TCP Validation

Master's Project Final Report

Author: Omkar Kasinadhuni

Email: okasinad@cs.odu.edu

Project Presentation Date: September 14, 2007
Department of Computer Science
Old Dominion University

Project Advisor: Dr. Michele Weigle
Email: mweigle@cs.odu.edu

Project Presentation Date: February 11, 2007

Department of Computer Science
Old Dominion University

Table of Contents

Acknowledgement

I would like to express my sincere gratitude and appreciation to my project advisor **Dr. Michele Weigle,** Assistant Professor, Department of Computer Science, Old Dominion University for providing me with an opportunity to work in the area of TCP Validations in NS-2. I would also wish to acknowledge my appreciation for her guidance, encouragement and patience throughout the duration of this project which lead to the success of this project.

Abstract

Simulations play a vital role in network research. The NS-2 network simulator is a discrete event simulator targeted at networking research, but its default TCP agent (one-way TCP) does not reflect how real TCP implementations perform. Full-TCP, the two-way TCP agent in NS-2 more closely emulates TCP. But Full-TCP is not fully validated. Its test suites need to be fully validated before Full-TCP could be made the default TCP Agent in NS-2. The goal of this project is to validate Full-TCP tests and bring Full-TCP a step closer to being the default TCP agent in NS-2.

1. Introduction

The behavior and the characteristics of networks have always been of keen interest to network researchers. The complexities involved in large-scale networks such as the Internet make it even more important to examine its dynamics and explore any unexpected behavior. Real time measurements and experiments are tough because of the network's great heterogeneity and rapid change. Developing a mathematical model to test and analyze Internet traffic is not feasible either. Hence, simulations play a vital role not only in testing, evaluating and correcting the current Internet, but also proposing the "*future Internet*".

Testing new protocols on real networks can endanger other people's traffic and, if not done with care, may also produce inaccurate and misleading results. Network simulations are widely used in network research to test new protocols and modify existing ones. Simulations allow examining performance of various networks with different topologies without any hardware setup. They provide means to emulate an entire network topology within just one machine. They help in predicting the expected performance of complex networks and also understanding the interactions of protocols. As networks continue to grow more complex, the need to simulate them increases.

The NS-2 network simulator is most widely-used simulator in networking research, and most of the traffic on the Internet is carried by TCP. Simulations in NS-2 pertaining to one-way TCP (where data flows only in one direction of the connection) have been done extensively. But, testing and validating Full-TCP (where data flows in both the directions of a connection) has not been ventured yet thoroughly in NS-2. For NS-2 to emulate the TCP implementation accurately, its Full-TCP tests should be completely validated. Hence, this project focuses on validation of Full-TCP tests in NS-2. The TCP variants dealt with in this project are Tahoe, Reno and New Reno TCP. The process of validation is carried out by examining the test to see if it performs according to the congestion control phases it implements. Also, the test results are compared to their one-way TCP equivalent test to see if they both produce the same result. Since the one-way TCP are already validated, this comparison helps in verifying the correct behavior of Full-TCP tests.

2. Background

2.1 Network Simulator NS-2

Network Simulator NS-2 is a public domain simulator targeted at the networking research. NS-2 was written in C++ and uses OTcl as a command and configuration interface. It supports simulations of TCP, routing and multicast protocols over wired and wireless networks. NS-2 allows simulating an entire network topology within one machine with multiple hops, packet losses and latency. It provides tools which allow researchers to

- Simulate complex networks in test labs
- Emulate the actions of several well-known flow control protocols (such as TCP)
- Visually analyze trends and potential trouble spots.

The creators of NS-2 claim that although they have complete confidence in it and in spite of it being one of the best network simulators available, NS-2 is not a polished and finished product, but the result of an on-going effort of research and development. Bugs in the software are still being discovered and corrected.

2.2 TCP Congestion Control

TCP is the dominant protocol used in the Internet today. Hence understanding its performance is important. The data flow in the Internet abides by TCP congestion control phases, slow start and congestion avoidance. The congestion window controls the amount of unacknowledged data allowed in the network. Therefore its size has a direct impact on the throughput achieved. The congestion control phases determine the allowed size of the congestion window and how much it must grow or shrink.

Slow start and congestion avoidance require that two variables be maintained for each connection: a slow start threshold (ssthresh) and a congestion window (cwnd). The combined algorithm operates as follows:

- Initialization for a given connection sets cwnd to one segment and ssthresh to 65535 bytes.
- The TCP output routine never sends more than the minimum of cwnd and the receiver's advertised window.

2.2.1 Slow Start

After connection setup, a TCP sender transmits one segment and waits for its ACK. For each new ACK received, TCP increments cwnd by one segment size. This results in doubling the congestion window each round-trip time (RTT). For example, after the first ACK is received, cwnd is incremented from one segment to two segments. Therefore, two segments can be sent. When each of those two segments is acknowledged, cwnd is increased to four segments. This provides an exponential growth. At some point the capacity of the network can be reached, and an intermediate router will start discarding packets. This tells the sender that its congestion window has gotten too large.

2.2.2 Congestion Avoidance

Slow start has cwnd begin at one segment and be incremented by one segment every time an ACK is received. As mentioned earlier, this opens the window exponentially: send one segment, then two, then four, and so on. Congestion avoidance

dictates that congestion window be incremented by (segsize/cwnd) each time an ACK is received, where segsize is the segment size and cwnd is maintained in bytes. This is a linear growth of congestion window, compared to slow start's exponential growth. The increase in congestion window should be at most one segment each RTT (regardless how many ACKs are received in that RTT), whereas slow start increments congestion window by the number of ACKs received in a round-trip time.

2.3 TCP Loss Recovery

Congestion can occur when data arrives on a large link and gets sent out on a smaller link. Congestion can also occur when multiple input streams arrive at a router whose output capacity is less than the sum of the inputs. TCP assumes that packet loss caused by damage is very small (less than 1%). Therefore the loss of a packet signals congestion somewhere in the network between the source and destination. There are two indications of packet loss: a timeout occurring and the receipt of duplicate ACKs.

When congestion occurs (indicated by a timeout or the reception of duplicate ACKs), one-half of the current window size (the minimum of congestion window and the receiver's advertised window, but at least two segments) is saved in ssthresh. Additionally, if the congestion is indicated by a timeout, congestion window is set to one segment (i.e., slow start).

When new data is acknowledged by the other end, the way cwnd increases depends on whether TCP is performing slow start or congestion avoidance. If cwnd is less than or equal to ssthresh, TCP is in slow start; otherwise TCP is performing congestion avoidance. Slow start continues until TCP is halfway to where it was when congestion occurred, and then congestion avoidance takes over.

2.3.1 Fast Retransmit

TCP generates an immediate duplicate ACK when an out-of-order segment is received. The purpose of this duplicate ACK is to let the other end know that a segment was received out of order, and to tell it what sequence number is expected. Since TCP does not know whether a duplicate ACK is caused by a lost segment or just a reordering

of segments, it waits for a small number of duplicate ACKs to be received. It is assumed that if there is just a reordering of the segments, there will be only one or two duplicate ACKs before the reordered segment is processed, which will then generate a new ACK. If three or more duplicate ACKs are received in a row, it is a strong indication that a segment has been lost. TCP then performs a retransmission of what appears to be the missing segment, without waiting for a retransmission timer to expire.

2.3.2 Fast Recovery

After fast retransmit sends what appears to be the missing segment, congestion avoidance, but not slow start is performed. This is called fast recovery. The reason for not performing slow start in this case is that the receipt of the duplicate acknowledgements tells TCP not only that packet has been lost but also that the packets after the lost one are delivered to the receiver. The receiver can only generate the duplicate acknowledgement when another segment is received. That segment has left the network and is in the receiver's buffer now. That is, there is still data flowing between the two ends, and TCP does not want to reduce the flow abruptly by going into slow start.

The fast retransmit and fast recovery phases are usually implemented together as follows:

1. When the third duplicate ACK in a row is received, set ssthresh to one-half the current cwnd, but no less than two segments. Retransmit the missing segment. Set cwnd to ssthresh plus 3 times the segment size. [cwnd = ssthresh + 3].

   This inflates the congestion window by the number of segments that have left the network and which the other end has cached.

2. Each time another duplicate ACK arrives, increment cwnd by the segment size. This inflates the congestion window for the additional segment that has left the network. Transmit a packet, if allowed by the new value of cwnd.

3. When the next ACK arrives that acknowledges new data, set cwnd to ssthresh. This ACK should be the acknowledgment of the retransmission from step 1, one round-trip time after the retransmission. Additionally, this ACK should acknowledge all the intermediate segments sent between the lost packet and the receipt of the first

duplicate ACK. This step is congestion avoidance, since TCP is down to one-half the rate it was at when the packet was lost.

2.4 TCP in NS-2

TCP is the dominant protocol used in the Internet today. Hence, understanding the performance of Transmission control protocol is important. Simulating TCP is difficult because of the wide range of variables, environments and implementations available.

NS-2 has had its models validated by its creators (Information Science Institute (ISI) 2004). However the models of protocols used by NS-2, specifically Transmission Control Protocol (TCP), are heavily abstracted.

Heidemann et al. [1] reports that the one-way TCP models included in NS-2 model have a simplified protocol supporting unidirectional data transfer without message fragmentation and do not attempt to model any particular TCP implementation or specification. These models suffice for many situations but do not reflect how real TCP implementations perform.

3. Full-TCP Validation

Full-TCP is the two-way TCP supported in NS-2 and is still under development. Full-TCP does not implement the entire reference TCP state machine. However, it tries to closely emulate the behavior of the reference implementation. It is different from the one-way TCP implementation in the following ways:

- Connections may be established and torn down (SYN/FIN packets are exchanged).
- This version of TCP currently sends data on the 3rd segment of an initial 3-way handshake which means that the third segment is an ACK with data. Many actual implementations do not do it until 4th segment.
- Bidirectional data transfer is supported
- The state variables wnd, wnd_init, cwnd, and ssthresh are in segment units.

- Sequence numbers are in bytes rather than packets.

The following the default values pertaining to Full-TCP. They are defined in the *ns-2.32/tcl/lib/nsdefault.tcl* file:

- "**set segsperack 1**": Acknowledges every segment. If set to an integer greater than 1, permits the TCP to delay sending an acknowledgment until that number of segments has been received
- "**set segsize 536**": With 40 bytes of TCP/IP header, the packet size becomes 576 bytes.
- "**set tcprexmtthresh 3**": Waits for 3 duplicate ACKs before entering fast retransmit.
- "**set iss 0**": Sets initial sequence number to 0.
- "**set nodelay false**": When set to true, disables Nagle algorithm.
- "**set data on syn false**": If set to true, allows data to be piggybacked on a SYN.
- "**set dupseg_fix true**": If set to true, fixes a bug that causes duplicate segments with duplicate ACKs not to trigger a fast retransmit.
- "**set dupack_reset false**": When true, sets the counter of consecutive duplicate acknowledgments to 0 when either a non-zero length segment or a segment with a window change is received during ACK processing.
- "**set interval 0.1**": Sets the delayed ACK interval to 100ms.

Although Full-TCP closely emulates the reference implementation, it is still not used as a standard TCP agent in NS-2 because its test suites are not validated. This brings us to the topic of this project which is TCP validation.

3.1 Project Tasks

Many tests in the Full-TCP implementation are yet to be written and validated. Also, there are many more tests that have been written, but need to be validated before Full-TCP can be used as a default TCP agent in NS-2. Hence, I would like to contribute

towards Full-TCP agent in NS-2 by validating Tahoe, Reno and New-Reno tests as my Masters project. The tasks involved in this project are the following:

1. Understand the test suites.
2. Understand the relationship between the various files included in each test suite.
3. Understand the different variables which affect the Full-TCP implementation.(*/ns-allinone-2.32/ns-2.32/tcl/lib/ns-default.tcl*)
4. Examine the control flow in each test suite.
5. Compare the logic of the Full-TCP implementation in NS-2 with the correct implementation (according to the congestion control phases [6]).
6. Check to see if the tests perform as they are supposed to.
7. Examine the output trace files for packet flow (which are obtained after running test files).
8. Examine the output graphs for packet flow (Graphs are obtained after running tests).
9. Compare the Full-TCP tests against the validated one-way TCP to verify the validation of the Full-TCP.
10. If necessary, make changes to Full-TCP, to avoid unexpected behavior, and change it to resemble the correct implementation.
11. Write tests which are yet to be written and validate them as well

The current state of validation tests of Full-TCP in NS-2 can be found in FullTCP.notes [2]. It contains the list of already written tests which need to be validated and also new tests which need to be written and validated.

3.2 Test Suites

Each test suite is named "test-suite-*TestSuiteName*.tcl" and is located at ns-allinone-2.32/ns-2.32/tcl/test/. Each test suite contains multiple tests. The file "misc_simple.tcl" in the same location contains the basic code which is needed for all the test-suite files. Almost all the test-suite files import this file. The rest are small enough to have this code in them. The representation "*ClassName ->MethodName()"* is used below to refer to method *MethodName()* of the Class *ClassName*.

3.2.1 Program control- flow

1. The test begins with TestSuite->runTest() method (in test-suite-*TestSuiteName*.tcl).
2. This calls Test/*testname*->init(). Test/*testname*->init() sets variables such as net_, test_. This method also calls its superclass init() method TestSuite-> init() in misc_simple.tcl**.** TestSuite->init() creates ns object and defines file pointers for output files.
3. Then, TestSuite->runTest() also calls Test/*testname*->run().
4. Test/*testname*->run() calls TestSuite->setup with arguments *testname*, *window*, *list-of-pkt-drops.*
5. TestSuite->setup() calls TestSuite->setTopo().
6. setTopo() creates Topology class object "topo_". Topology/net4->init() creates the whole topology.
7. Testsuite->setup() asks ns to start simulation. It also makes ns stop by calling Testsuite->cleanupAll().

3.2.2 Running validation tests

The tests in the TestSuites can be run by the following command (after setting Environment variables): `$ ns test-suite-`*`TestSuite-Name`*`.tcl `*`TestName`*

For example, to run the test "multiple_reno" in the Test suite test-suite-tcpVariants.tcl, the following command is used:

```
$ ns test-suite-tcpVariants.tcl multiple_reno
```

The command `$ ns test-suite-tcpVaraints.tcl` gives the list of tests available in the test suite.

3.2.3 Trace files

After running a test, each time, a packet trace file "all.tr" gets stored at the same location of the test file (*ns-allinone-2.32/ns2.32/tcl/test*). A typical trace file would look like the following:

```
+ 1.20088 0 2 tcp 40 -------   1 0.0 3.0 1 2 1 0x10 40 0
- 1.20088 0 2 tcp 40 -------   1 0.0 3.0 1 2 1 0x10 40 0
+ 1.20088 0 2 ack 576 ------   1 0.0 3.0 1 3 1 0x10 40 0
r 1.20092 0 2 tcp 40 -------   1 0.0 3.0 1 2 1 0x10 40 0
d 1.20092 2 3 tcp 40 -------   1 0.0 3.0 1 2 1 0x10 40 0
```

The first column is used to indicate either enque operation (+) or deque operation (-) or packet received operation (r) or that the packet has been dropped (d). The second column is used to specify the time in seconds. The next two fields indicate between which two nodes tracing is happening. The next field indicates the type of packet seen. The next field is the packet's size, as encoded in its IP header. The next field contains the flags, which not used in this example. The flags are defined in the flags[] array in trace.cc. Four of the flags are used for ECN: "E" for Congestion Experienced (CE) and "N" for ECN-Capable-Transport (ECT) indications in the IP header, and "C" for ECN-Echo and "A" for Congestion Window Reduced (CWR) in the TCP header.

The next field gives the IP *flow identifier* field as defined for IP version 6.1. The subsequent two fields indicate the packet's source and destination node addresses, respectively. The following field indicates the sequence number. The next field is a unique packet identifier, which as name suggests, is unique for every packet. The last three columns indicate the acknowledgement number, the TCP flags value (The 0x signifies that the number which follows is in hexadecimal format.), and the TCP header length.

3.2.4 Output Graphs

For each test simulation, a graph shows the results of each simulation. The x-axis of that graph shows the time in seconds. The y-axis shows (the sequence number) mod 20. There will be a markings on the graph for each packet as it arrives and departs from the congested gateway and also if the packet is dropped packet at the gateway. A typical output graph is shown in Figure 1.

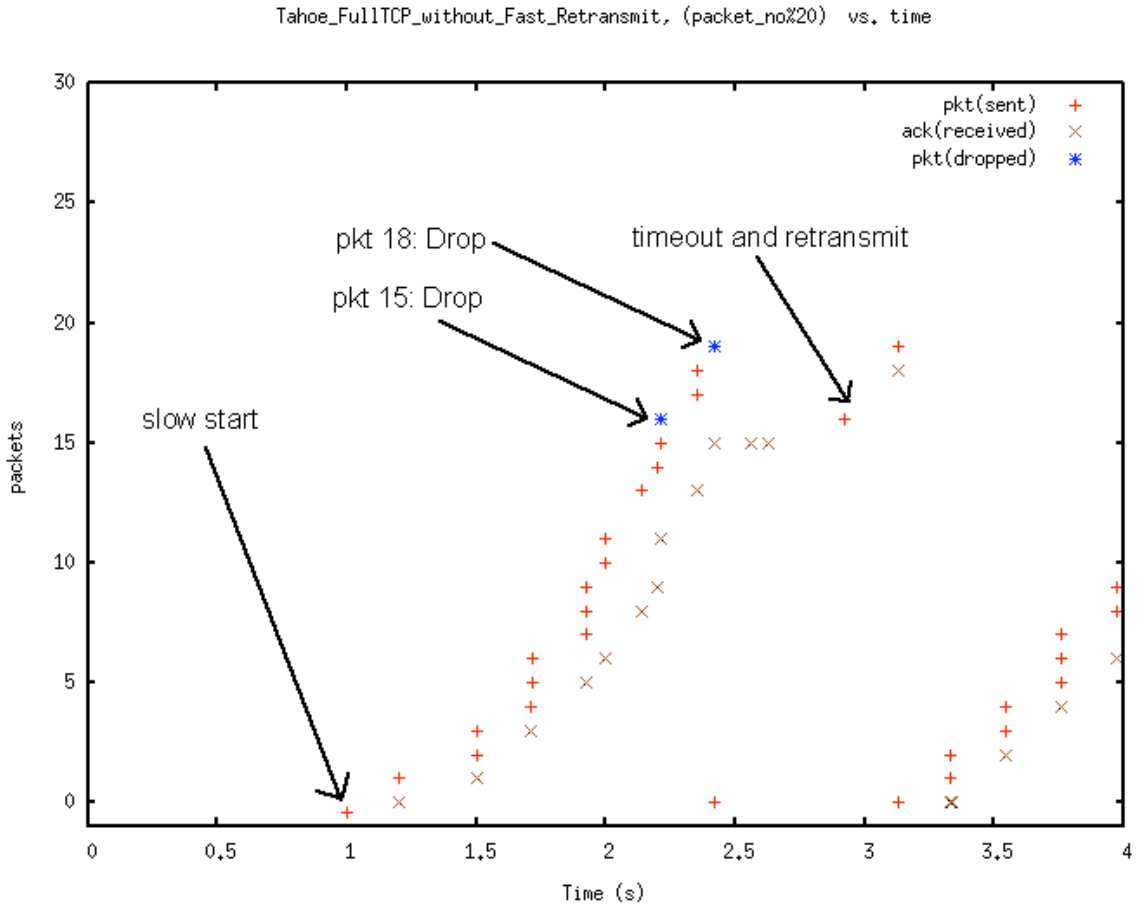

Figure 1: A graph of packet trace for the test "Tahoe Full-TCP without Fast retransmit" in the test suite "test-suite-testReno-full.tcl"

The methods TestSuite->enable_tracecwnd() and TestSuite->plot_cwnd() are responsible for enabling and plotting the congestion window graph for the test simulation. The congestion window graph of the above mentioned test (Tahoe Full-TCP without Fast retransmit" is shown in Figure 2.

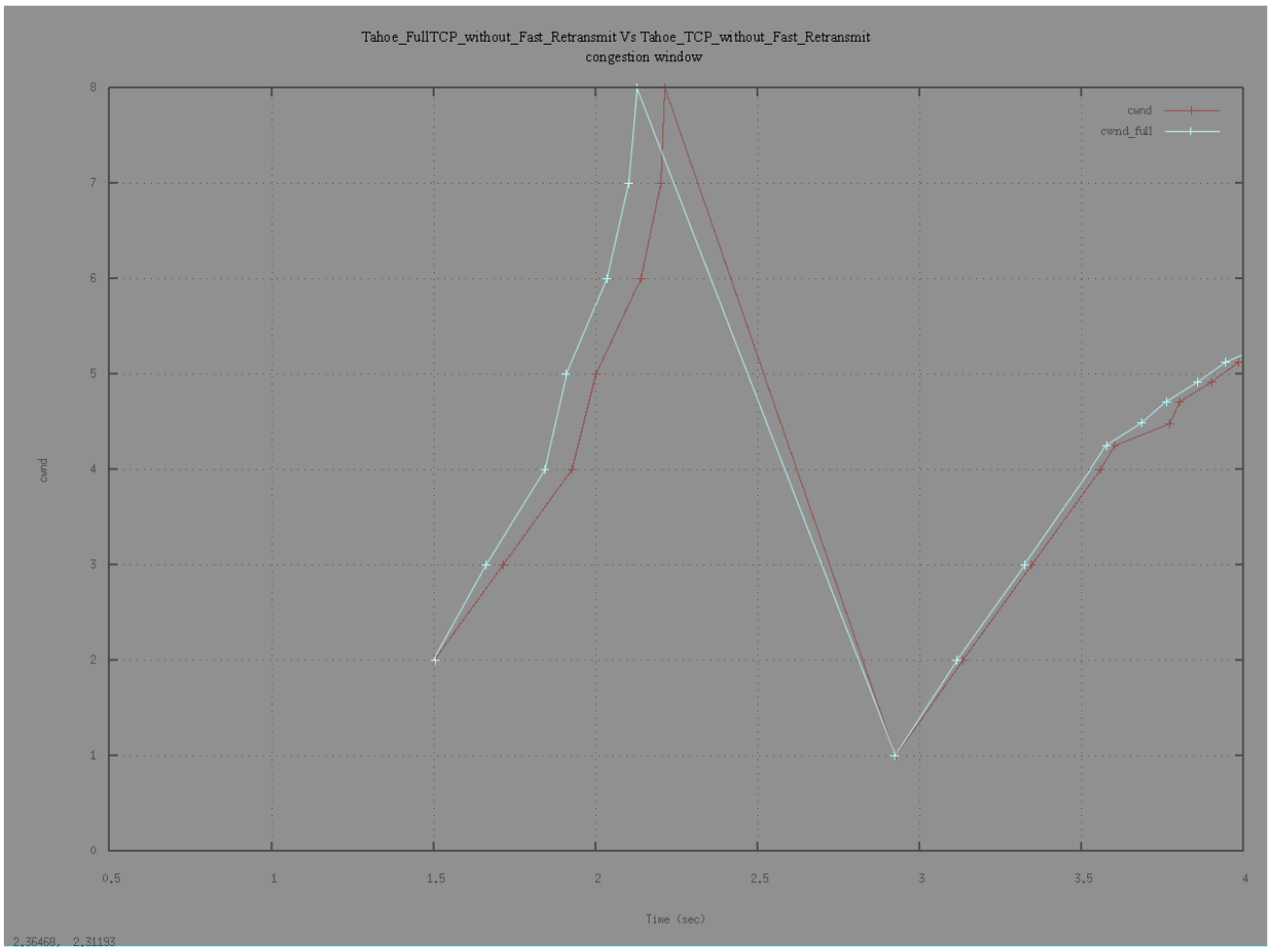

Figure 2: Congestion window of Tahoe Full-TCP with no fast retransmit versus its equivalent one-way TCP test.

3.2.5 Basics of Tahoe, Reno and new-Reno TCPs

Tahoe TCP: The Tahoe TCP implementation added new algorithms and refinements to earlier implementations. It includes slow-start, congestion avoidance, and fast retransmit.

Reno TCP: The Reno TCP includes slow-start, congestion avoidance, fast retransmit and fast recovery. The use of fast recovery phase avoids the need for slow-start after a single packet loss.

New-Reno TCP: New-Reno [3] makes simple changes to the sender's behavior during fast recovery in Reno TCP and avoids some of its performance problems. Apart from all the phases in Reno TCP, it also recognizes partial acknowledgements. When multiple packets are lost from a window, Reno waits for the retransmit timer to go off. In

Reno, partial ACKs, (which acknowledge some but not all of the packets that were outstanding at the start of that fast recovery period), take TCP out of fast recovery by "deflating" the usable window back to the size of the congestion window. In New-Reno, partial ACKs do not take TCP out of fast recovery. Instead, partial ACKs received during fast recovery are treated as an indication that the packet immediately following the acknowledged packet in the sequence space has been lost, and should be retransmitted. Thus, New-Reno can recover without a retransmission timeout when multiple packets are lost from a single window of data. It retransmits one lost packet per round-trip time until all of the lost packets from that window have been retransmitted.

3.3 Validations

As stated earlier, to validate a test, we may have to examine the trace files of the test to confirm its correct behavior. Also, we can compare the test files results with that of the equivalent one way TCP test results and check if they match. In fact, to compare the Full-TCP tests against their equivalent one way TCP tests would be the most efficient way to validate the Full-TCP as the one-way TCP has already been carefully validated and set as the default TCP agent in NS-2. Although the output trace files have been manually checked for each test before declare them validated, only output graphs of Full-TCP test and their equivalent one-way TCP tests are displayed in the following section.

If we notice that a Full-TCP test is behaving accordingly, we can prove its validity by showing that both the Full-TCP test and its equivalent one-way TCP test show similar graphs.

The one-way TCP tests do not drop the same packet as their Full-TCP equivalent. They drop the packet which is one number more than the packet specified. Therefore, I had to change the one-way TCP tests inputs by specifying one packet number less than required so that Full-TCP tests can be compared to their equivalent one-way TCP tests under the similar input conditions.

The tests which were validated are as follows:

- **Tahoe_FullTCP2_without_Fast_Retransmit:** Tahoe Full-TCP without fast retransmit with one packet drop.

- **Tahoe_FullTCP_without_Fast_Retransmit:** Tahoe Full-TCP without fast retransmit with two packet drop.
- **multiple_tahoe_full:** Tahoe Full-TCP with multiple packet drops (set 1).
- **multiple2_tahoe_full:** Tahoe Full-TCP with multiple packet drops (set 2).
- **multiple_reno_full:** Reno Full-TCP with multiple packet drops (set 1).
- **multiple2_reno_full:** Reno Full-TCP with multiple packet drops (set 2).
- **multiple_newreno_full:** New-Reno Full-TCP with multiple packet drops (set 1).
- **multiple2_newreno_full:** New-Reno Full-TCP with multiple packet drops (set 2).
- **timeouts_newreno1_full:** New-Reno Full-TCP with timeout and multiple packet drops (set 1).
- **timeouts_newreno2_full:** New-Reno Full-TCP with timeout and multiple packet drops (set 2).
- **timeouts_newreno3_full:** New-Reno Full-TCP with timeout and multiple packet drops (set 3).

In the following sections, validation of each one of the above mentioned tests are discussed in detail.

3.3.1 Test: Tahoe_FullTCP2_without_Fast_Retransmit

This test uses Tahoe TCP and has only one drop, packet 18. As the name suggests, this test is not supposed to use fast retransmit algorithm. That is, after 18^{th} packet is dropped, and the receiver gets 3 duplicate acknowledgements for the 17^{th} packet, the sender should not retransmit packet 18 (as it would if using fast retransmit). Rather, it should wait for a time out and then retransmit the dropped packet. This test uses a variable *noFastRetrans_* to avoid fast retransmission. When the test was run, the Figure 3 is obtained.

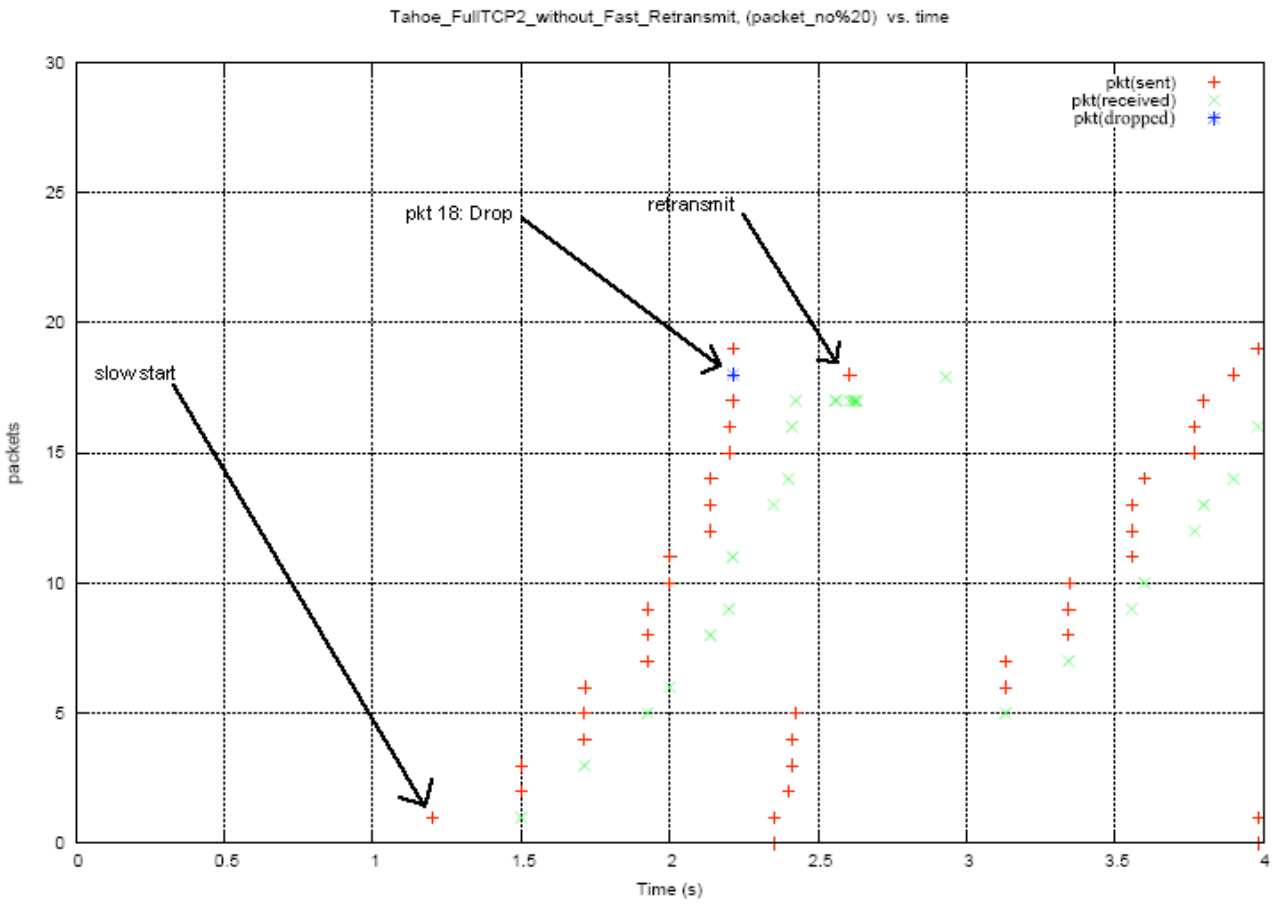

Figure 3: Tahoe_FullTCP2_without_Fast_Retranmit (Not Working properly)

After the simulation starts, the congestion window expands exponentially because of the slow start algorithm. After that, the 17th packet was dropped. Then, the sender receives 7 duplicate acknowledgements for the 16th packet indicating successful delivery of packets (18th -24th).

At this moment, the sender has to wait for the timeout and not retransmit the 17th packet right way after getting the 3rd duplicate ACK (since this test excludes fast retransmit). But Full-TCP incorrectly retransmits after receiving the 3rd duplicate acknowledgement.

Full-TCP is implemented in the file tcp/tcp-full.cc. The function dupack_action() in this file is responsible for the fast retransmit action. Before the function dupack_action() is invoked, it is not checked to see if the variable noFastRetrans_ (which is used to disable fast retransmit) is set. The function extra_ack() is responsible for the

fast recovery. Therefore, we have to check for noFastRetrans_ before invoking it as well. The code snippet looked like the Figure 4 earlier:

```
else if (++dupacks_ == tcprexmtthresh_) {
// ACK at highest_ack_ AND meets threshold
//trace_event("FAST_RECOVERY");
dupack_action(); // maybe fast rexmt
goto drop;

} else if (dupacks_ > tcprexmtthresh_) {
// ACK at highest_ack_ AND above threshole
//trace_event("FAST_RECOVERY");
extra_ack();
```

Figure 4: Full-TCP implementation code in tcp-full.cc (incorrect)

After modifying, it looked like Figure 5:

```
else if (++dupacks_ == tcprexmtthresh_ && !noFastRetrans_) {
// ACK at highest_ack_ AND meets threshold
//trace_event("FAST_RECOVERY");
dupack_action(); // maybe fast rexmt
goto drop;

} else if (dupacks_ > tcprexmtthresh_ && !noFastRetrans_) {
// ACK at highest_ack_ AND above threshole
//trace_event("FAST_RECOVERY");
extra_ack();
```

Figure 5: Full-TCP implementation code in tcp-full.cc (corrected)

After making the necessary changes, when the test was run, it produced the correct result. Figure 6 proves it.

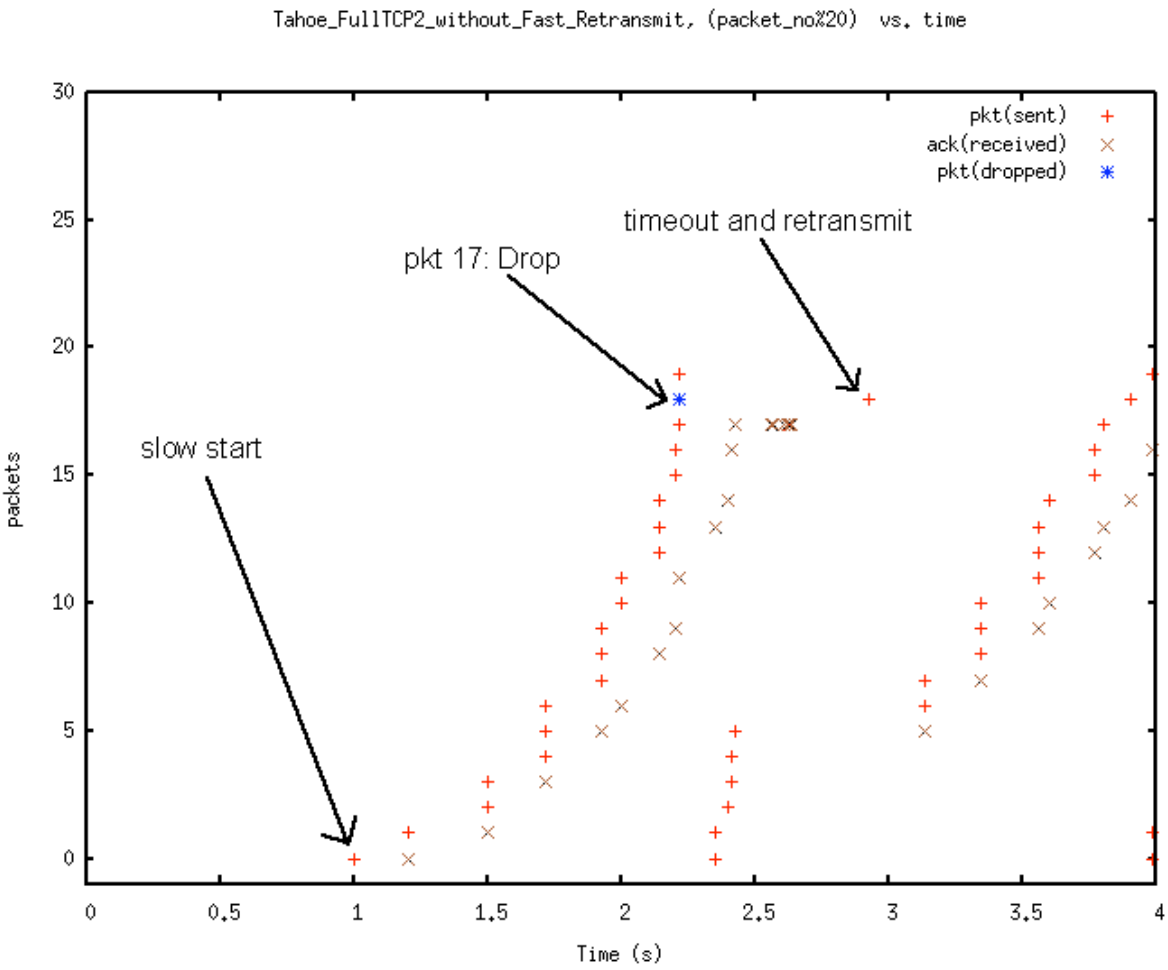

Figure 6: Tahoe_FullTCP2_without_Fast_Retranmit [4]

As we can see in the above graph, after the 17th packet is dropped, the sender waits for the timeout and only after that, retransmits the dropped (17th) packet. Finally, if we can prove that this test produces similar results as its one-way TCP equivalent test, we can rest assure that the test is validated. The equivalent one-way TCP test for this test is "Tahoe_TCP2_without_Fast_retransmit". It produces Figure 7.

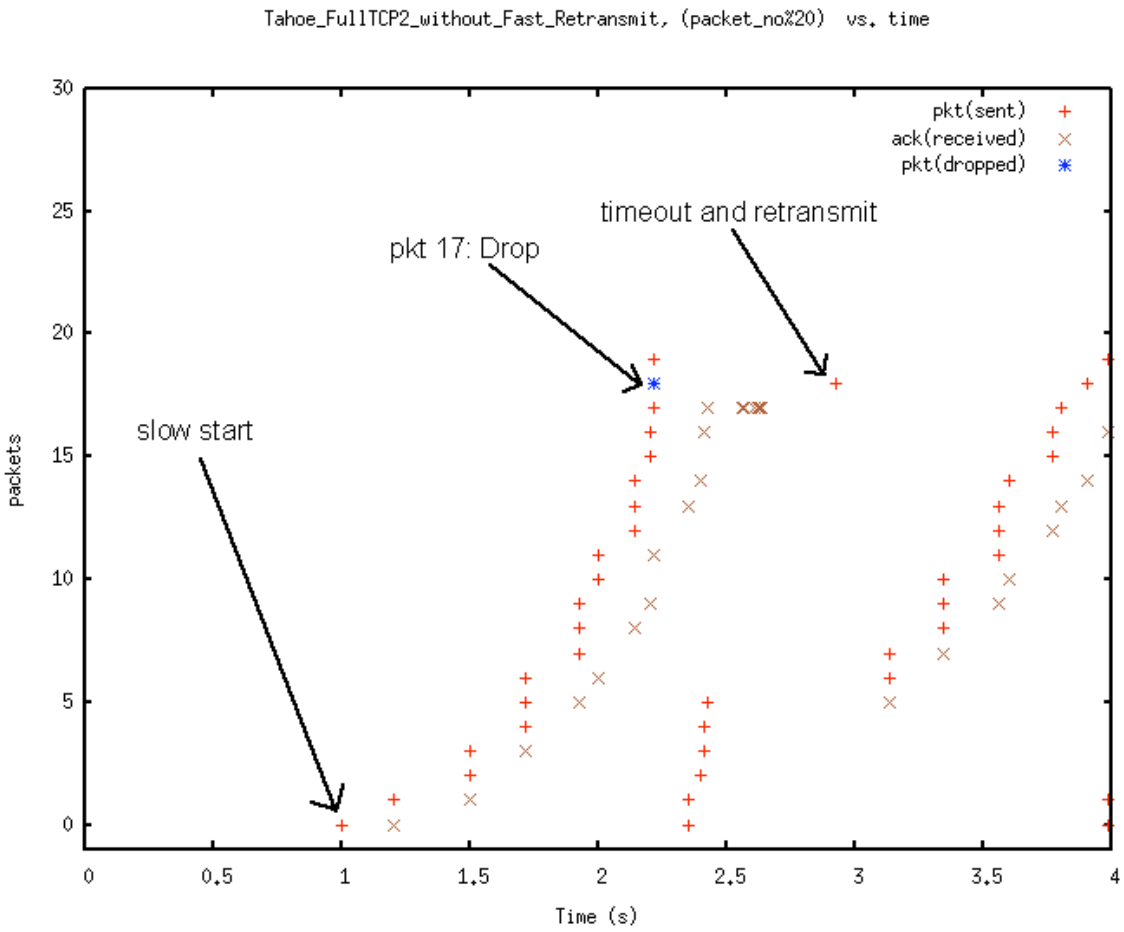

Figure 7: Tahoe_TCP2_without_Fast_Retranmit

We can see that the test Tahoe_FullTCP2_without_Fast_Retranmit produces same results as its one-way TCP equivalent. Let us also compare the congestion windows of both the test to confirm our result.

Figure 8 shows the graph for congestion window for the test Tahoe_FullTCP2_without_ Fast_Retransmit Vs Tahoe_TCP2_without_ Fast_Retransmit.

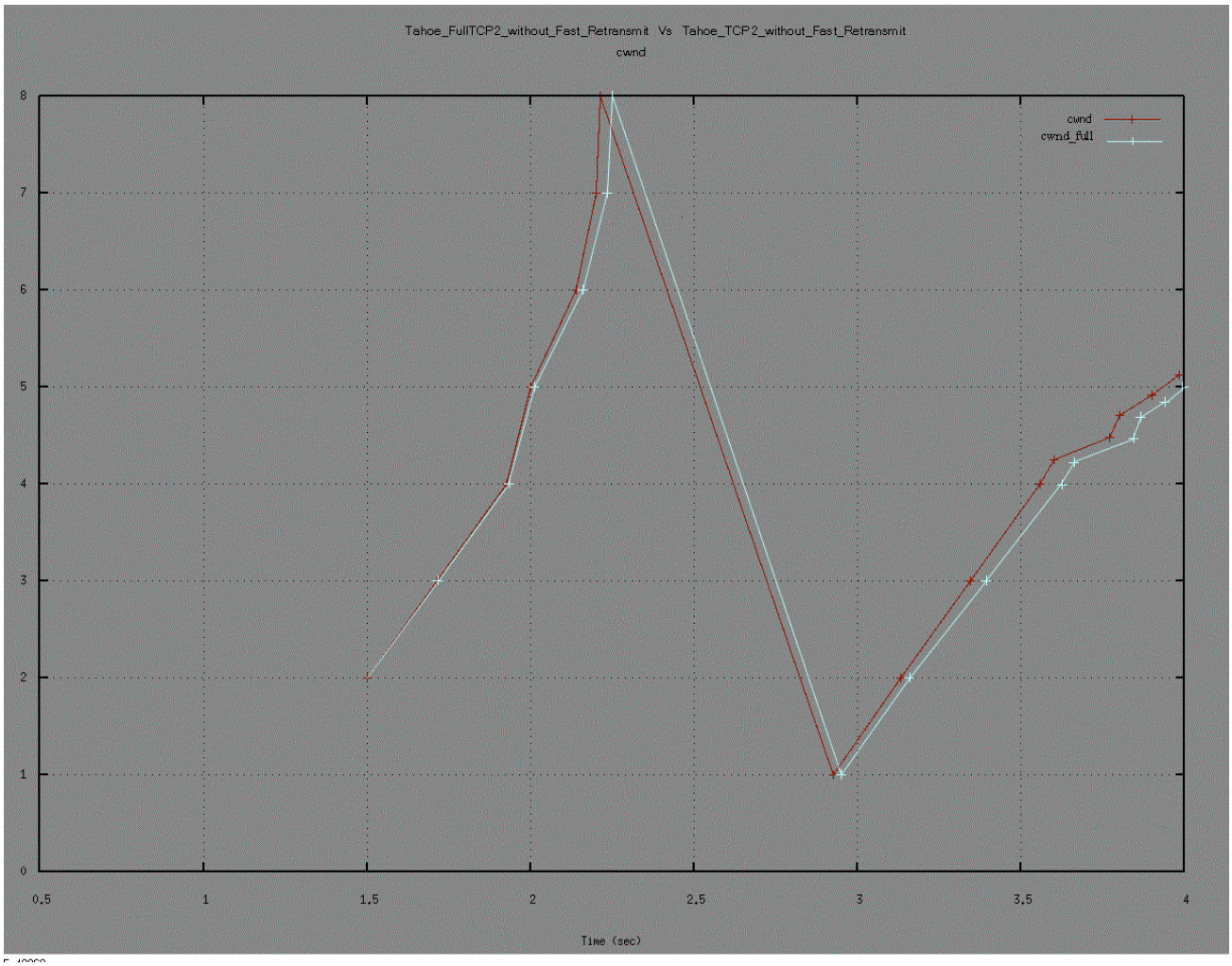

Figure 8: congestion window graph for the test
Tahoe_FullTCP2_without_Fast_Retransmit VS Tahoe_TCP2_without_Fast_Retransmit

The congestion window exponentially increases until 8 after the 3rd non-overlapping window of data. After the drop, the cwnd is set to 1 and slow start is initiated because of the time out occurrence. Later, the cwnd increases exponentially (slow start) until it reaches the threshold (ssthresh=8/2 = 4) and then, linearly increases from there (Congestion Avoidance). As we can see, both tests have the same congestion window graph. The packet trace dump (tcl/test/all.tr) also shows accurate behavior. Hence we can claim that the test Tahoe_FullTCP2_without_Fast_Restransmit has been validated.

3.3.2 Test: Tahoe_FullTCP_without_Fast_Retransmit

This test uses Tahoe TCP and has two drops (15th and 18th packets). As the name suggests, this test also, is not supposed to use the fast retransmit algorithm. That is, after

receiving 3 duplicate ACKs for the 14th packet, the sender should not retransmit packet 15 and should wait until timeout. This test also, like the previous one, uses a variable noFastRetrans_ to avoid fast retransmission (Figure 9).

```
Test/Tahoe_FullTCP_without_Fast_Retransmit instproc init {} {
        $self instvar net_ test_
        set net_      net4
        set test_     Tahoe_FullTCP_without_Fast_Retransmit
        Agent/TCP set noFastRetrans_ true
        Agent/TCP/FullTcp set segsperack_ 2
        $self next
}
```

Figure 9: code for the test Tahoe_FullTCP_without_Fast_Retransmit

When the test was run, Figure 10 was obtained.

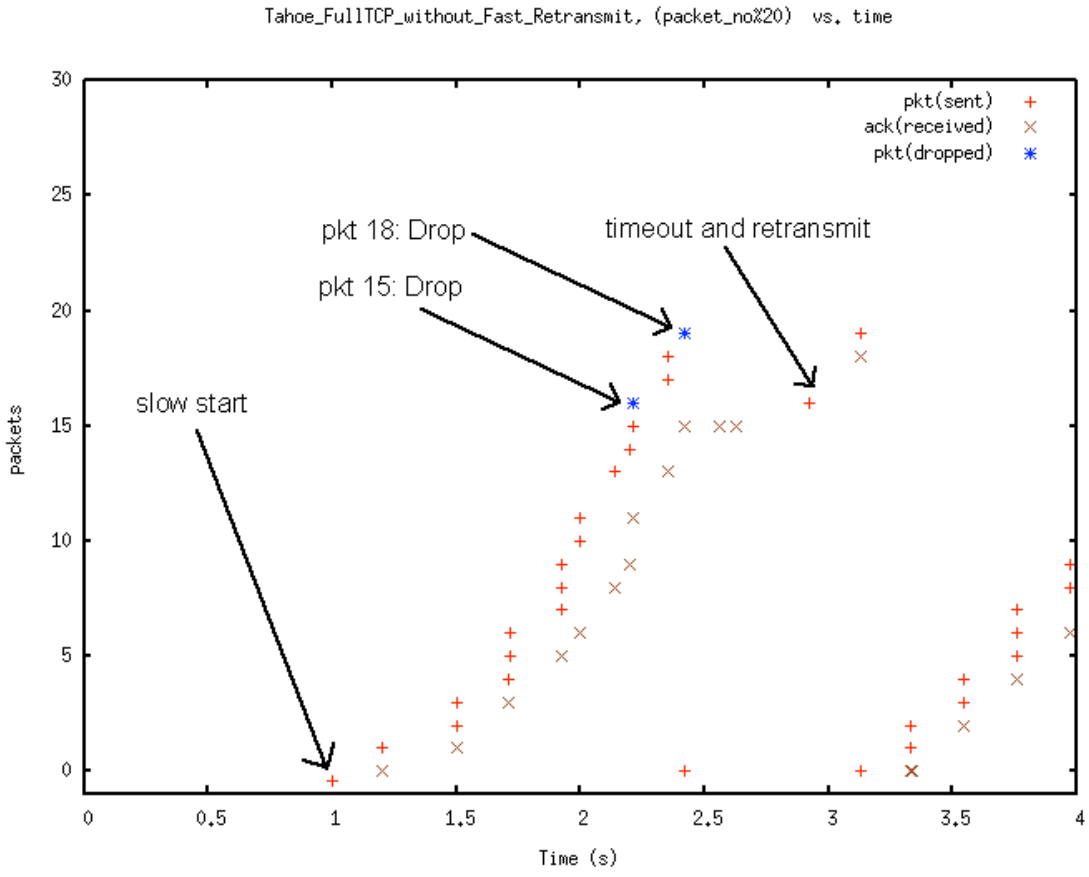

Figure 10: Tahoe_FullTCP_without_Fast_Retransmit

The packets 15 and 18 are dropped. Now that the Full-TCP implementation is corrected to check noFastRetrans_ variable, the sender waits for the timeout. Its congestion window is set to 1 again. Later, the sender retransmits packet 15 and packet 18.

Its equivalent one-way TCP test is Tahoe_TCP_without_Fast_Retransmit. It produces the Figure 11.

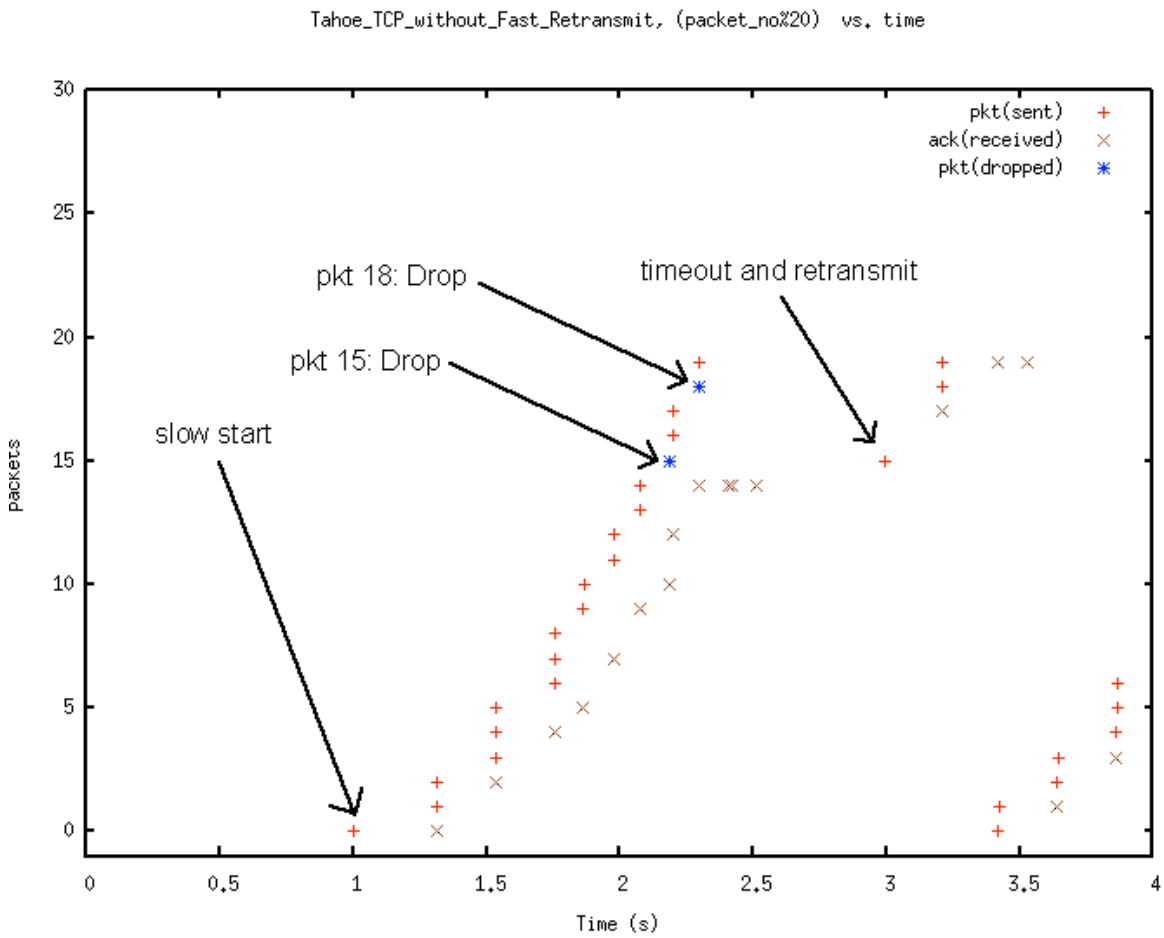

Figure 11: Tahoe_TCP_without_Fast_Retransmit

The graph for congestion window for the test Tahoe_FullTCP_without_Fast_Retransmit verses Tahoe_TCP_without_Fast_Retransmit is shown in Figure 12.

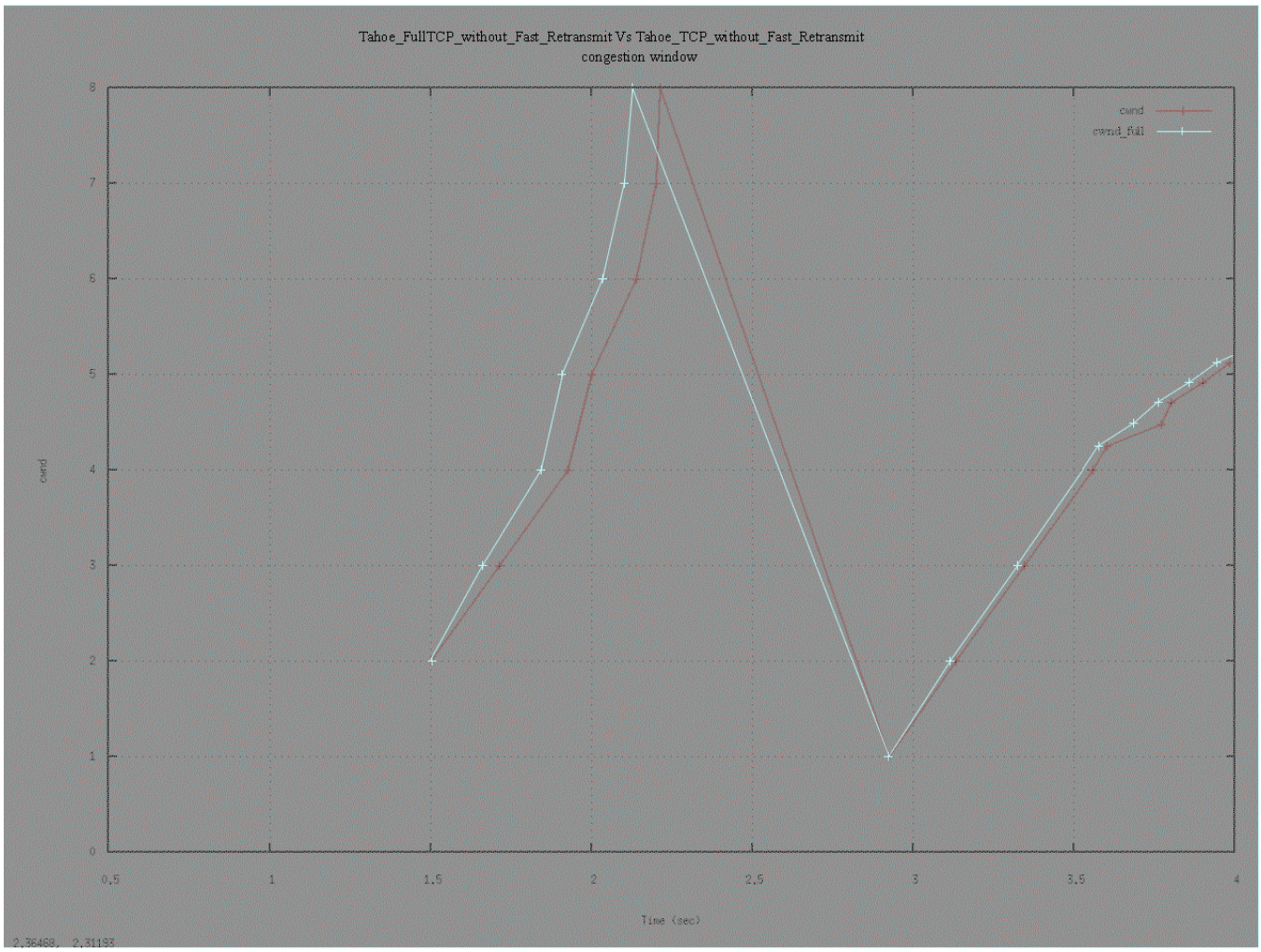

Figure 12: Congestion window for Tahoe_FullTCP_without_Fast_Retransmit Vs Tahoe_TCP_without_Fast_Retransmit

The congestion window increases exponentially according to the slow start, until the 15th packet is dropped. The congestion window drops to maximum segment size (1) and increases to 2 after receiving ACK for 15th packet and increases exponentially until it reaches the ssthresh and later increases accordingly after that. We can see that the both the tests have similar congestion window action. Hence, we can say that the test Tahoe_TCP_without_Fast_Retransmit is validated.

3.3.3 Test: multiple_tahoe_full

This test uses Tahoe TCP with multiple drops at packets 12, 13, 14, 15, 17, 18, 19 and 20. The test does not restrict and add anything new to Tahoe TCP. Therefore, the test is supposed to begin with slow start and fast retransmit after receiving 3rd duplicate ACK.

Figure 13 shows the graph obtained after running this test.

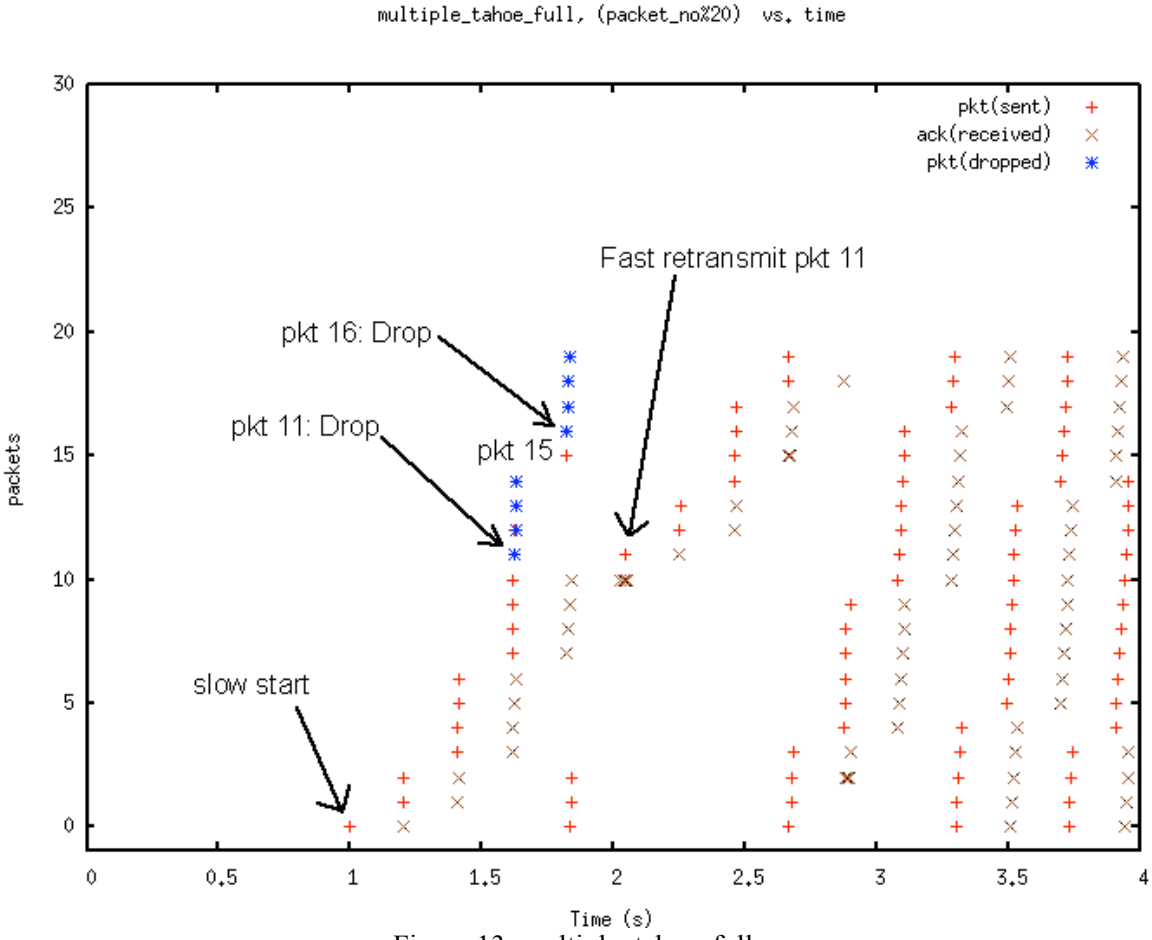

Figure 13: multiple_tahoe_full

cwnd starts with 1 and exponentially increases to 11, because ACKs for all packets until 10th (inclusive) have been received. Packets 11, 12, 13, 14, 16, 17, 18, 19 are dropped. On receiving 3rd duplicate ACK for packet 10 (which is the 4th ACK for packet 10), the threshold (ssthresh =12/2=6) is set to half the congestion window and packet 11 is retransmitted. After receiving the ACK for packet 11, the congestion window is set to 1 and slow start is initiated. After that, the congestion window increases exponentially until it reaches threshold (ssthresh=6) and later increases linearly according to congestion avoidance.

Let us compare this with its one-way TCP equivalent to prove its accuracy. The one-way TCP equivalent to multiple_tahoe_full is multiple_tahoe which is in the same test suite. Figure14 shows the graph obtained after running multiple_tahoe.

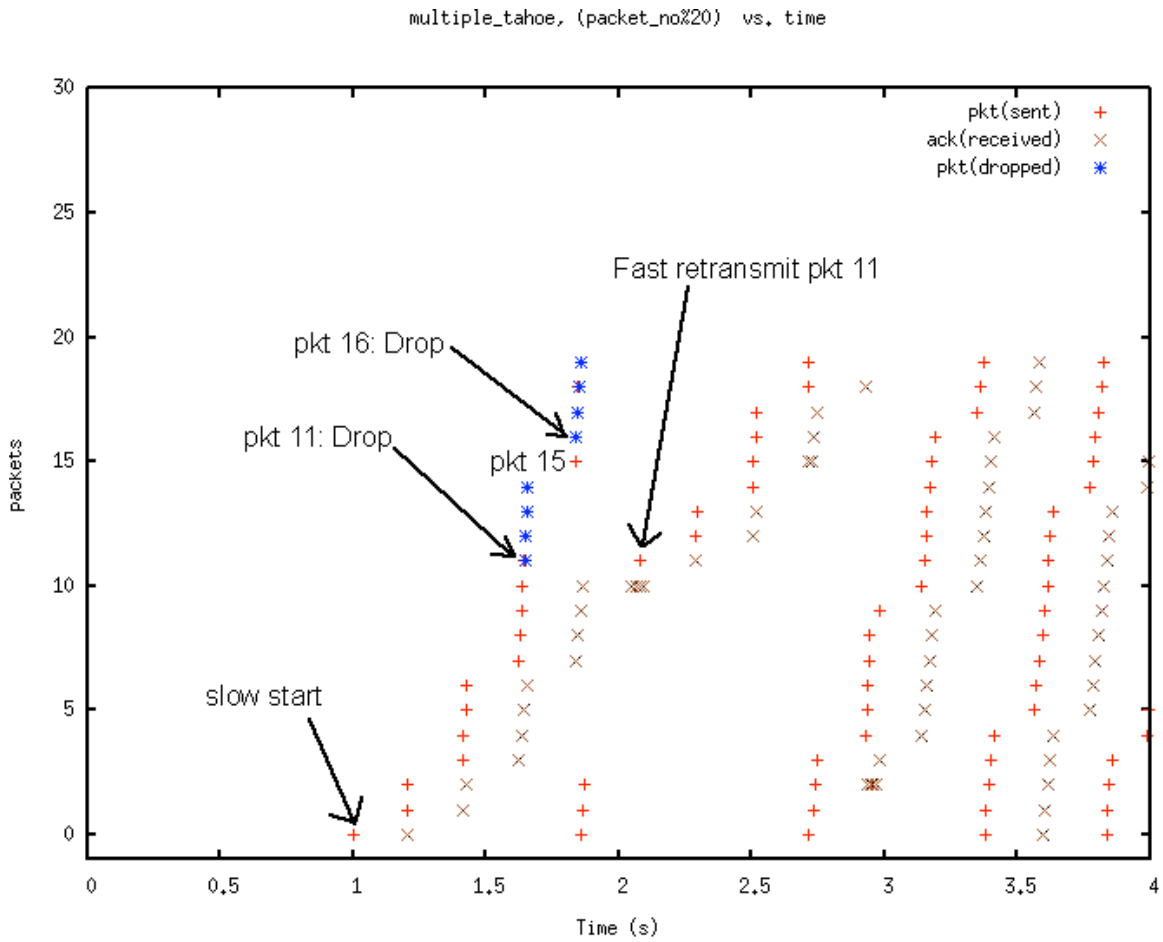

Figure 14: multiple_tahoe

The congestion window graph is shown in Figure 15.

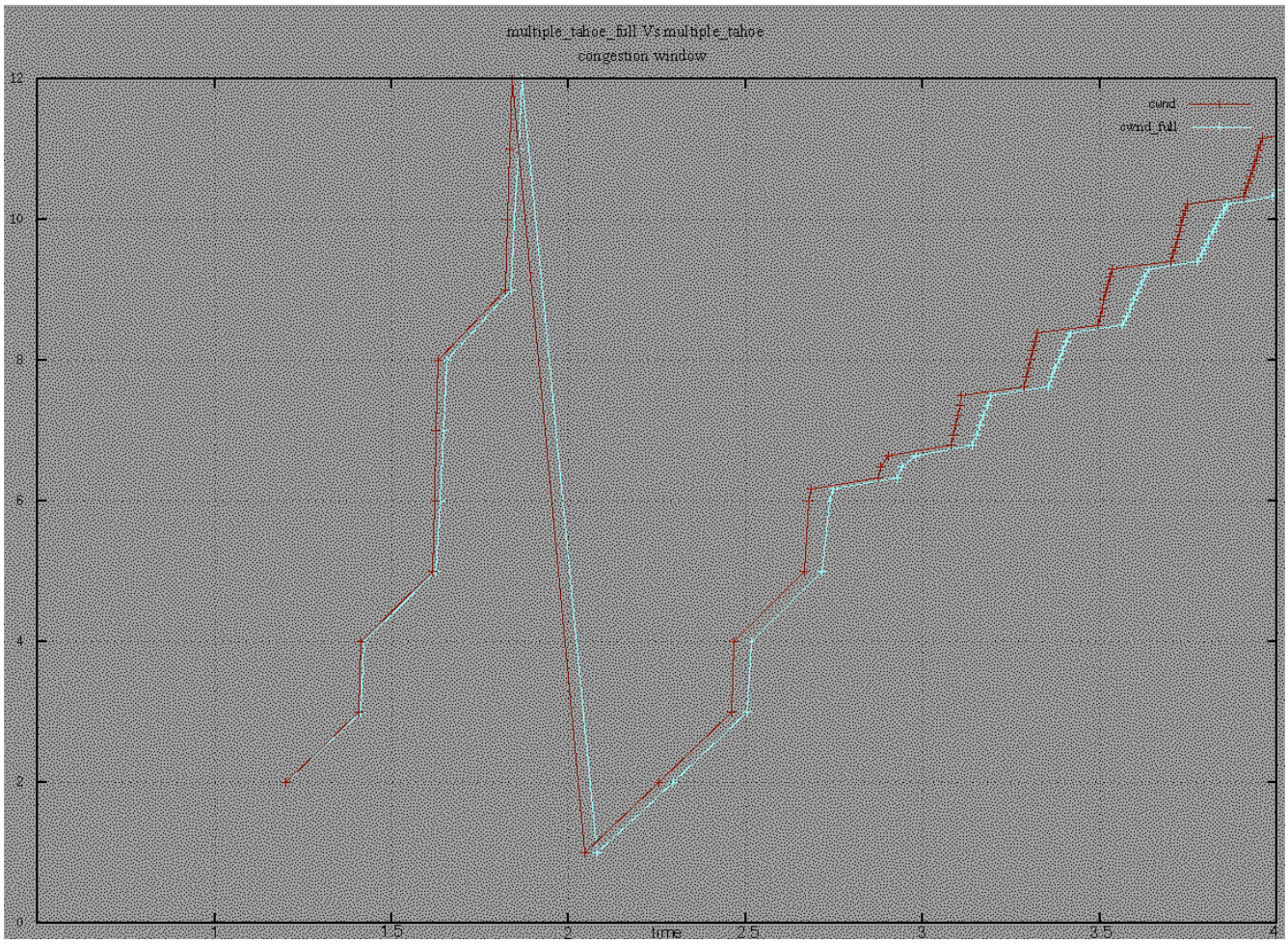

Figure 15: Congestion window for multiple_tahoe_full Vs multiple_tahoe.

As we can clearly see, both multiple_tahoe_full and multiple_tahoe give similar output graphs proving the point that multiple_tahoe_full is accurate. Thus we can claim that it has been validated.

3.3.4 Test: multiple2_tahoe_full

This test uses Tahoe TCP with multiple drops at packets 12, 13, 14, 15, 17. The test does not restrict and add anything new to Tahoe TCP. Therefore, the test is supposed to begin with slow start and fast retransmit after receiving 3rd duplicate ACK. Figure 16 shows the graph obtained after running this test.

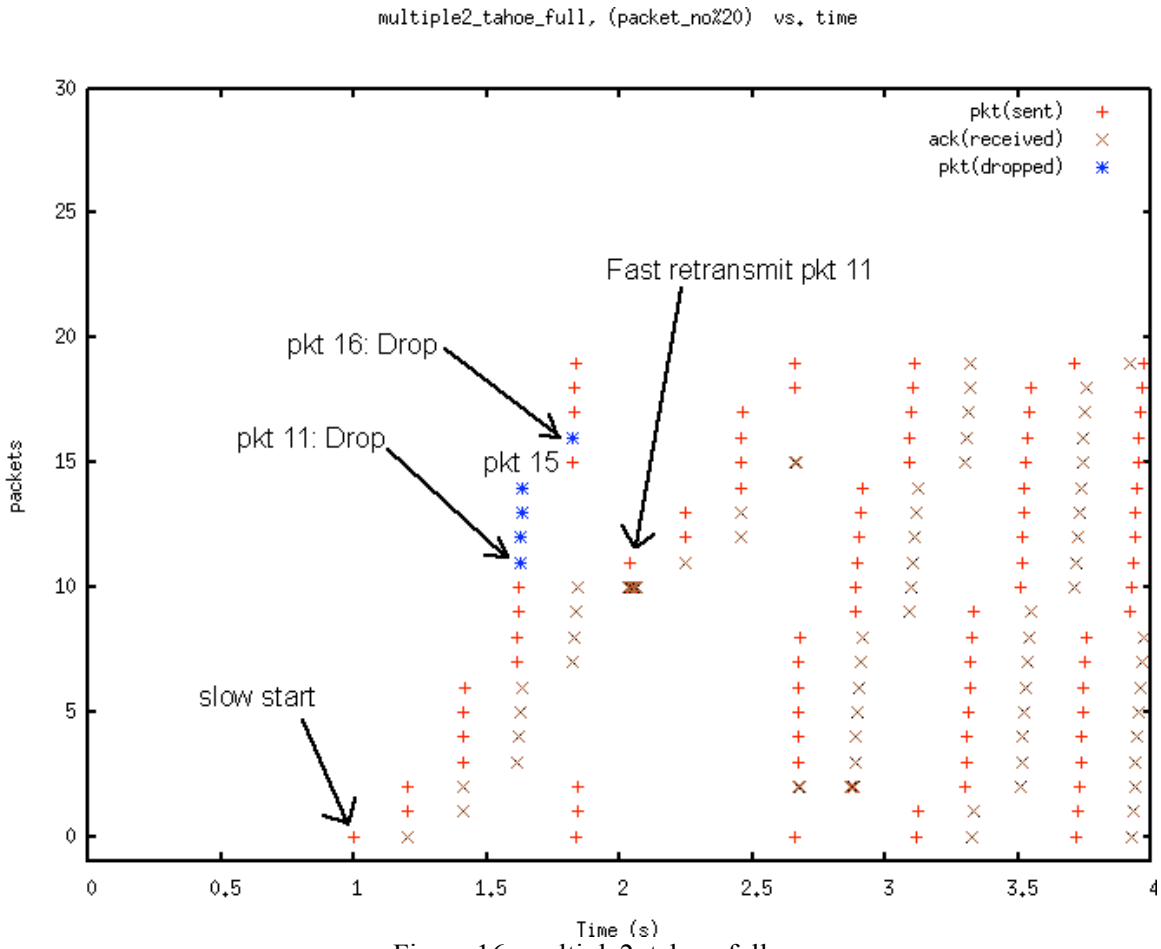

Figure 16: multiple2_tahoe_full

cwnd starts with 1 and exponentially increases to 11, since ACKs for all packets until 10th (inclusive) have been received. Packets 11, 12, 13, 14, 16 are dropped. On receiving 3rd duplicate ACK for packet 10 (which is the 4th ACK for packet 10), the threshold (ssthresh =12/2=6) is set to congestion window /2 and packet 11 is retransmitted. After receiving the ACK for packet 11, the congestion window is set to 1 and slow start is initiated. After that, congestion window increases exponentially until it reaches threshold (ssthresh=6) and later increases linearly according to congestion avoidance.

Let us compare this with its one-way TCP equivalent to prove its accuracy. The one-way TCP equivalent to multiple2_tahoe_full is multiple2_tahoe which is in the same test suite Figure 17 shows the graph obtained after running the test multiple2_tahoe.

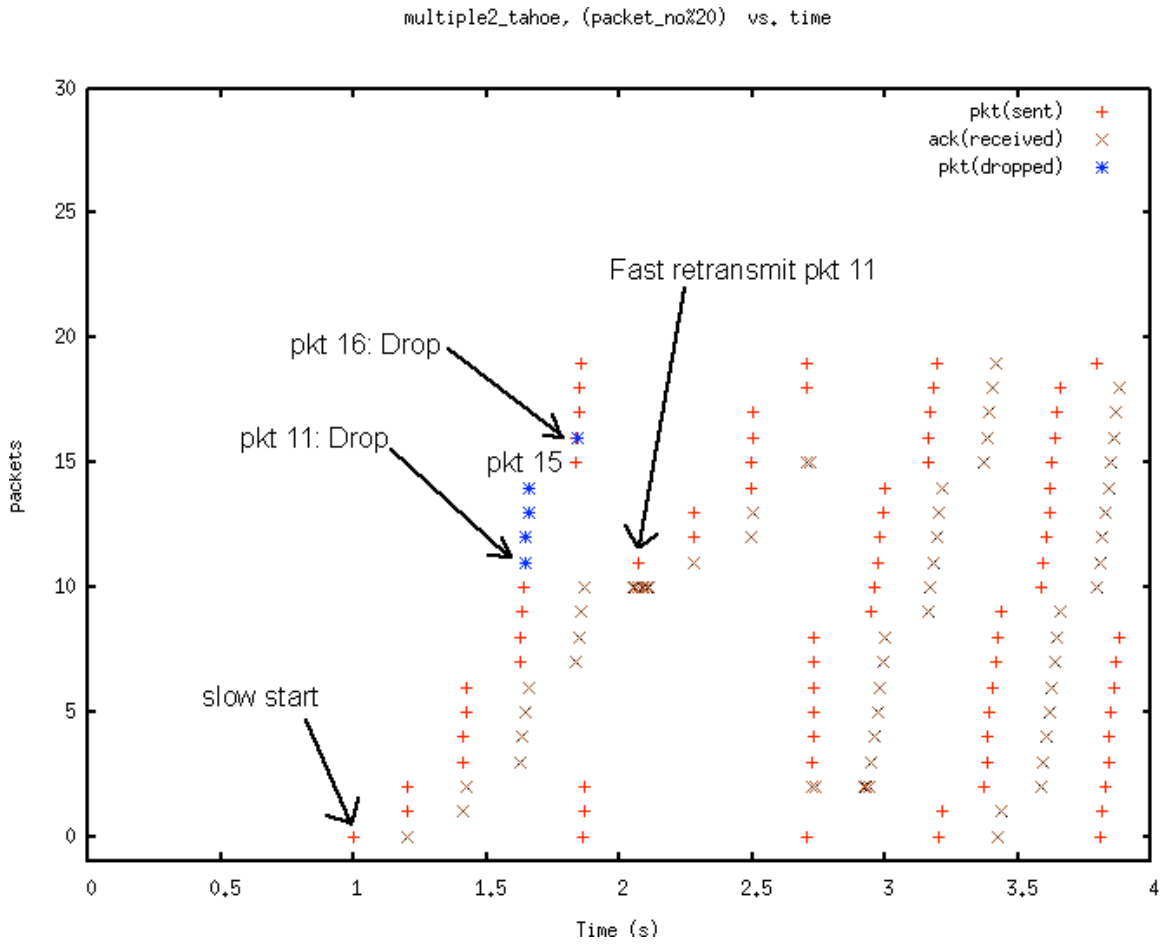

Figure 17: multiple2_tahoe

Figure 18 shows the congestion window graph.

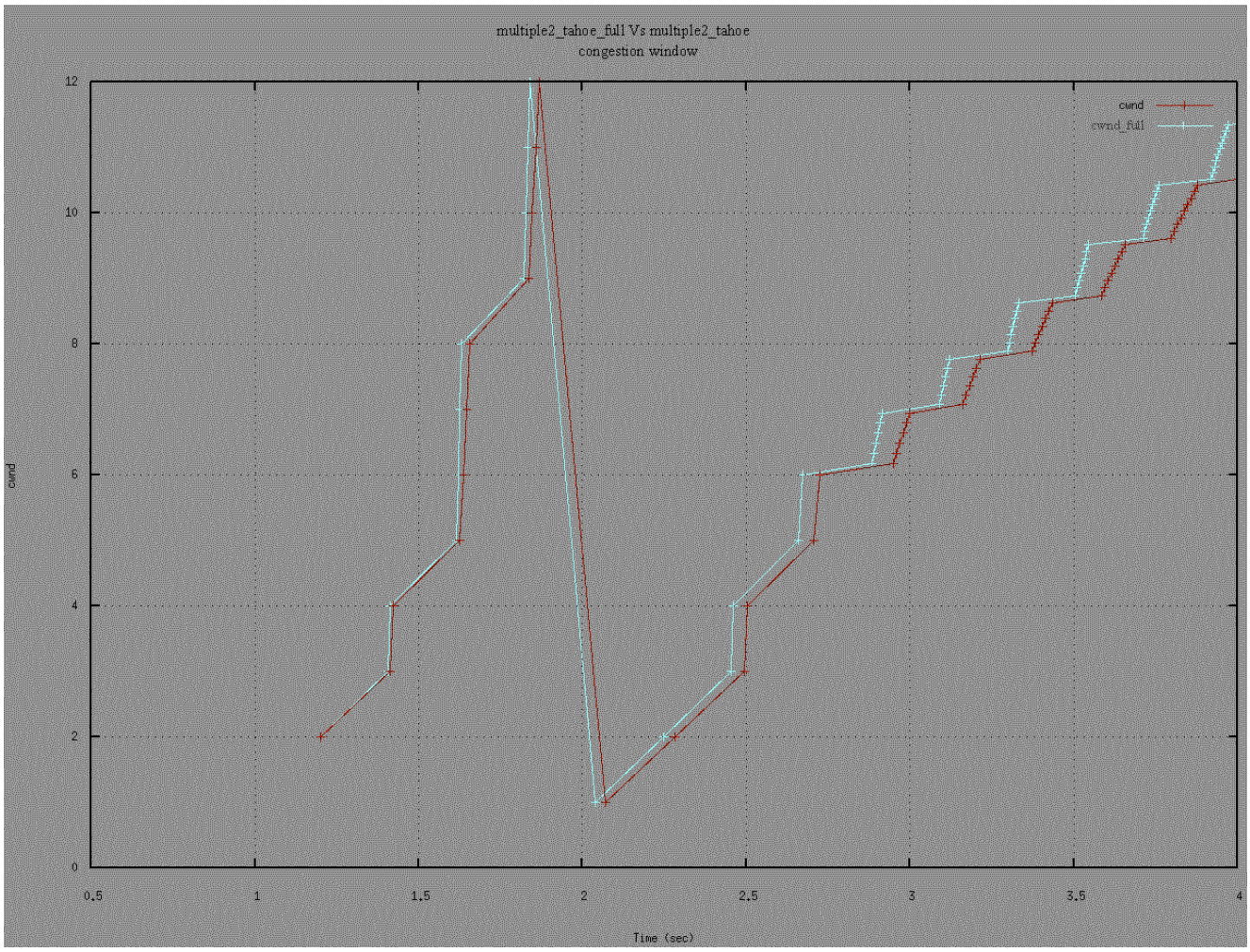

Figure 18: Congestion window for mulitple2_tahoe mulitple2_tahoe Vs mulitple2_tahoe

Both tests give same output which gives us the liberty to certify the test mulitple2_tahoe_full to be validated.

3.3.5 Test: multiple_reno_full

This test uses Reno TCP with multiple drops at packets 12, 13, 14, 15, 17, 18, 19 and 20. The test is supposed to begin with slow start and fast retransmit after receiving 3rd duplicate ACK and also use fast recovery algorithm. Figure 19 shows the graph obtained after running this test.

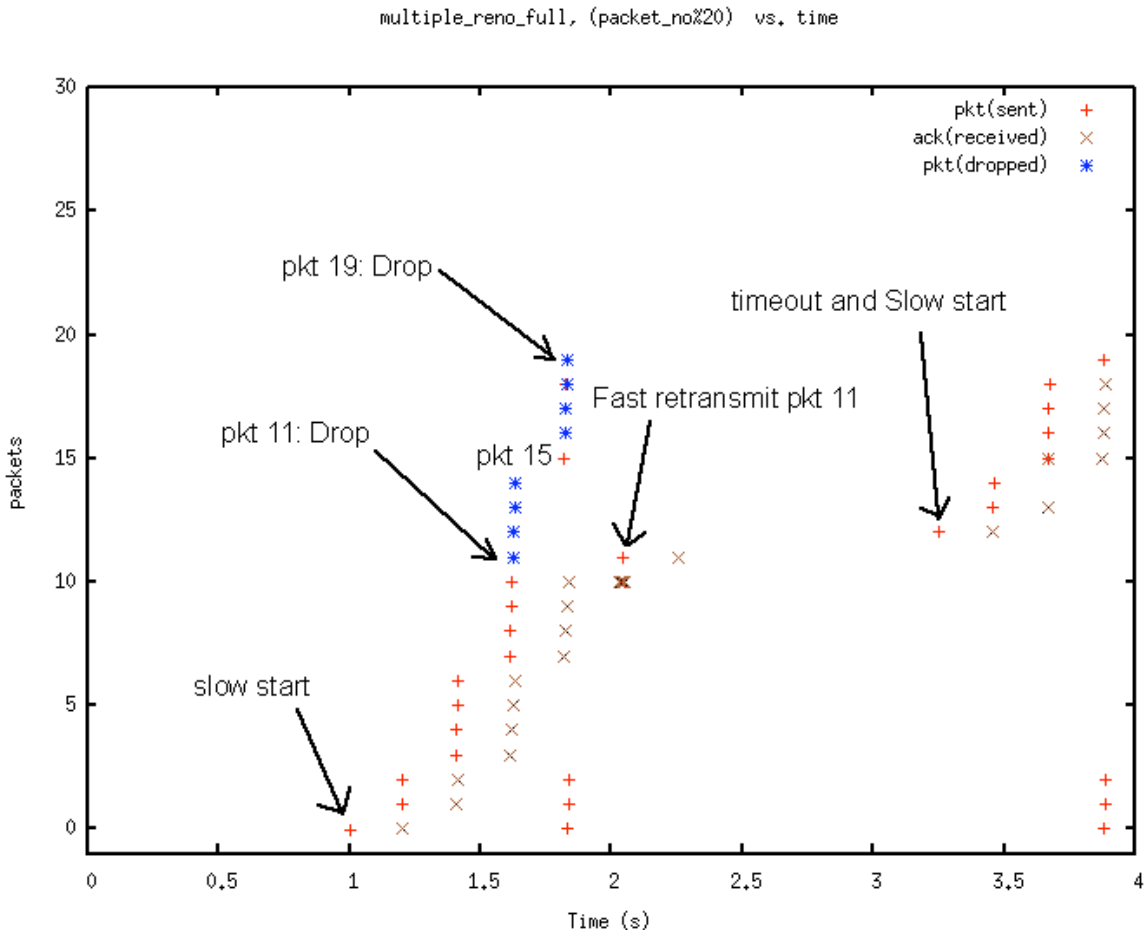

Figure 19: multiple_reno_full [5]

The sender starts with cwnd 1 and exponentially increases to 11, because ACKs for all packets until 10th (inclusive) have been received. Packets 11, 12, 13, 14, 16,17,18,19 are dropped. On receiving 3rd duplicate ACK for packet 10 (which is the 4th ACK for packet 10), ssthresh is set to congestion window /2 (ssthresh =12/2=6). cwnd is set to threshold (ssthresh+3). Then packet 11 is retransmitted. As stated earlier, Reno does not recover from multiple drops and has to wait for the retransmission timer to expire.

After receiving the ACK for packet 11 (new non-dup ACK), the usable window is "deflated" to ssthresh. Reno is now supposed to increases the congestion window linearly from now on with arrival of new ACKs. But there is no new ACK after ACK-11 (because, there were multiple drops after packet 11). Therefore, Reno waits will the timeout. After the retransmission timer expires, the congestion window is set to 1 and

slow start is initiated. congestion window increases exponentially until it reaches threshold (ssthresh=6) and later increases linearly according to congestion avoidance.

The equivalent one-way TCP test for this test is multiple_reno which is present in the same test suite. It produces a graph shown in Figure 20.

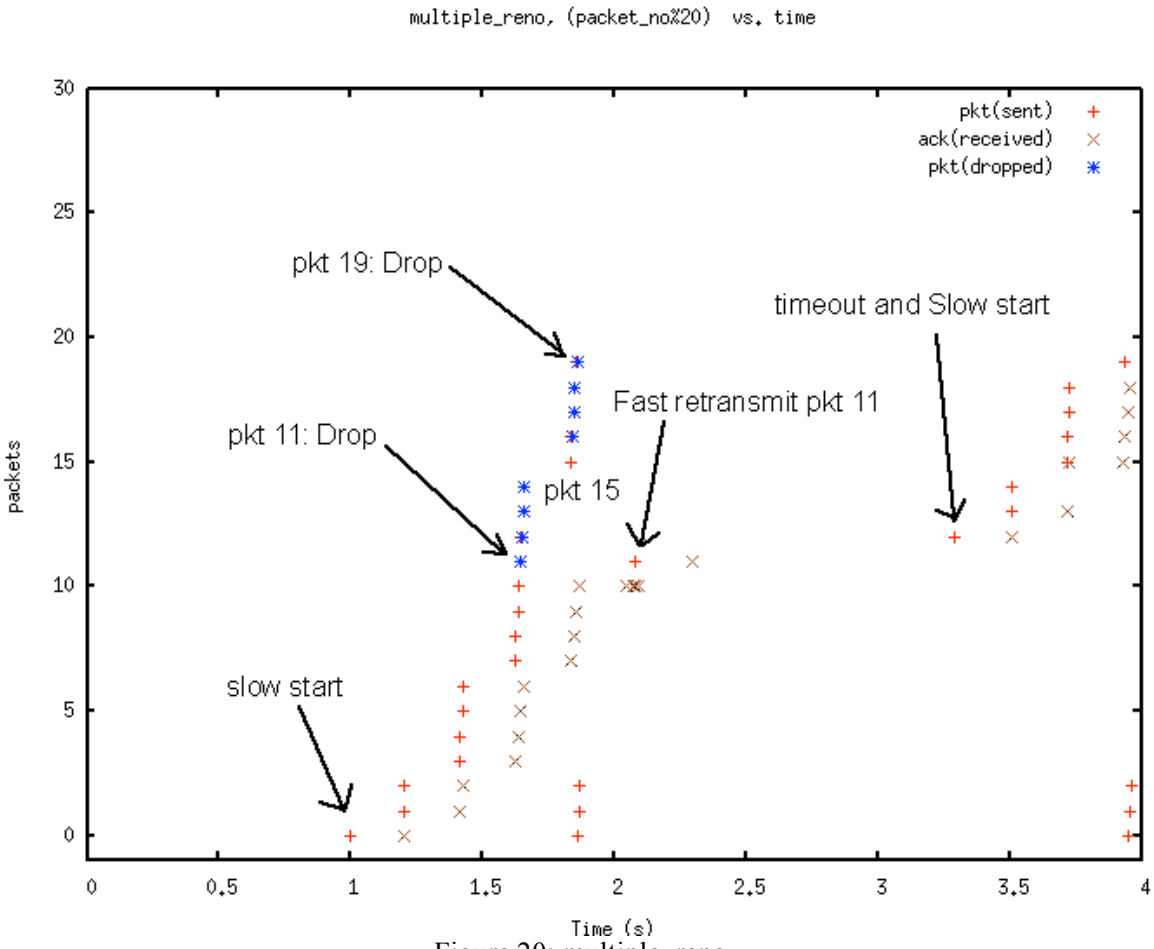

Figure 20: multiple_reno

The congestion window graph for multiple_reno_full Vs multiple_reno is shown in Figure 21.

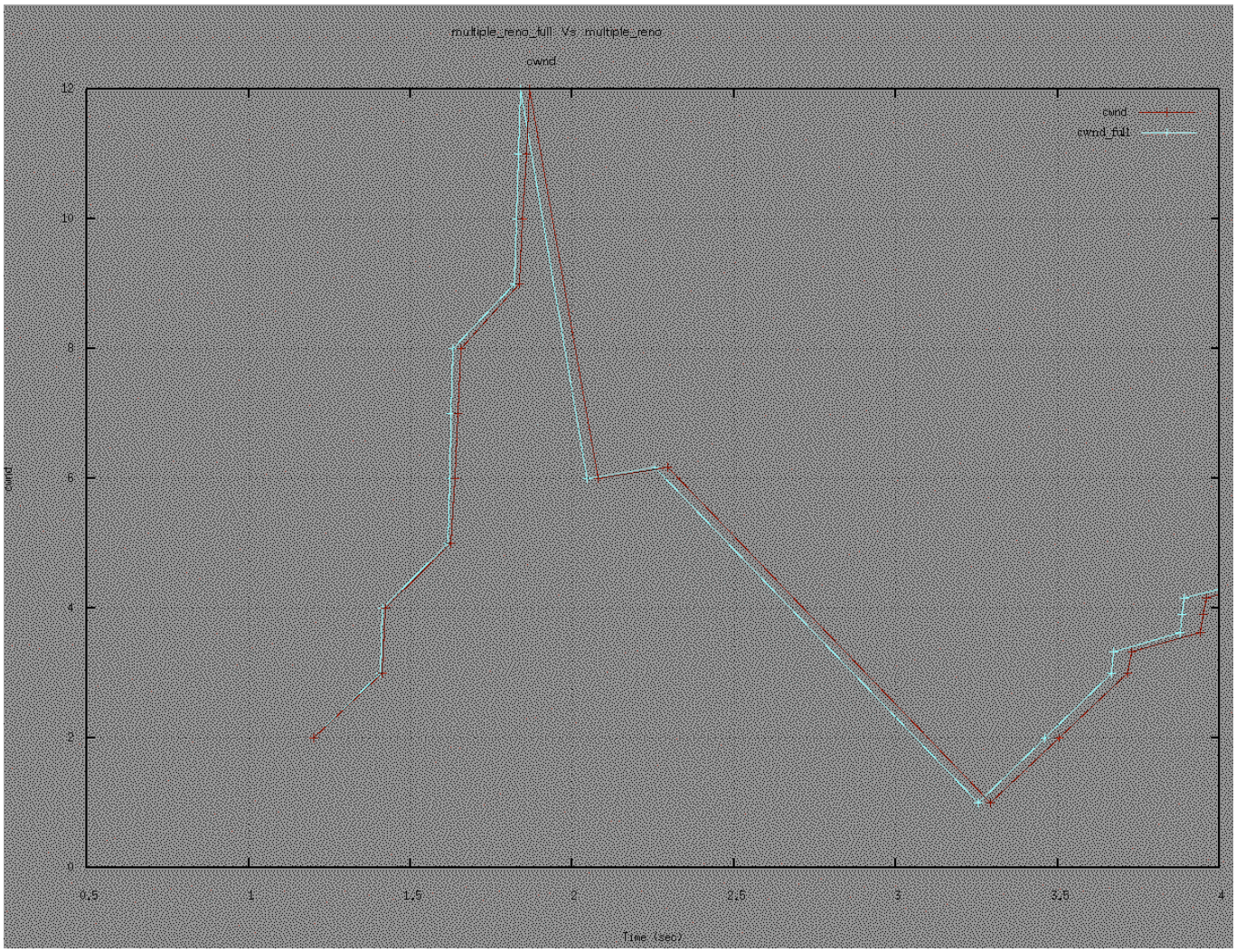

Figure 21: congestion window for multiple_reno_full Vs multiple_reno

As we can see from the graphs, both multiple_reno_full and multiple_reno produce similar results. Hence we can say that the test multiple_reno_full is validated.

3.3.6 Test: multiple2_reno_full

This test uses Reno TCP with multiple drops at packets 12, 13, 14, 15, 17. The test is supposed to begin with slow start and fast retransmit after receiving 3rd duplicate ACK and also use fast recovery algorithm.

Figure 22 shows the graph obtained after running this test.

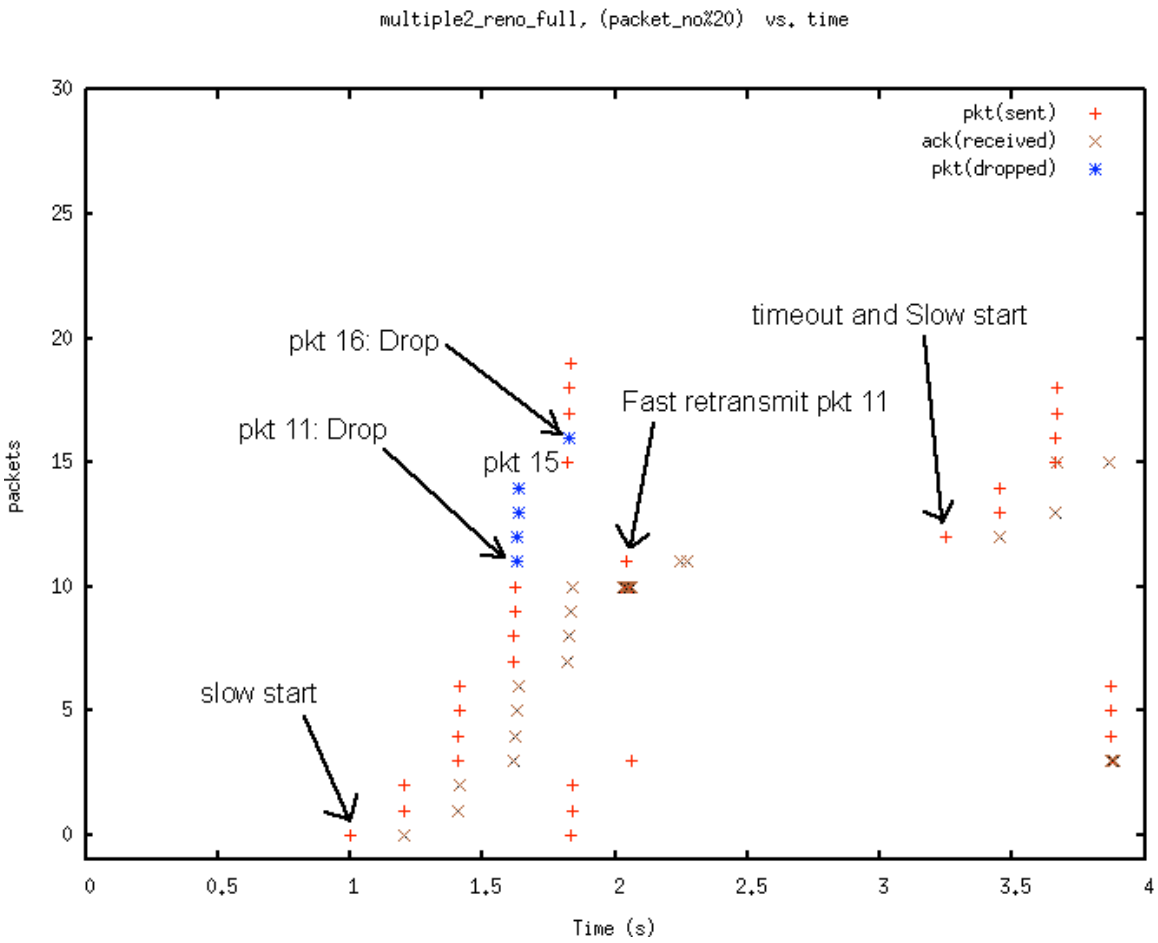

Figure 22: multiple2_reno_full

The sender starts with a cwnd 1 and exponentially increases to 11, because ACKs for all packets until 10th (inclusive) have been received. Packets 11, 12, 13, 14, 16 are dropped. On receiving 3rd duplicate ACK for packet 10 (which is the 4th ACK for packet 10), the ssthresh is set to congestion window/2 (ssthresh = 12/2 = 6). cwnd is set to threshold (ssthresh+3). Then packet 11 is retransmitted. As stated earlier, Reno does not recover from multiple drops and has to wait for the retransmission timer to expire.

After receiving the ACK for packet 11 (new non-dup ACK), the usable window is "deflated" to ssthresh. Reno is now supposed to increases the congestion window linearly from now on with arrival of new ACKs. But there is no new ACK after ACK-11 (because, there were multiple drops after packet 11). Therefore, Reno waits will the timeout. After the retransmission timer expires, the congestion window is set to 1 and slow start is initiated. congestion window increases exponentially until it reaches threshold (ssthresh=6) and later increases linearly according to congestion avoidance.

The equivalent one-way TCP test for this test is multiple2_reno which is present in the same test suite. It gives the graph shown in Figure 23.

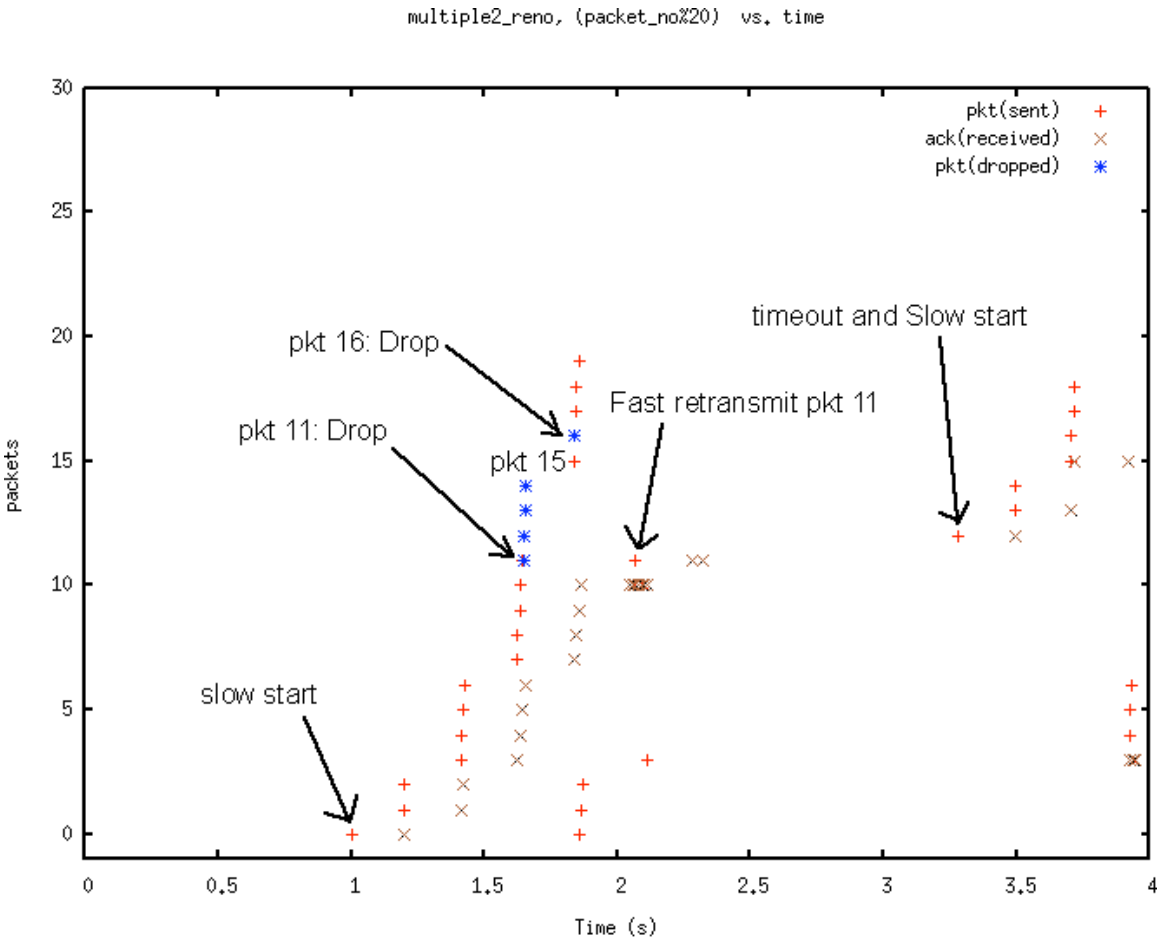

Figure 23: multiple2_reno

The graph for congestion window is shown in Figure 24.

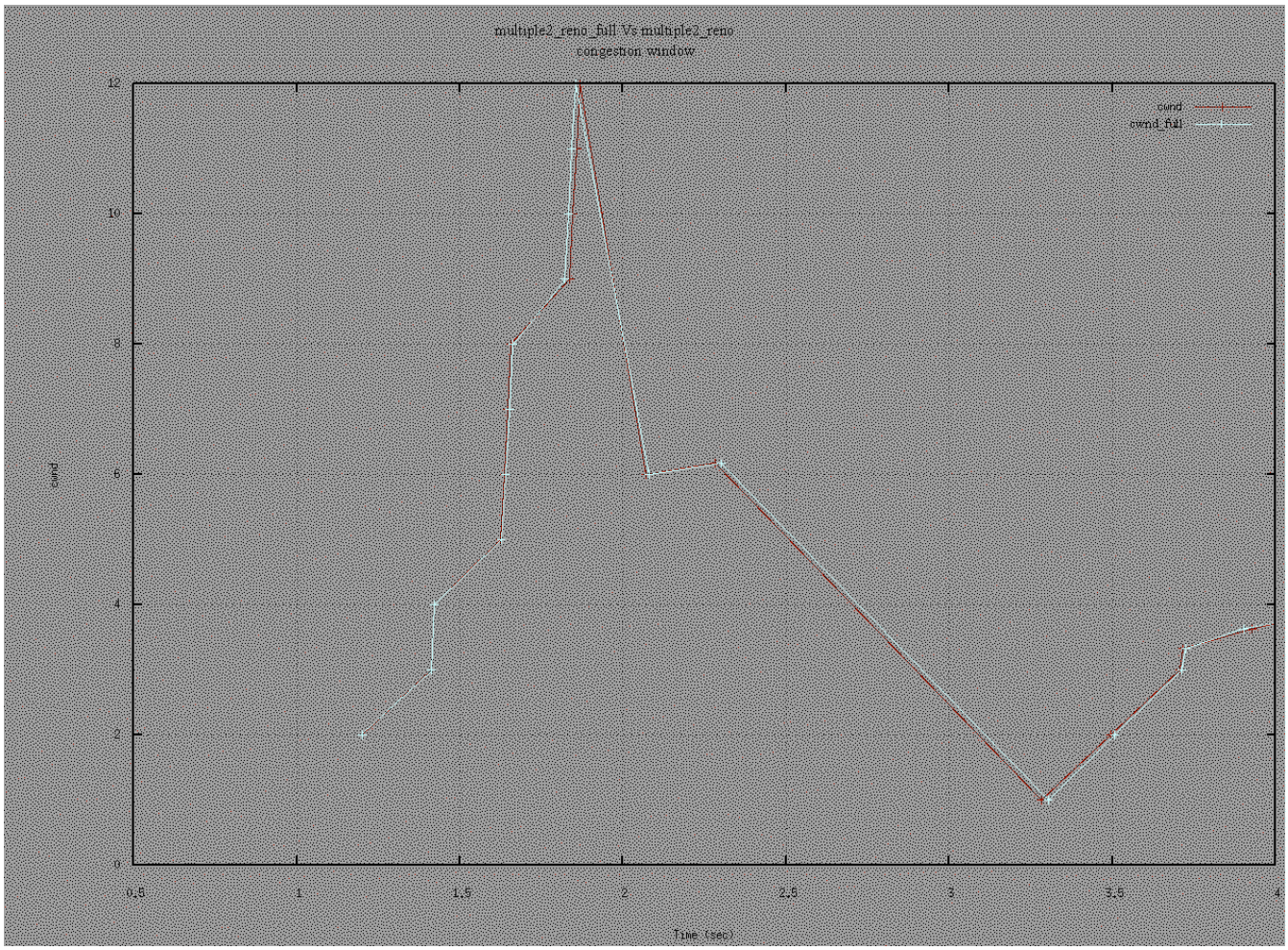

Figure 24: Congestion window for multiple2_reno_full Vs multiple2_reno

As we can see from the graphs, both multiple2_reno_full and multiple2_reno produce similar results. Hence we can say that the test multiple2_reno_full is validated.

3.3.7 Test: multiple_newreno_full

This test uses New-Reno TCP with multiple drops at packets 12, 13, 14, 15, 17, 18, 19 and 20. The test is supposed to begin with slow start and fast retransmit after receiving 3rd duplicate ACK. It should use fast recovery algorithm and consider partial acknowledgements as well. Figure 25 shows the graph obtained after running this test.

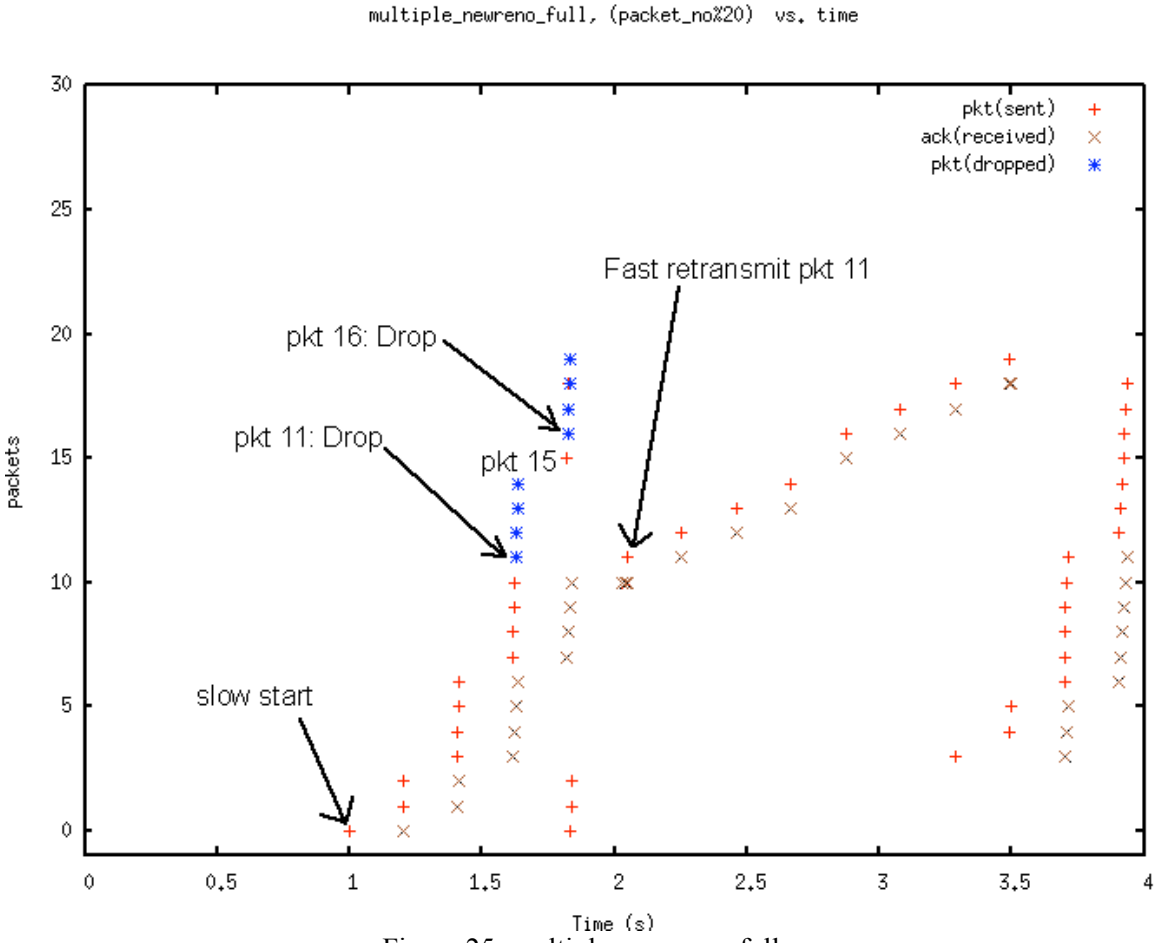

Figure 25: multiple_newreno_full

The sender starts with a cwnd 1 and exponentially increases to 11, because ACKs for all packets until 10th (inclusive) have been received. Packets 11, 12, 13, 14, 16, 17, 18, 19 are dropped. On receiving 3rd duplicate ACK for packet 10 (which is the 4th ACK for packet 10), the ssthresh is set to half the congestion window (ssthresh =12/2=6). cwnd is set to threshold (ssthresh+3). Then packet 11 is retransmitted. As stated earlier, Reno does not recover from multiple drops and has to wait for the retransmission timer to expire.

The ACK for packet 11 (new non-dup ACK) is a partial ACK. It indicates the sender that the packet after the 11th may have been lost. Therefore, the sender sends one dropped packet for round trip time until all the dropped packets are sent successfully. The usable window is not "deflated" to ssthresh after receiving ACK 11. We can see in the graph above that after receiving ACK 11, slow start is not initiated. Rather, one dropped packet is retransmitted per one round trip time until all the dropped packets have been resent successfully.

The equivalent one-way TCP test for this test is multiple_newreno. After running it, the graph obtained is shown in Figure 26.

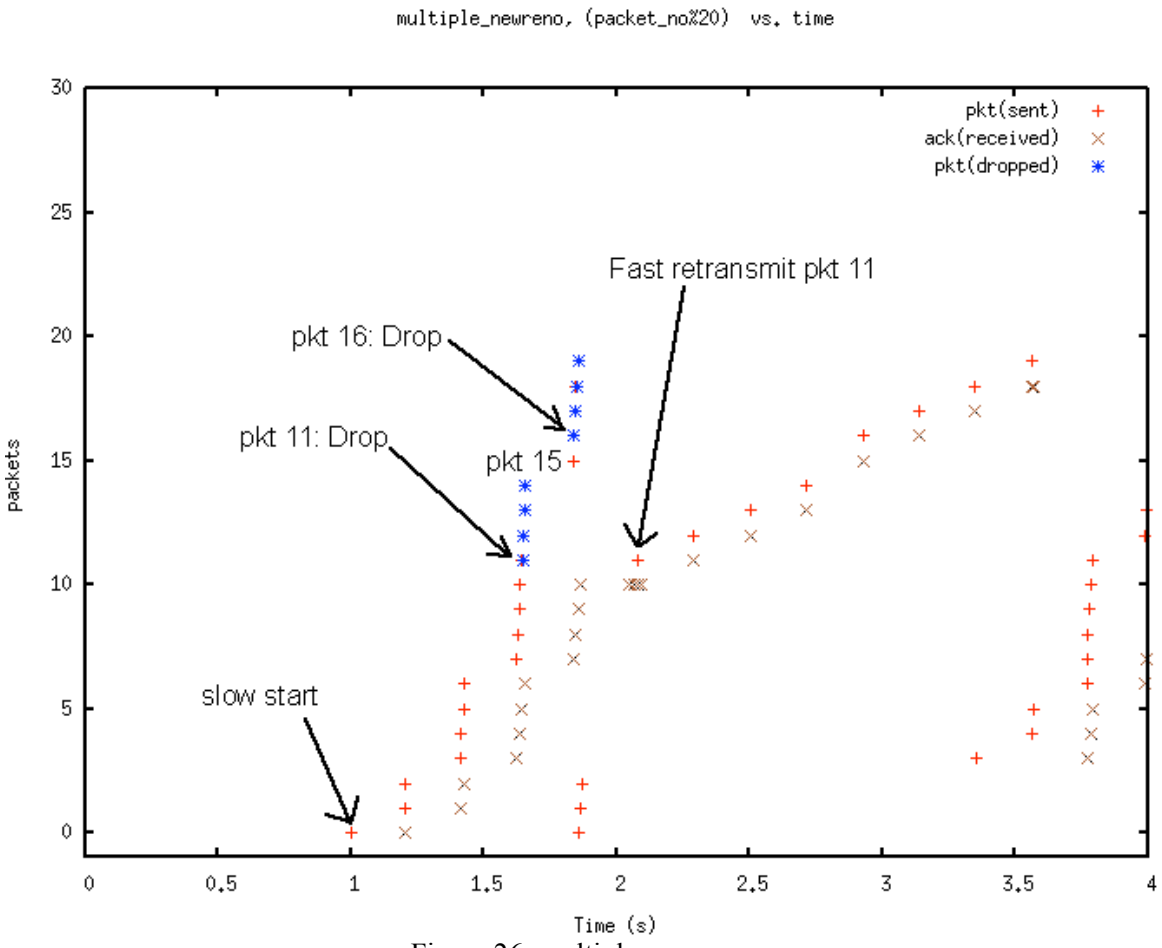

Figure 26: multiple_newreno

The congestion window for the test is shown in Figure 27.

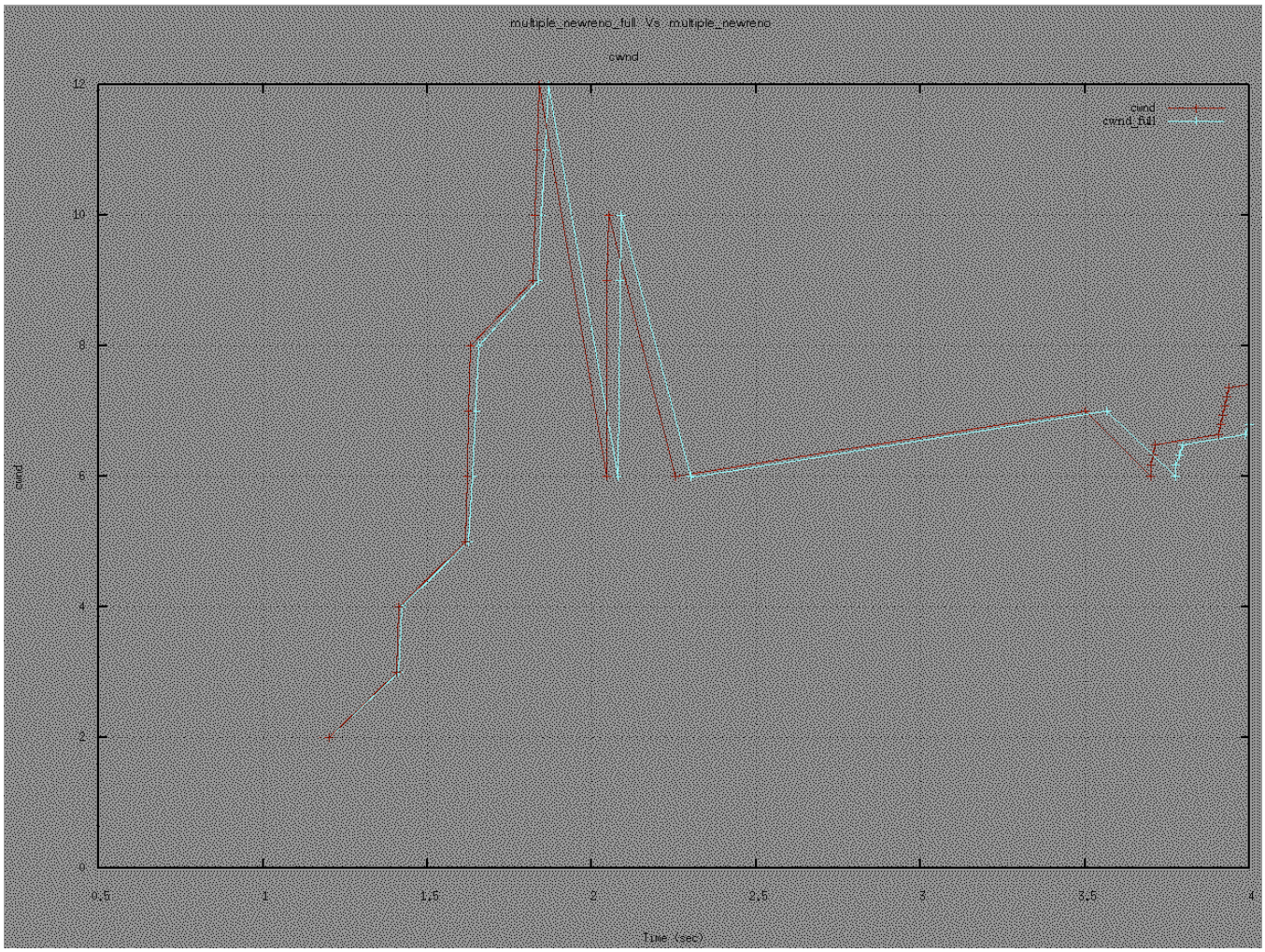

Figure 27: Congestion window graph for multiple_newreno

The congestion window graph evidently shows the advantage of using New-Reno as the cwnd does not fall to one. Slow start is avoided and after sending all the dropped packets, congestion avoidance is chosen.

As we can see from the graphs, both multiple_newreno_full and multiple_newreno produce similar results. Hence we can say that the test multiple_newreno_full is validated.

3.3.8 Test: multiple2_newreno_full

This test uses New-Reno TCP with multiple drops at packets 11, 12, 13, 14, 16. The test is supposed to begin with slow start and fast retransmit after receiving 3rd duplicate ACK-10. It should use fast recovery algorithm and consider partial acknowledgements as well.

Figure 28 shows the graph obtained after running this test.

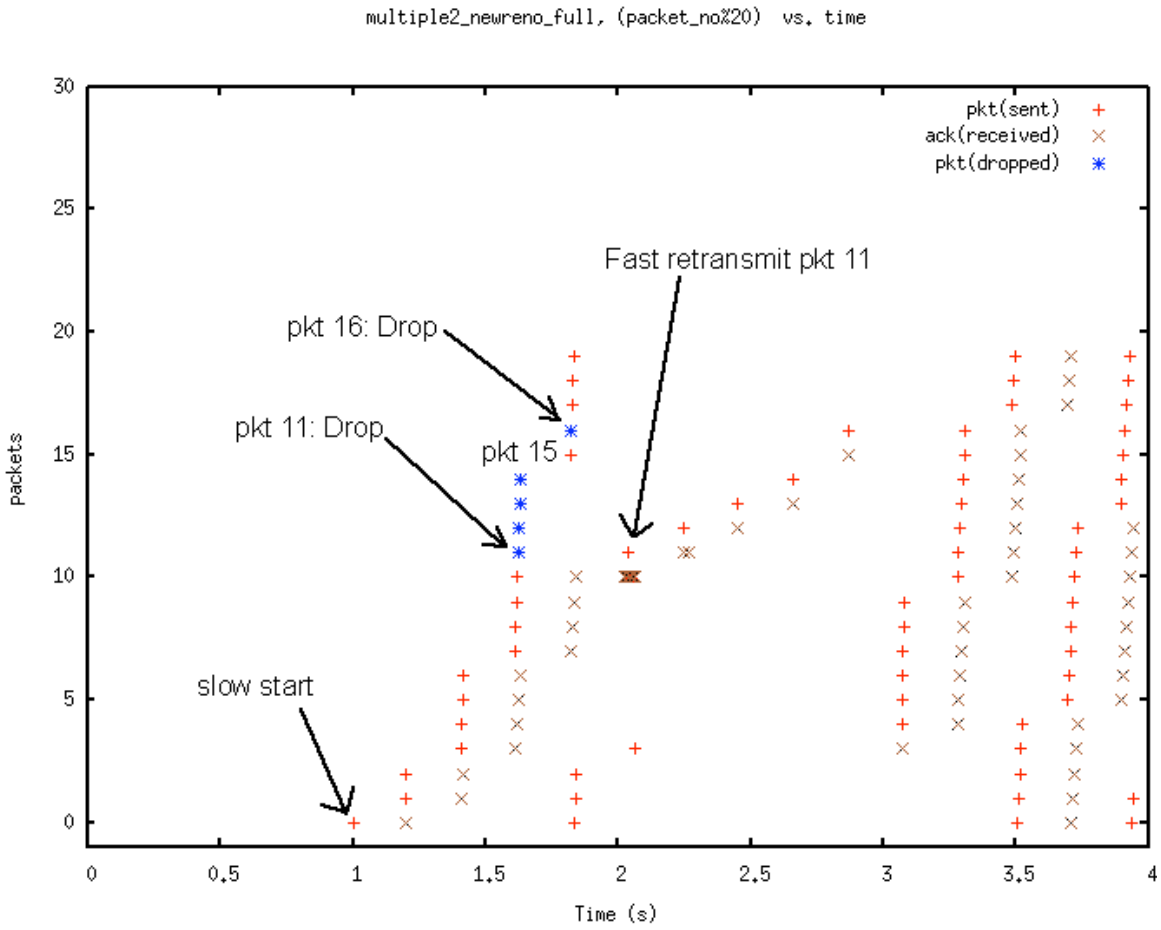

Figure 28: mulitple2_newreno_full

The sender starts with a cwnd 1 and exponentially increases to 11, because ACKs for all packets until 10th (inclusive) have been received. Packets 11, 12, 13, 14, 16 are dropped. On receiving 3rd duplicate ACK for packet 10 (which is the 4th ACK for packet 10), the threshold is set to half the congestion window (ssthresh =12/2=6). cwnd is set to threshold (ssthresh+3). Then packet 11 is retransmitted. As stated earlier, Reno does not recover from multiple drops and has to wait for the retransmission timer to expire. But here, The ACK for packet 11 (new non-dup ACK) is recognized as a partial ACK. It indicates the sender that the packet after the 11th may have been lost. Therefore, the sender retransmits one dropped packet for round trip time until all the dropped packets are resent successfully. The usable window is not "deflated" to ssthresh after receiving ACK 11. We can see in the graph above that after receiving ACK 11, slow start is not

initiated. Rather, one dropped packet is sent per one round trip time until all the dropped packets have been sent successfully.

The one-way TCP equivalent test for multiple2_newreno_full is multiple2_newreno. It produces the graph shown in Figure29.

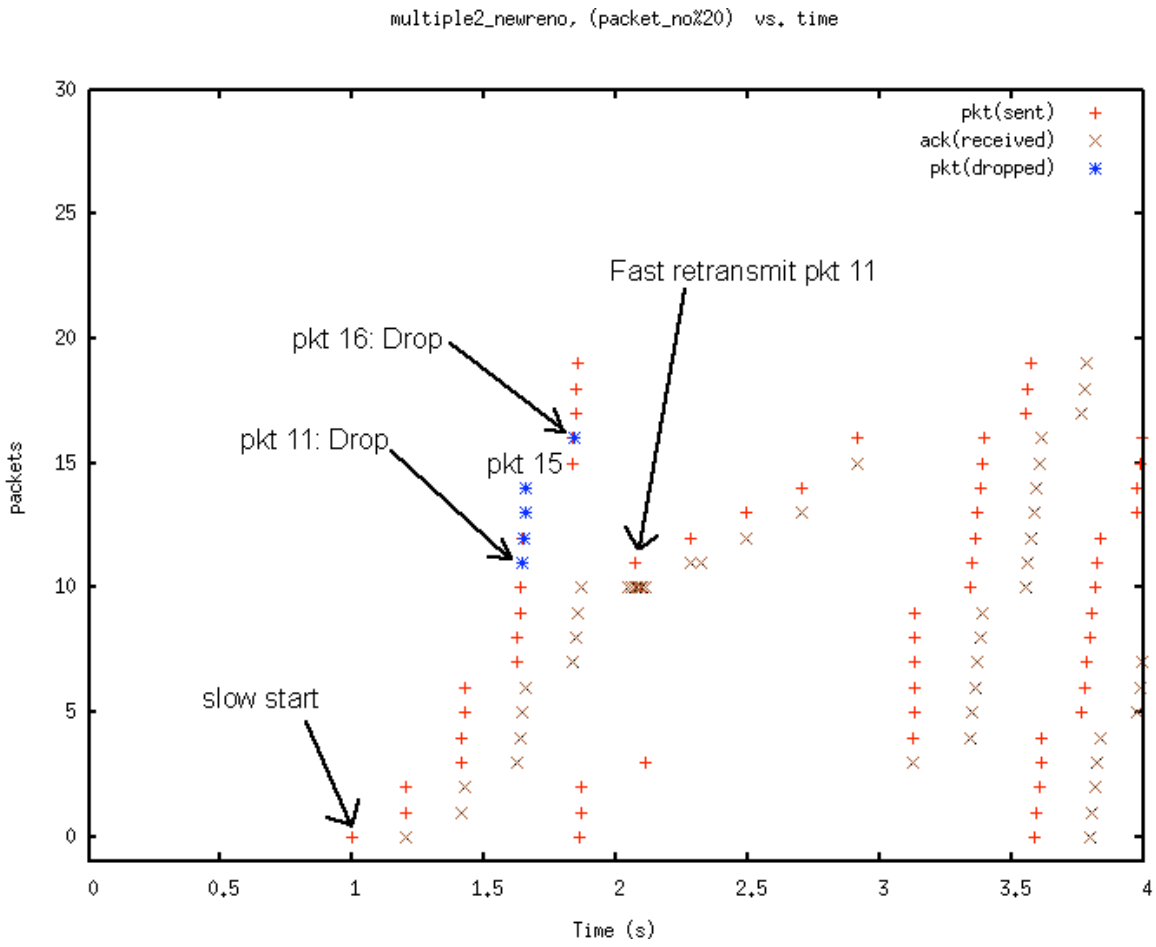

Figure 29: multiple2_newreno

The congestion window graph for this test is shown in Figure 30.

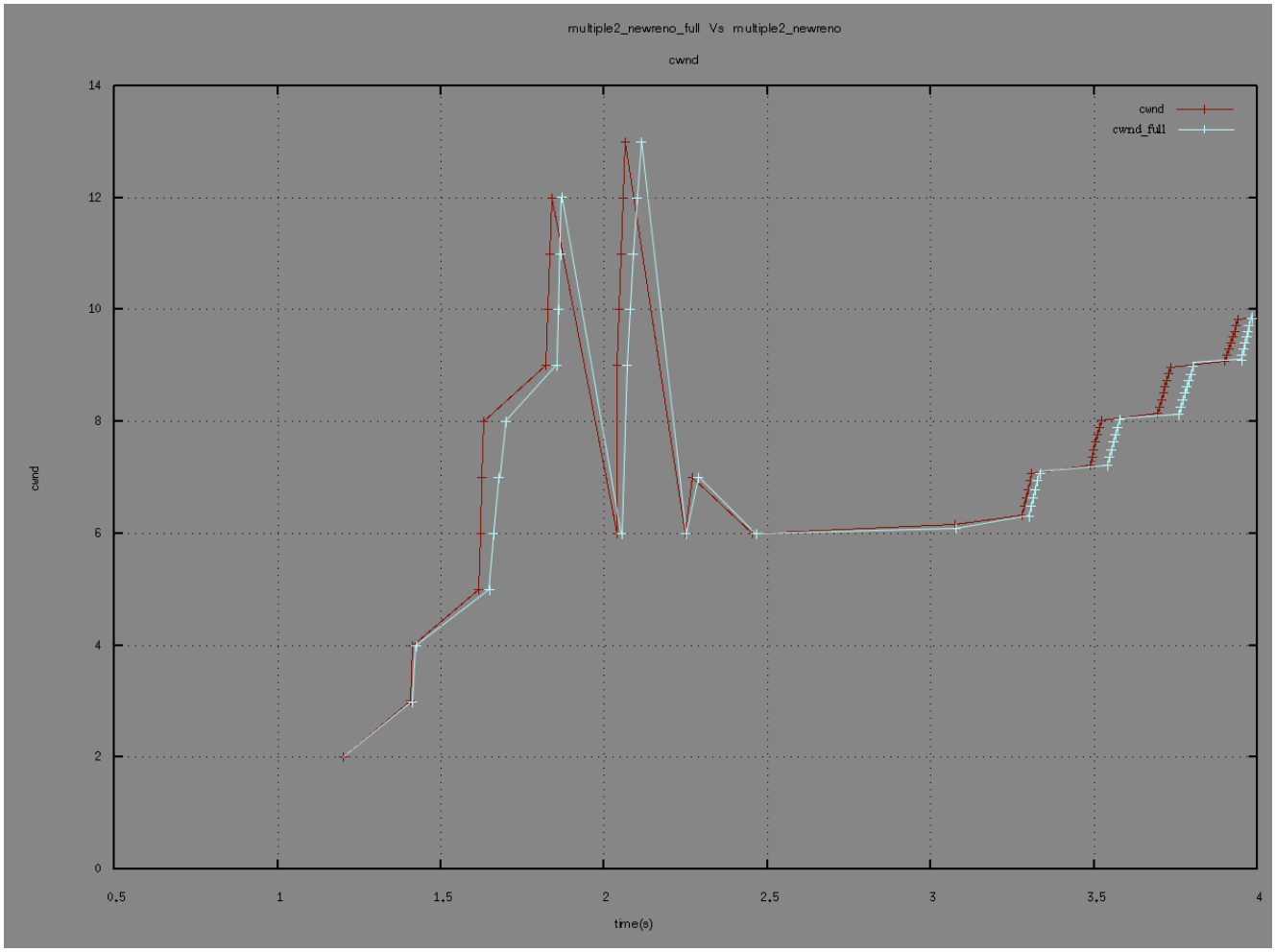

Figure 30: Congestion window for multiple2_newreno_full Vs multiple2_newreno

It is evident from the above figures that multiple2_newreno_full and mulitple2_newreno behave similar which ascertains that multiple2_newreno is validated.

3.3.9 Test: timeouts_newreno1_full

This test uses New-Reno TCP with multiple drops at packets 7, 8, 9, 10, 11, 12, 13, 14, 21. The test is supposed to begin with slow start and fast retransmit after receiving 3rd duplicate ACK and also use fast recovery algorithm. The following timeout scenarios are better without the bugfix_. Hence, this test sets bugfix_ to false which allows multiple fast retransmits. Figure 31 shows the graph obtained after running this test.

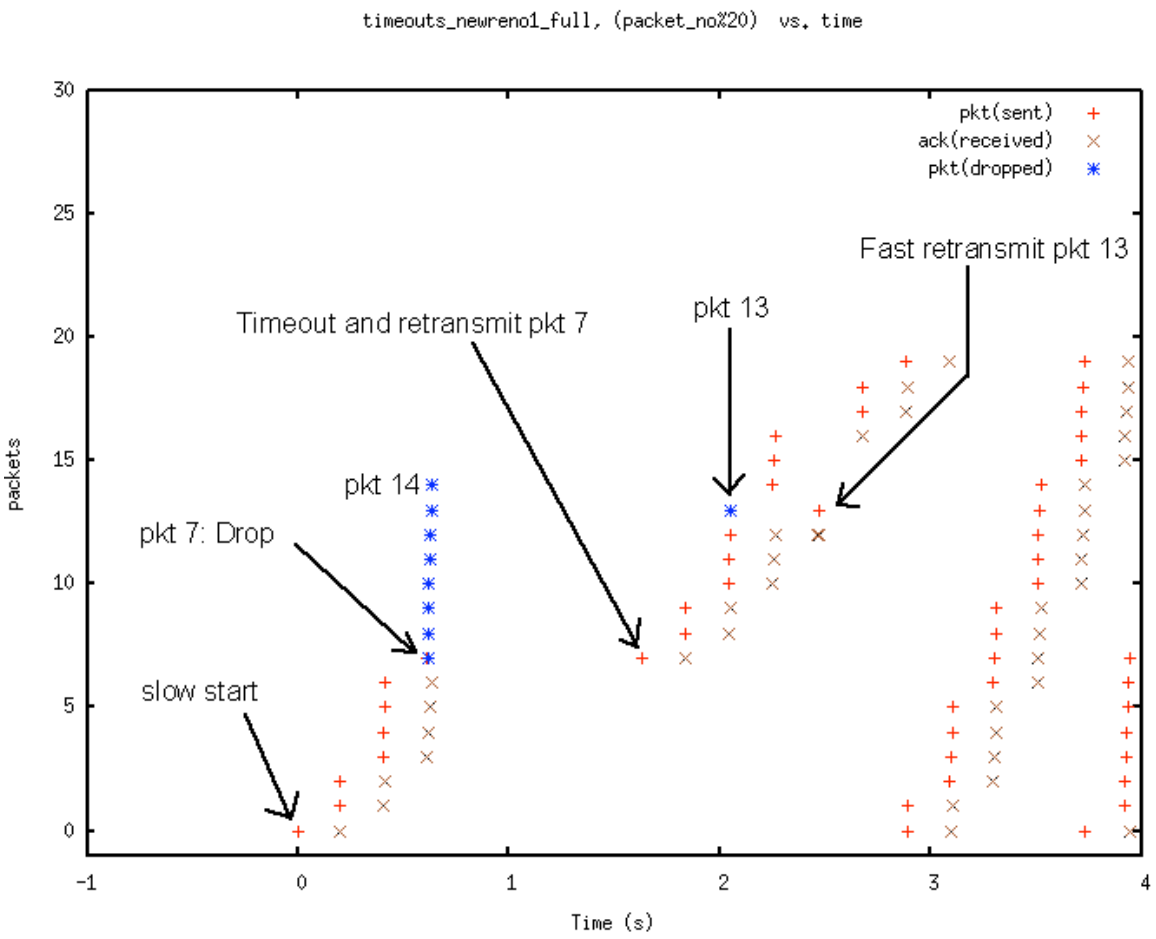

Figure 31: timeout_newreno1_full

In this test, the sender begins with slow start. It drops packet 7, 8, 9, 10, 11, 12, 13, 14. The congestion window expands exponentially until packet loss. After loosing packet 7-14 and retransmission timer expired, the cwnd is set to 1 and dropped packets are resent with slow start initiated. There are no partial ACKs. Therefore, New Reno waits for the retransmission timer to expire. The 21st packet which the sender transmits is the retransmitted packet 13 and it is dropped. The congestion window is 4 by then. After receiving 3 duplicate ACKs for packet 12, packet 13 is resent and congestion avoidance is chosen.

The one-way TCP equivalent test for this test is timeouts_newreno1 and it produces results shown in Figure 32.

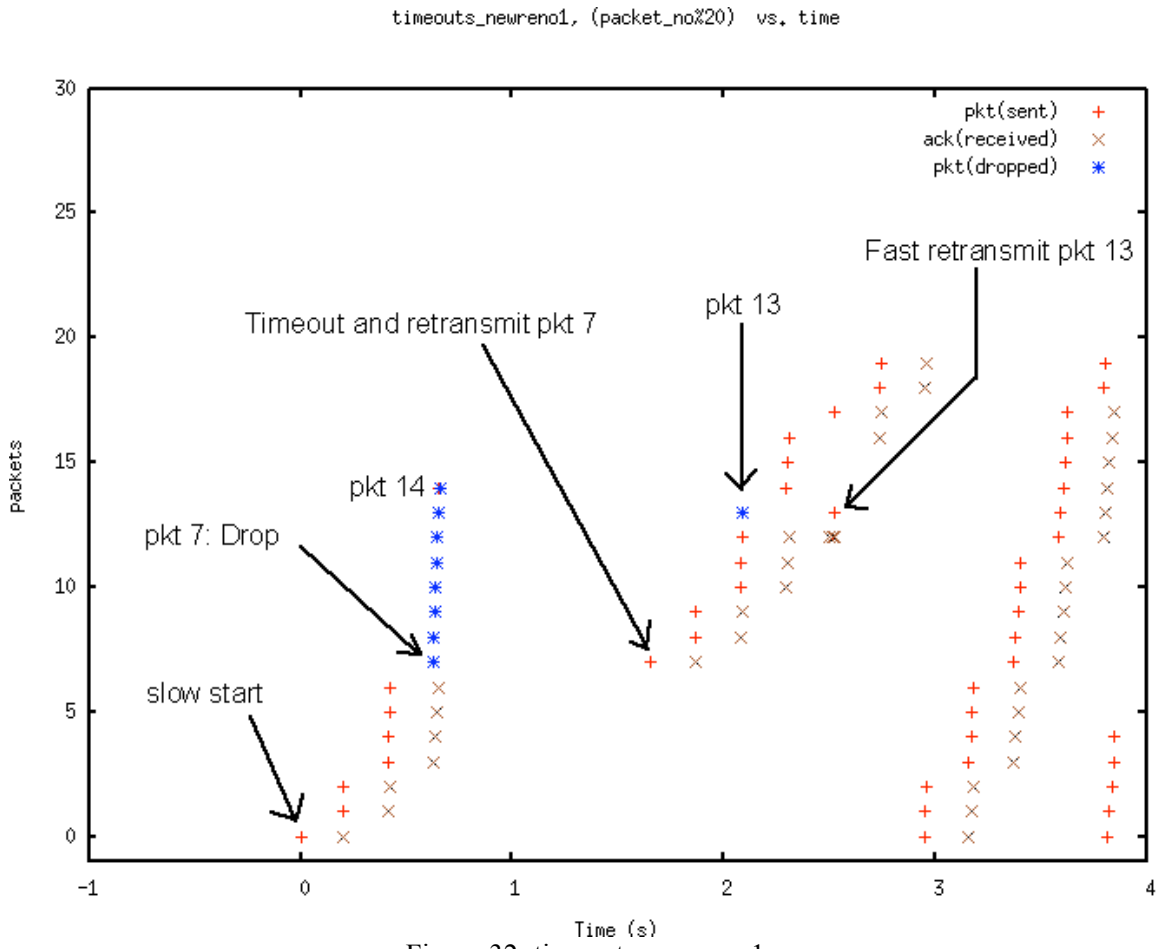

Figure 32: timeouts_newreno1

The congestion window graph for this test is shown in Figure 33.

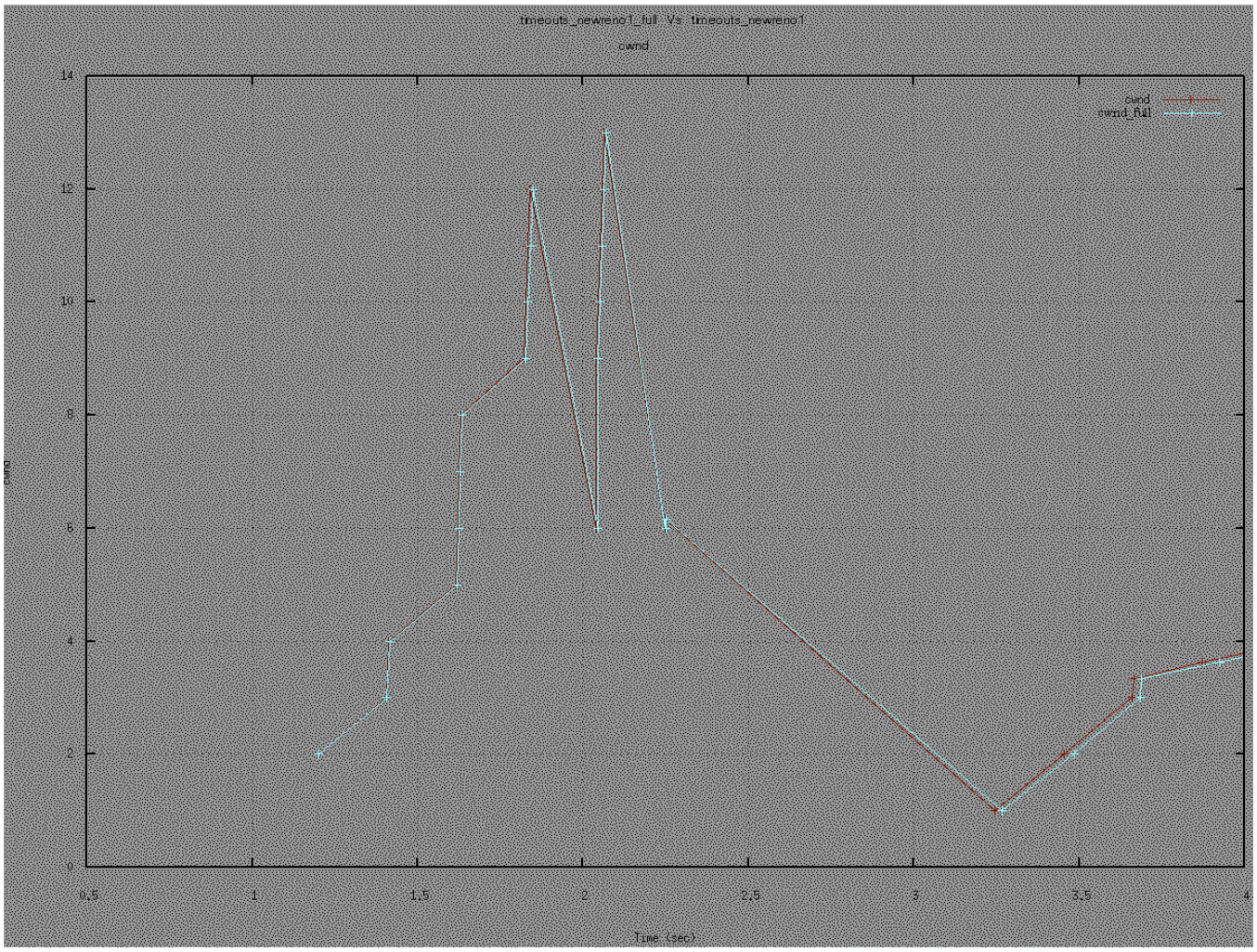

Figure 33: cwnd for timeout_newreno1_full Vs timeouts_newreno1

The graphs prove the fact that the tests timeouts_newreno1_full and timeouts_newreno1 produce the same results. Thus, we can declare that the test timeouts_newreno1_full.

3.3.10 Test: timeouts_newreno2_full

This test uses New-Reno TCP with multiple drops at packets 7, 8, 9, 10, 11, 12, 13, 14, 21. Multiple fast retransmits are not allowed in this test. The test sets the following variables:

Agent/TCP set timestamps_ true

Agent/TCP set ts_resetRTO_ true

This resets the RTO back-off after any valid Roundtrip time measurement. Figure 34 shows graph obtained after running this test.

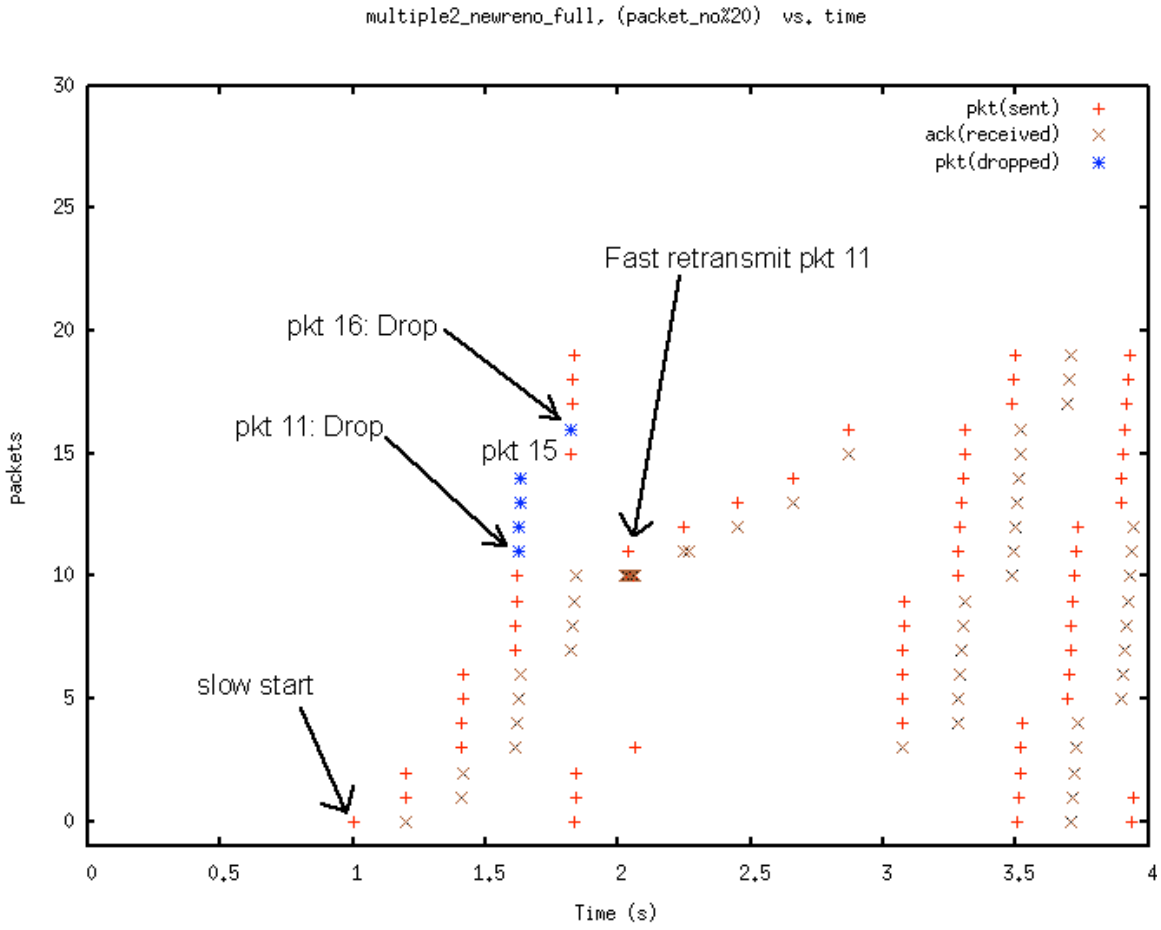

Figure 34: timeout_newreno2_full

This test disables multiple fast retransmissions. The sender begins with a congestion window 1 and slow start. Packets 7 – 14 and also retransmitted packet 13 are dropped. The congestion window expands exponentially until packet loss. After loosing packet 7-14 and retransmission timer expired, the cwnd is set to 1 and dropped packets are resent with slow start initiated. This test was written to see if the Full-TCP performs correctly when multiple retransmissions are disabled and with no-back-off RTO after any valid RTT measurement. The test works accordingly.

Its equivalent one-way TCP produces similar results (Figure 35).

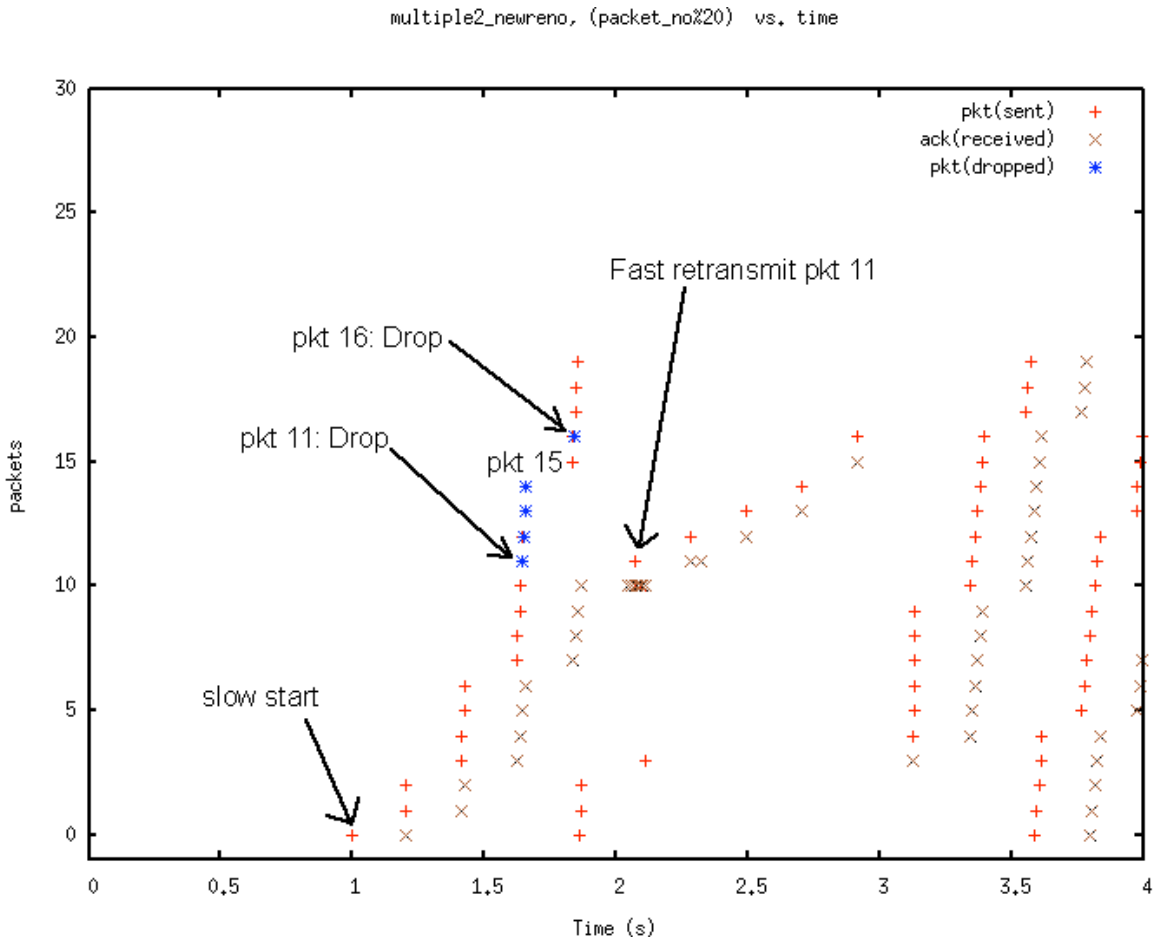

Figure 35: timeouts_newreno2

And the congestion window graph for this test is shown in Figure 36.

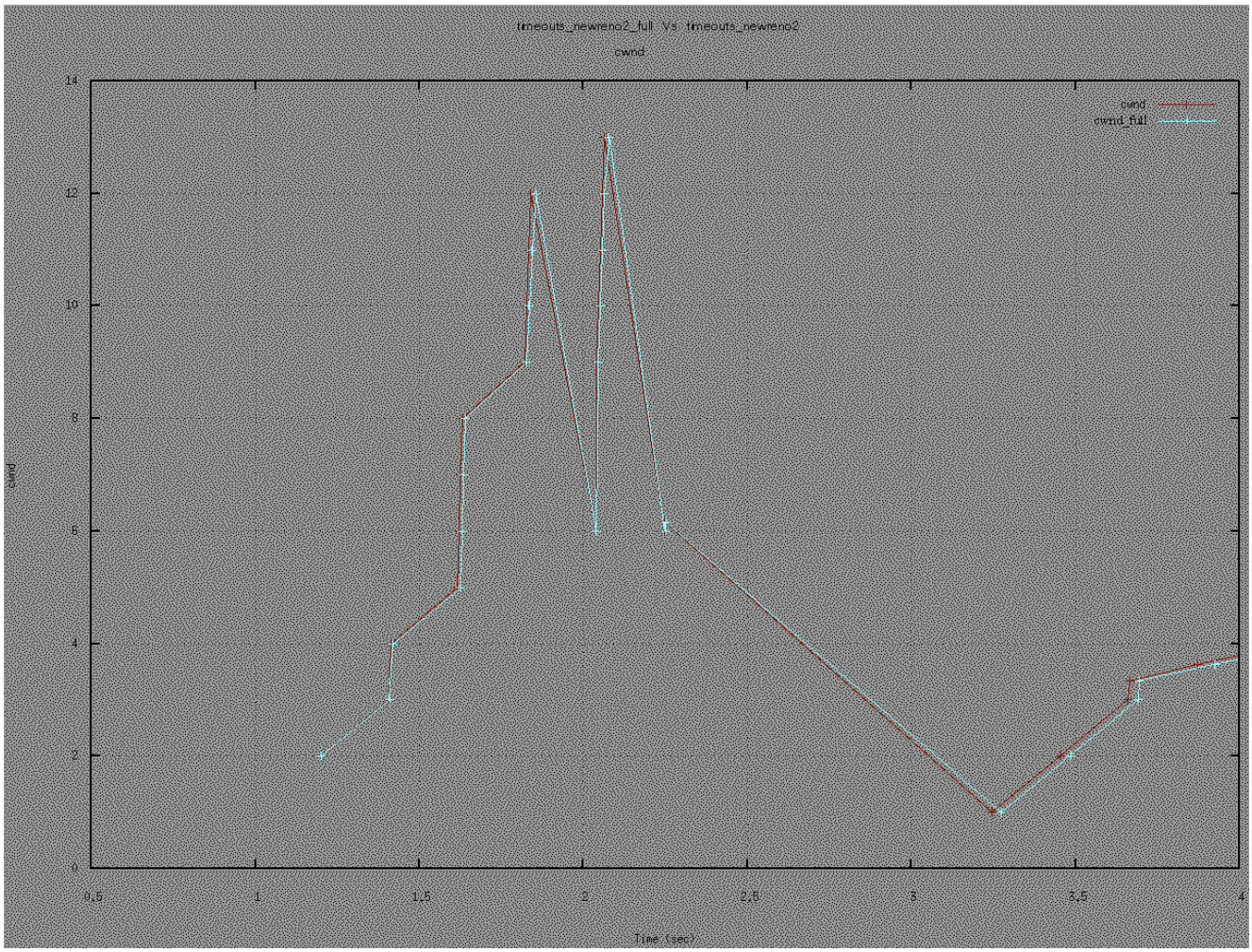

Figure 36: Congestion window for timeouts_newreno2_full Vs timeouts_newreno2

It is evident from the above graphs that the both timeouts_newreno2_full and timeouts_newreno2 produce similar results. Hence, we can declare timeouts_newreno2_full to be validated.

3.3.11 Test: timeouts_newreno3_full

This test is almost similar to timeouts_newreno2_full except for that it examines Full-TCP's behavior in the earlier test with the latest versions included in NS-2. The packets 7, 8, 9, 10, 11, 12, 13, 14, 21 are dropped. Figure 37 shows the graph obtained after running this test.

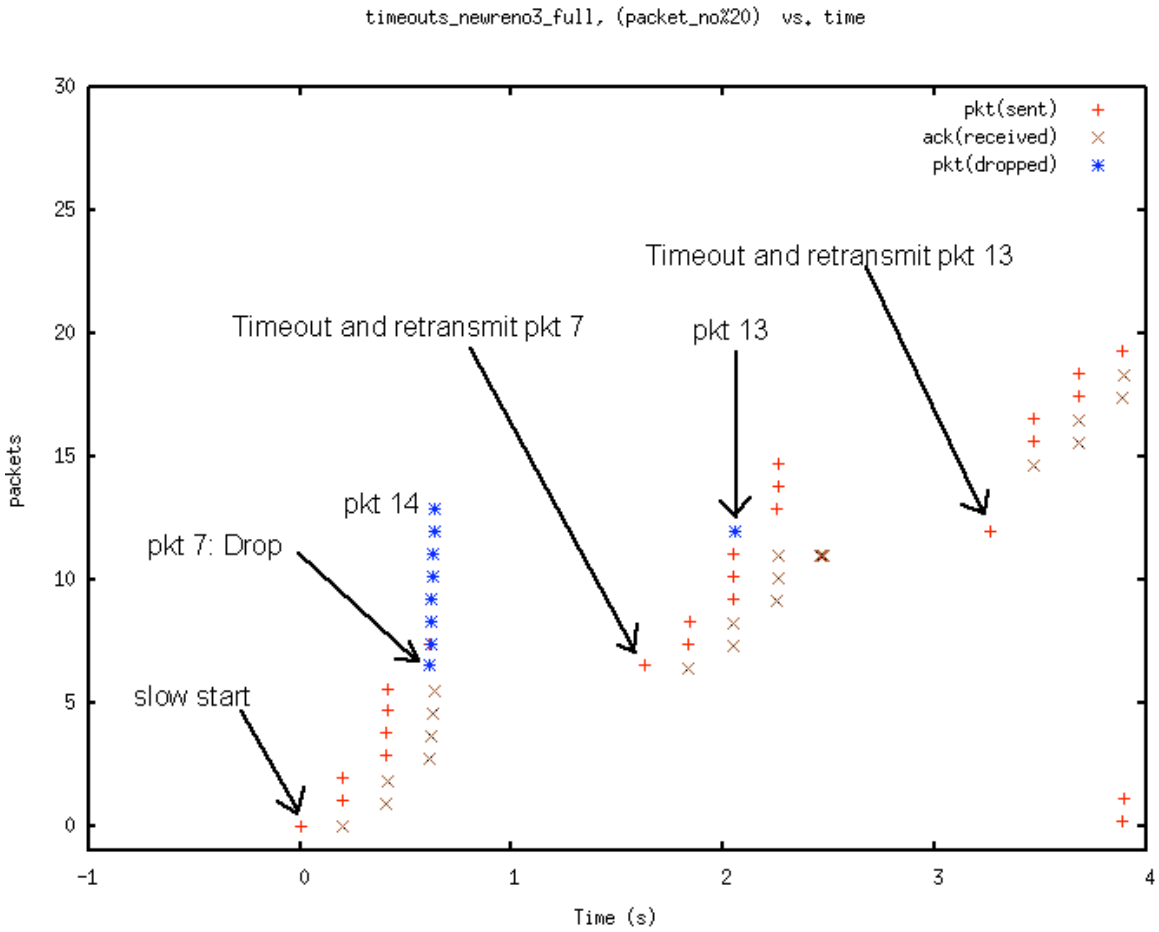

Figure 37: timeouts_newreno3_full

Its equivalent one-way TCP test produces similar results (Figure 38).

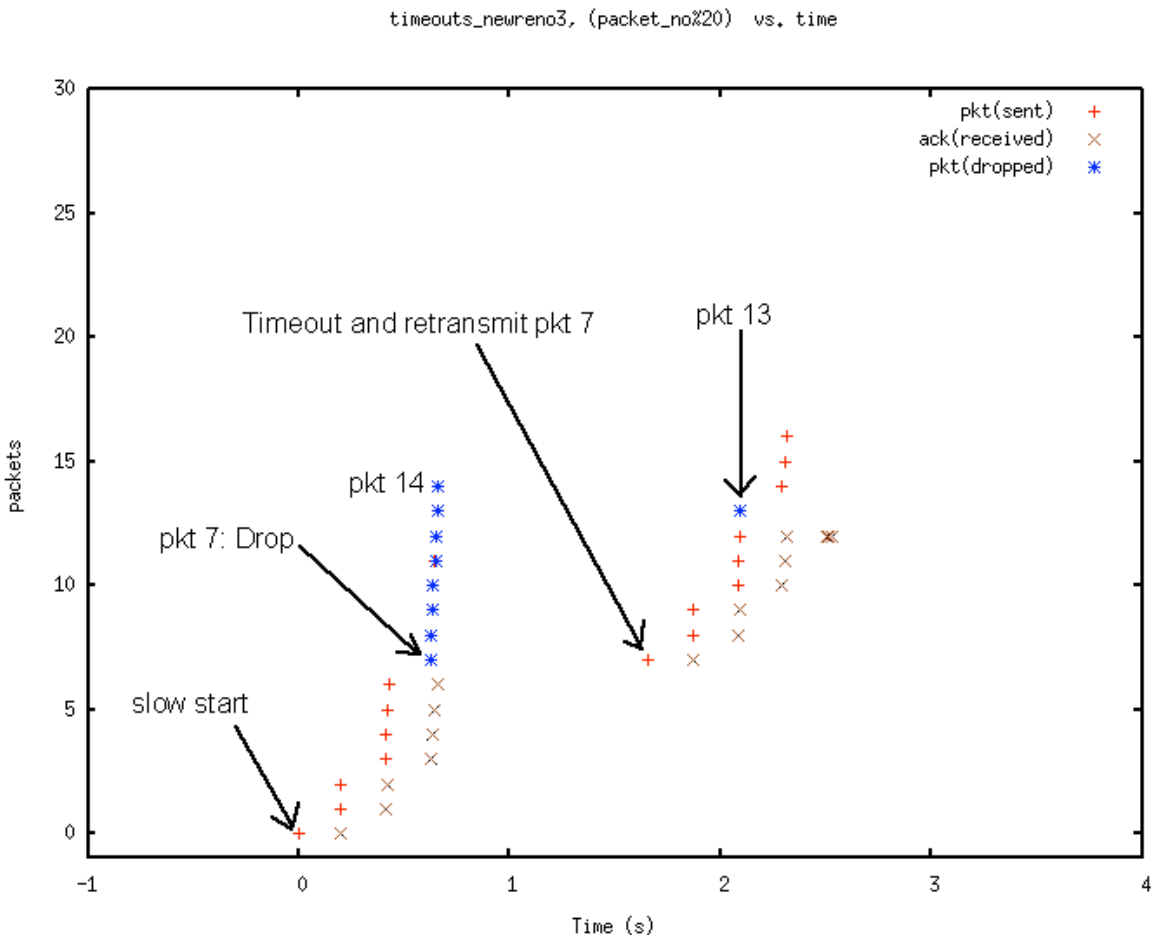

Figure 38: timeouts_newreno3

And the congestion window graph for this test is shown in Figure 39.

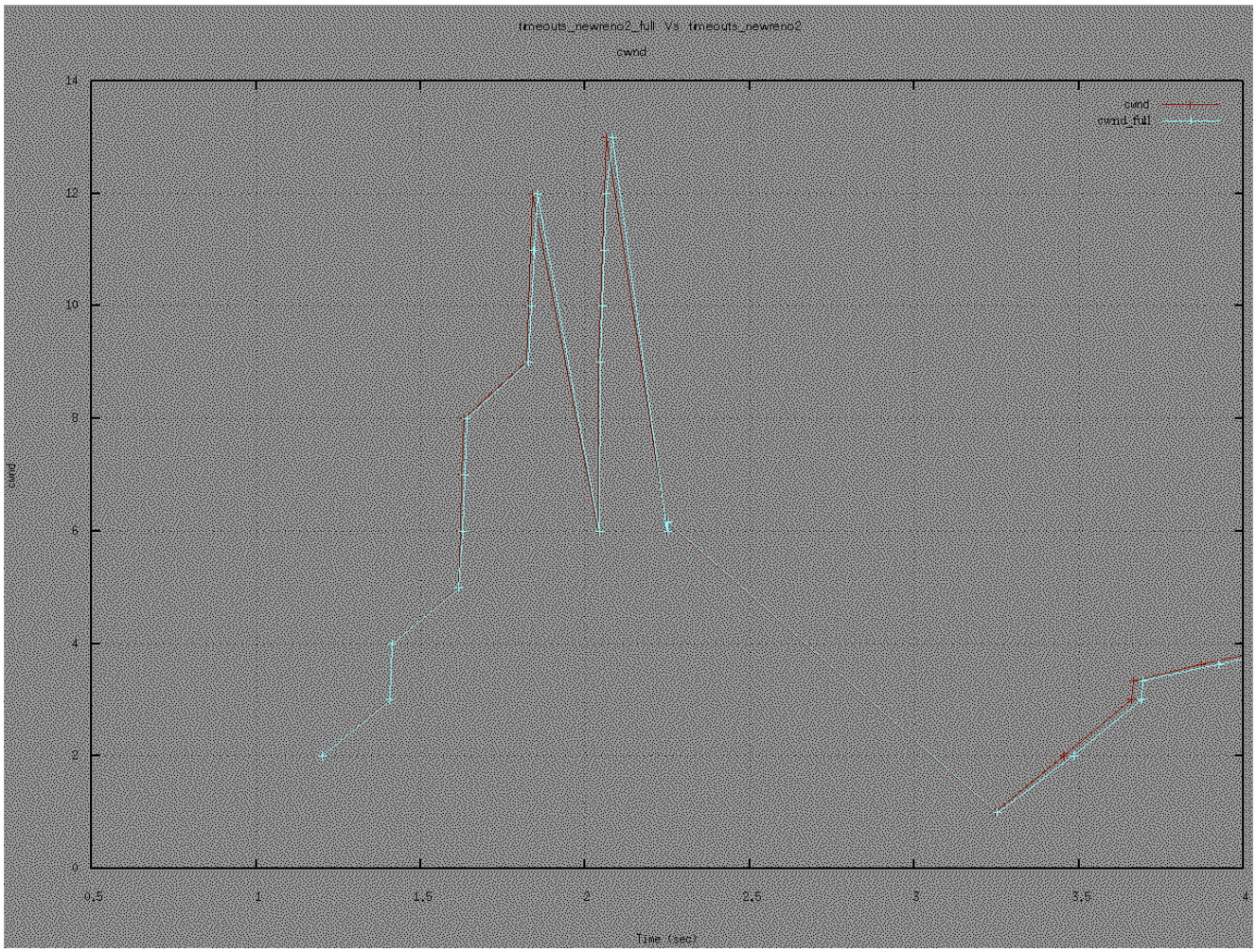

Figure 39: congestion window for timeouts_newreno3_full Vs timeouts_newreno3

Since the tests timeouts_newreno3_full and timeouts_newreno3, both produce same results, we can claim that timeouts_newreno3_full is working correctly and is validated.

4. Conclusion

After validating a few of the already written tests and also writing new tests, in total eleven tests pertaining to three variants of Full-TCP (Tahoe, Reno, New-Reno) have been validated in this project. Table 1 provides a summary of the tests that were validated and written in this project. But, this work does not address the issue completely. There are many more tests which need to be written and validated before Full-TCP can become the default TCP agent in NS-2. Future work for this project could

be to validate more un validated tests in Full-TCP and also address others variants of Full-TCP such as SACK TCP.

References

[1] Heidemann J., K. Mills, and S. Kumar, “Expanding confidence in network simulation,” *IEEE Network Magazine* 15 (5): 58{63}, Sept./Oct 2001.

[2] Stevens, W., "TCP Slow Start, Congestion Avoidance, Fast Retransmit, and Fast Recovery Algorithms," Request for comments(Experimental) RFC 2001, January 1997.

[3] S. Floyd, T. Henderson, “The NewReno Modification to TCP's Fast Recovery Algorithm,” Request for comments(Standards Track) RFC 2582, April 1999.

[4] Kevin Fall and Sally Floyd, “Simulation-based Comparisons of Tahoe, Reno, and SACK TCP,” July 1996, Berkeley, CA

[5] Kevin Fall, Sally Floyd, and Tom Henderson, “Ns Simulator Tests for Reno FullTCP,” July 1997.

[6] http://nsnam.cvs.sourceforge.net/nsnam/ns-2/tcl/test/FullTCP.notes?revision=1.2

Appendix

There were three tests which were newly written as part of this project. The remaining tests were already written but commented. The tests were written in Object oriented Tcl language. This section provides the source code for tests involved in the validation process.

Test	Source code	Comments

Tahoe_FullTCP2_without_Fast_Retransmit	Class Test/Tahoe_FullTCP2_without_Fast_Retransmit -superclass TestSuite
	Test/Tahoe_FullTCP2_without_Fast_Retransmit instproc init { } {
	$self instvar net_ test_
	set net_ net4
	set test_ Tahoe_FullTCP2_without_Fast_Retransmit
	Agent/TCP/FullTcp/Tahoe set noFastRetrans_ true
	Agent/TCP/FullTcp set segsperack_ 2
	$self next
	}
	Test/Tahoe_FullTCP2_without_Fast_Retransmit instproc run { } {
	$self setup FullTcpTahoe {8} {19}
	}
		Already written

Tahoe_FullTCP_without_Fast_Retransmit	Class Test/Tahoe_FullTCP_without_Fast_Retransmit -superclass TestSuite
	Test/Tahoe_FullTCP_without_Fast_Retransmit instproc init{ }
	{
	$self instvar net_ test_
	set net_ net4
	set test_ Tahoe_FullTCP_without_Fast_Retransmit
	Agent/TCP set noFastRetrans_ true
	Agent/TCP/FullTcp set segsperack_ 2
	$self next
	}
	Test/Tahoe_FullTCP_without_Fast_Retransmit instproc run{ }
	{
	$self setup FullTcpTahoe {5} {17 20}
	}
		Already written

multiple_tahoe_full	Class Test/multiple_tahoe_full -superclass TestSuite Test/multiple_tahoe_full instproc init { } { $self instvar net_ test_	Already written

	set net_ net4
	set test_ multiple_tahoe_full
	$self next pktTraceFile
	}
	Test/multiple_tahoe_full instproc run { } {
	$self setup FullTcpTahoe {12 13 14 15 17 18 19 20}
	}	

multiple_reno_full	Class Test/multiple_reno_full –superclass TestSuite
	Test/multiple_reno_full instproc init { } {
	$self instvar net_ test_
	set net_ net4
	set test_ multiple_reno_full
	$self next pktTraceFile
	}
	Test/multiple_reno_full instproc run { } {
	$self setup FullTcp {12 13 14 15 17 18 19 20}
	}
		Already written

multiple_newreno_full	Class Test/multiple_newreno_full -superclass TestSuite
	Test/multiple_newreno_full instproc init { } {
	$self instvar net_ test_
	set net_ net4
	set test_ multiple_newreno_full
	$self next pktTraceFile
	{
	Test/multiple_newreno_full instproc run { } {
	$self setup FullTcpNewreno {12 13 14 15 17 18 19 20}
	}
		Already written

multiple2_taho_full	Class Test/multiple2_tahoe_full -superclass TestSuite
	Test/multiple2_tahoe_full instproc init { } {
	$self instvar net_ test_
	set net_ net4
	set test_ multiple2_tahoe_full
	$self next pktTraceFile
	}
	Test/multiple2_tahoe_full instproc run { } {
	$self setup FullTcpTahoe {12 13 14 15 17 }
	}
		Newly written

multiple2_reno_full	Class Test/multiple2_reno_full -superclass TestSuite
	Test/multiple2_reno_full instproc init { } {
	$self instvar net_ test_
	set net_ net4
	set test_ multiple2_reno_full
	$self next pktTraceFile
	}
	Test/multiple2_reno_full instproc run { } {
	$self setup FullTcp {12 13 14 15 17 }
	}
		Newly written

multiple2_newreno_full	Class Test/multiple2_newreno_full -superclass TestSuite Test/multiple2_newreno_full instproc init { } { $self instvar net_ test_ set net_ net4 set test_ multiple2_newreno_full $self next pktTraceFile	Newly written

	}
	Test/multiple2_newreno_full instproc run { } {
	$self setup FullTcpNewreno {12 13 14 15 17 }
	}	

timeouts_newreno1_full	Class Test/timeouts_newreno1_full -superclass TestSuite
	Test/timeouts_newreno1_full instproc init { } {
	$self instvar net_ test_ guide_
	set net_ net4
	set test_ timeouts_newreno1_full
	set guide_ "NewReno, timeouts, better without bugfix"
	Agent/TCP set bugFix_ false
	$self next pktTraceFile
	}
	Test/timeouts_newreno1_full instproc run { } {
	$self setup2 FullTcpNewreno {8 9 10 11 12 13 14 15 22} 8
	}
		Already written

timeouts_newreno2_full	Class Test/timeouts_newreno2_full -superclass TestSuite
	Test/timeouts_newreno2_full instproc init { } {
	$self instvar net_ test_ guide_
	set net_ net4
	set test_ timeouts_newreno2_full
	set guide_ "NewReno, timeouts, bugfix, with timestamps, old version"
	Agent/TCP set timestamps_ true
	Agent/TCP set ts_resetRTO_ true
	$self next pktTraceFile
	}
	Test/timeouts_newreno1_full instproc run { } {
	$self setup2 FullTcpNewreno {8 9 10 11 12 13 14 15 22} 8
	}
		Already written

timeouts_newreno3_full	Class Test/timeouts_newreno3_full -superclass TestSuite
	Test/timeouts_newreno3_full instproc init {} {
	$self instvar net_ test_ guide_
	set net_ net4
	set test_ timeouts_newreno3_full
	set guide_ "NewReno, timeouts, bugfix, with timestamps, new version"
	Agent/TCP set timestamps_ true
	$self next pktTraceFile
	}
	Test/timeouts_newreno1_full instproc run { } {
	$self setup2 FullTcpNewreno {8 9 10 11 12 13 14 15 22} 8
	}
		Already written

Table 1: Source code of the tests validated in this project.