Fault Isolation in a Multicast Tree using DYSWIS
COMS 6181 Fall 2011 Final Report

Phil Sphicas
pgs2111@columbia.edu

December 20, 2011

Abstract

In an IPTV environment, in which video is delivered to subscribers via IP multicast, video quality problems can often be directly attributed to packet loss. When the video streams are encapsulated in RTP, this packet loss is easily detectable using RTP sequence number analysis. By correlating loss events across geographically diverse subscriber households, and overlaying these events on the multicast distribution tree, it is possible to quickly narrow down the search area where the fault may be occurring.

This paper present an extension of the DYSWIS tool, adding the functionality of RTP multicast stream monitoring, loss correlation between nodes, and fault isolation.

1 Introduction

The goal of this project was to use DYSWIS to isolate faults in a multicast video distribution network. To accomplish this, a new module has been added to the DYSWIS tool to monitor RTP multicast streams, and detect loss events using RTP sequence number analysis. DYSWIS nodes are able to query multiple remote peers for their session and fault histories, and through the use of traceroute probes towards the source of the multicast stream, can identify which links are part of the shared multicast distribution tree. By correlating the loss events, and overlaying this information on the discovered topology, the system can then determine where in the network the loss is not occurring, and point towards where the loss might be occurring.

1.1 Top-Down Fault Isolation Approach

Consider the network shown in Figure 1. In this diagram, 'src' indicates the multicast source, 'rtr$N$' represents a router, and 'sub$N$' represents a subscriber.

Assuming that the multicast tree is strictly source-based, each link would carry only one copy of every packet, and if a loss occurred on a router or link, all downstream subscribers would experience that loss.

For example, if a loss occurred on the link between rtr2 and rtr5, then both sub3 and sub4 would experience the exact same loss. Similarly, if a loss occurred on the link between rtr1 and rtr3, the sub5, sub6, sub7, and sub8 would all experience that shared loss event.

In fact, for a single packet loss, it is possible to build a matrix that maps the location of the fault in the network to the subscribers that experience the loss. This matrix is represented in Figure 2.

Turning this around and approaching it from the perspective of a subscriber, if sub1 and sub2 experience the same loss event, then it is trivial to look at the matrix and determine that the loss is either at src-rtr1, rtr1-rtr2, or rtr2-rtr4. Given full knowledge from all subscribers, it would be possible to uniquely determine where the loss occurred.

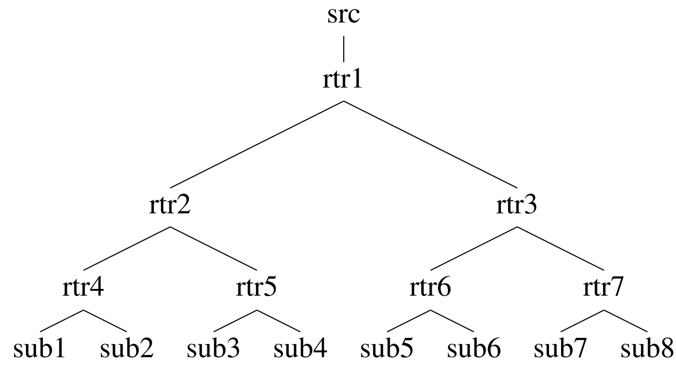

Figure 1: Simplified multicast distribution tree

	sub1	sub2	sub3	sub4	sub5	sub6	sub7	sub8
src-rtr1	x	x	x	x	x	x	x	x
rtr1-rtr2	x	x	x	x				
rtr1-rtr3					x	x	x	x
rtr2-rtr4	x	x						
rtr2-rtr5			x	x				
rtr3-rtr6					x	x		
rtr3-rtr7							x	x
rtr4-sub1	x							
rtr4-sub2		x						
rtr5-sub3			x					
rtr5-sub4				x				
rtr6-sub5					x			
rtr6-sub6						x		
rtr7-sub7							x	
rtr7-sub8								x

Figure 2: Observed loss at subscribers by network fault location

The main challenge with this approach is that the topology must be known in advance in order to build the matrix. Also, one factor that has been ignored until this point is that not every subscriber is always watching the same channel. A subscriber that is not joined to the stream will not experience the loss event, but that information is not taken into account.

1.2 An Alternative Fault Isolation Approach

If we start again from the subscriber end system, it is logical that if a fault occurred, then it occurred somewhere on the path between the source and the receiver. Again assuming a source-based tree, the tree would have been built from the receiver towards source using the unicast routing table of the routers.

Therefore, the end system can reasonably approximate the path that multicast would have taken from the source to the receiver by simply running a traceroute towards the multicast source. The routers will forward the traceroute probes out the same interface into which the multicast would be flowing, as that would be the interface that satisfied the reverse path forwarding check.

Given any set of end systems, then, by comparing what they know about a loss event, and their respective paths to the multicast source, they can reduce the set of possible links where the loss occurred.

This is the approach taken by the DYSWIS multicast RTP module.

1.3 DYSWIS Multicast Fault Isolation Approach in Detail

To briefly solidify the concept of a fault, or a loss event, recall that subscribers may or may not be joined to a multicast stream whose contents are RTP packets. RTP packets have monotonically increasing sequence numbers that wrap when they reach a maximum value. Therefore, it is relatively easy to detect a loss, since there will be a gap in sequence numbers. In general, a fault can be detected whenever the RTP sequence number is not what was expected, which could potentially include packet mis-ordering.

Since this is multicast (presumably not dense mode, and certainly not broadcast), a subscriber that has not joined a stream will not receive any of the packets associated with that stream. This is not particularly useful. However, if a receiver was in fact joined to a stream during the time another subscriber experienced a loss event, and did not experience any loss, this is useful information indeed.

Given a node that experienced a loss event, our goal is to identify a set of "suspicious" links where that loss may have occurred, and to reduce the size of this set as much as possible given the data available.

With no additional data, if node a experiences a loss event and has a set of hops H_a in the traceroute towards the source of the stream, then we can initialize the set of possible bad hops $B = H_a$.

If a different node b experiences the same loss event, and has a set of hops H_b in the traceroute towards the source of the stream, then we can now say that:

$$B = B \cap H_b$$

That is, the set of bad hops can only be the hops that are shared between the receivers that experienced the fault.

Alternatively, if a node c can positively acknowledge that it did not experience the same fault, then we can say that the suspect nodes cannot be in the path H_c, since those hops are known to be "good":

$$B = B \setminus H_c$$

For example, returning for a moment to Figure 1, let us assume that sub1 and sub3 experience a correlated loss. The hops towards the source can be calculated as:

$$H_{sub1} = \{rtr4sub1, rtr2rtr4, rtr1rtr2, srcrtr1\}$$

$$H_{sub3} = \{rtr5sub3, rtr2rtr5, rtr1rtr2, srcrtr1\}$$

Computing the bad hops from the perspective of sub1, B is initialized as H_{sub1}.

$$B = H_{sub1} = \{rtr4sub1, rtr2rtr4, rtr1rtr2, srcrtr1\}$$

Then taking into account the data from sub3, we have:

$$B = B \cap H_{sub3} = \{rtr1rtr2, srcrtr1\}$$

Thus we have reduced the list of possible bad hops to just two.

Taking a separate example, let us assume that sub1 experiences a loss event, but sub3 is able to positively confirm that it did not experience that loss event. In this case:

$$B = H_{sub1} \setminus h_{sub3} = \{rtr4sub1, rtr2rtr4\}$$

1.4 Caveats

Clearly the idea of determining the path of the multicast via traceroute is a simplification, and ignores common issues like routers that don't always respond to traceroute requests, parallel links, links that don't have multicast routing enabled, and countless others.

For the purpose of this project, a correlated loss actually means that the exact same RTP sequence number fault on the same multicast stream is experienced at two or more subscribers. In particular, if one receiver does not receive packets 1 - 5, and another receiver does not receive packet 2 - 5, this is not considered a correlated loss.

2 Project Team

Phil Sphicas	pgs2111@columbia.edu	Design, Engineering, Testing, Documentation
Kyung-Hwa Kim	kk2515@columbia.edu	DYSWIS Subject Matter Expert
Prof. Schulzrinne	hgs@columbia.edu	Course Instructor, Project Sponsor

3 Platform

3.1 Framework

This project is built on the DYSWIS framework. DYSWIS, which stands for "Do You See What I See", is a distributed system that automatic detects and diagnoses faults, by collecting and comparing data from multiple end systems. The following components were added within the DYSWIS framework:

1. A detector to monitor the RTP multicast streams for loss events, and record a history.
2. A probe to run traceroute back towards a multicast source address.
3. A probe to check if other peers experienced the same loss by checking their history, or if they explicitly did not see a loss - i.e. they were watching the same stream and had no sequence number errors.
4. A set of diagnosis rules to correlate the results.

3.2 Operating System

Development was done on both Windows XP and OS X Lion. The resulting code was also tested on Windows 7 and OS X Snow Leopard. Unfortunately, the jpcap packet capture libraries do not seem to fully support OS X at this time. I was able to compile a version of the library for OS X that does function, but only allows capture on the default capture device, which may or may not be the desired NIC.

3.3 Programming Languages

The detector and probe components were written in Java, using Sun Java Version 6 Update 29. The diagnosis rules were written in Jess, a LISP-like language. The Eclipse IDE was used for the development.

4 Schedule

Week of	Milestone
10/03	Project plan complete
10/10	Access DYSWIS repository, review existing modules
10/17	Implementation of RTP Detect module started
10/24	Implementation of RTP Session module started
10/31	Implementation of RTP Fault module started
11/07	Implementation of RTP Diagnosis module started
11/14	Implementation of RTP Analysis module started
11/21	Implementation of Probes to retrieve historical RTP faults started
11/28	Implementation of topology discovery started
12/05	Implementation complete
12/12	Testing and documentation complete

5 Architecture and Design

The DYSWIS framework as a whole is described in Figure 3.

The following sections describe the classes that make up the module_mcast_rtp package, and how they fit in to the framework. Complete code listings can be found in the appendices.

5.1 McastRtpDetect

The McastRtpDetect java class performs several important functions.

This detection module is registered with the capture engine to receive multicast RTP packets. When a packet arrives, it attempts to parse the packet. If the packet is successfully parsed as an McastRtpPacket, it is matched against a map of previously observed sessions. If an existing session is found, the packet is dispatched to that McastRtpSession object. If no existing session is found, a new session is created, and added to the map.

This is described in Figure 4. Note that this is a slight variation from the model defined by the framework, in which the Session instances would register with the capture engine.

As an optimization, when new session is detected, a traceroute probe is initiated towards the source of the multicast stream. The results are stored for later retrieval. This is important, because a traceroute can take over 30 seconds to complete, especially if there are many timeouts along the path. Having this information cached allows nodes to respond to probe requests more quickly, as will be seen later.

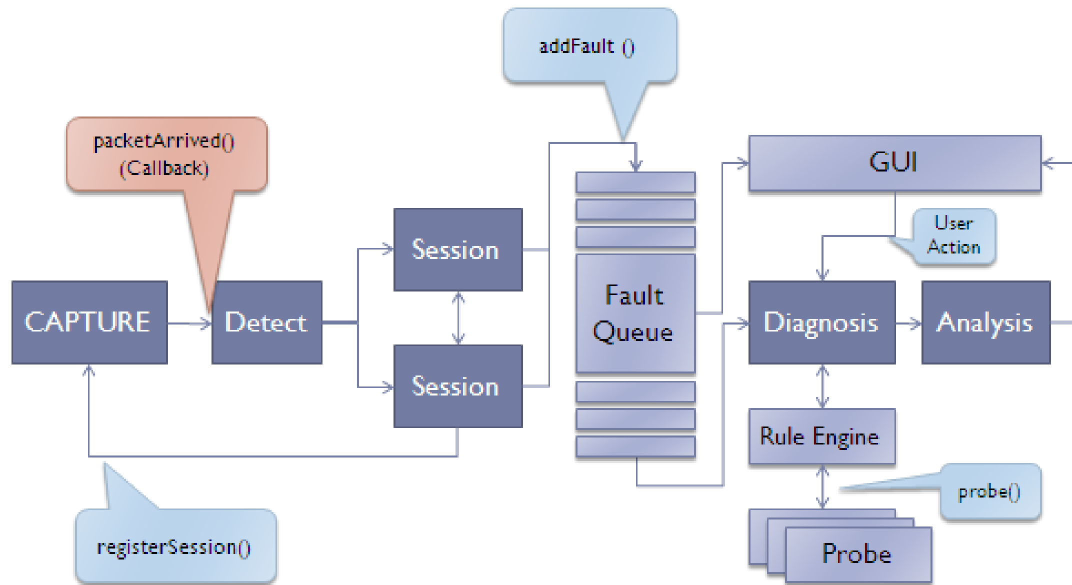

Figure 3: DYSWIS architecture (from http://www.cs.columbia.edu/irt/project/dyswis/)

As an implementation note, the DYSWIS framework support for packet capture filters does not seem to be working. Although it appears that classes implementing the DetectInterface are allowed to specify a capture filter, in testing it appeared that the McastRtpDetect class was receiving all packets, not just its own.

5.2 McastRtpPacket

The McastRtpPacket java class is used to represent multicast RTP packets. Its constructor can parse a captured UDPPacket and extract the various RTP header components. A simplified model of the interaction between McastRtpDetect and McastRtpPacket is shown in Figure 4.

In terms of implementation, the most interesting part of this class was implementing the checks to ensure that the received packet was, in fact, a multicast RTP packet. McastRtpPacket actually exposes a

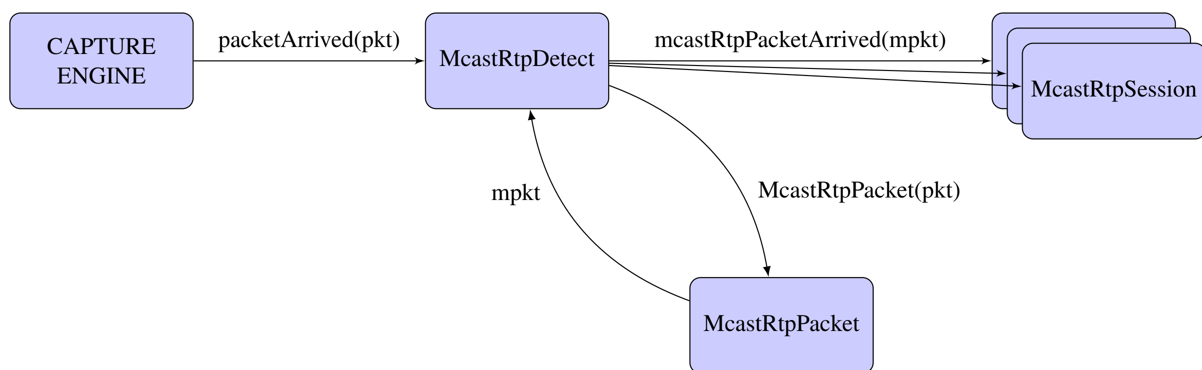

Figure 4: McastRtpDetect and its interfaces

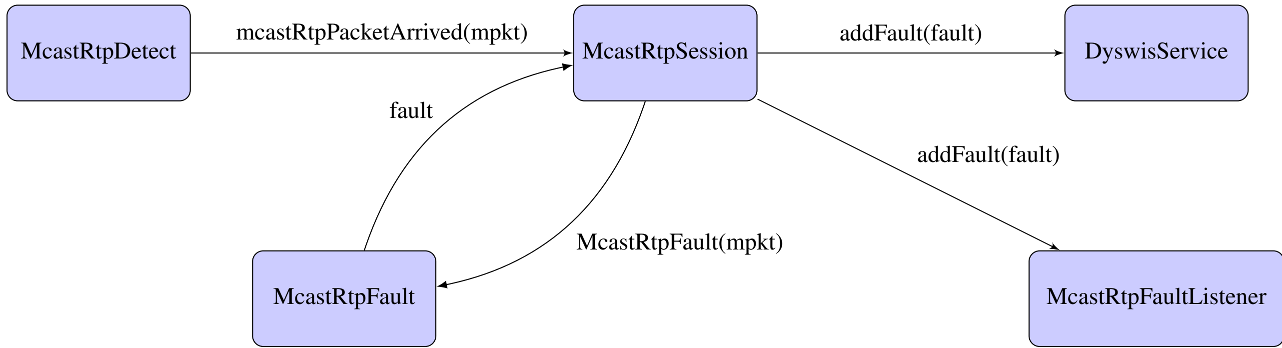

Figure 5: McastRtpSession and its interfaces

static method, checkPacket(), that allows McastRtpDetect to perform a quick check on the packet prior to passing it to the McastRtpPacket constructor.

5.3 McastRtpSession

The McastRtpSession java class is used to represent multicast streams. A session is uniquely identified by the source IP address, the source UDP port, the destination IP address (i.e. multicast group), and destination UDP port. McastRtpPackets that have all these values in common are considered to belong to the same session. For the purposes of this paper, the terms stream and session may be used interchangeably.

In addition to these session identifiers, information is also kept about the time the session started, the time the last packet was received as part of this session, the last RTP sequence number observed, and the next RTP sequence number expected.

McastRtpSession is responsible for detecting sequence number gaps. When a gap is detected, a McastRtpFault must be created. The fault is then added to the Dyswis GUI, as well as to the fault history maintained by the McastRtpFaultListener. This is illustrated in Figure 5.

5.4 McastRtpFault

The McastRtpFault java class is used to represent faults that occur within a specific McastRtpSession. Currently, the only fault type supported is a sequence number fault. RTP sequence numbers increase monotonically, wrapping when they reach a maximum value. An McastRtpFault is generated by the McastRtpSession module when an RTP packet is received with a sequence number that is unexpected, i.e. not one more than the previously received sequence number for that stream.

Note that stream loss alone does not cause an McastRtpFault to be generated. There is no timer mechanism that ensures that the next packet in a stream is received within a specific timeframe. It is only when another packet in the stream is received that the fault, or sequence number gap, is detected.

5.5 McastRtpFaultListener

The McastRtpFaultListener java class is used to maintain a history of McastRtpFaults. Although some similar functionality exists in the DYSWIS framework, the interfaces to search and retrieve faults from the history were not exposed or available to non-“core” modules.

When a fault is added by the McastRtpSession module to the global fault history, it is also added to the McastRtpFaultListener history. The faults can then be retrieved by the Analysis (i.e. Probe) modules.

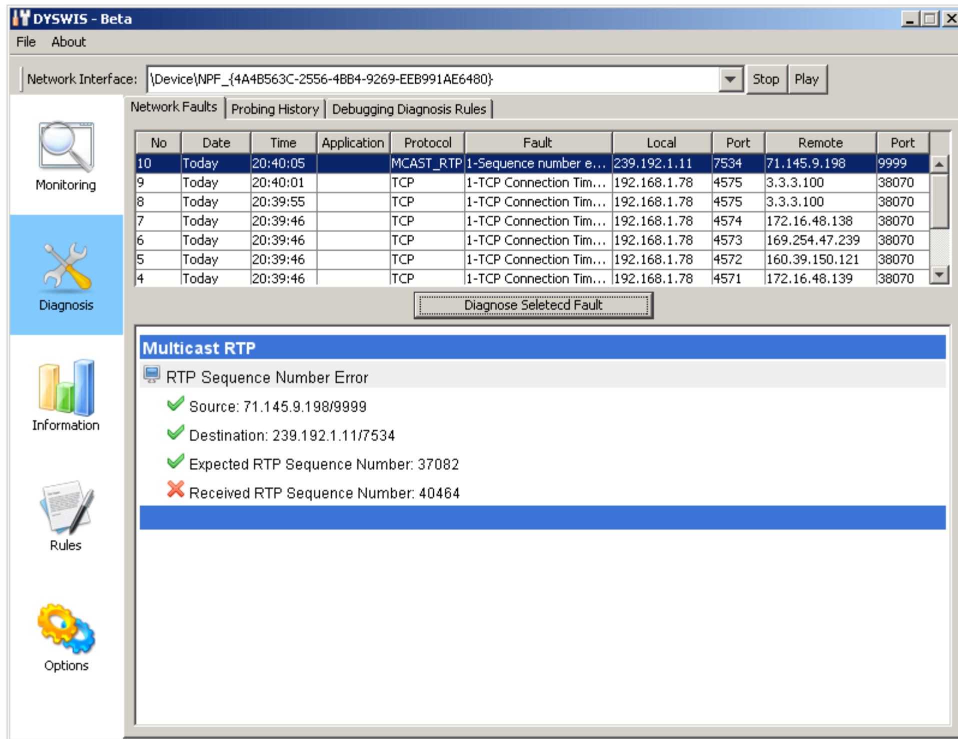

Figure 6: McastRtpProbe output

5.6 Probes

In the DYSWIS architecture, a Probe is used to gather information about a fault. When a fault occurs, it is displayed in the GUI in the Network Faults table. The user can then select the fault and "diagnose" it, which causes the Rule Engine to initiate the appropriate probe (or probes) for that type of fault.

Probes can run on the local node (LocalProbe), they can run on a remote node (RemoteProbe), and they can run on several remote nodes at once (MultiProbe).

Several probes were written for this project:

- McastRtpProbe, which displays information about the fault in the GUI
- McastRtpReversePathDiscovery, which calculates the path from the local or remote node to the multicast source
- McastRtpFaultHistoryProbe, which queries the local or remote fault history
- McastRtpSessionHistoryProbe, which queries the local or remote session history
- McastRtpRemoteProbe, which incorporates the functionality of the previous 3 probes

5.6.1 McastRtpProbe

The McastRtpProbe java class is designed to be invoked by the Jess rules engine, and displays basic information about an McastRtpFault in the Diagnosis window of the GUI. Figure 6 shows a sample of the output displayed. Included is the source IP address and port, the multicast group IP address and destination UDP port, the RTP sequence number that was expected, and the RTP sequence number that was received.

5.6.2 McastRtpReversePathDiscovery

This probe calculates the path to the multicast source. It can be run on the local node, or it can be invoked as a remote probe, in which case the functionality is the same, but no GUI messages are displayed.

There were a few optimizations made here that are worth mentioning. One is that the traceroute results are cached. When the probe is called, the cache is checked first. The other optimization, mentioned previously, is that the traceroute itself is performed as soon as a new session is detected, which can greatly reduce the time it takes for a remote probe to respond.

5.6.3 McastRtpSessionHistoryProbe

This probe queries the local or remote session history to see whether the client was receiving packets belonging to a stream during a specific time range. This is not entirely trivial. Although it is easy to look up a stream in the session history by source and destination IP address and UDP port, the question of whether or not the client was joined to that stream at a specific time is somewhat more tricky. The logic used is that if the first packet of the stream was received prior to the time of the fault, and at least one packet was received after the time of the fault, then this is considered a match.

At one point, this Probe was called directly by the Jess rules engine, but now its functionality has been moved to McastRtpRemoteProbe.

5.6.4 McastRtpFaultHistoryProbe

This simple probe queries the local or remote fault history to see whether a specific fault is present. For a fault to be considered a match, the session must match, and the expected and received sequence numbers must match. Note that in this case, no consideration is given to the time of the fault. It is theoretically possible that due to sequence number wrapping, two distinct faults may erroneously be considered to be the same, but this is unlikely.

At one point, this Probe was called directly by the Jess rules engine, but now its functionality has been moved to McastRtpRemoteProbe.

5.6.5 McastRtpRemoteProbe

This is a monolithic probe that gathers all the necessary data from a remote node about a fault, i.e. whether or not the remote node was joined to the same session, whether or not the fault was observed, and the hop list towards the multicast source. The motivation for combining these probes has already been discussed, but to restate briefly, individual probes executed in sequence are no guaranteed to communicate with the same remote node. In order to get all the information from the same node, a single probe is used.

5.7 Analysis Rules: MCAST_RTP_SEQ

Final analysis of the MultiProbe results is handled by the Jess rules engine. The decision logic for the identification of bad hops is shown in Figure 7, and matches very closely the algorithm described earlier.

6 Testing

Testing was performed throughout the development process. The initial focus was on successfully detecting and parsing multicast RTP packets. Once this was working, the traceroute functionality was implemented. After this, the fault history features were added, and then, the session history. Finally, the use of the MultiProbe and the analysis of the results was fleshed out.

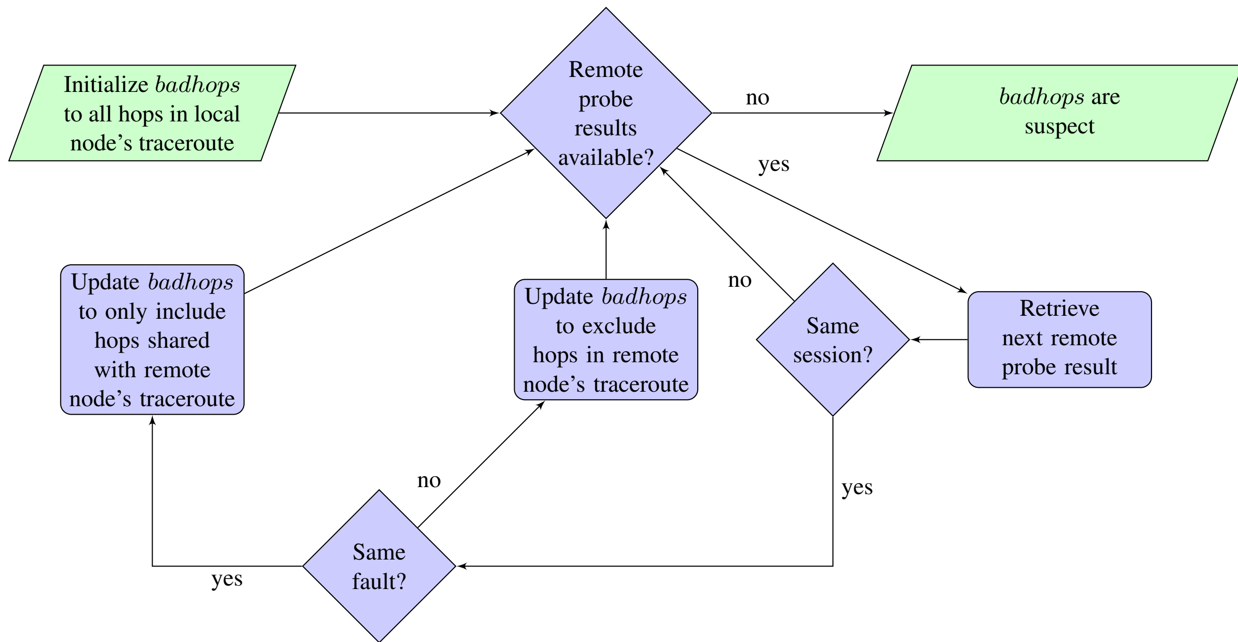

Figure 7: Decision tree for identifying suspect hops

6.1 Initial Test Environment

The majority of the development took place on a Windows XP virtual machine guest, hosted on a Mac OS X Lion host, using bridged networking. A nice side effect of this arrangement was that any packets destined for the guest also passed through the host NIC.

The other convenient aspect of the setup is that as an actual IPTV subscriber, I have multicast RTP streams available on my home network.

Therefore, in order to generate sequence number faults, I used VLC on the guest to join a multicast stream. This is illustrated in Figure 8. By simply stopping and then restarting the stream, a sequence number gap would occur. With the DYSWIS client running on both the XP guest and the OS X Lion host, both nodes would observe the exact same fault.

Figure 9 shows the network faults view in the DYSWIS client on the XP guest. An MCAST_RTP

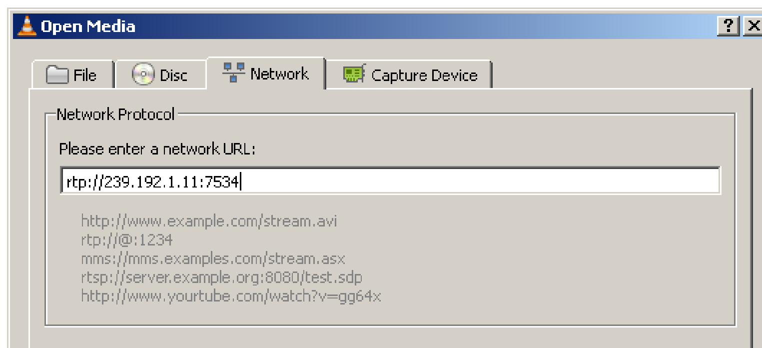

Figure 8: Opening a stream using VLC

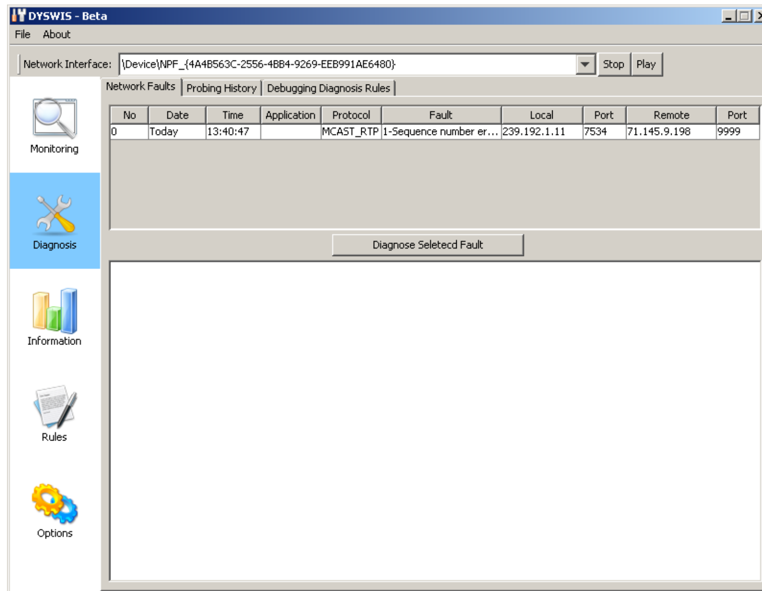

Figure 9: DYSWIS Network Faults view

sequence number fault is displayed, which can be diagnosed using the "Diagnose Seletecd Fault" button.

Once diagnosed, the results are displayed in the results window. Figure 10 and Figure 11 shows the results of the probe. Unfortunately this window is not resizable at this time.

Note that the OS X node responds that yes, it observed the same stream, and yes, it observed the same fault, and provides its traceroute results. Interestingly, the traceroute is not identical, it is missing a hop. This is due to the fact the the routers in the network respond to only a limited number of traceroute requests per minute.

6.2 Final Test Environment

Once the functionality was in place, the focus was placed on building a more interesting test topology with more nodes. This topology is illustrated in Figure 12, which was built using actual Cisco routers and Mac and Windows laptops, and Figure 13, which was completely virtualized, using GNS3 to emulate the routers and VirtualBox to host Windows XP virtual machines.

From a logical standpoint, the topologies are virtually identical. The only difference is that the physical topology has the addition of R3, which performs a NAT function. In the virtual topology, routers R1 and R2 perform NAT. In both cases, this is NATing 10.x.x.x to 192.168.x.x, and the residential gateway performs another NAT operation to from 192.168.x.x to a public IP address.

6.2.1 Multicast Topology

Only the routers R1 and R2 participate in multicast routing. They are configured with PIM sparse mode. R1 is the RP for the multicast groups 239.1.0.0/16. R2 is configured as the RP for multicast groups 239.2.0.0/16. SRC10 is the multicast source, and RCVR20, RCVR30, and RCVR40 are multicast receivers. VLC is used on the multicast server to stream an MPEG file. VLC was also used on the receivers to view the stream. The receivers run DYSWIS.

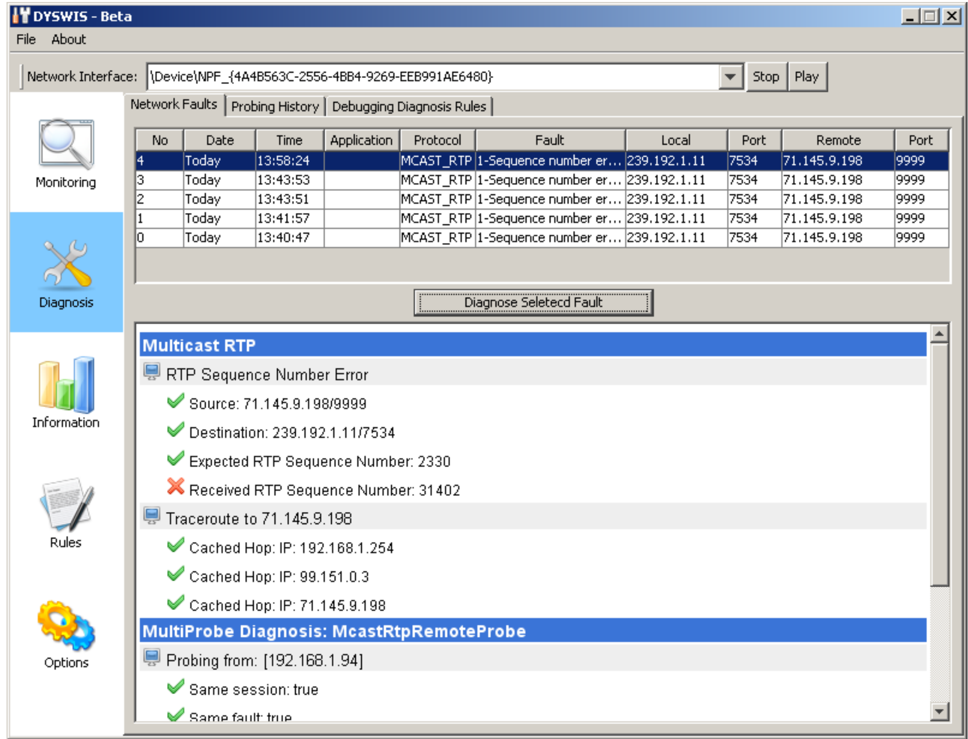

Figure 10: Diagnosis results - top half

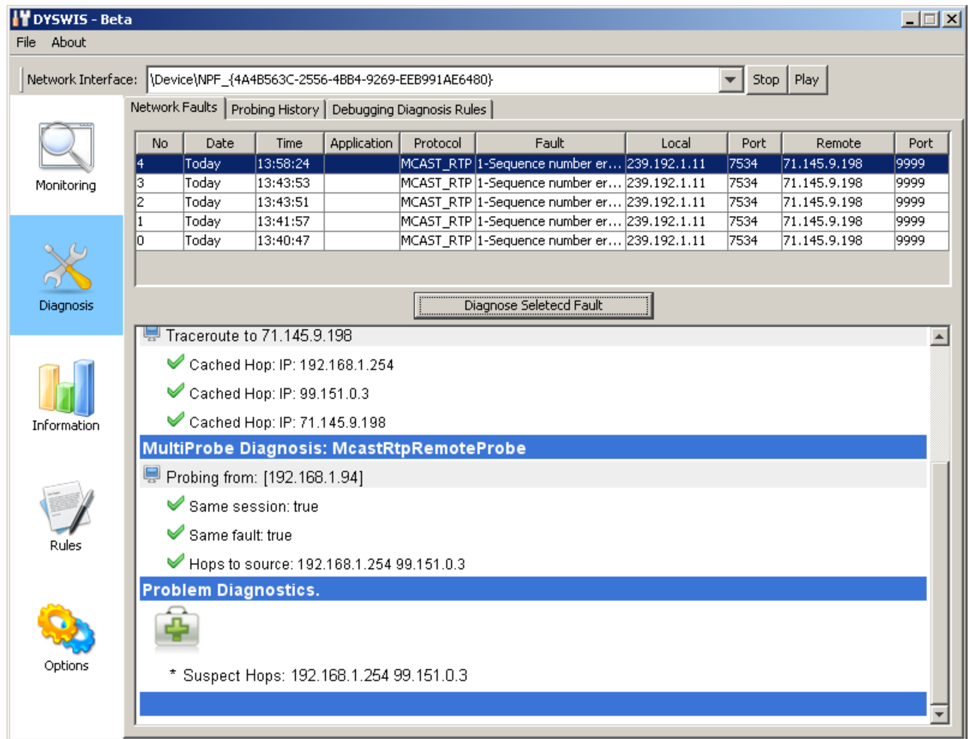

Figure 11: Diagnosis results - bottom half

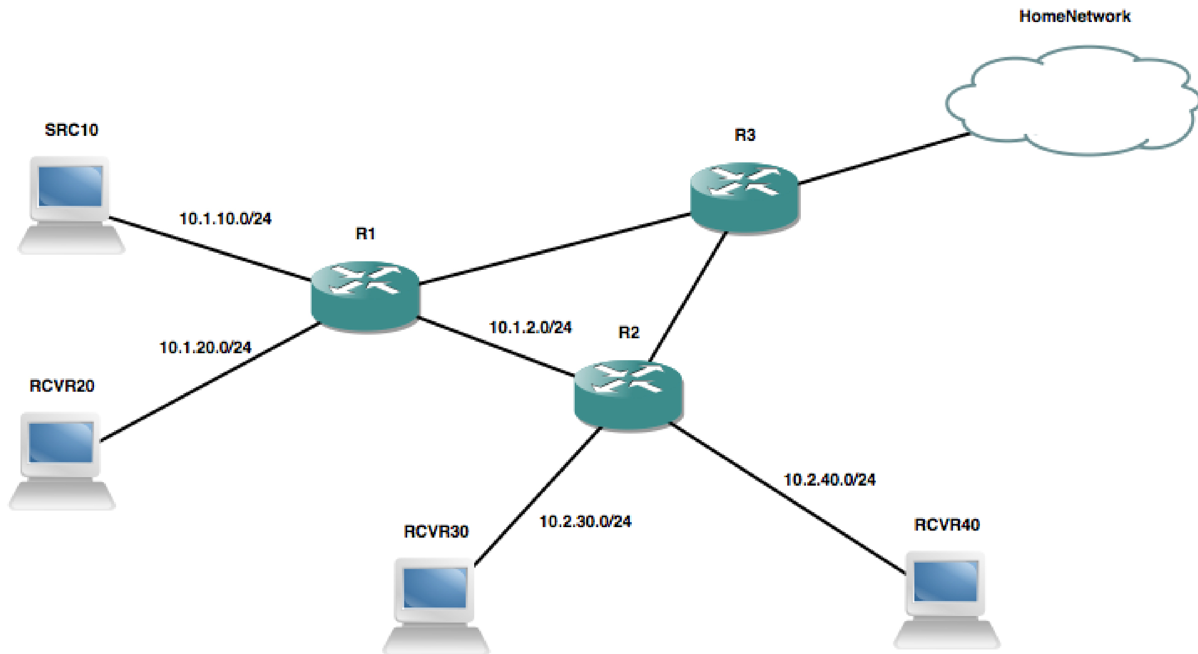

Figure 12: Test network topology - physical

6.2.2 IP Addressing

The test topology uses the 10.0.0.0/8 private address space. Each node is numbered, and the last octet of the IP address of the node is its number. The subnet connecting node a and node b is 10.a.b.0/24. The IP addresses are summarized in the following table:

IP Address	Node	Interface	Connected To	
10.1.2.1	R1	fa3/0	R2	fa3/0
10.1.2.2	R2	fa3/0	R1	fa3/0
10.1.10.1	R1	fa1/0	SRC10	
10.1.10.10	SRC10		R1	fa1/0
10.1.20.1	R1	fa2/0	RCVR20	
10.1.20.20	RCVR20		R1	fa2/0
10.2.30.2	R2	fa1/0	RCVR30	
10.2.30.30	RCVR30		R2	fa1/0
10.2.40.2	R2	fa2/0	RCVR40	
10.2.40.40	RCVR40		R2	fa2/0

6.2.3 Generating Loss Events

In order to generate loss events, an access-list was applied on the router links one at a time. Specifically, the following access list was used:

```
ip access-group extended BLOCK_MULTICAST
 deny   ip any host 239.0.0.0 0.255.255.255
 permit ip any any
```

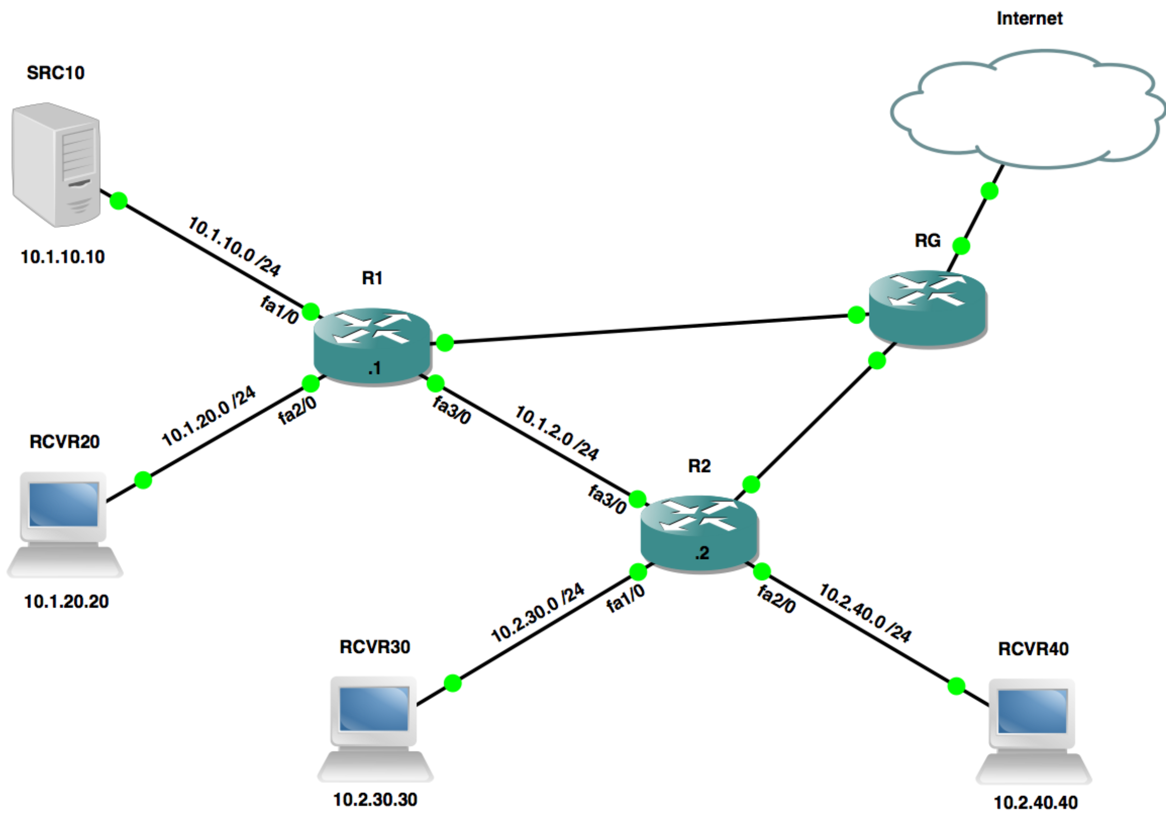

Figure 13: Test network topology - virtualized

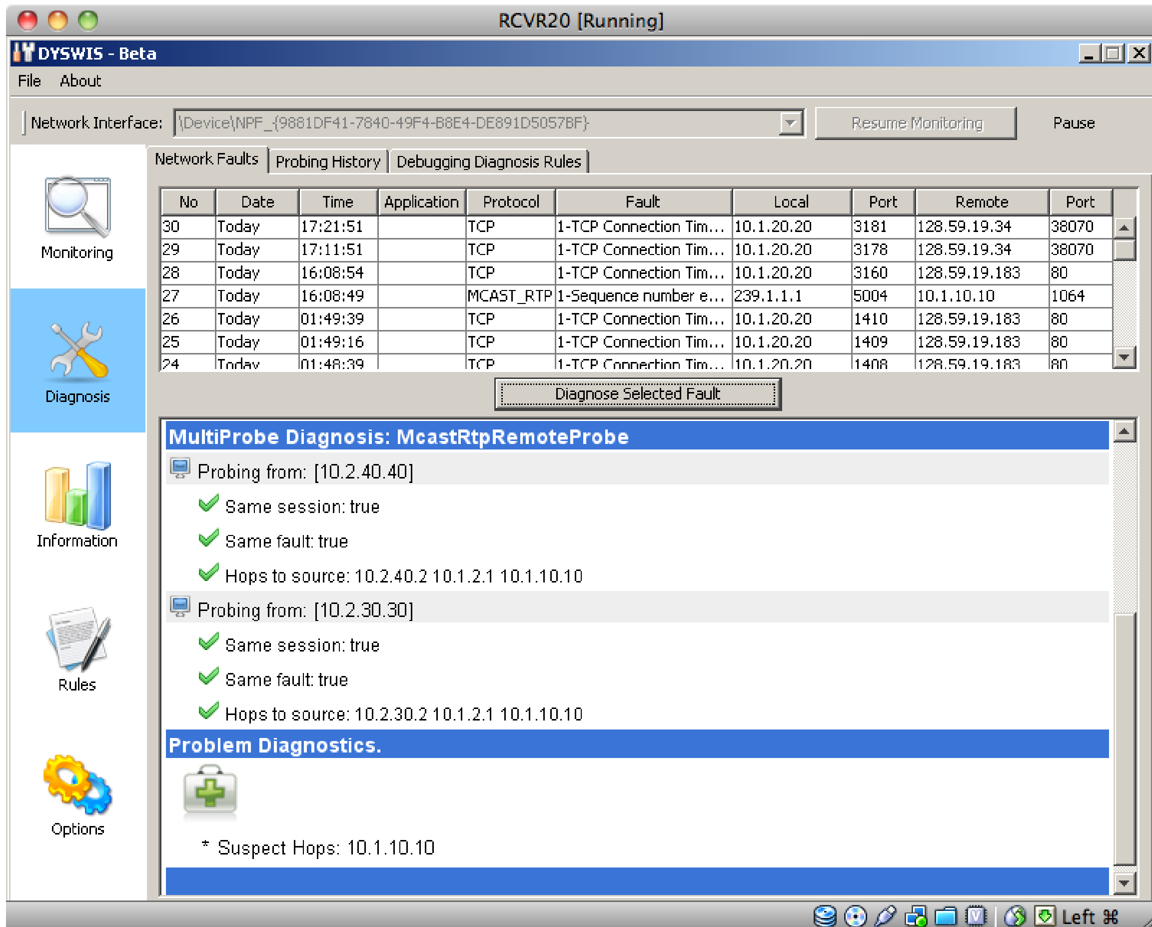

Figure 14: Test Case 1 - Loss between SRC10 and R1

For example, when applied on R1 inbound on interface fa1/0, this would block all the multicast coming in to the router from SRC10, and all downstream receivers would experience loss. Or, when applied outbound on fa1/0 on R2, towards RCVR30, only RCVR30 would experience loss. The loss would be readily observably as a freeze in video.

6.3 Test Results

The testing was largely sucessful. There were some early problems due to the video file that was used - I attempted to stream a high definition home movie at close to 10 Mbps, which proved to be too much for the CPUs on the test machines to encode and decode. There were many video artifacts in the absence of packet loss, which wasn't really the goal.

Other than that, the system works as expected. DYSWIS was able to correctly identify the links on which the access-lists were applied.

6.3.1 Test Case 1: Multicast blocked inbound on R1 fa1/0 (between SRC10 and R1)

Loss was observed on RCVR20, RCVR30, and RCVR40. Diagnosis was performed on RCVR20.

The following table summarizes the information that was reported by each node:

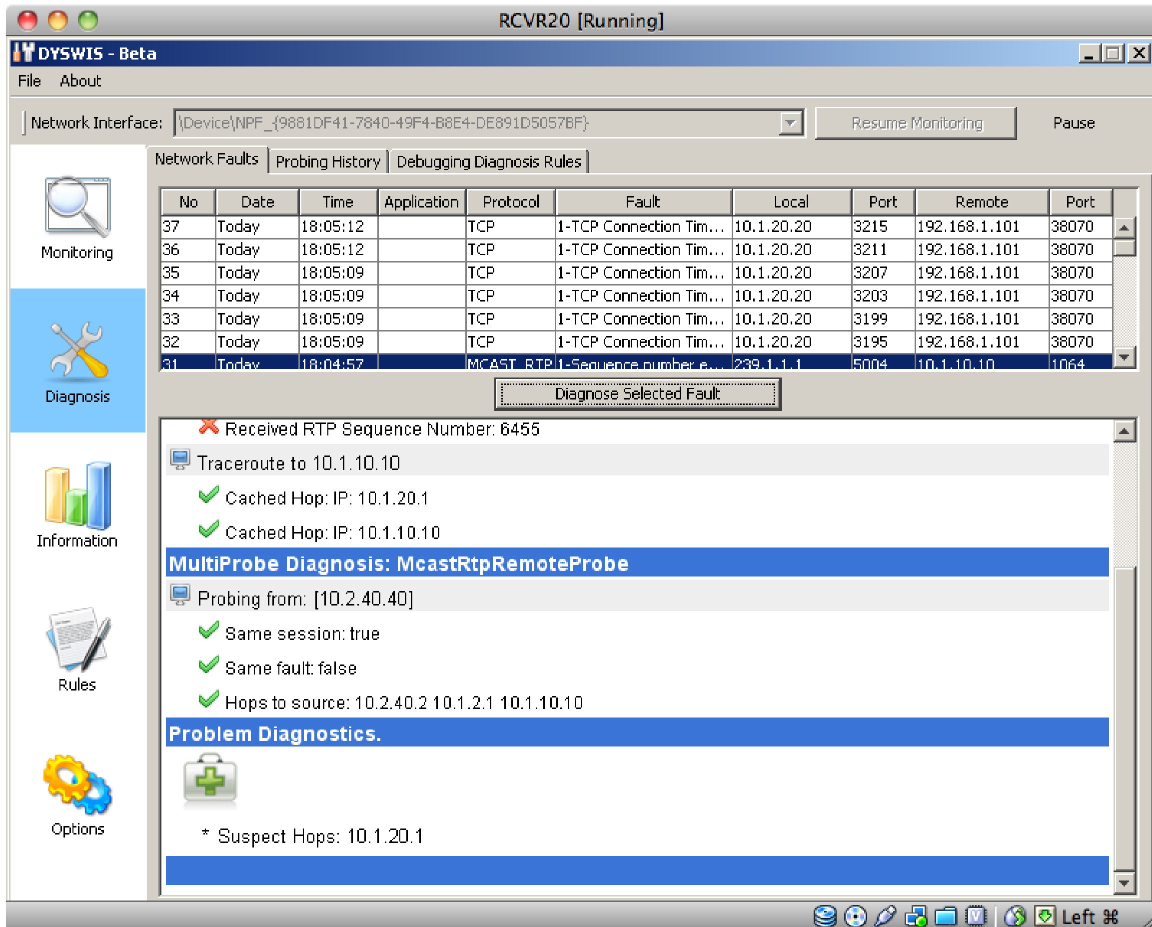

Figure 15: Test Case 2 - Loss between R1 and RCVR20

Node	Same Session	Same Fault	Hops
RCVR20			{ 10.1.20.1, 10.1.10.10 }
RCVR30	Y	Y	{ 10.2.30.2, 10.1.2.1, 10.1.10.10 }
RCVR40	Y	Y	{ 10.2.40.2, 10.1.2.1, 10.1.10.10 }

Using this information, RCVR20 was able to concluded that the hop at fault was 10.1.10.10, the only one common to all the traceroutes. Referring to the IP address table, this points to the link between R1 and SRC10, which is in fact where the loss occurred.

6.3.2 Test Case 2: Multicast blocked outbound on R1 fa2/0 (between R1 and RCVR20)

Loss was observed on RCVR20, and diagnosis was performed on RCVR20.

The following table summarizes the information that would have been reported by each node:

Node	Same Session	Same Fault	Hops
RCVR20			{ 10.1.20.1, 10.1.10.10 }
RCVR30	Y	N	{ 10.2.30.2, 10.1.2.1, 10.1.10.10 }
RCVR40	Y	N	{ 10.2.40.2, 10.1.2.1, 10.1.10.10 }

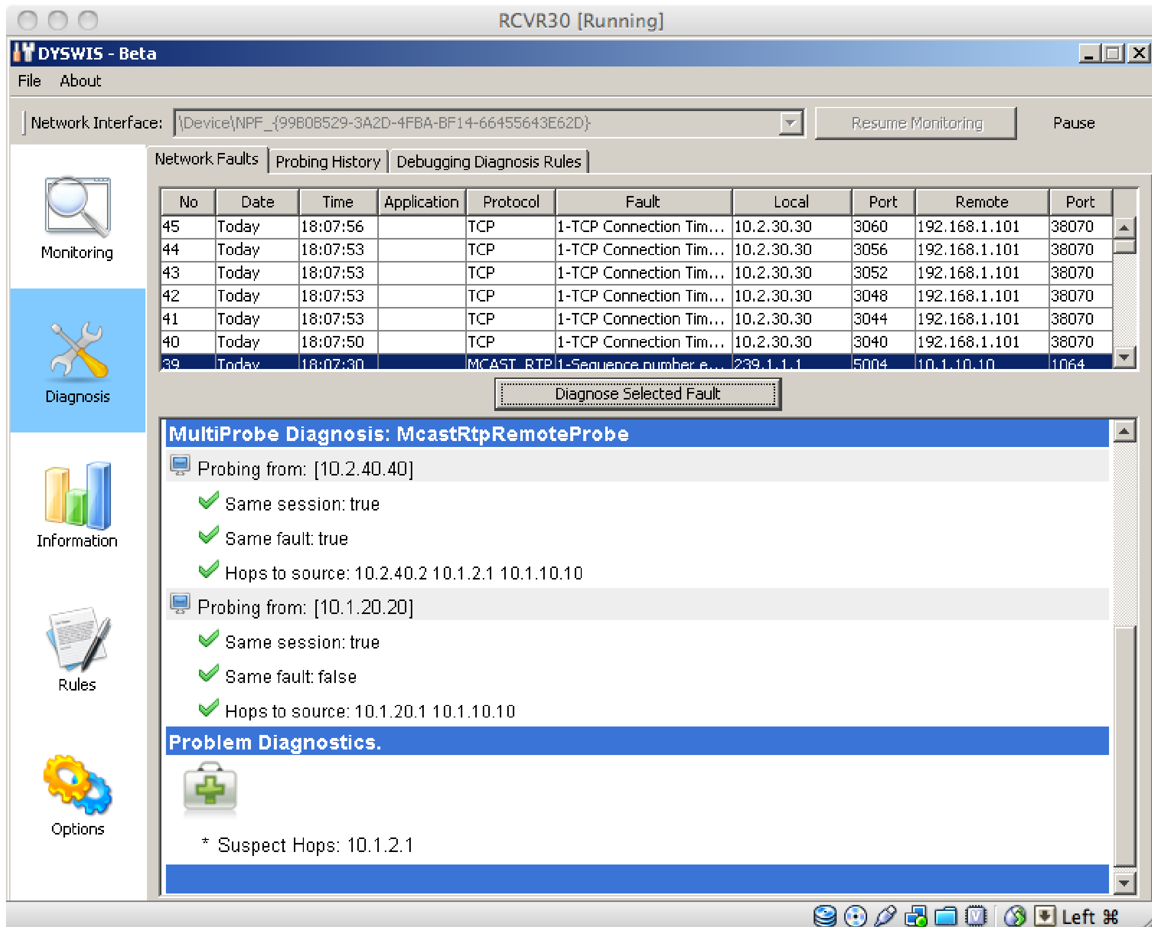

Figure 16: Test Case 3 - Loss between R1 and R2

In fact, as Figure 15 shows, RCVR20 only needed the information from RCVR40 to compute the bad hop as the link between R1 and RCVR20.

$$\{10.1.20.1, 10.1.10.10\} \setminus \{10.2.40.2, 10.1.2.1, 10.1.10.10\} = \{10.1.20.1\}$$

6.3.3 Test Case 3: Multicast blocked outbound on R1 fa3/0 (between R1 and R2)

Loss was observed on RCVR30 and RCVR40. Diagnosis was performed on RCVR30.

The table below summarizes the results from each probe:

Node	Same Session	Same Fault	Hops
RCVR30			{ 10.2.30.2, 10.1.2.1, 10.1.10.10 }
RCVR20	Y	N	{ 10.1.20.1, 10.1.10.10 }
RCVR40	Y	Y	{ 10.2.40.2, 10.1.2.1, 10.1.10.10 }

Taking the information from RCVR40 first, we have:

$$\{10.2.30.2, 10.1.2.1, 10.1.10.10\} \cap \{10.2.40.2, 10.1.2.1, 10.1.10.10\} = \{10.1.2.1, 10.1.10.10\}$$

Then factoring in the information from RCVR20, we have:

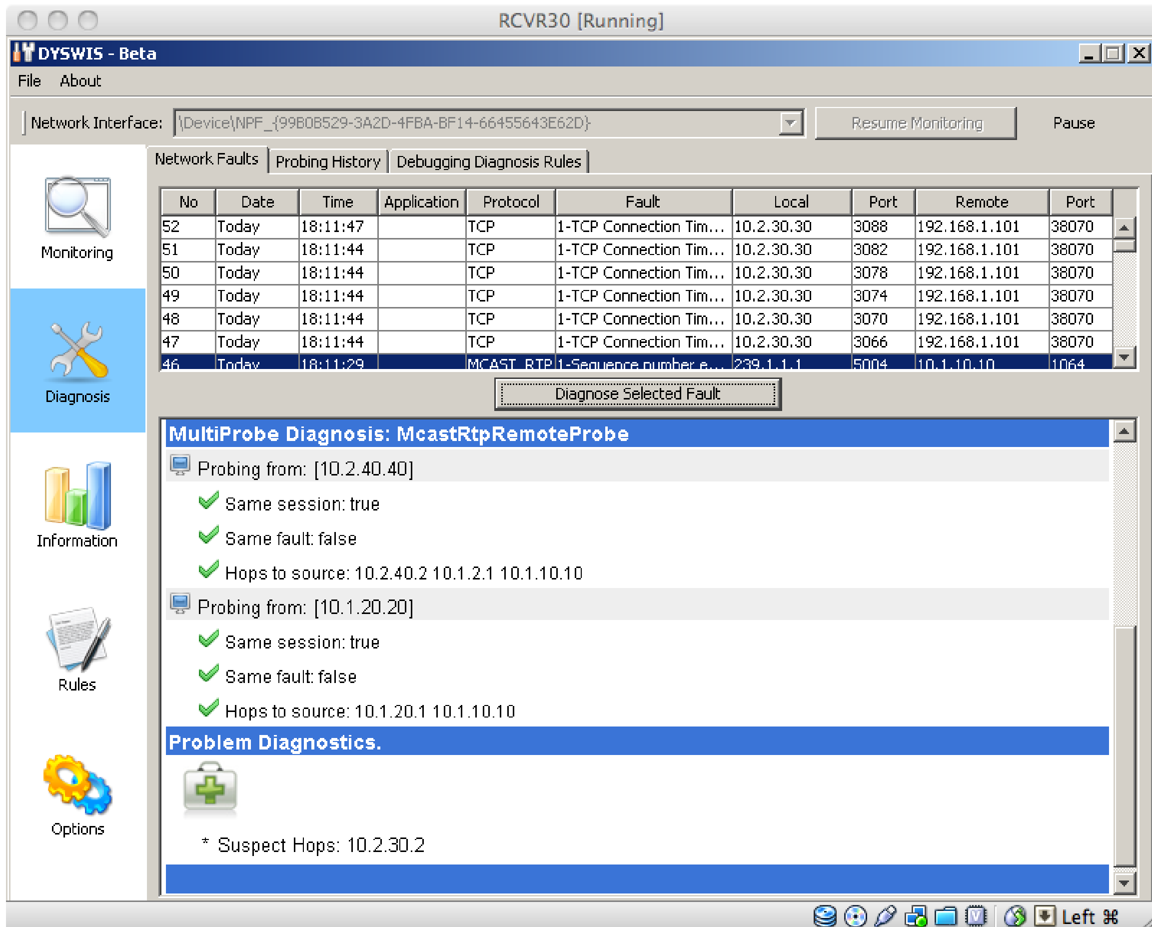

Figure 17: Test Case 4 - Loss between R2 and RCVR30

$$\{10.1.2.1, 10.1.10.10\} \setminus \{10.1.20.1, 10.1.10.10\} = \{10.1.2.1\}$$

10.1.2.1 is the link between R1 and R2, which is where the multicast was blocked.

6.3.4 Test Case 4: Multicast blocked outbound on R2 fa1/0 (between R2 and RCVR30)

Loss was observed on RCVR30 only, and the diagnosis was applied there.

Again, the table summarizes the results:

Node	Same Session	Same Fault	Hops
RCVR30			{ 10.2.30.2, 10.1.2.1, 10.1.10.10 }
RCVR20	Y	N	{ 10.1.20.1, 10.1.10.10 }
RCVR40	Y	N	{ 10.2.40.2, 10.1.2.1, 10.1.10.10 }

Just using the information from RCVR40, we can see that

$$\{10.2.30.2, 10.1.2.1, 10.1.10.10\} \setminus \{10.2.40.2, 10.1.2.1, 10.1.10.10\} = \{10.2.30.2\}$$

Indeed, this is where the loss occurred - between R2 and RCVR30.

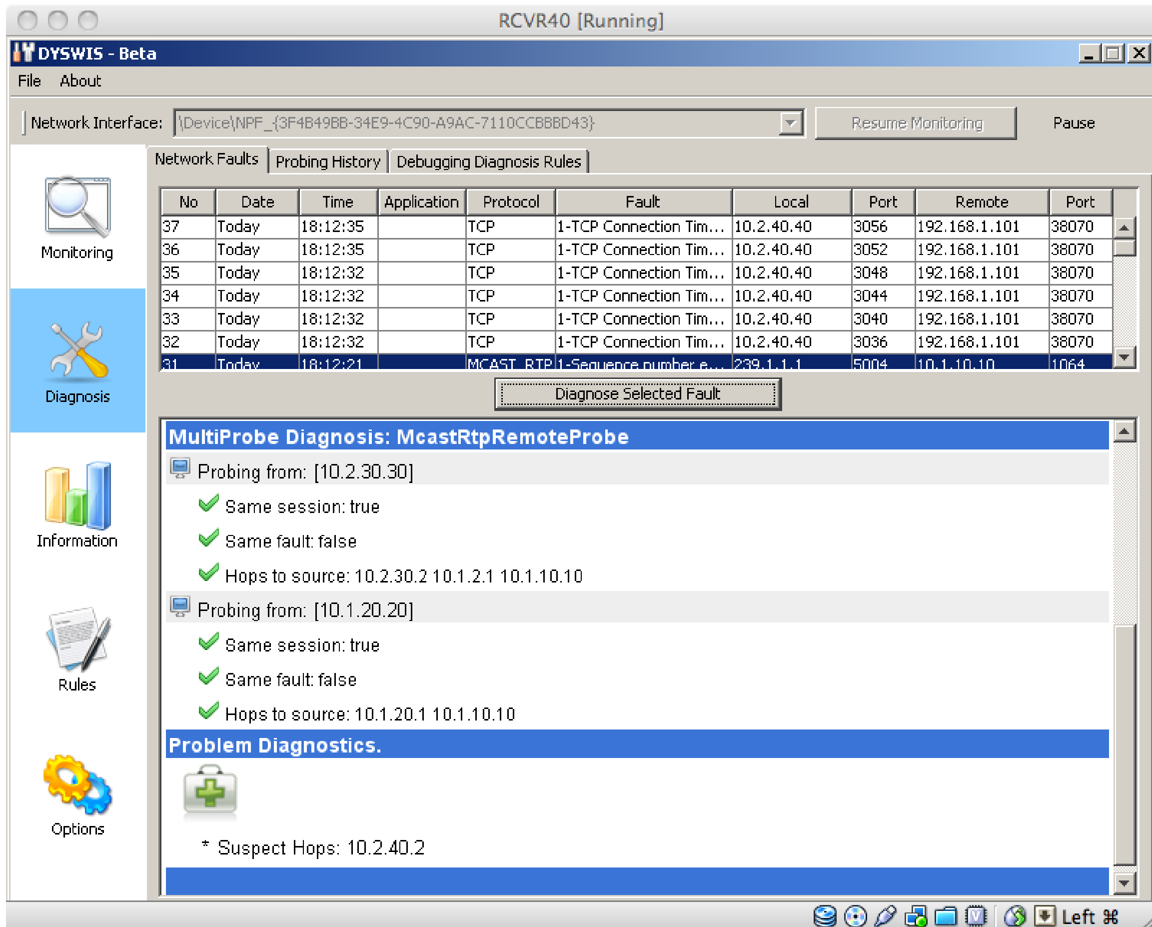

Figure 18: Test Case 5 - Loss between R2 and RCVR40

6.3.5 Test Case 5: Multicast blocked outbound on R2 fa2/0 (between R2 and RCVR40)

This is essentially the same as the previous test case. DYSWYS diagnosis on RCVR40 correctly identified the loss between R2 and RCVR40.

7 Future Work

The current model does not take into account all the information that may potentially be available. For example, if a node is joined to a different stream than the one on which a fault was detected, but it is coming from the same source, this information could be useful.

The current implementation is also somewhat overly trusting, in that the information received from the remote node is assumed to be accurate. This may not be true, especially in the case where a remote node claims to be joined to the same session *at the time the fault occurred* but did not see a fault - this is largely dependent on the time synchronization between the nodes, and that is not guaranteed. It may be better to apply some sort of weight to the information and calculate the probabilities of various hops being bad.

8 Conclusion

In this paper, we proposed an approach to isolate faults in a multicast distribution tree. We showed that it was feasible to use the DYSWIS framework to perform this fault isolation, and successfully demonstrated this capability.

9 References

1. DYSWIS - http://www.cs.columbia.edu/irt/project/dyswis/
2. Jess rules engine - http://www.jessrules.com/jess/index.shtml
3. RTP - http://www.cs.columbia.edu/irt/software/rtptools/
4. JPCAP - http://sourceforge.net/projects/jpcap/files/jpcap/v0.01.16/
5. GNS3 - http://www.gns3.net/
6. VirtualBox - https://www.virtualbox.org/

A Source Code: MCAST_RTP_SEQ.jess

This section contains the Jess source code that calls the probes and correlates the results, implementing the algorithm described in Figure 7. One thing that is worth noting is that the MultiProbe specifies "same_nat" as the type of remote node to query. This is correct given the test environment, since the entire testbed is behind a single public IP address. This would likely need to change in other environments.

The code has been checked in to the SVN repository at svn://erie.cs.columbia.edu/dyswis.

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;; COMS E6181: Advanced Internet Services (Fall 2011)
;; pgs2111 at columbia.edu
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(load-function edu.columbia.cs.dyswis.core.diagnosis.LocalProbe)
(load-function edu.columbia.cs.dyswis.core.diagnosis.RemoteProbe)
(load-function edu.columbia.cs.dyswis.core.diagnosis.MultiProbe)
(load-function edu.columbia.cs.dyswis.core.diagnosis.Analysis)

;; maximum number of remote nodes to query
(bind ?MAX_REMOTE_PROBES 3)

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;; Analyze an Multicast RTP sequence number gap
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
(deffunction MCAST_RTP_SEQ (?fault)

    ;; Display basic information about the multicast stream that experienced the fault:
    ;;    - Source ip/port
    ;;    - Multicast ip/port
    ;;    - Expected sequence number
    ;;    - Received sequence number
    (bind ?result (LocalProbe "McastRtpProbe" ?fault))

    ;; Display the path towards the multicast source IP
    (bind ?traceresult (explode$ (LocalProbe "McastRtpReversePathDiscovery" ?fault)))
```

```
    ;; For up to MAX_REMOTE_PROBES remote nodes:
    ;;   - checks if the remote was joined to the same session
    ;;   - checks if the same loss occurred on the remote
    ;;   - gets the traceroute from the remote to the source
    (bind ?allresults (MultiProbe "McastRtpRemoteProbe" "same_nat" ?MAX_REMOTE_PROBES
            "false" ?fault))

    ;; This is our list of suspect hops
    ;; At first, anything node in our traceroute could be bad
    (bind ?badhops ?traceresult)

    ;; Loop through each result.
    ;; Each result consists of:
    ;;    -  true/false : whether or not the remote was joined to the same session
    ;;    -  true/false : whether or not the remote experienced the same fault
    ;;    -  nodelist   : the remote traceroute to the source
    (foreach ?remoteresult ?allresults

        ;; explode the space-separate string into a list
        (bind ?remoteresultlist (explode$ ?remoteresult))

        ;; grab the samesession true/false value from the head of the list
        (bind ?samesession (nth$ 1 ?remoteresultlist))
        (bind ?remoteresultlist (rest$ ?remoteresultlist))

        ;; grab the samefault true/false value from the head of the list
        (bind ?samefault (nth$ 1 ?remoteresultlist))
        (bind ?remoteresultlist (rest$ ?remoteresultlist))

        ;; was the remote node joined to the same multicast group at the same time?
        (if (eq ?samesession true) then

            ;; if this node and the remote node experienced the same fault,
            ;; then the common hops are suspect
            (if (eq ?samefault true) then
                (bind ?badhops (intersection$ ?remoteresultlist ?badhops))

                ;; but if the remote node did not experience the same fault,
                ;; the remote traceroute hops are ok.
                else
                ;; BE CAREFUL - ORDER MATTERS!
                ;; complement$ returns all elements of the second list
                ;; that are not in the first list
                (bind ?badhops (complement$ ?remoteresultlist ?badhops))
                )

            )
        )

    ;; Report out the list of suspicious hops
    (Analysis (str-cat "Suspect Hops: " (implode$ ?badhops)))
    )

;;;;;;;;;;;;;;;
;; Main Program
;;;;;;;;;;;;;;;
(defrule MAIN::MCAST_RTP_SEQ
    (declare (auto-focus TRUE)) => (MCAST_RTP_SEQ (fetch FAULT))
```

```
)
```

B Source Code: module_mcast_rtp

This section contains the Java source code for the module_mcast_rtp package, which was authored for this project. The code has been checked in to the SVN repository at svn://erie.cs.columbia.edu/dyswis.

B.1 Activator.java

```
/**
 * COMS E6181: Advanced Internet Services (Fall 2011)
 * pgs2111 at columbia.edu
 */
package edu.columbia.cs.dyswis.module_mcast_rtp;

import java.util.Properties;

import org.apache.log4j.Logger;
import org.osgi.framework.BundleActivator;
import org.osgi.framework.BundleContext;
import org.osgi.util.tracker.ServiceTracker;

import edu.columbia.cs.dyswis.common.baseclass.DysProbe;
import edu.columbia.cs.dyswis.service.DyswisService;
import edu.columbia.cs.dyswis.service.MainService;

/**
 * Bundle activator for the module_mcast_rtp package.
 *
 * @author phil
 */
public class Activator implements BundleActivator {

    /** logger */
    private static Logger logger = Logger.getLogger("Module McastRtp");

    /** The name of this package */
    private static final String packageName = "module_mcast_rtp";

    /**
     * The execution context of this bundle
     */
    private BundleContext m_context;

    /**
     * Dyswis Core Service tracker
     */
    public static ServiceTracker m_tracker;

    /**
     * Dyswis main service tracker
     */
    public static ServiceTracker m_mainTracker;

    /**
     * Multicast RTP detection module
```

```
     */
    public static McastRtpDetect mcastRtpDetect = null;

    /**
     * Multicast RTP fault listener
     */
    public static McastRtpFaultListener mcastRtpFaultListener = null;

    /**
     * Get a handle on the DyswisService
     *
     * @return the DyswisService
     */
    public static DyswisService getDyswis() {
        return (DyswisService) m_tracker.getService();
    }

    /**
     * Get a handle on the MainService
     *
     * @return the MainService
     */
    public static MainService getMain() {
        return (MainService) m_mainTracker.getService();
    }

    /**
     * Start the module_mcast_rtp bundle
     * <p>
     * Registers probes, starts the fault listener, and starts the detection
     * module.
     */
    @Override
    public void start(BundleContext arg0) throws Exception {

        logger.debug("Module McastRtp loading..");

        m_context = arg0;
        m_tracker =
                new ServiceTracker(m_context,
                        m_context.createFilter("(&(objectClass="
                                + DyswisService.class.getName() + ")"
                                + "(Name=DyswisService))"), null);
        m_tracker.open();

        m_mainTracker =
                new ServiceTracker(m_context,
                        m_context.createFilter("(&(objectClass="
                                + MainService.class.getName() + ")"
                                + "(Name=MainService))"), null);
        m_mainTracker.open();

        logger.debug("Module McastRtp has been loaded");

        /*
         * register the probes with the DyswisService
         */
        contextRegisterService("McastRtpProbe", m_context);
        contextRegisterService("McastRtpFaultHistoryProbe", m_context);
```

```
    contextRegisterService("McastRtpReversePathDiscovery",
            m_context);
    contextRegisterService("McastRtpSessionHistoryProbe", m_context);
    contextRegisterService("McastRtpRemoteProbe", m_context);

    /*
     * Start the McastRtpFaultListener
     */
    mcastRtpFaultListener = new McastRtpFaultListener();

    /*
     * Start the McastRtpDetect module
     */
    mcastRtpDetect = new McastRtpDetect();
    getMain().addPacketCapture(mcastRtpDetect);
}

/**
 * Stops the module_mcast_rtp bundle
 */
@Override
public void stop(BundleContext arg0) throws Exception {

    logger.debug("Module McastRtp is being unloading..");
    getMain().removePacketCapture(mcastRtpDetect);
    mcastRtpFaultListener = null;
    mcastRtpDetect = null;

    m_tracker.close();
    m_mainTracker.close();
    logger.debug("Module McastRtp has been unloaded");
}

/**
 * Register the probe.
 *
 * @param className
 *            java class of the probe
 * @param context
 *            the execution context of the bundle
 */
public void contextRegisterService(String className,
        BundleContext context) {

    Properties props = new Properties();

    props.put("Package", packageName);
    props.put("Probe", className);

    String classFullPath =
            "edu.columbia.cs.dyswis." + packageName + "."
                    + className;
    try {

        Class<?> obj;
        obj = Class.forName(classFullPath);
        DysProbe dp = (DysProbe) obj.newInstance();
        context.registerService(DysProbe.class.getName(), dp, props);
```

```
        } catch (Exception e) {
            e.printStackTrace();
        }
    }
}
```

B.2 McastRtpDetect.java

```
/**
 * COMS E6181: Advanced Internet Services (Fall 2011)
 * pgs2111 at columbia.edu
 */
package edu.columbia.cs.dyswis.module_mcast_rtp;

import java.util.Collections;
import java.util.LinkedHashMap;
import java.util.Map;

import org.apache.log4j.Logger;

import net.sourceforge.jpcap.net.Packet;
import edu.columbia.cs.dyswis.common.baseclass.DetectInterface;
import edu.columbia.cs.dyswis.common.baseclass.DysDetect;

/**
 * Detector and Session History maintainer for multicast RTP packets.
 * <p>
 * Listens for multicast RTP packets. Maintains a history of all multicast RTP
 * sessions. Raises faults if sequence number gaps are observed.
 *
 * @author phil
 */
public class McastRtpDetect extends DysDetect implements
        DetectInterface {

    /** for log messages */
    private static Logger logger = Logger.getLogger("McastRtpDetect");

    /**
     * Map of session identifiers to detected sessions.
     *
     * @see McastRtpSession#makeSessionIdentifier(McastRtpPacket)
     */
    private Map<String, McastRtpSession> mcastRtpSessionMap = null;

    /**
     * Default pcap filter. Looks for multicast destination address and RTP
     * version 2.
     */
    private static final String defaultPcapFilter =
            "net 224.0.0.0/4 and (udp[8:1]&0x80)==0x80";

    /**
     * Name of protocol
     */
    private static final String protocol = "MCAST_RTP";

    /**
     * Default constructor
```

```
 */
public McastRtpDetect() {
    super();

    filter = defaultPcapFilter;
    mcastRtpSessionMap =
            Collections
                    .synchronizedMap(new LinkedHashMap<String, McastRtpSession>());

    logger.info("McastRtpDetect starting");
}

/**
 * Handles a received packet. If it is a Multicast RTP packet, finds an
 * existing session or creates a new one. Then notifies the session that a
 * packet has arrived.
 *
 * @param packet
 *            the Packet that just arrived.
 * @see McastRtpSession#packetArrived(Packet)
 */
@Override
public void packetArrived(Packet packet) {

    // logger.debug("packetArrived: " + packet.toString());

    /*
     * this is a weird one. why are we getting packets that don't match our
     * filter?
     */
    if (!McastRtpPacket.checkPacket(packet)) {
        // logger.debug("packetArrived: checkPacket failed");
        return;
    }
    McastRtpPacket mcastRtpPacket = null;

    /*
     * Even if checkPacket succeeds, this still might not be a valid packet
     * .. we won't know for sure until we parse it.
     */
    try {
        mcastRtpPacket = new McastRtpPacket(packet);
    } catch (IllegalArgumentException iae) {
        logger.debug(iae);
        return;
    }

    /* moved by phil - don't count it unless it is valid */
    packetCount++;

    /*
     * Try to match the packet against an existing session, or create a new
     * session if necessary.
     */
    String mcastRtpSessionId =
            McastRtpSession.makeSessionIdentifier(mcastRtpPacket);

    if (mcastRtpSessionMap.containsKey(mcastRtpSessionId)) {
        mcastRtpSessionMap.get(mcastRtpSessionId)
```

```java
                .mcastRtpPacketArrived(mcastRtpPacket);
    } else {
        mcastRtpSessionMap.put(mcastRtpSessionId,
                new McastRtpSession(mcastRtpPacket));

        // run the RPF check now. we'll probably need it.
        Thread rpfThread =
                new McastRtpReversePathDiscoveryThread(
                        mcastRtpPacket.getSourceAddress());
        rpfThread.start();
    }

}

/**
 * Retrieve a session from the session history
 *
 * @param sessionId
 *            the session identifier
 * @return the associated session, or null if not found
 */
public McastRtpSession getSession(String sessionId) {
    return mcastRtpSessionMap.get(sessionId);
}

/**
 * Retrieve the number of multicast RTP sessions
 *
 * @return the number of multicast RTP sessions
 */
@Override
public int getSessionSize() {
    return mcastRtpSessionMap.size();
}

/**
 * Required override. Not used.
 */
@Override
public void loadConfig() {
    // TODO Auto-generated method stub
}

/**
 * Get the total number of McastRtpPackets for statistics collection
 *
 * @return the number of McastRtpPackets detected.
 */
@Override
public int getPacketCount() {
    return packetCount;
}

/**
 * Get the name of the protocol.
 *
 * @return the number of this protocol.
 */
@Override
```

```
    public String getProtocolName() {
        return protocol;
    }

}
```

B.3 McastRtpPacket.java

```
/**
 * COMS E6181: Advanced Internet Services (Fall 2011)
 * pgs2111 at columbia.edu
 */
package edu.columbia.cs.dyswis.module_mcast_rtp;

import java.io.Serializable;
import java.nio.ByteBuffer;

import org.apache.log4j.Logger;

import net.sourceforge.jpcap.net.Packet;
import net.sourceforge.jpcap.net.UDPPacket;
import edu.columbia.cs.dyswis.common.baseclass.DysPacket;

/**
 * This class represents a multicast RTP packet.
 * <p>
 * Specifically, this is an RTP version 2 packet destined for a multicast group
 * address.
 *
 * @author phil
 */
public class McastRtpPacket extends DysPacket implements Serializable {

    /** serialVersionUID so we can serialize the McastRtpPacket */
    private static final long serialVersionUID = 7184723097372608037L;

    /** for log messages */
    private static Logger logger = Logger.getLogger("McastRtpPacket");

    /** rtp header field */
    private byte version;
    /** rtp header field */
    private byte p;
    /** rtp header field */
    private byte x;
    /** rtp header field */
    private byte cc;
    /** rtp header field */
    private byte m;
    /** rtp header field */
    private byte pt;
    /** rtp header field */
    private int seq; /* 16 bits, but java has no unsigned short :( */
    /** rtp header field */
    private long ts; /* 32 bits, but java has no unsigned int :( */
    /** rtp header field */
    private long ssrc; /* 32 bits, but java has no unsigned int :( */
    /** rtp header field */
    private int csrc[];
```

```
/** rtp header field */
private short ext_type;
/** rtp header field */
private short ext_len;
/** rtp header field */
private int ext_data[];

/**
 * arrival time of the packet (millis)
 */
private long arrivalTime;

/**
 * @param packet
 */
public McastRtpPacket(Packet packet) {

    super(packet);
    arrivalTime = System.currentTimeMillis();

    transProtocol = P_UDP;
    applProtocol = P_MCAST_RTP;
    protocolName = "MCAST_RTP";

    if (!__parse(packet)) {
        throw new IllegalArgumentException(packet.toString());
    }

    // logger.debug("McastRtpPacket: " + udpPacket.toString());
}

/**
 * Mandatory override. Not really applicable for McastRtpPacket - always
 * returns false.
 *
 * @return false
 */
@Override
public boolean containsErrorResponse() {
    return false;
}

/**
 * Quick check to see if a packet meets the criteria for a multicast RTP
 * packet.
 * <p>
 * Needs to be: <br>
 * 1. UDP <br>
 * 2. Have a class D destination address <br>
 * 3. Be RTP version 2 <br>
 * 4. Not an RTCP packet <br>
 * <p>
 * Not a bad idea to call this before handing a packet to the constructor :)
 *
 * @param packet
 *            the packet to examine
 * @return false for an invalid packet, true if it might be valid
 */
public static boolean checkPacket(Packet packet) {
```

```
    /* make sure it's udp */
    if (!(packet instanceof UDPPacket)) {
        // logger.debug("checkPacket: NOT UDP: " + packet.toString());
        return false;
    }
    UDPPacket udpPacket = (UDPPacket) packet;

    /* check the first octet to make sure it is a class D */
    int b = udpPacket.getDestinationAddressBytes()[0] & 0xff;
    if (b < 224 || b > 239) {
        // logger.debug("checkPacket: NOT Multicast: " + packet.toString());
        return false;
    }

    /* check the length of the packet */
    if (udpPacket.getLength() < 12) {
        // logger.debug("checkPacket: NOT RTP version 2 (too short): " +
        // packet.toString());
        return false;
    }

    /* check the RTP version */
    b = udpPacket.getData()[0];
    if ((b & 0xc0) != 0x80) {
        // logger.debug("checkPacket: NOT RTP version 2 (incorrect version): "
        // + packet.toString());
        return false;
    }

    /* check if it might be RTCP */
    b = udpPacket.getData()[1];
    if ((b & 0xfe) == 0xc8) {
        // logger.debug("checkPacket: Looks like RTCP: " +
        // packet.toString());
        return false;
    }

    /* well, if it matches this far, that's a pretty good sign .. */
    return true;
}

/**
 * Actually parse the packet. Called by the constructor.
 *
 * @param packet
 *            the packet to parse
 * @return true if parse is successful, false otherwise
 */
public boolean __parse(Packet packet) {

    if (!checkPacket(packet)) {
        return false;
    }

    ByteBuffer buf = ByteBuffer.wrap(udpPacket.getUDPData());
    int len = buf.remaining();
    if (len < 12) {
        logger.error("packet too short");
        return false;
```

```
}

int word = buf.getInt();
version = (byte) ((word >> 30) & 0x03);
p = (byte) ((word >> 29) & 0x01);
x = (byte) ((word >> 28) & 0x01);
cc = (byte) ((word >> 24) & 0x0f);
m = (byte) ((word >> 23) & 0x01);
pt = (byte) ((word >> 16) & 0x7f);
seq = word & 0xffff;
ts = (buf.getInt() & 0xffffffffL);
ssrc = (buf.getInt() & 0xffffffffL);

/*
 * check the version first (checkPacket should have caught this)
 */
if (version != 2) {
    logger.error("RTP version incorrect");
    return false;
}

/*
 * Make sure this isn't RTCP. Check the payload type, and make sure that
 * it is neither RTCP_SR (200 = 0xc8) nor RTCP_RR (201 = 0xc9) In RTP,
 * the second byte consists of unsigned int m:1 unsigned int pt:7 In
 * RTCP, the second byte is just unsigned int pt:8
 */
if (m == 0x1 && (pt == 0x48 || pt == 0x49)) {
    logger.error("M: " + m + " PT: " + pt
            + ": possible RTCP packet?");
    return false;
}

/* any csrcs? */
if (cc > 16) {
    logger.error("CSRC count invalid");
    return false;
}

int hlen = 12 + 4 * cc;
if (len < hlen) {
    logger.error("packet too short");
    return false;
}

if (cc > 0) {
    csrc = new int[cc];
    for (int i = 0; i < cc; i++) {
        csrc[i] = buf.getInt();
    }
}

/* header extensions? */
if (x == 1) {

    if (len < hlen + 4) {
        logger.error("packet too short");
        return false;
    }
```

```
            ext_type = buf.getShort();
            ext_len = buf.getShort();

            if (len < hlen + 4 + ext_len * 4) {
                logger.error("packet too short");
                return false;
            }

            if (ext_len > 0) {
                ext_data = new int[ext_len];
                for (int i = 0; i < ext_len; i++) {
                    ext_data[i] = buf.getInt();
                }
            }
        }

        return true;
    }

    /**
     * Mandatory override. Does nothing - the packet should already be parsed by
     * the constructor.
     */
    @Override
    public void parse() {
        /*
         * do nothing. the packet should already be parsed.
         */
        return;
    }

    /**
     * @return the src_addr
     */
    public String getSourceAddress() {

        return udpPacket.getSourceAddress();
    }

    /**
     * @return the src_port
     */
    public int getSourcePort() {

        return udpPacket.getSourcePort();
    }

    /**
     * @return the dst_addr
     */
    public String getDestinationAddress() {

        return udpPacket.getDestinationAddress();
    }

    /**
     * @return the dst_port
     */
```

```
public int getDestinationPort() {

    return udpPacket.getDestinationPort();
}

/**
 * @return the seq
 */
public int getSeq() {

    return seq;
}

/**
 * @return the ssrc
 */
public long getSsrc() {

    return ssrc;
}

/**
 * @return a string representation of the packet.
 */
@Override
public String toString() {

    StringBuffer buffer = new StringBuffer();

    buffer.append("[");
    buffer.append("McastRtpPacket");
    buffer.append(": ");
    buffer.append("v=" + version);
    buffer.append(":p=" + p);
    buffer.append(":x=" + x);
    buffer.append(":cc=" + cc);
    buffer.append(":m=" + m);
    buffer.append(":pt=" + pt);
    buffer.append(":seq=" + seq);
    buffer.append(":ts=" + ts);
    buffer.append(":ssrc=" + ssrc);

    for (int i = 0; i < cc; i++) {
        buffer.append(":csrc[" + i + "]=" + csrc[i]);
    }

    if (x == 1) {
        buffer.append(":ext_type=" + ext_type);
        buffer.append(":ext_len=" + ext_len);
        for (int i = 0; i < ext_len; i++) {
            buffer.append(":ext_data[" + i + "]=" + ext_data[i]);
        }
    }

    buffer.append("]");
    return buffer.toString();
}

/**
```

```
     * @return the arrivalTime
     */
    public long getArrivalTime() {
        return arrivalTime;
    }

    /**
     * @param arrivalTime
     *            the arrivalTime to set
     */
    public void setArrivalTime(long arrivalTime) {
        this.arrivalTime = arrivalTime;
    }

}
```

B.4 McastRtpSession.java

```
/**
 * COMS E6181: Advanced Internet Services (Fall 2011)
 * pgs2111 at columbia.edu
 */
package edu.columbia.cs.dyswis.module_mcast_rtp;

import org.apache.log4j.Logger;

import edu.columbia.cs.dyswis.common.baseclass.DysSession;

/**
 * An McastRtpSession represents a unique multicast stream.
 * <p>
 * The stream is uniquely identified by the 4-tuple of source IP address, source
 * port, destination multicast group address, destination port.
 * <p>
 * McastRtpSessions are created by McastRtpDetect when an McastRtpPacket is
 * received that does not match an existing session.
 *
 * @author phil
 */
public class McastRtpSession extends DysSession {

    /** for log messages */
    private static Logger logger = Logger.getLogger("McastRtpSession");

    /**
     * The SSRC associated with this session
     */
    private long ssrc;

    /**
     * The last sequence number that was received for this session
     */
    private int lastSeqReceived;

    /**
     * The next sequence number that is expected for this session
     */
    private int nextSeqExpected;
```

```java
/**
 * Time that the session was started (in millis)
 */
private long sessionStartTime;

/**
 * Time that the last packet was received
 */
private long lastPacketTime;

/**
 * Create a new Multicast RTP session with an identifier that matches the
 * received packet.
 *
 * @param mcastRtpPacket
 *            the packet used to build the session identifier
 */
public McastRtpSession(McastRtpPacket mcastRtpPacket) {
    super();
    sessionStartTime = mcastRtpPacket.getArrivalTime();

    ssrc = mcastRtpPacket.getSsrc();
    lastSeqReceived = mcastRtpPacket.getSeq();
    nextSeqExpected = (lastSeqReceived + 1) & 0xffff;
    logger.info("NEW SESSION: "
            + makeSessionIdentifier(mcastRtpPacket));
}

/**
 * Generate a session identifier from a Multicast RTP packet
 *
 * @param mcastRtpPacket
 *            the multicast RTP packet
 * @return a String that can be used as a session identifier
 */
public static String makeSessionIdentifier(
        McastRtpPacket mcastRtpPacket) {

    if (mcastRtpPacket == null) {
        return "INVALID MCAST_RTP_PACKET";
    }

    return makeSessionIdentifier(mcastRtpPacket.getSourceAddress(),
            Integer.toString(mcastRtpPacket.getSourcePort()),
            mcastRtpPacket.getDestinationAddress(),
            Integer.toString(mcastRtpPacket.getDestinationPort()));
}

/**
 * Generate a session identifier from the required identifiers
 *
 * @param srcAddr
 *            source IP address
 * @param srcPort
 *            source port
 * @param dstAddr
 *            destination (multicast) IP address
 * @param dstPort
 *            destination port
```

```java
 * @return a String that can be used as a session identifier
 */
public static String makeSessionIdentifier(String srcAddr,
        String srcPort, String dstAddr, String dstPort) {
    StringBuffer buffer = new StringBuffer();

    buffer.append(srcAddr);
    buffer.append("/");
    buffer.append(srcPort);
    buffer.append("->");
    buffer.append(dstAddr);
    buffer.append("/");
    buffer.append(dstPort);

    return buffer.toString();
}

/**
 * Handles a McastRtpPacket that belongs to this session. Verifies that the
 * sequence number is what was expected. Logs a fault if it doesn't match.
 * <p>
 * This is called directly by McastRtpDetect when it gets an appropriate
 * packet matching this session.
 *
 * @param mcastRtpPacket
 *            - the packet that arrived
 * @see McastRtpDetect#packetArrived(net.sourceforge.jpcap.net.Packet)
 */
public void mcastRtpPacketArrived(McastRtpPacket mcastRtpPacket) {

    /* warn on SSRC changes, but don't log a fault */
    if (mcastRtpPacket.getSsrc() != ssrc) {
        logger.warn("SSRC changed: was " + ssrc + ": now "
                + mcastRtpPacket.getSsrc());
        logger.debug(mcastRtpPacket.toString());
    }

    /* raise an McastRtpFault if there is a sequence number gap */
    if (mcastRtpPacket.getSeq() != nextSeqExpected) {
        logger.warn("Sequence number error: expected "
                + nextSeqExpected + ": got "
                + mcastRtpPacket.getSeq());
        logger.debug(mcastRtpPacket.toString());

        McastRtpFault fault = new McastRtpFault(mcastRtpPacket);
        fault.setErrCode("1");
        fault.setErrMsg("Sequence number error");
        fault.setRuleFileName("MCAST_RTP_SEQ");
        fault.setExpected(nextSeqExpected);
        fault.setReceived(mcastRtpPacket.getSeq());
        fault.setGap(lastPacketTime,
                mcastRtpPacket.getArrivalTime());
        addFault(fault);
    }
    ssrc = mcastRtpPacket.getSsrc();
    lastSeqReceived = mcastRtpPacket.getSeq();
    nextSeqExpected = (lastSeqReceived + 1) & 0xffff;
    lastPacketTime = mcastRtpPacket.getArrivalTime();
}
```

```
    /**
     * Add a fault to the GUI and also to the multicast RTP fault history.
     *
     * @param fault
     *            the fault to add
     */
    private void addFault(McastRtpFault fault) {

        /*
         * Add the fault to the diagnosis module
         */
        Activator.getDyswis().addFault(fault);

        /*
         * in a perfect world, this would be a call-back from the diagnosis
         * module
         */
        Activator.mcastRtpFaultListener.faultAdded(fault);
    }

    /**
     * @return the sessionStartTime
     */
    public long getSessionStartTime() {
        return sessionStartTime;
    }

    /**
     * @param sessionStartTime
     *            the sessionStartTime to set
     */
    public void setSessionStartTime(long sessionStartTime) {
        this.sessionStartTime = sessionStartTime;
    }

    /**
     * @return the lastPacketTime
     */
    public long getLastPacketTime() {
        return lastPacketTime;
    }

    /**
     * @param lastPacketTime
     *            the lastPacketTime to set
     */
    public void setLastPacketTime(long lastPacketTime) {
        this.lastPacketTime = lastPacketTime;
    }

}
```

B.5 McastRtpFault.java

```
/**
 * COMS E6181: Advanced Internet Services (Fall 2011)
 * pgs2111 at columbia.edu
 */
```

```java
package edu.columbia.cs.dyswis.module_mcast_rtp;

import java.text.SimpleDateFormat;
import java.util.HashMap;

import org.apache.log4j.Logger;

import edu.columbia.cs.dyswis.common.baseclass.DysFault;
import edu.columbia.cs.dyswis.common.baseclass.DysPacket;
import edu.columbia.cs.dyswis.core.diagnosis.faultoperator.DysFaultOperator;

/**
 * This class represents a fault while receiving a multicast RTP stream.
 * <p>
 * Currently, the only type of fault defined is a sequence number gap.
 *
 * @author phil
 */
public class McastRtpFault extends DysFault {

    /** serialVersionUID required since DysFault is serializable */
    private static final long serialVersionUID = -5583071235806013543L;

    /** RTP sequence number of the next packet we are expecting to receive */
    private int expected;

    /** RTP sequence number of the packet we received */
    private int received;

    /** the time the of the last packet we received before the gap */
    private long gapStart;

    /** the time of the first packet we received after the gap */
    private long gapEnd;

    /** the McastRtpSession identifier */
    private String sessionId;

    /** for log messages */
    private static Logger logger = Logger.getLogger("McastRtpFault");

    /**
     * Generates a fault from a multicast rtp packet.
     * <p>
     * Only populates the information about the session, but does not specify
     * any details about the fault itself.
     *
     * @param packet
     *            the packet that triggered the fault
     */
    public McastRtpFault(DysPacket packet) {
        super(packet);

        assert (packet instanceof McastRtpPacket);

        protocolName = "MCAST_RTP";
        hostAddr = ((McastRtpPacket) packet).getSourceAddress();
        hostPort = ((McastRtpPacket) packet).getSourcePort();
        myAddr = ((McastRtpPacket) packet).getDestinationAddress();
```

```
        myPort = ((McastRtpPacket) packet).getDestinationPort();
        sessionId =
                McastRtpSession
                        .makeSessionIdentifier((McastRtpPacket) packet);
        type = DysFaultOperator.MCAST_RTP;
    }

    /**
     * Store information about the fault in a hash map. The hashmap is then sent
     * to the remote probe receiver.
     *
     * @see edu.columbia.cs.dyswis.common.baseclass.DysFault#makeHashMap()
     */
    @Override
    public HashMap<?, ?> makeHashMap() {
        logger.debug("creating hash map");
        HashMap<String, String> hm = new HashMap<String, String>();
        hm.put("FAULT", this.toString());
        hm.put("SOURCE", this.getHostAddr());
        hm.put("SOURCE_PORT", Integer.toString(this.getHostPort()));
        hm.put("DEST", this.getMyAddr());
        hm.put("DEST_PORT", Integer.toString(this.getMyPort()));
        hm.put("SESSIONID", sessionId);
        hm.put("GAP_START", Long.toString(gapStart));
        hm.put("GAP_END", Long.toString(gapEnd));
        return hm;
    }

    /**
     * Sets the sequence number that was expected
     * <p>
     * Called by McastRtpDetect when creating the fault
     *
     * @param i
     *            the expected sequence number.
     */
    public void setExpected(int i) {
        expected = i;
    }

    /**
     * Sets the sequence number that was received
     * <p>
     * Called by McastRtpDetect when creating the fault
     *
     * @param i
     *            the received sequence number.
     */
    public void setReceived(int i) {
        received = i;
    }

    /**
     * Retrieve the expected sequence number from a fault
     *
     * @return the expected sequence number
     */
    public int getExpected() {
        return expected;
```

```
}

/**
 * Retrieve the received sequence number from a fault
 *
 * @return the received sequence number
 */
public int getReceived() {
    return received;
}

/**
 * Display information about the fault
 *
 * @return a String representation of the fault.
 */
@Override
public String toString() {
    McastRtpPacket mcastRtpPacket = (McastRtpPacket) this.packet;

    String result =
            McastRtpSession.makeSessionIdentifier(mcastRtpPacket);
    result += " Expected seq: " + getExpected();
    result += " Received seq: " + getReceived();

    return result;
}

/**
 * Set the start and end time of the gap
 * <p>
 * Time values are in the format of System.currentTimeMillis()
 *
 * @param start
 *            the start time
 * @param end
 *            the end time
 */
public void setGap(long start, long end) {

    SimpleDateFormat fmt =
            new SimpleDateFormat("yyyy-MM-dd HH:mm:ss");

    logger.debug("gap start: " + fmt.format(start));
    logger.debug("gap end: " + fmt.format(end));

    gapStart = start;
    gapEnd = end;
}

/**
 * @return the sessionId
 */
public String getSessionId() {
    return sessionId;
}

/**
 * @return the gapStart
```

```
     */
    public long getGapStart() {
        return gapStart;
    }

    /**
     * @return the gapEnd
     */
    public long getGapEnd() {
        return gapEnd;
    }

}
```

B.6 McastRtpFaultListener.java

```
/**
 * COMS E6181: Advanced Internet Services (Fall 2011)
 * pgs2111 at columbia.edu
 */
package edu.columbia.cs.dyswis.module_mcast_rtp;

import java.util.Map;
import java.util.concurrent.ConcurrentHashMap;

import org.apache.log4j.Logger;

import edu.columbia.cs.dyswis.common.baseclass.DysFault;

/**
 * Maintains a searchable history of McastRtpFaults.
 * <p>
 * I'd really like this to implement the FaultListener interface, so that faults
 * that are added via addFault are automatically added here too. But, that
 * interface isn't exported, and I don't know if all the hooks would work.
 *
 * @see edu.columbia.cs.dyswis.core.diagnosis.FaultListener
 * @author phil
 */
public class McastRtpFaultListener /* implements FaultListener */{

    private static Logger logger = Logger
            .getLogger("McastRtpFaultListener");

    private Map<String, String> faultMap;

    /**
     * Default constructor
     */
    public McastRtpFaultListener() {

        logger.debug("McastRtpFaultListener started");
        faultMap = new ConcurrentHashMap<String, String>();
    }

    /**
     * Right now this is called by faultAdded in McastRtpSession
     *
     * @see edu.columbia.cs.dyswis.module_mcast_rtp.McastRtpSession#addFault
```

```
     */
    public void faultAdded(DysFault fault) {

        if (!(fault instanceof McastRtpFault)) {
            return;
        }

        logger.debug("fault added: " + fault.toString() + ": ts :"
                + fault.getTimestamp());
        faultMap.put(fault.toString(), fault.getTimestamp());
    }

    public boolean findFault(String faultString) {

        logger.debug("finding fault: " + faultString);

        if (faultMap.containsKey(faultString)) {
            logger.debug("found!");
            return true;
        } else {
            logger.debug("not found!");
            return false;
        }
    }

    public boolean findFault(McastRtpFault fault) {
        if (fault == null) {
            logger.debug("fault is null");
            return false;
        }
        String faultString = fault.toString();
        return findFault(faultString);
    }

}
```

B.7 McastRtpProbe.java

```
/**
 * COMS E6181: Advanced Internet Services (Fall 2011)
 * pgs2111 at columbia.edu
 */
package edu.columbia.cs.dyswis.module_mcast_rtp;

import java.util.HashMap;

import org.apache.log4j.Logger;

import edu.columbia.cs.dyswis.common.baseclass.DysProbe;
import edu.columbia.cs.dyswis.service.DyswisService;

/**
 * Local probe that simply displays information about a McastRtpFault.
 * <p>
 * Specifically, it displays the Source IP and port, the Destination multicast
 * group IP and port, the expected RTP sequence number, and the received RTP
 * sequence number.
 *
 * @author phil
```

```java
 */
public class McastRtpProbe implements DysProbe {

    /** for log messages */
    private static Logger logger = Logger.getLogger("McastRtpProbe");

    /**
     * Display information about an McastRtpFault
     * @param faultMap the McastRtpFault
     * @return true if the input was a valid McastRtpFault, false otherwise
     */
    @Override
    public Object probe(Object faultMap) {

        logger.debug("probe: " + faultMap);

        DyswisService gui = Activator.getDyswis();
        gui.resetDiagnoseMessage();
        gui.addDiagnosisMessageTitle("Multicast RTP");

        McastRtpFault fault;

        if (faultMap instanceof HashMap<?, ?>) {
            /* don't do anything for a remote probe */
            return false;
        } else if (faultMap instanceof McastRtpFault) {
            /* normal case - a local probe */
            fault = (McastRtpFault) faultMap;
        } else {
            /* not sure why we would get this */
            gui.addDiagnosisMessageContent("Unknown fault type", 0);
            return false;
        }

        /*
         * Display the information in the GUI
         */
        gui.addDiagnosisMessageContent("RTP Sequence Number Error", 0);
        gui.addDiagnosisMessageDescribe(
                "Source: " + fault.getHostAddr() + "/"
                        + fault.getHostPort(), 0);
        gui.addDiagnosisMessageDescribe(
                "Destination: " + fault.getMyAddr() + "/"
                        + fault.getMyPort(), 0);
        gui.addDiagnosisMessageDescribe(
                "Expected RTP Sequence Number: " + fault.getExpected(),
                0);
        gui.addDiagnosisMessageDescribe(
                "Received RTP Sequence Number: " + fault.getReceived(),
                1);

        return true;
    }

    /**
     * Required override. Not used.
     */
    @Override
    public void displayResult(Object request, Object result) {
```

```
        logger.debug("display result: " + request + ": " + result);

    }

}
```

B.8 McastRtpReversePathDiscovery.java

```
/**
 * COMS E6181: Advanced Internet Services (Fall 2011)
 * pgs2111 at columbia.edu
 */
package edu.columbia.cs.dyswis.module_mcast_rtp;

import java.io.BufferedReader;
import java.io.IOException;
import java.io.InputStreamReader;
import java.util.HashMap;
import java.util.Map;
import java.util.concurrent.ConcurrentHashMap;
import java.util.regex.Matcher;
import java.util.regex.Pattern;

import org.apache.log4j.Logger;

import edu.columbia.cs.dyswis.common.baseclass.DysProbe;
import edu.columbia.cs.dyswis.common.util.Functions;

/**
 * Find the path towards the multicast source IP. (by running a traceroute)
 *
 * @author phil
 */
public class McastRtpReversePathDiscovery implements DysProbe {

    /** for log messages */
    private static Logger logger = Logger
            .getLogger("McastRtpReversePathDiscovery");

    /**
     * traceroute command for Windows
     * <p>
     * -d no names resolution<br>
     * -w 1000 1 second timeout<br>
     * -h 30 maximum of 30 hops
     */
    private static final String tracerouteCmdWindows =
            "tracert -d -w 1000 -h 30";
    /**
     * traceroute command for OSX/Linux/other platforms
     * <p>
     * -n no names resolution<br>
     * -w 1 1 second timeout<br>
     * -m 30 maximum of 30 hops
     */
    private static final String tracerouteCmdGeneric =
            "traceroute -n -w 1 -m 30";
```

```java
/**
 * Maps IP address (of multicast source) => traceroute hops towards the
 * source (hops are separated by ' ')
 */
private static Map<String, String> sourceMap =
        new ConcurrentHashMap<String, String>();

/**
 * Perform a traceroute towards the multicast source. Store the nodes in a
 * space-separated string.
 *
 * @param faultMap
 *            either an McastRtpFault, or a HashMap containing a mapping
 *            from key "SOURCE" to a String with the multicast source IP
 */
@Override
public Object probe(Object faultMap) {

    logger.debug("probe: " + faultMap);

    /*
     * if called directly from the GUI with an actual McastRtpFault, probe
     * towards the source and display messages on the GUI
     */
    if (faultMap instanceof McastRtpFault) {
        McastRtpFault fault = (McastRtpFault) faultMap;
        return probe(fault.getHostAddr(), true);
    }
    /*
     * if called by a remote probe, probe towards the source and don't
     * display messages
     */
    else if (faultMap instanceof HashMap<?, ?>) {
        String mcastRtpSource =
                (String) ((HashMap<?, ?>) faultMap).get("SOURCE");
        return probe(mcastRtpSource, false);
    } else {
        logger.debug("Unknown object: " + faultMap);
        return null;
    }
}

/**
 * Runs the traceroute, displaying messages if desired.
 * <p>
 * Cache the results, since traceroute can be slow if you have lots of
 * intermediate hops that time out.
 *
 * @param mcastRtpSource
 * @param displayGuiMessages
 * @return a String containing space-separated hops
 */
public static String probe(String mcastRtpSource,
        boolean displayGuiMessages) {
    logger.debug("RPF check towards: " + mcastRtpSource);
    if (displayGuiMessages) {
        Activator.getDyswis().addDiagnosisMessageContent(
                "Traceroute to " + mcastRtpSource, 0);
    }
```

```java
/* check if we have seen this source before */
if (sourceMap.containsKey(mcastRtpSource)) {

    logger.debug("Already seen source: " + mcastRtpSource);
    return getCachedResult(mcastRtpSource, displayGuiMessages);
}

/* run the traceroute and store the result */
StringBuilder hops = new StringBuilder();

Runtime rt = Runtime.getRuntime();

String tracerouteCmd;
if (Functions.getMyOSType() == Functions.OS_WINDOWS) {
    tracerouteCmd = tracerouteCmdWindows;
} else {
    tracerouteCmd = tracerouteCmdGeneric;
}

try {
    Process process =
            rt.exec(tracerouteCmd + " " + mcastRtpSource);
    BufferedReader bf =
            new BufferedReader(new InputStreamReader(
                    process.getInputStream()));
    String line = null;
    String regexp =
            "(\\d+)\\b.*?\\b(\\d+\\.\\d+\\.\\d+\\.\\d+)\\b";

    Pattern p = Pattern.compile(regexp);
    logger.debug("pattern: " + p.toString());

    while ((line = bf.readLine()) != null) {
        logger.debug(line);
        Matcher m = p.matcher(line);

        if (m.find()) {
            int hop = Integer.parseInt(m.group(1));
            String ip = m.group(2);
            logger.debug("hop: " + hop + " ip: " + ip);
            if (displayGuiMessages) {
                Activator
                        .getDyswis()
                        .addDiagnosisMessageDescribe(
                                "Hop: " + hop + " IP: " + ip, 0);
            }
            hops.append(ip + " ");
        }
    }
    logger.debug("traceroute complete");
} catch (IOException e) {
    System.out.println(e.getStackTrace());
}

String result = hops.toString();
sourceMap.put(mcastRtpSource, result);
logger.debug("returning: result: " + result);
return result;
```

```java
    }

    /**
     * Retrieves the cached traceroute results. May be called locally or
     * remotely.
     *
     * @param mcastRtpSource
     *            the source IP address
     * @param displayGuiMessages
     *            if true, update the GUI
     * @return the list of hops separates by spaces
     */
    private static String getCachedResult(String mcastRtpSource,
            boolean displayGuiMessages) {

        /* retrieve the cached source */
        String result = sourceMap.get(mcastRtpSource);

        /* update the GUI if necessary */
        if (displayGuiMessages) {
            String[] hopArray = result.split(" ");
            for (int i = 0; i < hopArray.length; i++) {
                Activator.getDyswis().addDiagnosisMessageDescribe(
                        "Cached Hop: IP: " + hopArray[i], 0);
            }
        }

        return result;
    }

    /**
     * Display a message that the traceroute is complete.
     * <p>
     * Doesn't need to do too much, since the probe itself updates the GUI as it
     * goes along.
     *
     * @param request
     *            ignored
     * @param result
     *            ignored
     */
    @Override
    public void displayResult(Object request, Object result) {
        Activator.getDyswis().addDiagnosisMessageDescribe(
                "Traceroute complete", 0);
        Activator.getDyswis().addDiagnosisMessageDescribe(
                result.toString(), 0);
    }

}
```

B.9 McastRtpReversePathDiscoveryThread.java

```java
/**
 * COMS E6181: Advanced Internet Services (Fall 2011)
 * pgs2111 at columbia.edu
 */
package edu.columbia.cs.dyswis.module_mcast_rtp;
```

```
import org.apache.log4j.Logger;

/**
 * Separate thread to run the traceroute probes.
 *
 * @author phil
 */
public class McastRtpReversePathDiscoveryThread extends Thread {

    /** for log messages */
    private static Logger logger = Logger
            .getLogger("McastRtpReversePathDiscoveryThread");

    /** the multicast source address */
    private String mcastRtpSource;

    /**
     * Create a new thread to run a traceroutes towards the given address
     *
     * @param s
     *            the source IP (to which the traceroute will be run)
     */
    public McastRtpReversePathDiscoveryThread(String s) {
        logger.debug("New Thread. Source: " + s);
        mcastRtpSource = s;
    }

    /**
     * Launch the traceroute probe
     */
    @Override
    public void run() {
        logger.debug("running probe towards: " + mcastRtpSource);

        /* don't display GUI messages */
        McastRtpReversePathDiscovery.probe(mcastRtpSource, false);
    }

}
```

B.10 McastRtpFaultHistoryProbe.java

```
/**
 * COMS E6181: Advanced Internet Services (Fall 2011)
 * pgs2111 at columbia.edu
 */
package edu.columbia.cs.dyswis.module_mcast_rtp;

import java.util.HashMap;

import org.apache.log4j.Logger;

import edu.columbia.cs.dyswis.common.baseclass.DysProbe;
import edu.columbia.cs.dyswis.service.DyswisService;

/**
 * Query the Fault History to see if a loss event was detected.
 * <p>
 * At one point, this was intended to be called directly as a remote probe, and
```

```java
 * it can still be used in this way. However, now it is called locally (by the
 * remote receiver) as part of the McastRtpRemoteProbe.
 *
 * @author phil
 */
public class McastRtpFaultHistoryProbe implements DysProbe {

    /** for log messages */
    private static Logger logger = Logger
            .getLogger("McastRtpFaultHistoryProbe");

    /**
     * Check the history for a given McastRtpFault
     *
     * @param faultMap
     *            either an McastRtpFault, or a HashMap containing an entry with
     *            a key of "FAULT" that maps to a fault string
     * @return true if the fault was found, or false otherwise
     */
    @Override
    public Object probe(Object faultMap) {

        logger.debug("probe: " + faultMap);

        String faultString;

        if (faultMap instanceof HashMap<?, ?>) {
            logger.debug("hashmap detected");
            faultString =
                    (String) ((HashMap<?, ?>) faultMap).get("FAULT");
        } else if (faultMap instanceof McastRtpFault) {
            faultString = faultMap.toString();
        } else {
            logger.debug("Unknown fault type: " + faultMap.toString());
            return null;
        }

        logger.debug("checking fault history for: " + faultString);
        return Activator.mcastRtpFaultListener.findFault(faultString);
    }

    /**
     * Add a message to the GUI with the result of the probe
     *
     * @param request
     *            the initial request that was sent to the remote
     * @param result
     *            the result of the probe
     */
    @Override
    public void displayResult(Object request, Object result) {

        DyswisService gui = Activator.getDyswis();
        gui.addDiagnosisMessageTitle("Multicast RTP Fault History Probe");
        if ((Boolean) result) {
            gui.addDiagnosisMessageContent("Remote side found fault", 0);
        } else {
            gui.addDiagnosisMessageContent(
                    "Remote side didn't find fault", 0);
```

```
        }
    }

}
```

B.11 McastRtpSessionHistoryProbe.java

```
/**
 * COMS E6181: Advanced Internet Services (Fall 2011)
 * pgs2111 at columbia.edu
 */
package edu.columbia.cs.dyswis.module_mcast_rtp;

import java.util.HashMap;

import org.apache.log4j.Logger;

import edu.columbia.cs.dyswis.common.baseclass.DysProbe;
import edu.columbia.cs.dyswis.service.DyswisService;

/**
 * Probe to query the multicast RTP session history for a matching session.
 * <p>
 * A fault occurs during specific time range. It is useful to know whether or
 * not a remote node was joined to the same stream and the same time.
 *
 * @author phil
 */
public class McastRtpSessionHistoryProbe implements DysProbe {

    /** for logging */
    private static Logger logger = Logger
            .getLogger("McastRtpSessionHistoryProbe");

    /**
     * Check the session history for the session
     * <p>
     *
     * @param faultMap
     *            either an McastRtpFault, or a HashMap containing the
     *            SESSION_ID, GAP_START, and GAP_END
     * @return true if a matching session was found
     */
    @Override
    public Object probe(Object faultMap) {

        logger.debug("probe: " + faultMap);

        String sessionId;
        long gapStart;
        long gapEnd;

        if (faultMap instanceof HashMap<?, ?>) {
            HashMap<?, ?> map = (HashMap<?, ?>) faultMap;
            sessionId = (String) map.get("SESSIONID");
            gapStart = Long.parseLong((String) map.get("GAP_START"));
            gapEnd = Long.parseLong((String) map.get("GAP_END"));
        } else if (faultMap instanceof McastRtpFault) {
            McastRtpFault fault = (McastRtpFault) faultMap;
```

```
            sessionId = fault.getSessionId();
            gapStart = fault.getGapStart();
            gapEnd = fault.getGapEnd();
        } else {
            logger.debug("Unknown fault type: " + faultMap.toString());
            return null;
        }

        McastRtpSession session =
                Activator.mcastRtpDetect.getSession(sessionId);

        /*
         * If we don't have a matching session at all, return false
         */
        if (session == null) {
            logger.debug("No matching session found");
            return false;
        }
        /*
         * If the session started after the gap occurred, the information isn't
         * useful
         */
        if (session.getSessionStartTime() > gapStart) {
            logger.debug("Start time was after gap");
            return false;
        }

        /*
         * If we haven't received a packet after the gap ended, the information
         * isn't useful
         */
        if (session.getLastPacketTime() < gapEnd) {
            logger.debug("Session may have ended prior to gap");
            return false;
        }

        /*
         * We have a good match - the same session, encompassing the entire gap
         */
        logger.debug("Good session match!");
        return true;
    }

    /**
     * Display a result, if called directly by a LocalProbe or RemoteProbe.
     * <p>
     * Not really expecting this to be called .. this is handled through
     * MultiProbe now.
     */
    @Override
    public void displayResult(Object request, Object result) {

        DyswisService gui = Activator.getDyswis();
        gui.addDiagnosisMessageTitle("Multicast RTP Session History Probe");
        if ((Boolean) result) {
            gui.addDiagnosisMessageContent("Remote side found session",
                    0);
        } else {
            gui.addDiagnosisMessageContent(
```

```
                    "Remote side didn't find session", 0);
        }
    }

}
```

B.12 McastRtpRemoteProbe.java

```
/**
 * COMS E6181: Advanced Internet Services (Fall 2011)
 * pgs2111 at columbia.edu
 */
package edu.columbia.cs.dyswis.module_mcast_rtp;

import org.apache.log4j.Logger;

import edu.columbia.cs.dyswis.common.baseclass.DysProbe;
import edu.columbia.cs.dyswis.service.DyswisService;

/**
 * Monolithic remote probe that consolidates all the data that may relate to a
 * fault.
 * <p>
 * Since there is no concept of a "conversation" between nodes, we cannot
 * guarantee that multiple remote probes executed in series will talk to the
 * same remote. Therefore, it is best to get all the data via a single remote
 * query.
 * <p>
 * This probe is intended to be called remotely, either by RemoteProbe, or most
 * likely as by MultiProbe.
 *
 * @author phil
 */
public class McastRtpRemoteProbe implements DysProbe {

    /** for logging */
    private static Logger logger = Logger
            .getLogger("McastRtpRemoteProbe");

    /**
     * Given a hash map relating to an McastRtpFault, returns a string
     * describing what the remote probe experienced.
     * <p>
     * This is a little kludgy right now, but the String contains the following:
     * <br>
     * <SAME_SESSION> <SAME_FAULT> <PATH_TO_SOURCE><br>
     * where<br>
     * SAME_SESSION => "true" or "false"<br>
     * SAME_FAULT => "true" or "false"<br>
     * PATH_TO_SOURCE => space separated traceroute values<br>
     *
     * @return a String describing what the remote probe experienced.
     */
    @Override
    public Object probe(Object faultMap) {
        logger.debug("probe called!");
        logger.debug(faultMap.toString());

        DysProbe probe;
```

```
        probe = new McastRtpSessionHistoryProbe();
        boolean sameSession = (Boolean) probe.probe(faultMap);

        probe = new McastRtpFaultHistoryProbe();
        boolean sameFault = (Boolean) probe.probe(faultMap);

        probe = new McastRtpReversePathDiscovery();
        String result = (String) probe.probe(faultMap);

        return new String(sameSession + " " + sameFault + " " + result);
    }

    /**
     * Called by MultiProbe (or RemoteProbe, theoretically) to display the
     * result.
     */
    @Override
    public void displayResult(Object request, Object result) {

        if (result == null || !(result instanceof String)) {
            return;
        }

        /* parse the result */
        String[] fields = ((String) result).split(" ", 3);
        if (fields.length < 3)
            return;

        /* display the GUI messages */
        DyswisService gui = Activator.getDyswis();
        if (gui != null) {
            gui.addDiagnosisMessageDescribe("Same session: "
                    + fields[0], 0);
            gui.addDiagnosisMessageDescribe("Same fault: " + fields[1],
                    0);
            gui.addDiagnosisMessageDescribe("Hops to source: "
                    + fields[2], 0);
        }
    }

}
```

C Other contributions to DYSWIS

In addition to the Multicast RTP module, other contributions to DYSWIS were made during the course of the project.

C.1 Mac OS X Support

The DYSWIS framework depends on the jpcap.sourceforge.net package to capture packets to pass to the detection modules. The project was missing a valid libjpcap.jnilib JNI library file for current versions of Mac OS X. After compiling a new version on OS X Lion, testing showed that the library was working, but the packet capture engine was unable to look up the device list properly. Adding a few lines of code to DysPacketCapture.java to use findDevice() instead of lookupDevices() allowed the Mac to capture packets successfully, but only using the default interface. On the iMac and MacBook, this ended up being en0, which was fine, but on the MacBook Air this defaulted to the wireless interface.

C.2 MultiProbe

During the initial development and testing, I was only using two nodes, and using a sequence of RemoteProbes to check for the same session, same fault, and traceroute results. This was fine with only two nodes, but it later became apparent that multiple queries sent in sequence would not necessarily target the same remote nodes.

The MultiProbe code was intended to fire off the same probe to multiple remote nodes. Unfortunately, the code was not fully fleshed out. I added some improved thread management, as well as modifications to how the probe gets its set of remote nodes.