Midyear Report for Implementing the Interoperable IETF/IDWG/IDXP Protocol with Proxy/Tunnel Capability

Team Members:

Nick Hertl – (Project Manager)
Will Berriel
Richard Fujiyama
Chip Bradford

Project Advisor: Professor Michael Erlinger
Project Liaisons: Joseph Betser, Ph.D
Rayford Sims

December 2002

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

ABSTRACT

The archival and correlation of network intrusion detection data is becoming more and more important for the timely recognition of intrusion attempts and prompt response to such threats. Often, however, the central repository of intrusion data is separated from a network administrator who wishes to examine this data. Currently, either transient, but insecure, holes are opened in the firewall, or permanent, and secure, channels are established in order to bypass the firewall. Clearly a transient and secure tunnel would be the best solution. Our project's goal is to implement the Blocks Extensible Exchange Protocol (BEEP) tuning profile for the Tunnel protocol. This promises to provide secure tunneling capabilities through firewalls for intrusion detection analysis as well as general use.

1. BACKGROUND INFORMATION

1.1. TCP

Transmission Control Protocol (TCP) (Information Sciences Institute) is the main transport protocol that computers use to move information across the Internet. It provides a reliable full duplex stream of data from one host to another, potentially using intermediate hosts (routers) to reach the eventual goal. Many application level protocols already use TCP, such as HTTP (web pages) or SMTP (email). To communicate, such protocols use a standard interface or port. These ports are generally not blocked by firewalls because they are so commonly used and necessary for regular network operation. TCP provides the transport for the BEEP implementations that we will use to write our profiles, but BEEP could also be mapped to other transport layer protocols.

1.2. FIREWALL

Firewalls are computers that protect internal networks from potential hackers, as well as prevent private data from leaking out to the Internet. Every firewall has a list that specifies which ports or protocols are blocked or filtered. If you want to connect to a computer inside a firewall from outside a firewall on a port that the firewall does not allow, you are not allowed to connect. Tunnel proposes a solution to this problem for BEEP.

1.3. BEEP

BEEP (Rose) is a new general protocol for application development. In the past, application protocol designers have spent a large majority of their time discussing the details of how

packets will look. Once all the fighting has died down, nobody has any energy to actually develop the protocol. BEEP helps with this problem by providing a very general set of capabilities for application protocols together with a comprehensive Application Programming Interface (API) for application protocol development. It allows the user to employ tuning profiles to enable security if needed, or a number of other tuning variables. This application level protocol runs on top of TCP, integrating smoothly into the TCP/IP protocol stack. BEEP allows programmers to take the best parts of application protocols and use them in their own protocols. It follows the axiom of code reuse, allowing developers to write a profile that will perform a function like security, and then simply including that profile whenever they want an application to have security features.

1.4. IDXP

Intrusion Detection Exchange Protocol (IDXP) (Feinstein) is implemented as a BEEP profile. IDXP packets transmit data encoded using Intrusion Detection Message Exchange Format (IDMEF), the focus of a previous clinic. The main purpose of IDXP is to transport intrusion detection alert data. This is currently only possible if no firewalls block the packets. IDXP is the current motivation for the development of the Tunnel profile, since IDXP requires the ability to safely pass through firewalls. Tunnel can also be used by any other BEEP profiles because it is a tuning profile for BEEP. In the future, any application that needs to get through a firewall should be able to easily do so simply by using the Tunnel profile.

1.5. SECURITY

It is important to ensure secure and authenticated transmission of data, especially when it involves traversing a firewall intended to increase security. BEEP provides tools and methods for easily negotiating these properties.

1.6. TUNNEL

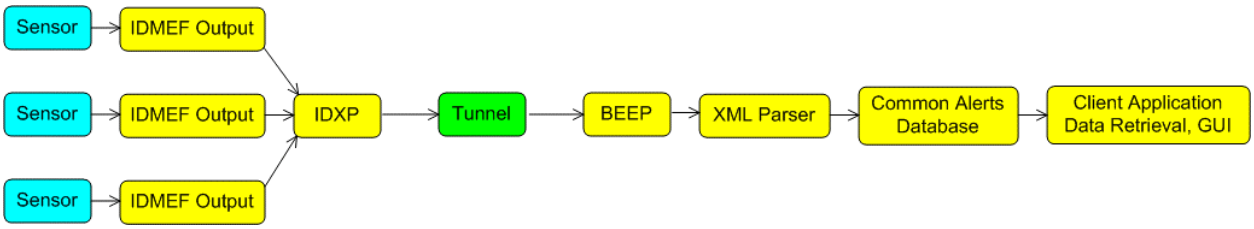


Figure 1: Conceptual Layout

Light Blue (Sensors): 3rd party IDS
Yellow (Light): Assumed to work properly
Green (Tunnel): The focus of our clinic

Figure 1 shows the ideal intrusion detection situation. Intrusion Sensors use their innate technology to generate alerts. All Sensors use the IDMEF syntax to specify their alerts. Alerts are then transported via the IDXP/BEEP protocols to Analyzers, where a database and user interface exist to allow the manager to recognize intrusions and to take appropriate action. Tunnel exists to provide a means to transport the intrusion alerts through any number of firewalls or proxies that might exist between the Sensor and Analyzer.

Tunnel (New) provides a way for BEEP peers to form an application-layer-tunnel. Peers exchange Tunnel packets to establish a connection between them that acts as a point-to-point

connection between the two peers. It is a profile that falls between IDXP and BEEP, tuning the BEEP session to have the necessary characteristics for creating the connection between the peers. In the diagram above, each sensor will want a point-to-point connection with the Extensible Markup Language (XML) Parser, and thus each will form its own tunnel over BEEP to allow the IDXP messages to be exchanged between the sensors and the parser.

Tunnel, like most other BEEP profiles, uses XML for data transfer. Its elements are layered, so that the outermost element specifies the next hop in the connection, or through the use of an empty Tunnel element, that it has reached an endpoint. A proxy refers to any BEEP peers between the start and endpoint in the tunnel; firewalls are a common example. Once the initial connection has been setup between the begin and end points, the proxies between them transparently transmit whatever is sent to them, not checking for BEEP syntax, allowing each peer to encrypt their messages without the proxies being able to view them. In addition once a tunnel is established, peers can send non-BEEP data through it. The proxies can also use whatever security features they wish to manage their immediate connections. A proxy can limit tunnels to certain machines or to only those hosts that are authorized and authenticated through Simple Authentication and Security Layer (SASL).

The Internet Engineering Task Force (IETF) Tunnel Internet Draft provides examples of the expected behavior for some common expected scenarios. The simplest example would be two hosts separated by one peer, with the initiator knowing the path to the listener. As shown in the following figure:

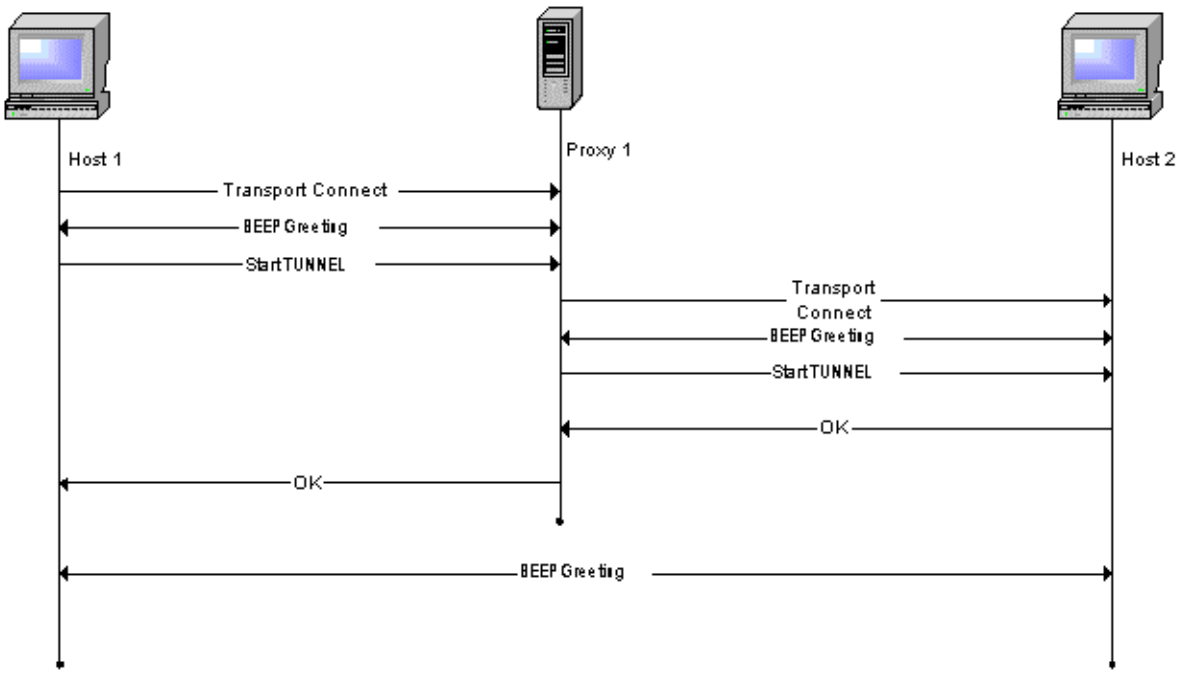


Figure 2: One Hop Proxy

Host 1 is separated from Host 2 by Proxy 1. Host 1 connects to the proxy, starts a BEEP session, and sends the Tunnel start message with two layers: The outermost being the identifier for Host 2 and the innermost being an empty Tunnel element, signifying that the second hop is the endpoint (Host 2).

After the initial connection between Host 1 and Proxy 1, Proxy 1 connects to Host 2 and starts a second BEEP session. Proxy 1 sends an initial Tunnel message, signifying that Host 2 is an endpoint, at which point Host 2 replies with an OK message (assuming it is willing to accept the tunnel). After receiving the OK, Proxy 1 sends it's own OK back to Host 1 and begins transparently forwarding all messages between Host 2 and Host 1.

One major aspect of Tunnel is that it can be directed to a host based on various criteria, the simplest would be fully qualified domain names (e.g. cs.hmc.edu) or IPv4 addresses with a port number, but it can also work if given a requested service (e.g. email). The proxy may know which internal machine hosts such a service, or may reply with an error.

For a more complicated example, we can use two proxies between the two hosts. The following diagram shows such an exchange:

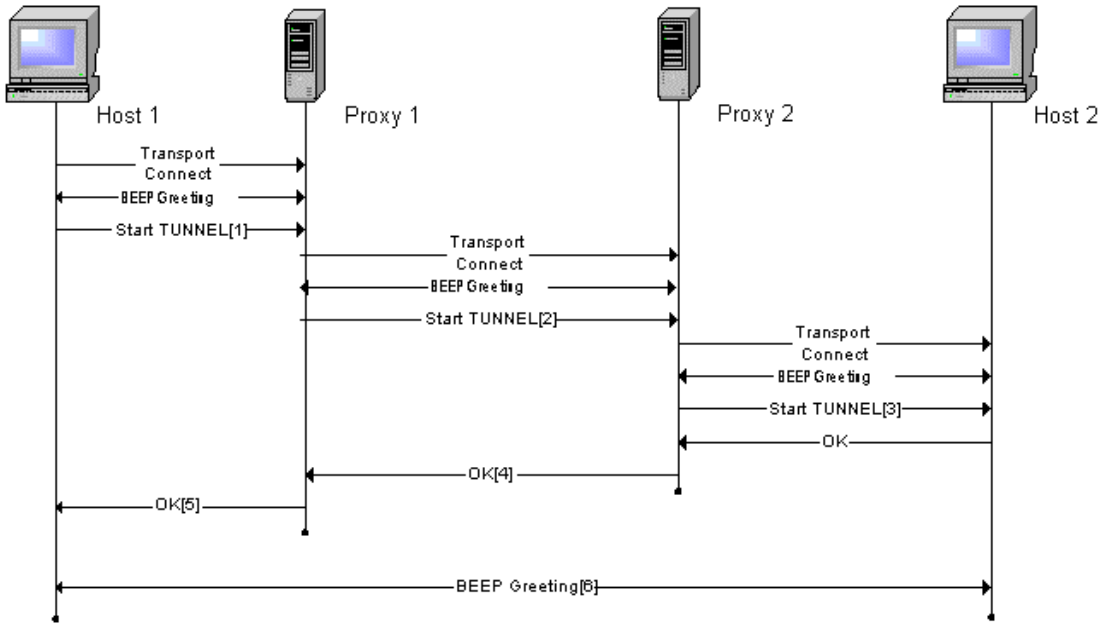


Figure 3: 2 Proxies

If Host 1 knows the full path to Host 2 through the two proxies, then it connects to Proxy 1 the same way as it would in the 1-hop example, and when it sends the initial Tunnel start message (line #1) it will send a message with three layers: The outermost being the identifier of Proxy 2, the middle being the identifier of Host 2, and the innermost being blank. Proxy 1 then strips off the outermost Tunnel element and initiates a connection to Proxy 2. After starting a BEEP session, it sends Proxy 2 the Tunnel message consisting of the inner two Tunnel elements (line #2). Proxy 2

connects to Host 2, initiates the BEEP session, and sends the innermost Tunnel element (line #3). This informs Host 2 that it is the expected endpoint. If Host 2 accepts the connection, it sends an OK to Proxy 2. Proxy 2 sends an OK to Proxy 1 and anything that Proxy 2 receives from Host 2 after the OK (line #4) is sent transparently to Proxy1. Upon receiving the OK from Proxy 2, Proxy 1 sends an OK to Host 1 and immediately after begins transparently forwarding messages from Proxy 2 to Host 1.

If Host 1 does not know the full path to Host 2, but knows that a path exists to it through Proxy 1, it can still establish a tunnel to Host 2. The connection between Host 1 and Proxy 1 begins as above, but when sending the Tunnel start message to Proxy 1, (line #1 in the diagram) Host 1 sends only the identifier for Host 2 (the service running on Host 2, the requested profile on Host 2, or a specific identifying string for Host 2) and no empty Tunnel element. Proxy 1 should be able to determine that Host 2 lies somewhere beyond Proxy 2, and thus connects to Proxy 2 as above, and sends a Tunnel start message with a single element, the identifier for Host 2 (line #2). Upon receiving this message, Proxy 2 will realize that Host 2 is connected directly to it, and thus should connect as above, and as above send a single, empty Tunnel message to Host 2, signifying that it is the endpoint (line #3). The rest of the example carries on the same way as above.

Finally, the message sent after the tunnel is built (line # 6) does not actually have to be a BEEP message, although it is in these examples. The proxies do not read any of the messages passing between them, so any type of data may be sent over the tunnel as needed. In addition, the hosts can work out some form of encryption preventing the proxies or anything else between them from reading the messages, keeping them secure.

The most useful features that Tunnel provides for use in intrusion detection come about when a manager and an analyzer are separated by a proxy. It allows the proxy to authenticate the manager, verifying that the manager is authorized to connect to the analyzer. It can also insulate the analyzer that is behind the proxy from outside attacks, since the analyzer's IP address does not ever need to be revealed to anyone outside the proxy.

2. EVALUATION OF ALTERNATIVES TO TUNNEL

2.1. SSL/TLS

Transport Layer Security (TLS) (Dierks, Rescorla) is the IETF version of Secure Sockets Layer version 3 (SSLv3) and was mainly intended for secure transportation of HTTP traffic. In practice it is also used to secure NNTP (news), IMAP (email), and POP (email) traffic. In the protocol stack, TLS lies between TCP and the application layers and usually provides an API similar to the BSD socket API for secured communication. Applications that wish to make use of SSL will require minimal changes to work properly. In addition to encryption, TLS provides server authentication via certificates and optionally client authentication as well. While using certificates allows a client and server to authenticate without having a pre-shared secret, spoofed certificates make it more prone to man-in-the-middle attacks. However, TLS (in the SSLv3 incarnation) is widely deployed, because client configuration is simple, since TLS does not address any access control issues. Applications using Tunnel are also able to use TLS, since they are both BEEP tuning profiles. A secure application would most likely negotiate the TLS profile before starting the tunnel.

2.2. SASL

Simple Authentication and Security Layer (SASL) (Meyers) is a method for adding authentication and security support to connection-based protocols. It is a framework for providing a protocol with mechanisms for authentication, integrity checking, and encryption. Some SASL mechanisms will negotiate which services to provide for the protocol, while others have a

predetermined set of services. SASL allows the network administrator to configure the proper level of security for that environment; Tunnel benefits from this since BEEP supports SASL as a tuning profile.

2.3. SSH TUNNELING

SSH (Secure Shell) is a pair of applications in the client/server model that are used to replace the *rlogin* and *telnet* programs. All communication between the client and server are encrypted, thus providing data confidentiality in addition to client authentication. The mass adoption of SSH implies that in most firewalls, port 22 is left open for SSH communication, and thus SSH is often used to create secure tunnels through firewalls. However, this approach is not without its drawbacks: SSH is an application level tool and applications that wish to use SSH tunnels must manually create the tunnel through SSH, the client must name an explicit endpoint for connection, SSH only provides client authentication but no host authentication, and all traffic is encrypted. Tunnel improves upon SSH tunneling by being more flexible in authentication and encryption details, providing address anonymity for machines behind the firewall, and for being able to create tunnels without an explicit endpoint (as in the case of services).

2.4. VPN

A VPN (Virtual Private Network) is a secure, permanent, private network built on a publicly accessible infrastructure such as the Internet or telephone network. A VPN is transparent in that the traffic it carries is unaware of any intermediate nodes between the endpoints and the intermediate nodes are unaware they are carrying traffic that is part of the VPN. In addition, a

VPN provides some combination of encryption and strong authentication of remote users and hosts, and thus most VPN implementations are fairly intrusive on the client node. Tunnel is easier than a VPN to administer and deploy since most configuration is done only on the firewall. In addition, Tunnel provides more policy flexibility and is easier to configure than a VPN.

2.5. IPSEC

IPsec (IP Security) is a protocol designed to protect IP (Internet Protocol) from attack. In doing so, it also protects all protocols that run on IP such as TCP and UDP. Thus, applications running on IP benefit from increased security without recompiling. However, by its nature as a protocol-level enhancement, IPsec requires modification of the IP stack, which usually resides in the kernel and is thus very invasive to operating systems. Combined with the fact that IPsec is a peer-to-peer protocol, deployment of IPsec is very difficult unless all machines are running the same operating system or all operating systems have interoperable IPsec implementations. Due to the strict enforcement of IP address consistency, IPsec does not operate correctly behind a Network Address Translator (NAT). In comparison, Tunnel is easier to deploy, more configurable, operates properly with NAT, and handles proxies.

3. WORK COMPLETED

3.1. EVALUATION OF TUNNEL DRAFT

We have read through the Tunnel draft to search for any glaring problems with the current Tunnel specification. Since no one has fully implemented Tunnel before, we did not know if there were any major problems that would prevent implementation. After reading through and discussing problems, we found 3 major issues with the draft. The first was that there was no IPv6 support in the XML Document Type Definition (DTD) that describes Tunnel, nor was there a common way to extend the DTD. Secondly, there is a possibility for misconfigured proxies to enter into a loop, passing a message indefinitely. Thirdly, there is no possibility for a Time To Live (TTL) to be specified that would function like TCP's time to live and set a maximum number of hops, allowing for the detection of cycles. We contacted the author of the Tunnel draft, and since speaking to him, he has added support for IPv6 to the draft.

3.2. CHOOSING BEEP IMPLEMENTATION

For both C and Java there are multiple BEEP implementations in various states of development. On the C side both Beepcore-C and RoadRunner exist. Beepcore-C is developed by the creators of BEEP, but is very far from finished and does not have most of the necessary features. RoadRunner is developed by a company in Scandinavia. It is a fully featured BEEP implementation and is fairly close to actually being complete. On the Java side Beepcore-JAVA and PermaBEEP exist. Beepcore-JAVA is also developed by the creators of BEEP and seems to have

all of the security features necessary for an effective Tunnel profile. PermaBEEP is developed by a private company, but is released as open source. It is much easier to use than Beepcore-JAVA, but does not have all of the necessary security features for Tunnel. We have decided to use RoadRunner for the C version and begin development using PermaBEEP to speed up early development with an eventual port to Beepcore-JAVA.

3.3. SINGLE HOST BEEP COMMUNICATION

Our initial work on implementing the Tunnel profile began with RoadRunner. Since RoadRunner uses *Glib* and *libxml* we had to learn those libraries (as well as the RoadRunner interface) before we could begin working on the actual BEEP profile. Once we had some general knowledge about these APIs, we began work on the simplest subset of the Tunnel profile we could imagine—a loopback connection. This type of communication had several advantages for our first pass at a BEEP profile since it only required a single computer (thus eliminating any networking issues) and it required support for only the empty `<tunnel />` element to be fully functional. Implementing these features was relatively trivial after we figured out how to use the new tools and libraries. In addition to the C implementation, we have an equally functional Java implementation using PermaBEEP.

3.4. PEER-TO-PEER BEEP COMMUNICATION

Once we had a simple Tunnel session working on the loopback device, it was fairly trivial to get this same simple communication working between two separate machines. This required no changes to the profile itself, only modifications to the client program that utilizes the profile. We did

run into a problem with the RoadRunner library while migrating the server to a separate machine. We were unable to get the server to accept connections from remote machines and, at first, thought this was a bug in the RoadRunner libraries since we could find no documentation or samples of dealing with this issue (unfortunately, since all BEEP implementations are still in development bugs are not uncommon.) Later, however, (after patching the RoadRunner source code to fix the issue) we discovered that our problem was really just caused by a lack of documentation for the RoadRunner API. Additionally, we have included this functionality in the Java version.

3.5. ONE HOP BEEP COMMUNICATION

Our next step in completing the Tunnel profile was to add support for proxying. The goal was to allow a connection from client to server with a single machine in between. This jump was significantly more complicated than the previous milestones for several reasons: 1) It required parsing the XML Tunnel message in order to strip the outer element before forwarding. 2) Once the tunnel was established, the proxy must begin forwarding packets transparently, bypassing the normal BEEP framing and other interpretations.

Parsing the XML and forwarding the correctly modified tunnel message turned out to be a simple exercise in learning the *libxml* API. We verified this part of the process was done using *Ethereal* to examine the network packets before work even started on the second phase, message passing. Currently, our implementation can only handle `ip4` (IP Address) or `fqdn` (fully qualified domain name) tunnel attributes with the `port` attribute. The more advanced routing functionality still remains to be done. Since there is no specific way these features must be implemented, we plan to simply add the ability to register callbacks with the profile code in order for different server

applications to handle these address translations in different ways. We also plan to add some simple default handling routines whenever possible.

Once the Tunnel session was established, we were able to utilize the G*lib* event loop and socket interfaces in order to pass data transparently from one socket to the other. This was a bit tricky, however, since we needed to close the RoadRunner functionality on top of the sockets without actually closing the sockets themselves.

We are in the process of completing the multihop profile functionality in Java using PermaBEEP. It should operate just like the C version, but probably less efficient.

4. FUTURE WORK

4.1. MULTIPLE HOP TUNNELING

While we have not yet successfully tested our profile with multiple proxies between the client and server, we see no reason why this should be fundamentally different from a single proxy since the proxy serves as both a client and server as the tunnel is established. We should be able to easily finish this part of the project on schedule.

4.2. FIREWALL PROXY HANDLING

A typical use for the Tunnel profile is for an application to create a secure, authenticated, and transparent tunnel that originates at the initiator host, passes through a firewall, travels through the Internet, passes through another firewall, and finally terminates at the listener host. In order to simulate this scenario, we will implement a BEEP daemon in C that makes use of our Tunnel profile as well as the SASL, TLS, and IDXP profiles that are provided with the RoadRunner BEEP library. The daemon is intended to be a long-lived process that runs on the firewall host and handles all incoming TCP connections on the Internet Assigned Numbers Authority (IANA) assigned port for BEEP. As a proxy, the daemon must handle multiple, concurrent incoming and outgoing connections to connect hosts on opposite ends of the tunnel. In addition, to be a useful prototype the daemon must be robust, configurable, and portable.

We chose the C language for implementing the daemon because of the portability, standardization, and speed that well written C code provides, as well as the fact that C is the

canonical language for implementing daemons and other programs requiring high performance. The IANA has not assigned a port number for BEEP and thus we will make use of an unassigned port for testing purposes. Robustness is necessary because of the fact that the daemon will allow users to bypass the normal firewall rules. Configurability of the daemon is a concern because of the many decisions that such a daemon must make regarding local network policies such as allowing only certain users to bypass the firewall and mandating encryption of all outgoing data. Portability is important, as it will allow firewalls running on various operating systems such as *Linux*, *OpenBSD*, and *Solaris* to use the same code base. At this time we see no reason for a kernel based implementation of the daemon considering that code running in kernel space is: subject to much smaller memory bounds, not allowed to use user space libraries, and will probably not offer greater performance than user space code for this application. In addition, many firewalls are implemented as daemons and thus also run in user space.

The BEEP proxy daemon is a complicated software project that will require much time and testing. Our current Tunnel server will evolve into this daemon over the course of the second semester.

4.3. INTEROPERABILITY

The IETF requires a Proposed RFC to have at least two interoperable implementations before it can become a Draft RFC, which could then eventually become a Standard RFC. We are creating two Tunnel implementations in the Java and C languages and intend for them to be fully interoperable in order to fulfill this requirement. In addition, interoperability aids in confirming the

proper operation of the underlying BEEP libraries and confirming our understanding of the Tunnel protocol.

Interoperability is not a trivial task given that BEEP is a relatively new protocol, both BEEP libraries are in the beta phase, and the APIs are still changing. In addition, two team members had minor interoperability problems with earlier versions of the BEEP libraries in research conducted during the summer.

4.4. FINAL TESTING

We will need to test multiple scenarios with different network configurations to ensure that our product functions properly. First, we will need to verify single host and host-to-host communication, which is configuration independent. We will test C and Java in this form, as well as using both C and Java on the client and server for the host-to-host test. This ensures the most basic compatibility and functionality. Further testing will include a more complicated network setup.

In the ideal situation, a machine on a particular network wants to access a machine on a protected network that it would otherwise have no access to. We will demonstrate this inability to communicate using traditional methods including *ping* or *telnet*. The firewall that protects that secure network will execute either the C or Java version of our proxy application, and the client and protected server will get the C or Java version of our client or server applications. The specific tests are outlined below. In each configuration, we will test a Tunnel request based upon each of the valid combinations of Tunnel attributes. This includes but is not limited to `fqdn` and `port`, IP address and port, and specific service requests.

4.4.1. Single Host C

This test will ensure basic functionality of the C profile. As a trivial case, it has little real relevance, but provides a basis for further testing.

4.4.2. Single Host Java

This test serves the same purpose as the equivalent C test.

4.4.3. Client C, Server Java

This test ensures that a C client can interact with a Java server. This case proves that the BEEP implementations are compatible but provides little real gain for the Tunnel profile since it's not actually tunneling.

4.4.4. Client Java, Server C

This test serves a similar purpose as the previous test, only in reverse.

4.4.5. Client C, Proxy C, Server C

This is the first actual example of a tunneling situation. This test ensures that the C version of the Tunnel software works properly. It will first show that a direct connection between the client and server is not possible, and then show that a connection using an application implementing the Tunnel profile will allow such a thing. Since all code is in C using the C libraries, this test will only prove functionality of the C code.

4.4.6. Client Java, Proxy Java, Server Java

This test mirrors the previous test except that it applies to the Java versions instead of the C versions.

4.4.7. Client C, Proxy Java, Server C

This test proves that C clients and servers can communicate using a Java Proxy. Once we have shown the single hop communication to work, this will only be an exercise in interoperability testing.

4.4.8. Client Java, Proxy C, Server Java

This test mirrors the previous test except with a Java client and server using a C proxy for tunneling. Again, the main purpose is to show interoperability.

4.4.9. Client C, Proxy 1 C, Proxy 2 C, Server C

Once we have shown some basic interoperability, and single hop proxy hopping, the next task is to show that multiple hop proxy hopping works. In this case, we will have two proxies, each representing a network firewall. The client and server in this test will not have the ability to reach past their own firewall on the required port. But then when they use the Tunnel profile, the client is passed all the way through to the server.

4.4.10. Client C, Proxy 1 C, Proxy 2 C, Server C

This test mirrors the previous test only using Java instead of C. There is no need to prove language interoperability here since previous tests have already established this.

APPENDIX A: REFERENCES

Dierks T, Allen C. RFC 2246: "The TLS Protocol Version 1.0". http://www.ietf.org/rfc/rfc2246.txt?number=2246. January 1999.

Feinstein B, Matthews G, White J. "The Intrusion Detection Exchange Protocol (IDXP)". http://www.ietf.org/internet-drafts/draft-ietf-idwg-beep-idxp-07.txt. October 22, 2002.

Information Sciences Institute – University of Southern California. RFC 793: "Transmission Control Protocol". http://www.ietf.org/rfc/rfc0793.txt?number=793. September 1981.

IP Security Protocol (ipsec) Charter. http://www.ietf.org/html.charters/ipsec-charter.html. October 1, 2002.

Meyers J. RFC 2222: "Simple Authentication and Security Layer (SASL)". October 1997.

New D. "The TUNNEL Profile". http://www.ietf.org/internet-drafts/draft-ietf-idwg-beep-tunnel-05.txt. December 5, 2002.

Pennington, Havoc. GTK+/Gnome Application Development. Riders, Indianapolis 1999.

Rescorla, Eric. SSL and TLS: Designing and Building Secure Systems. Addison Wesley.

Rose, Marshall T. BEEP: The Definitive Guide. O'Reilly 2002.

Rose, Marshall T. RFC 3081: "Mapping the BEEP Core onto TCP". http://www.ietf.org/rfc/rfc3081.txt?number=3081. March 2001.

Rose, Marshall T. RFC 3082: "The Blocks Extensible Exchange Protocol Core". http://www.ietf.org/rfc/rfc3080.txt?number=3080. March 2001

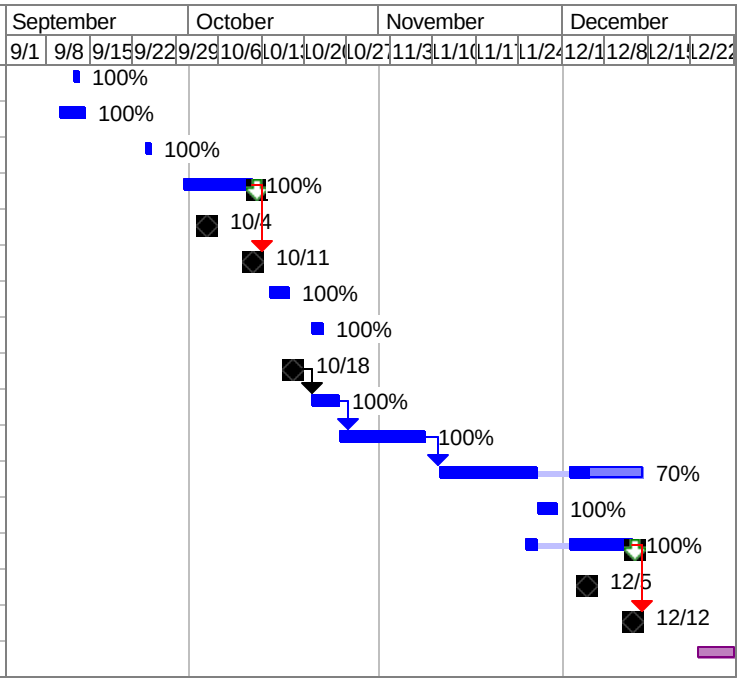
Secure Shell IETF Charter. http://www.ietf.org/html.charters/secsh-charter.html. October 28, 2002.

Wright, Peter. GTK+/GNOME Programming. Wrox, Birmingham 2000.

APPENDIX B: SCHEDULE

Table 1: First Semester Schedule

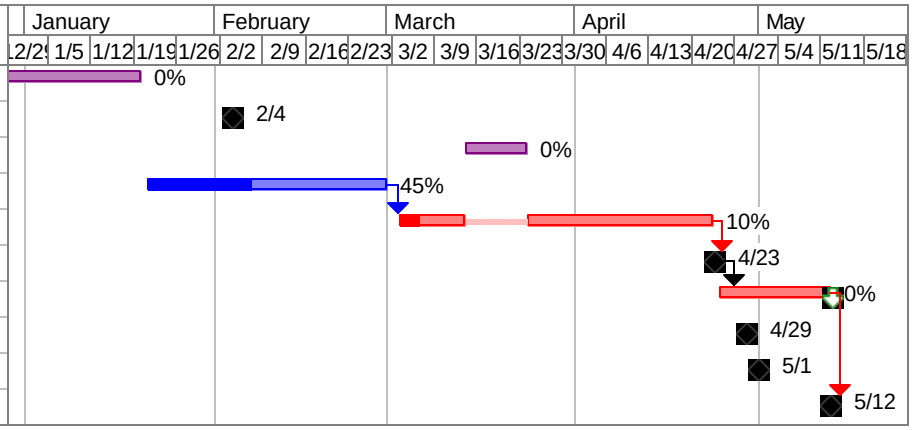
ID	Task Name
1	Site Visit To Aerospace
2	Prepare Development Machine
3	Group Presentation
4	Proposal
5	Draft Proposal Due
6	Proposal Due
7	Evaluate Tunnel Draft
8	Fall Break
9	Choose BEEP Implementation
10	Single Host BEEP Communication
11	Peer-to-peer BEEP Communication
12	One-Hop BEEP Communication
13	Thanksgiving Break
14	Midyear Report
15	Draft Midyear
16	Midyear Report
17	Winter Break



First semester, we stayed fairly close to schedule, with some slight variations to the original. Although the C version has passed our initial tests, One-Hop BEEP Communications are not quite finished yet since the Java version of the Tunnel profile has not been tested.

Table 2: Second Semester Schedule

ID	Task Name
17	Winter Break
18	Presentation
19	Spring Break
20	Multiple Hop Tunneling
21	Firewall Proy Hanlding
22	Code Freeze
23	Final Report and CD
24	Draft Presentation
25	Draft Final Report and CD
26	Final Report and CD



Preliminary work has already begun on second semester tasks. Multiple Hop Proxy works to some degree in C. We think that Firewall Proxy Handling will be trivial after the normal proxy hopping works. Most of our scheduled time will be devoted to regular testing.

APPENDIX C: CAPABILITIES

Dependencies

Implementation of Tunnel as a BEEP profile depends upon a working version of BEEP. Some versions exist, but are still works in progress and cannot be guaranteed to function properly. We will start with the most trusted one available in both Java and C and go from there. If BEEP cannot operate in the way that it should, this may become a problem, but we hope that only the obscure functionality will have problems, if any.

Hardware

We currently have a single consumer grade personal computer running Redhat 7.3, which has served as our first host for testing BEEP profiles. It has an Intel Pentium 4 processor and 256MB of RAM, which should be enough for the relatively lightweight open-source software we anticipate running. If this is not powerful enough to run the Tunnel Protocol as we write it, we cannot expect any adoption. Additionally, we have a second personal computer similar to the first, running Redhat 8 with two network interface cards allowing for firewall testing. For more complicated network topologies, we may need still more hardware, which we are currently trying to acquire.

Software

We plan to use Java and C to implement the BEEP Profiles for Tunnel. The two languages are needed to fulfill not only the desires of Aerospace, but also the requirements for acceptance as an Internet Standard. We plan to use a free web-server daemon (Apache) for communicating our

work to others via the Internet. Additionally, we have a variety of Microsoft operating system licenses available to us, which may become useful for cross-platform compatibility testing.

Team Member Profiles

- Nick Hertl (Project Manager)

Nick is a senior Computer Science major at Harvey Mudd College. He has worked in System Administration for three years, mostly for the HMC Computer Science Department. He spent the past two summers working at Microsoft, the most recent of which involved some high-level network programming. His main experience with low level network protocols comes from a Computer Networks class taken in the Spring of 2002 with Professor Mike Erlinger. He skis on both snow and water in his rare free time.

- Will Berriel

Will is senior Computer Science major at Harvey Mudd College. He has taken courses in Operating Systems and Computer Networks. He has worked with BEEP before in developing an IDXP profile for PermaBEEP and client and server applications to transmit IDMEF messages between an analyzer and a viewer. He currently competes for the Claremont Colleges in Cross Country and Track.

- Richard Fujiama

Richard is a senior Computer Science major at Harvey Mudd College. He has several years of experience as a Windows and Unix system administrator as well as experience gained from research with Mike Erlinger on the IDXP protocol. He is comfortable with network programming

in the C and Java languages. His interests include operating systems, martial arts, and entrepreneurship.

- Chip Bradford

Chip is a senior Computer Science major at Harvey Mudd College. He has extensive experience programming C/C++, especially in low-level environments such as Operating Systems and the network stack. His summer was spent setting up a framework for automated testing of a piece of billing software developed by LPASystems for Xerox. He is currently tutoring the Networks class at Harvey Mudd. His other interests include computer games and parties.

APPENDIX D: TUNNEL.H

```
#ifndef __RR_TUNNEL_H__
#define __RR_TUNNEL_H__

typedef struct _RRTunnel RRTunnel;
typedef struct _RRTunnelClass RRTunnelClass;

#define RR_TUNNEL_URI "http://xml.resource.org/beep/profiles/TUNNEL"

#include <librr/rr-channel.h>

G_BEGIN_DECLS

#define RR_TYPE_TUNNEL (rr_tunnel_get_type ())
#define RR_TUNNEL(obj) (G_TYPE_CHECK_INSTANCE_CAST((obj), RR_TYPE_TUNNEL,
RRTunnel))
#define RR_TUNNEL_CLASS(klass) (G_TYPE_CHECK_CLASS_CAST ((klass),
RR_TYPE_TUNNEL, RRTunnelClass))
#define RR_IS_TUNNEL(obj) (G_TYPE_CHECK_INSTANCE_TYPE((obj),
RR_TYPE_TUNNEL))
#define RR_IS_TUNNEL_CLASS(klass) (G_TYPE_CHECK_CLASS_TYPE ((klass),
RR_TYPE_TUNNEL))
#define RR_TUNNEL_GET_CLASS(obj) (G_TYPE_INSTANCE_GET_CLASS ((obj),
RR_TYPE_TUNNEL, RRTunnelClass))

struct _RRTunnel {
        RRChannel parent_object;

        GError *response_error;
};

struct _RRTunnelClass {
        RRChannelClass parent_class;
};

GType rr_tunnel_get_type (void);

gboolean rr_tunnel_start (RRConnection *connection, GError **error,
                          gchar* payload);


G_END_DECLS

#endif /* __RR_TUNNEL_H__ */
```

APPENDIX E: TUNNEL.C

```
#include <libxml/xmlmemory.h>
#include <libxml/parser.h>
#include <librr/rr.h>
#include "tunnel.h"

#include <stdio.h>
#include <string.h>

static GObjectClass *parent_class = NULL;

static gboolean frame_available (RRChannel *channel, RRFrame *frame,
                                 GError **error);
static gboolean client_init (RRChannel *channel, GError **error);

static gboolean replyOK(RRConnection * conn, RRChannel* channel,
                        GError** error);

static void sendError(gint code,  gchar* text, RRChannel* channel,
                      GError** error);

static RRConnection *
init_connection (const gchar *hostname, gint port);

static gchar ok_msg[]  = RR_BEEP_MIME_HEADER "<ok />\r\n";

static gboolean
xmlValidTunnelNode (xmlNodePtr node, GError **error)
{
        int numAttrs;
        xmlAttrPtr cur;

        /* Make sure we can parse from the node */
        if (!node)
                return FALSE;

        /* Make sure its a tunnel node */
        if (xmlStrcmp (node->name, "tunnel") != 0)
                return FALSE;

        /* Count the number of attributes in this node */
        numAttrs = 0;
        cur = node->properties;
        while (cur) {
                ++numAttrs;
               cur = cur->next;
        }

        /*
         * Return TRUE for valid attribute combos
```

```
         * According to the tunnel draft, the only allowable combos are:
         *    fqdn + port;
         *    fqdn + srv;
         *    ip4 + port;
         *    profile, but only on the innermost element;
         *    endpoint, but only on the innermost element; or,
         *    no attributes, but only on the innermost element.
         */
        if (((numAttrs == 2)
             && ((xmlHasProp (node, "fqdn")
                  && (xmlHasProp (node, "port")
                      || xmlHasProp (node, "srv")))
                 || (xmlHasProp (node, "ip4")
                     && xmlHasProp (node, "port"))))
            || ((numAttrs == 1) && !node->children
                && ((xmlHasProp (node, "profile")
                     || xmlHasProp (node, "endpoint"))))
            || ((numAttrs == 0) && !node->children))
                return TRUE;

        /* No other set of attributes should be allowed */
        return FALSE;
}

/* function intentionally left blank... it doesn't do anything for now...
*/
static void
rr_tunnel_init (GObject *object)
{
}

/* sets up function pointers for tunnel "objects" */
static void
rr_tunnel_class_init (GObjectClass *klass)
{
        RRChannelClass *channel_class = (RRChannelClass *)klass;

        channel_class->frame_available = frame_available;
        channel_class->client_init = client_init;

        parent_class = g_type_class_peek_parent (klass);
}

/* returns the glib type of the RRTunnel object  */
GType
rr_tunnel_get_type (void)
{
        static GType rr_type = 0;

        if (!rr_type) {
                static GTypeInfo type_info = {
                        sizeof (RRTunnelClass),
                        NULL,
                        NULL,
```

```
                        (GClassInitFunc) rr_tunnel_class_init,
                        NULL,
                        NULL,
                        sizeof (RRTunnel),
                        16,
                        (GInstanceInitFunc) rr_tunnel_init
                };
                rr_type = g_type_register_static (RR_TYPE_CHANNEL,
"RRTunnel",
                                                  &type_info, 0);

                rr_channel_set_uri (rr_type, RR_TUNNEL_URI);
        }
        return rr_type;
}

/* Allows proxy to connect incoming and outgoing sockets */

static gboolean
pass_through (GIOChannel *source, GIOCondition condition, gpointer data)
{
  GIOChannel *dest = (GIOChannel*)data;
  const gsize BUF_SIZE = 1024;
  gchar buffer[BUF_SIZE];
  gsize read, written;

  do {
    g_io_channel_read_chars(source, buffer, BUF_SIZE, &read, NULL);
    g_io_channel_write_chars(dest, buffer, read, written, NULL);
  }
  /* Continue reading while the buffer is full and all the bytes
   * are successfully copied. If an error occurs, just exit quietly
   * since tunnel_close should be called by the glib event loop. */
  while (read >= BUF_SIZE && read == written);
}

/* Terminates second socket when the first one in a proxy connection dies
*/

static gboolean
tunnel_close (GIOChannel *source, GIOCondition condition, gpointer data)
{
  GIOChannel *dest = (GIOChannel*)data;

  g_io_channel_unref(source);
  g_io_channel_unref(dest);
}

/* Automatically gets called when a new frame comes in  */

static gboolean
frame_available (RRChannel *channel, RRFrame *frame, GError **error)
{
        RRTunnel *tunnel = RR_TUNNEL (channel);
```

```
        RRMessage *msg;
        RRConnection *conn = channel->connection;
        RRConnection *out_going;
        GIOChannel *in, *out;
        gchar *body;
        gint32 size;
        xmlDocPtr doc;
        xmlNodePtr cur;

        g_return_val_if_fail (RR_IS_TUNNEL (channel), FALSE);


        /* Remove MIME headers */
        body = rr_frame_mime_get_body (frame);
        size = rr_frame_mime_get_body_size (frame);

        /* Parse xml in the frame */
        if (!(doc = xmlParseMemory (body, size))) {
          sendError(500, "Malformed XML", channel, error);
               goto error;
        }

        /* Get the document root object */
        if (!(cur = xmlDocGetRootElement (doc))) {
          sendError(500, "Malformed XML: No root element", channel, error);
               goto error;
        }

        if (frame->type == RR_FRAME_TYPE_MSG) {
                /* Make sure this is a tunnel message */
                 if (!xmlValidTunnelNode (cur, error)) {
                        sendError(501, "Syntax error in parameters",
channel, error);
                       goto error;
                }

                /* Check for empty tunnel element */
                if (!cur->children && !cur->properties) {
                  if(!replyOK(conn, channel, error))
                    goto error;
                }
                /* If a non-empty tunnel element */
                else {
                  xmlChar* next_host;
                  xmlChar* port;
                  xmlNodePtr* child;
                  xmlChar* next_msg;
                  if(xmlHasProp(cur, "ip4")){ /* ip4 + port */
                    next_host = xmlGetProp(cur, "ip4");
                    port = xmlGetProp(cur, "port");
                    child = cur->xmlChildrenNode;
                    next_msg = xmlNodeListGetString(doc, child, 1);
                  }
```

```
                else if (xmlHasProp(cur, "fqdn") && xmlHasProp(cur,
"port")) { /* fqdn + port */
                  next_host = xmlGetProp(cur, "fqdn");
                  port = xmlGetProp(cur, "port");
                  next_msg = xmlNodeListGetString(doc, child, 1);
                }
                else {
                  sendError(504, "Parameter not implemented", channel,
error);
                  goto error;
                }


                out_going = init_connection(next_host, atoi(port));

                if(!rr_tunnel_start(out_going, &error, next_msg)){
                  sendError(450, "Failed to connect to next hop",
                            channel, error);
                  goto error;
                }

                if(!replyOK(conn,channel, error))
                  goto error;

                in = RR_TCP_CONNECTION(conn)->iochannel;
                out = RR_TCP_CONNECTION(out_going)->iochannel;

                g_io_channel_ref(in);
                g_io_channel_ref(out);

                g_io_add_watch(in, G_IO_IN | G_IO_PRI, pass_through,
out);
                g_io_add_watch(out, G_IO_IN | G_IO_PRI, pass_through,
in);

                g_io_add_watch(in, G_IO_HUP | G_IO_ERR | G_IO_NVAL,
                               tunnel_close, out);
                g_io_add_watch(out, G_IO_HUP | G_IO_ERR | G_IO_NVAL,
                               tunnel_close, in);

                xmlFree(next_msg);
                xmlFree(next_host);
                xmlFree(port);
              }
              /* FIXME: proxies should handle other tunnel elements */
        }

        else if (frame->type == RR_FRAME_TYPE_RPY) {
              /* Make sure its an ok node */
              if (xmlStrcmp (cur->name, "ok") != 0) {
                sendError(500, "Wrong content for reply, expected :
<ok/>",
                          channel, error);
                    goto error;
```

```
                }

                rr_connection_complete_tuning_reset (conn, channel);
        }

        else if (frame->type == RR_FRAME_TYPE_ERR) {
                xmlChar* code = xmlGetProp(cur, "code");
                xmlChar* errtext =
                  xmlNodeListGetString(doc, cur->xmlChildrenNode, 1);

                // Make sure errtext does not contain % signs.
                g_set_error(error, RR_ERROR, atoi(code), errtext);


                xmlFree(errtext);
                xmlFree(code);

                rr_connection_disconnect (conn, error);
                goto error;
        }

        else /* Something else? */ {
                sendError(500, "Wrong message type", channel, error);
                goto error;
        }

        xmlFreeDoc (doc);
        return TRUE;

 error:
        xmlFreeDoc (doc);
        return FALSE;
}

/* sends a packaged ok message.  saves from repeating this many times  */

static gboolean replyOK(RRConnection * conn, RRChannel* channel,
                        GError** error) {
  RRMessage * msg;
  msg = rr_message_static_new (RR_FRAME_TYPE_RPY,
                               ok_msg,
                               sizeof (ok_msg),
                               FALSE);

  /* Send the message and perform a tuning reset */
  if (!rr_connection_begin_tuning_reset (conn, error))
    return FALSE;
  if (!rr_channel_send_message (channel, msg, error))
    return FALSE;
  rr_connection_complete_tuning_reset (conn, channel);

  return TRUE;
}
```

```
/* sends an error message.  It takes the error code, error text, and then
 * the channel and error pointers so you know which channel to send it on.
 */

static void sendError(gint code,  gchar* text, RRChannel* channel,
                      GError** error) {
  RRMessage* msg = (RRMessage*)rr_message_error_new(code,
                                                    NULL,
                                                    text);
  rr_channel_send_message (channel, msg, error);

}

/* initializes client for tunnelling  */

static gboolean
client_init (RRChannel *channel, GError **error)
{
        rr_connection_begin_tuning_reset (channel->connection, NULL);
        return TRUE;
}

/* initializes connection */

static RRConnection *
init_connection (const gchar *hostname, gint port)
{
        RRProfileRegistry *profreg;
        RRConnection *conn;
        GError *error = NULL;
        gint use_tunnel = TRUE;

        /* Tell roadrunner which profiles we want to support */
        profreg = rr_profile_registry_new ();
        rr_profile_registry_add_profile (profreg, RR_TYPE_TUNNEL, NULL);

        /* Create a connection object */
        if ((conn = rr_tcp_connection_new (profreg, hostname, port,
                                           &error)) == NULL)
                g_error ("connection failed: %s\n", error->message);
        return conn;
}

/* starts the tunnel.  This function gets called by the user application
*/

gboolean
rr_tunnel_start (RRConnection *connection, GError **error, gchar* payload)
{
        RRMessage *msg;
        RRManager *manager;
        RRTunnel *tunnel;
        gchar*  str;
        gsize str_len;
```

```
        if(!payload)
          payload = "<tunnel /> \r\n";

        /* Add the MIME header to the message. */
        str_len = RR_BEEP_MIME_HEADER_LEN + strlen(payload);
        str = g_malloc(str_len);
         g_stpcpy(str, RR_BEEP_MIME_HEADER);
        g_stpcpy(str+RR_BEEP_MIME_HEADER_LEN, payload);
        printf("%d:%s\n", str_len, str);

        g_return_val_if_fail (RR_IS_CONNECTION (connection), FALSE);
        manager = rr_connection_get_manager (connection);
        g_return_val_if_fail (RR_IS_MANAGER (manager), FALSE);

        if ((tunnel = (RRTunnel *)rr_connection_start (connection, NULL,
                                                       RR_TYPE_TUNNEL,
                                                       NULL, error)) == NULL)
                goto error;

        msg = rr_message_static_new (RR_FRAME_TYPE_MSG, str, str_len,
                                     FALSE);
        /* Don't free str yet... its used by reference in the msg object!
*/

        if (!rr_channel_send_message (RR_CHANNEL (tunnel), msg, error))
                goto error;

        if (!rr_manager_wait_for_greeting (manager, error))
                goto error;

        /* FIXME: Are we sure we don't need this? */
        /*
        if (!rr_manager_wait_for_greeting_sent (manager, error))
                goto error;
        */

        if (tunnel->response_error) {

                g_propagate_error (error, tunnel->response_error);
                tunnel->response_error = NULL;
                goto error;
        }

        g_object_unref (G_OBJECT (tunnel));
        g_free(str);
        return TRUE;
 error:
        g_object_unref(G_OBJECT(tunnel));
        g_free(str);
        return FALSE;
}
```