```
1.    FILTER_TABLE
2.    tr_token : (12 2 0x9900), (14 2 0x0001)
3.    tr_token_ack : (12 2 0x9900), (14 2 0010)
4.    TCP_data : (34 2 0x6000), (36 2 0x4000)
                 (47 1 0x10 0x10)
5.    END

6.    SCENARIO Test_Single_Node_Failure 1sec
7.    CNT_DATA : (TCP_data, node1, node4, RECV)
8.    TokensTo2 : (tr_token, node1, node2, RECV)
9.    TokensFrom2 : (tr_token, node2, node3, SEND)
10.   TokensTo4 : (tr_token, node2, node4, RECV)
11.   TokensTo1 : (tr_token, node4, node1, RECV)

12.   ((CNT_DATA > 1000 ))  >>
            ENABLE_CNTR( TokensTo2 );
13.   ((TokensTo2 = 1))  >>  FAIL( node3 );
14.                        ENABLE_CNTR( TokensFrom2 );
15.                        RESET_CNTR( TokensTo2 ) ;
16.   ((TokensFrom2 = 3)) >> ENABLE_CNTR(TokensTo4);
17.   ((TokensTo4 = 1)) >> ENABLE_CNTR(TokensTo1);

         /*** ANALYSIS SCRIPT ***/
18.   ((TokensFrom2 > 3 ))  >>  FLAG_ERROR;
19.   ((TokensTo2 = 1) && (TokensTo4 = 1)
            && (TokensTo1 = 1))  >>  STOP ;
20.   END
```

Fig. 6. **Script to test the implementation of token recovery in Rether. The script knocks one node out of the ring and checks if the token passing mechanism can recover by reconstructing the ring without the "crashed" node.**

time TCP-based client-server communication and $node2$ and $node3$ do not have any real time data to send.

In this test scenario, we let $node3$ "crash" using the fault injection script, and observe through the analysis script, if Rether detects the failure and the token cycle is reconstructed among the remaining nodes. As shown in the script in Figure 6, a fault is injected by crashing $node3$ when $node2$ receives a token following exchange of 1000 TCP data packets from $node1$. Since $node2$ receives the token, and as it is in best-effort mode it must send the token to $node3$, which has crashed. Hence fault detection mechanism should be able to reconstruct the ring by detecting that there is no $token - ack$ from $node3$ inspite of 3 retransmissions. The analysis script verifies the fault detection mechanism by checking for the 3 token retransmissions by $node2$. The recovery protocol is verified by checking the packet sequence for a round-robin visit of the token to the 3 remaining nodes within an inactivity timeout period set at 1 sec. Since the fault detection and recovery should complete within 1 sec, an error is flagged if the scenario is terminated due to inactivity.

7. Performance Evaluation

An important design constraint for a fault injection tool is that it should not unintentionally introduce excessive delay that eventually distorts the behavior of the protocol implementation under test. In this section, we study the performance impact of VirtualWire on network protocols and applications. We measure the throughput and latency of TCP/UDP connections with and without VirtualWire between two Pentium-4 machines connected through a 100 Mbps switch. For all the experiments, we have varied the number of packet type definitions (or filters) from 1 to 25, and allowed 25 actions to be triggered for each packet.

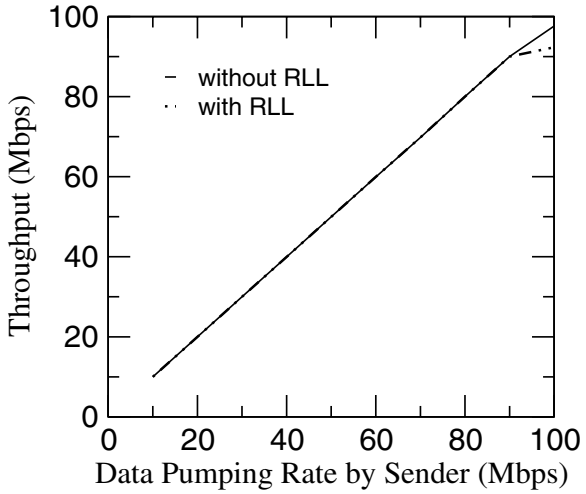

Fig. 7. **The plot shows the throughput that can be achieved with the Fault Injection Layer inserted in the protocol stack and using a TCP connection between 2 Pentium-4 hosts connected using a 100Mbps switch.**

In Figure 7, the throughput of a TCP connection between the 2 test machines is plotted against the offered data pumping rate. There is a noticeable drop in throughput beyond 90 Mbps. This is because the Reliable Link Layer encapsulates both the TCP data and the TCP ack packets. This generates ACKs at the RLL level in both directions, increasing the chances of collisions when the offered load is high, and degrading the throughput. However, the throughput loss in this case is within 10%.

The other important metric for assessing the intrusiveness of VirtualWire is the extra latency it adds to protocol processing. For this experiment, we used an echo connection using UDP between the 2 test machines and measured the packet round-trip latency with and without VirtualWire. Figure 8 shows the percentage increase in protocol processing time due to VirtualWire. The time to process a packet grows linearly with the number of packet types in each curve because the current VirtualWire implementation searches linearly through the packet type definitions for the exact match. With actions added, the overhead is increased as VirtualWire has to update all the tables that are affected. Turning on the Reliable Link Layer further increases the overhead. However, with up to 25 packet types, the additional packet processing overhead never goes beyond 7% of the normal round-trip time.

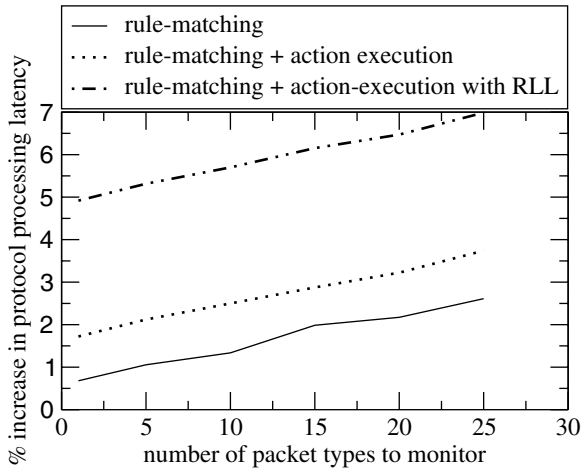

Fig. 8. **The plot shows the additional overhead in protocol processing due to the insertion of the Fault Injection layer. The test was conducted by measuring the round-trip latency for a packet over a UDP connection between 2 Pentium-4 hosts with VirtualWire installed. The test was repeated with (i) 25 packet matching rules, (ii) 25 packet matching rules plus each packet match triggering 25 actions, and (iii) case(ii) with RLL turned on. The percentage increase in latency even with RLL is around 7%.**

8. Conclusion and Future Work

Testing a distributed application or a network protocol implementation is fundamentally hard. It is not sufficient to devise meaningful test cases. Developers are also plagued with the lack of tools to automate the process of injecting targeted faults into the network under test and of analyzing the large amount of resulting packet traces for non-compliant behaviors. As a result, the fault injection and packet trace analysis tasks are performed in an ad hoc and manual fashion. Worse yet, they have to be repeated from scratch every time a new network protocol or new distributed application is to be tested.

In this paper, we present a distributed network fault injection and analysis tool called **VirtualWire**, which can automate most of the testing processes in a protocol-independent way. Instead of instrumenting the protocol under test and manually inspecting packet traces, users can write a script that describes the faulty scenarios that should take place at certain time and the expected resulting packet sequences in response to these faulty scenarios. The script is in the form of event and action pairs, also called rules. In this scripting language, events and actions may be distributed over multiple network nodes and implicitly ordered through special language constructs, thus providing the flexibility to accommodate complex test case specification.

Using this tool, the task of protocol testing is elevated to a much higher level in defining proper test cases rather than spending time in writing low level customized code to inject faults and to analyze packet traces. By pushing the low-level chores of network protocol testing to an automated script-driven tool, VirtualWire significantly improves the productivity of protocol implementation development, and greatly facilitates regression testing of newer versions of protocol implementations.

Finally, as a long term goal of the VirtualWire project it will be interesting to investigate the possibility of generating the fault injection and packet trace analysis scripts directly from the protocol specification. This will truly make the testing process completely automated by relieving the testing team of the task of script development.

Acknowledgement

This research is supported by NSF awards ANI-9814934, ACI-9907485, and ACI-0083497, USENIX student research grants, as well as fundings from National Institute of Standards and Technologies, Siemens, and Rether Networks Inc.

References

[1] G. Holzmann, *Design and Validation of Computer Protocols*. Prentice Hall, 1991.
[2] "Vint," http://netweb.usc.edu/vint.
[3] S.Keshav, "Real: a network simulator," University of California at Berkeley, Berkeley, California, Tech. Rep., 1988.
[4] C. Venkatramani and T. Chiueh, "The design, implementation and evaluation of a software-based real-time ethernet protocol," in *ACM SIGCOMM*, 1995.
[5] S. Sharma, K. Gopalan, N. Zhu, G. Peng, P. De, and T. Chiueh, "Implementation experiences of bandwidth guarantee on a wireless lan," in *ACM/SPIE Multimedia Computing and Networking (MMCN)*, 2002.
[6] Z. Segall and T. Lin, "Fiat: Fault-injection based automated testing environment," in *18th Int'l Symposium on Fault Tolerant Computing*, 1988.
[7] J. Carreira, H. Madeira, and J. G. Silva, "Xception: Software fault injection and monitoring in processor functional units," in *5th IFIP Int'l Working ConferenceDependable Computing of Critical Applications (DCCA)*, Sept. 1995.
[8] S. Han, H. A. Rosenberg, and K. G. Shin, "Doctor: An integrated software fault injection environment for distributed systems," in *IEEE International Computer Performance and Dependability Symposium*, 1995.
[9] D. Stott, B. Floering, Z. Kalbarczyk, and R. K. Iyer, "Nftape: A framework for assessing dependability in distributed systems with lightweight fault injectors," in *4th IEEE International Computer Performance and Dependability Symposium (IPDS-2K)*, 2000.
[10] D. Engler, B. Chelf, A. Chou, and S. Hallem, "Checking system rules using system-specific, programmer-written compiler extensions," in *OSDI*, 2000.
[11] V. Paxson, "Automated packet trace analysis of tcp implementations," in *ACM SIGCOMM*, 1997.
[12] D. E. Comer and J. C. Lin, "Probing tcp implementations," in *USENIX*, June 1994.
[13] S. Dawson, F. Jahanian, and T. Mitton, "Orchestra: A fault injection environment for distributed systems," in *26th International Symposium on Fault-Tolerant Computing (FTCS)*, 1996.
[14] J. L. Griffin, "Testing protocol implementation robustness," in *29th International Symposium on Fault-Tolerant Computing*, 1999.
[15] T. Tsai and N. Singh, "Reliablity testing of applications on windows nt," in *International Conference on Dependable Systems and Networks (DSN)*, 2000.
[16] D.B.Ingham and G.D.Parrington, "Delayline: A wide-area network emulation tool," in *Computing Systems*, 1994.
[17] L. Rizzo, "Dummynet: a simple approach to the evaluation of network protocols," in *ACM computer Communications Review*, 1997.
[18] P. De, A. Neogi, and T. Chiueh, "Virtualwire: A fault injection and analysis tool for network protocols," Stony Brook University, Tech. Rep., 2002, http://www.ecsl.cs.sunysb.edu/tr/TR132.ps.gz.
[19] W.Stevens, "Tcp slow start, congestion avoidance, fast retransmit, and fast recovery algorithms," 1997.