An Implementation of CASP – A Technology Independent Lightweight Signaling Protocol

Master's Project Report
Computer Science Department
University of Kentucky

By:
Shahid Saleem Mohammed

Under guidance of:

Dr. Henning Schulzrinne
Associate Professor, Dept. of Computer Science
Columbia University

Dr. Kenneth Calvert
Associate Professor, Dept. of Computer Science
University of Kentucky

Abstract:
This document describes a prototype implementation of Cross-Application signaling protocol (CASP).
CASP [2] is a general-purpose protocol for managing state information in network devices. It can be used for inter-domain and intra-domain QoS signaling, configuration of middle-boxes, for collecting measurement data and any other application where state management is required. The framework for CASP is defined as a modular protocol, which includes a general purpose messaging layer (M-layer), that in turn supports a number of client layers for particular signaling applications (e.g. QoS, MIDCOM). There is also a special purpose client component for next-peer discovery. My objective in this project was to implement a secure interface for a CASP implementation using OpenSSL [6] library, to develop some of the functionality for Messaging layer and to integrate various components of CASP prototype developed by others involved in CASP development [5].

1. Introduction

1.1 An Introduction to CASP [1, 2]

In the beginning of the Internet, all packets were equal and received the same treatment. As the Internet evolves, it is being used for an increasing number of different applications; now we know that flows originating from some applications or users need special treatment. Hence, flow-specific state needs to be established in network nodes. This implies flow-related signaling is gaining importance in the Internet. For example, signaling is necessary for installing QoS, NAT and firewall control, MPLS label distribution, collection of measurement data or VPN set-up. It would be desirable to satisfy all these signaling needs with a single protocol. In fact, RSVP has been extended so it can be employed for most of the above-mentioned signaling applications. However, RSVP originally was designed to signal QoS for IntServ, and not for general-purpose signaling. Therefore, not surprisingly, it is not ideally suited for that purpose. Moreover, the multitude of RSVP extensions were designed in independent efforts and it is unclear whether they would collaborate in a single implementation. For its original purpose, QoS signaling, RSVP is not widely deployed either. The reason is often believed to be excessive transport and processing overhead, although much of this could be reduced by yet other RSVP extensions. In order to advance the state of signaling, particularly regarding QoS signaling, in November 2001 the NSIS (Next Steps In Signaling) Working Group was chartered by the IETF. It is currently creating requirements and frameworks for a next-generation signaling protocol. The Signaling problem addressed by CASP is the same as the overall problem being addressed by the NSIS activities.

1.2 A CASP Framework

The CASP framework is defined as a modular protocol, which includes a general purpose messaging layer (M-layer), which supports a number of client layers for particular signaling applications (e.g. QoS, MIDCOM). The messaging layer is responsible for delivering signaling messages from the initiator to the responder, typically the data source and the data sink, respectively. (The initiator and responder can also be represented by proxies close-by, e.g., to support end systems that do not themselves have CASP capabilities.) The client layers perform the actual signaling function, e.g., reserve resources or open firewall ports. A CASP client can, however, send back responses that allow the client that sent a message to confirm that the initial message was delivered and to determine whether the operation was successful or encountered an error. This offers end-to-end reliability. To separate signaling message delivery from discovery, CASP also uses a special client, the scout protocol, which is used for next-peer discovery. Figure 1 describes shows different components of a CASP framework. Note that not every CASP node will have a client layer running. Some CASP nodes may only have Messaging (M) layer running, these nodes simply forward the packets that they do not understand to next CASP node.

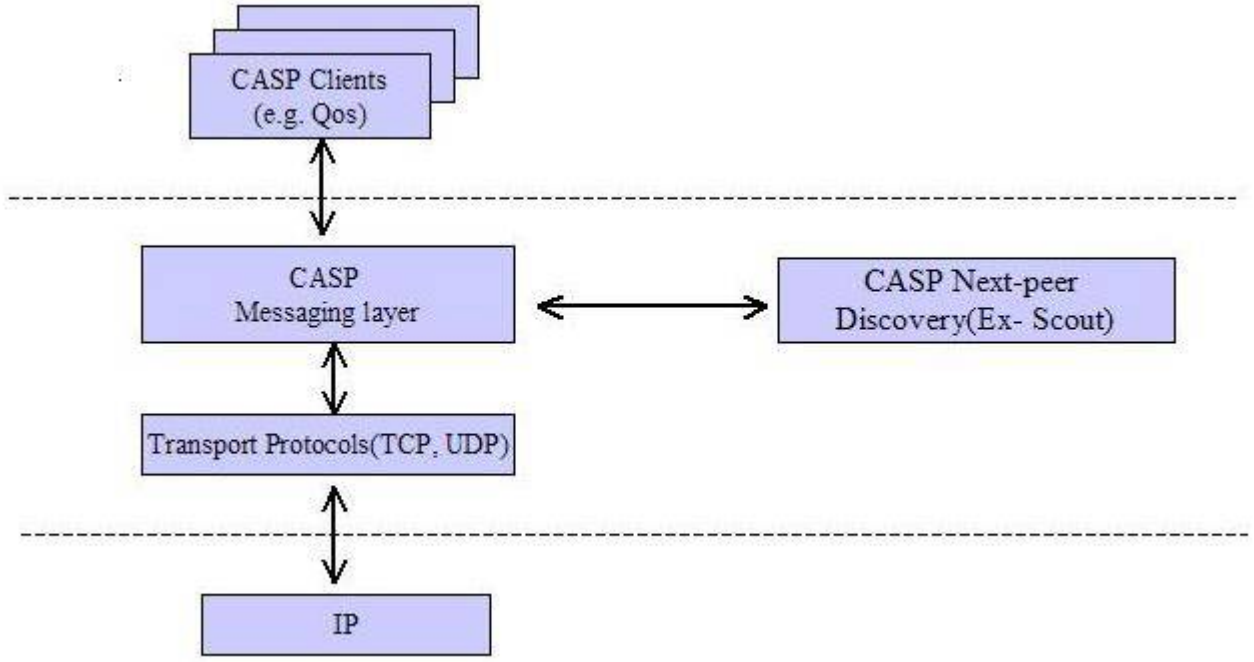

Figure 1: A CASP Framework

A CASP message is handed from one CASP node to another. Typically, nodes are connected by a reliable transport protocol, such as TCP or SCTP. CASP designers [2] chose a reliable transport since signaling requires many of its functions, such as reliability, congestion control, flow control and fragmentation. This may seem surprising since it is often assumed that signaling applications have low data rates, with small, infrequent packets. However, while this is often true, not all signaling applications are that well behaved all the time. For example, authentication tokens and user authorization certificates for AAA can easily push the message size to several kilobytes. A CA-signed certificate including the principal's public key weighs in at about 5 kB, without signed data. Such large messages will likely require fragmentation and may make congestion control advisable. Also, end systems may decide to rapidly probe for available resources if the network is busy. Since CASP nodes may need to perform time-consuming AAA operations, the processing time for each request can vary, so that flow control is needed to keep a neighboring node from overwhelming it with requests. Transport connections in CASP are simple reliable channels, shared between any numbers of CASP sessions, sequentially and in parallel. Nodes can tear down transport connections without affecting the CASP session. Sharing a transport session amortizes the connection setup overhead across many sessions and improves the round-trip-time estimate. The use of a reliable transport also makes it possible to use TLS for channel confidentiality and integrity. Not all CASP nodes need to support all client layers. Figure 2 shows an example where not all nodes support the required client (in this case the `Foo' client). This may occur, for example, where an end-system knows that its first router is CASP aware, but this router does not support all possible clients.

Each CASP node is responsible for passing the signaling message on to its next peer. If it does not already know the address of the next peer, then it must perform a discovery operation by sending a scout request, or using some other mechanism. CASP signaling messages are addressed peer-to-peer. Usually, a single TCP connection or SCTP association is created between each pair of peers. If a connection to the next peer is already available (whether it is from the same signaling session or another) then it is reused. If a connection is not available, then one is opened.

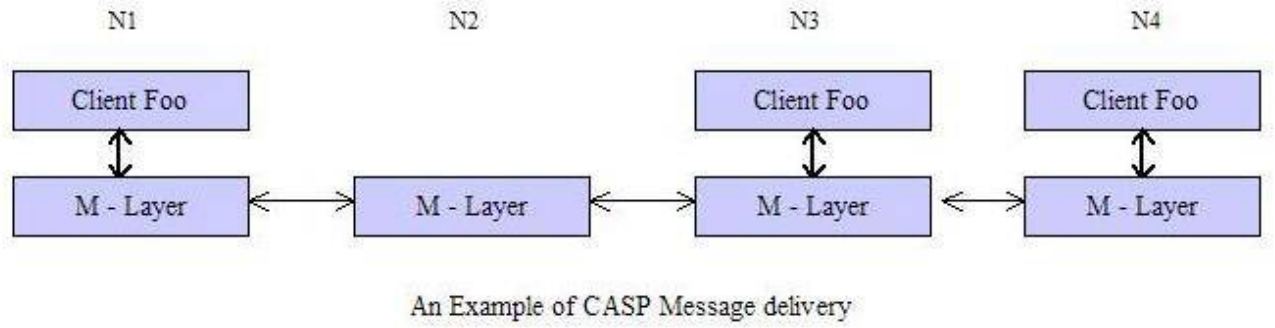

Figure 2

1.3 CASP Message Format [2]

A CASP message consists of a common header, followed by a body consisting of a variable number of variable-length objects, which are identified as being of a particular type.

1.3.1 Length Header (4 Bytes):

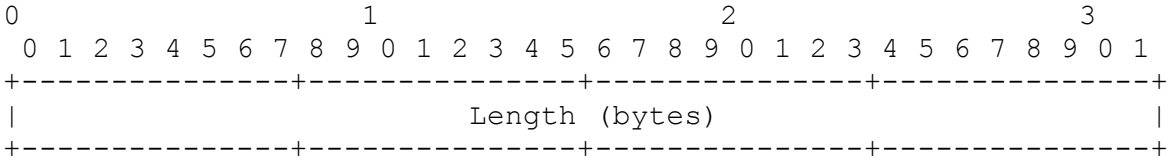

Figure 3

This is the length of the CASP header in bytes. This includes the length of the common header and this length header.

1.3.2 Common Header (20 Bytes):

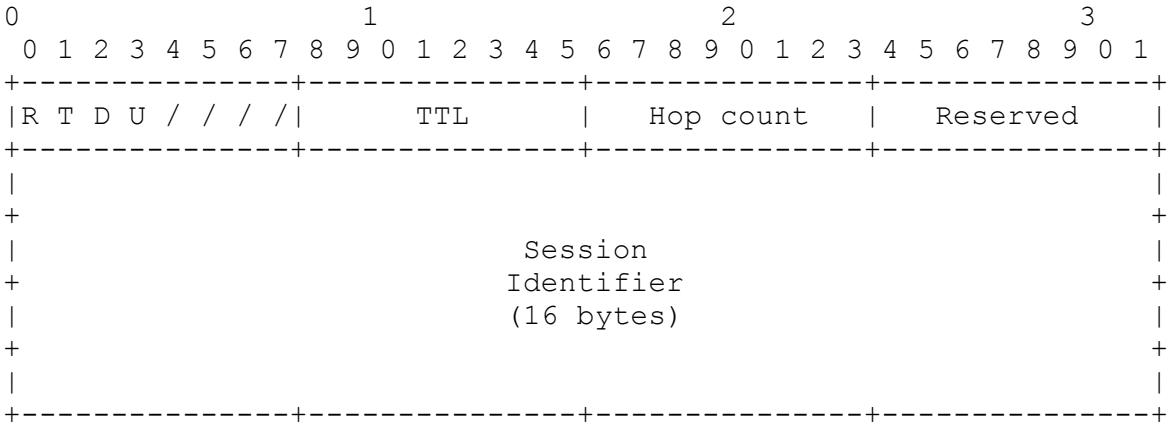

Figure 4

The common header comprises of following fields

1.3.3 FLAGS: The CASP IETF draft [2] currently defines four flag bits

- Reverse (R): The reverse bit indicates that a node should route in the opposite direction to the data flow.
- Tear-down (T): The tear-down (T) bit indicates that this message tears down all CASP M-layer state (and any associated client state) for the CASP M-session. If not set, the message establishes or refreshes M-layer state.
- Discovery (D): The discovery (D) bit requests that the node perform a new discovery operation. If not set, the old next-hop should be used if possible. (This bit cannot be set if R is set, and would usually not be set if the T bit is set).
- Unsecure (U): The un-secure (U) (or "tainted") bit indicates that the message has traversed a hop without channel security.

1.3.4 TTL: **The** TTL (Time to Live) value is decremented by each CASP hop. If the TTD reaches zero then the message should be discarded.

1.3.5 Hops: The Hop count is incremented by each CASP node.

1.3.6 M-Layer objects
The basic CASP message also comprises of a number of M-layer objects, including

- A mandatory CASP source address (NI), either IPv4 or IPv6 (note that this is **not** necessarily the data source address)
- A mandatory CASP destination address (NR), either IPv4 or IPv6 (note that this is **not** necessarily the data destination address)
- An optional session identifier (to allow moving session endpoints)

The session identifier is a 128-bit cryptographically random integer. It is optional and is not needed for one-off messages that are routed based on their destination address only. It MUST appear in messages were the T or R flags are set. A random identifier, together with channel protection, makes it easier to securely identify the session owner.

1.4 Object Formats

An object header has the following fields:

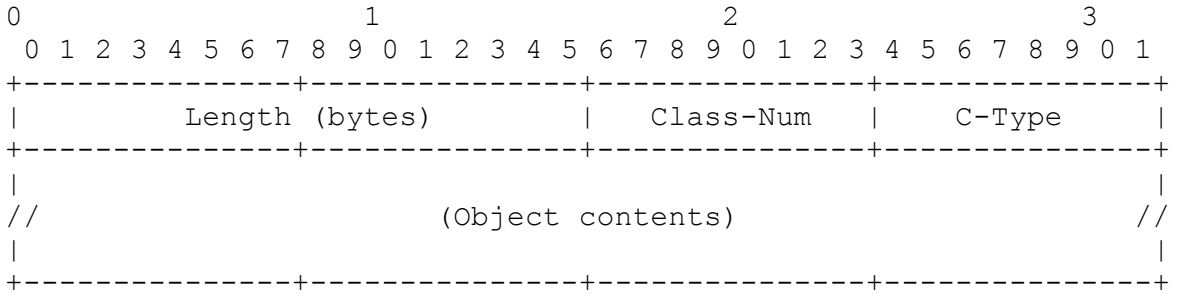

Figure 5

The Class-Num describes what the object does, while the C-Type describes how it looks like (e.g., whether it is IPv4 or IPv6). The Length includes the fixed object header and is thus at least 4. If there is padding, the full

object is the length rounded to the next multiple of four. Each object consists of one or more 32-bit words with a one-word header.

Each CASP object MUST be padded to align on a 32-bit (word) boundary, using the minimal number of additional bytes. Up to three zero-valued bytes are added to the end of the data object field until a word boundary is reached. The length of the padding is not included in the Length field of the object.

At an intermediate CASP node, the basic processing procedure is:

- Look up M-layer state using Flow ID. (If Flow ID is not present, then message is malformed and must be discarded.)
- If M-layer state not found:
  - If R bit set, message is malformed, and must be discarded. Processing is then complete.
  - Otherwise, create M-layer state and record previous peer in it
- If the appropriate client for the given Client Data object is available, pass message up to it
- If R bit is set:
  - Send message on to 'previous' hop. This should use pre-existing transport connection (e.g. TCP or SCTP) if available, otherwise open a new one. Processing is then complete.
- If 'next' hop not recorded in M-layer state:
  - Perform discovery operation
  - Record 'next' hop in M-layer state
- Send message on to 'next' hop (using pre-existing TCP connection if available, otherwise open a new one). Processing is then complete.

1.5 Discovery

As mentioned above, CASP nodes need to discover the next hop. CASP distinguishes between *active, passive* and *directory-based* discovery. For active discovery, each node sends out a UDP CASP "scout" message addressed to the CASP responder IP address, again marked with an IP router alert option similar to RSVP. The next CASP node intercepts the scout message and responds to the IP source address. This is then used by the previous hop to establish a connection to that node. (This mode of operation will work through many, but not all, NATs.)

In passive discovery, the CASP node uses existing routing information to determine the next CASP node. In the simplest case, a node knows that the next IP forwarding hop is CASP-aware and simply picks that address. This case may be uncommon except in edge networks. Within a network, a node may have access to the intra-domain routing table if provided by a link-state protocol such as OSPF. The node can then track which other nodes in the network are CASP-aware, determine the path for the data packet with the responder's IP address and connect to this CASP node.

Finally, with directory-based discovery, a node consults a directory, such as DNS or LDAP, to find the next hop. This mode is particularly interesting for path-decoupled signaling, where the message does not follow the router path, but rather visits, say, only the same autonomous systems (AS) along the way. BGP paths contain the AS number of the next hop. As one technique, we proposed to use DNS SRV records. Each AS has one such DNS entry, e.g., AS 1248 would have the entry 1248.as.arpa. Each entry then identifies a number of CASP servers that can handle CASP messages, ensuring load-balancing and redundancy.

2. Motivation and Problem Description:

Our goal in this project was to implement a CASP prototype to demonstrate the capabilities of CASP protocol. It was an excellent case of teamwork, because a CASP implementation was simply too big a project for a single person to implement. Accordingly then we split it into different components namely M-layer, Client QoS layer, Scout, Firewall Client, AAA and M-layer security [5]. I was mainly involved with security at M-layer, part of

M-layer functionality and then worked with others to integrate the entire package. I will be focusing the rest of the document my work.

The CASP M-layer plays a very important role in the overall operation of CASP protocol, It is responsible for maintaining state information, keeping track of various client layers running on each node, forwarding messages to appropriate client applications and forwarding messages to other CASP nodes if there are no registered client applications on current CASP node. The challenge in this case was to design a multiplexing system that could keep track of all the incoming and out going messages along with the State information. A typical CASP node might have more than 1000 connections open at the same time.

Then comes the interesting part of providing security, due to the inherent nature of the protocol itself, any host running a signaling protocol like CASP is susceptible to various security threats unless suitable security measures are provided, as the signaling messages have to travel through different networks. In an open system such as the Internet, where the identity of the communicating partners is not easy to define, it is very difficult if not impossible to provide protection against all kinds of attacks. The CASP draft lists some of the potential security issues and their fixes. CASP security could be broadly classified into three different categories, (a) Scout protocol security (b) Securing the signaling messages at various places in network and (c) Client layer security. For our initial implementation we were mostly concerned with (b) signaling message security. The security mechanisms provide means to protect the different signaling messages in different portions of the network, including first-peer, intra and inter domain. The security protection of signaling messages at the messaging layer can be classified into authentication, integrity and replay protection.
An example threat scenario would be where an adversary eavesdrops and collects signaling messages and replays them at a latter point in time (or at a different place, or uses parts of them at a different place or in a different way e.g. cut and paste attacks). Without proper replay protection an adversary might be able to mount denial and/or theft of service attacks.

Another example threat scenario would be an adversary being able to inject/modify messages. An adversary modifies signaling messages (e.g. by acting as a man-in-the-middle) to cause an unexpected network behavior with a bogus signaling message. Possible actions are reordering, delaying, dropping, injecting and modifying. An adversary may inject a signaling message requesting a large amount of resources (using a different user identity). If granted it causes other user's resource-request not to be successful and a different initiator (for example a user) to pay for the QoS reservation.

If proper integrity protection is not provided, an adversary might be able to mount denial of service attacks against network elements participating in the QoS protocol, it could also hijack a connection and from forging reservation requests and the corresponding replies.

A signaling protocol like CASP thus must provide strong authentication, integrity protection, protection against replay attacks, prevent denial-of-service attacks against signaling entities and provide confidentiality of signaling messages.

3. Design Overview, Options and Choices

This section briefly describes the design of our implementation along with the design issues and choices we made.
The primary responsibility of the messaging layer is to multiplex messages coming from different CASP peers onto different client applications and vice versa. A single CASP messaging layer session can be used by multiple client states, to ensure that all client states are removed at the same time. Initially our plan was to implement the CASP messaging layer a group of threads, which accept incoming connections, parse the incoming messages and maintain state information. We started of the implementation based on a threaded model using the pthreads library, but at a later point in the interest of simplicity and time constraints, we switched to a poll based model. Also the openssl library which we were using to provide TLS security has known issues with multithreading. Even though OpenSSL is thread-safe, it has some limitations like, an SSL

connection may not be concurrently used by multiple threads. For the multiplexing system, between select system call and poll, we preferred because poll scales more efficiently and we were expecting each CASP node to have thousands of connections.

3.1 IPSEC or SSL or TLS?

Since TCP and SCTP are the main protocols for exchanging signaling messages, as per the CASP paper, we had the option of using either IPSec or SSL/TLS to maintain the integrity of signaling messages. SSL/TLS provides security on top of transport layer. IPSec provides much of the same services, but it works on top of network layer.

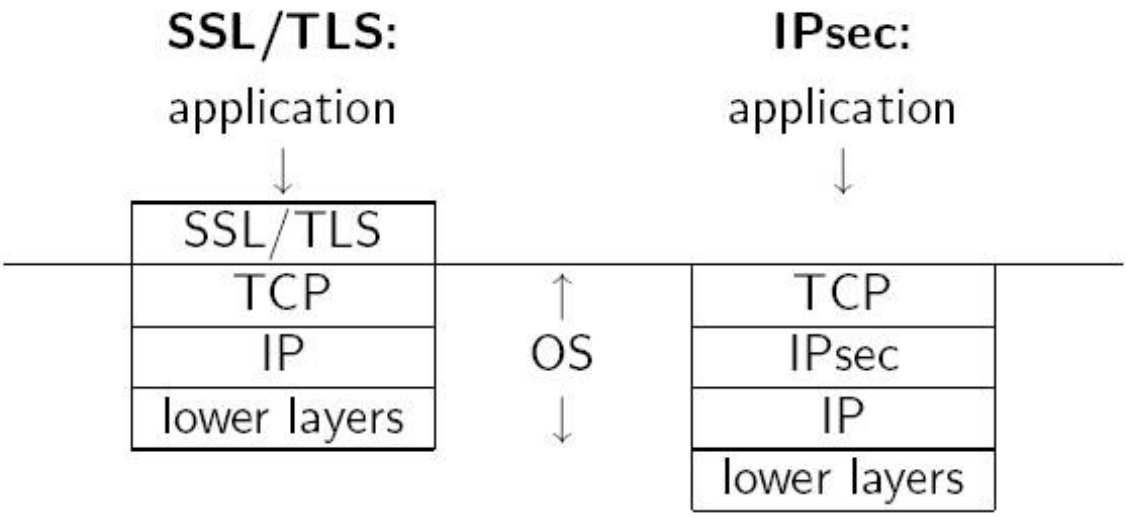

Figure 6: Comparison of SSL/TLS vs. IPSec

IPSec is based on the philosophy that implementing security with in the operating system automatically causes all the applications to be protected without the applications having to be modified. On the down side this also means that IPSec must be implemented in all the Operating Systems. Also with the current commonly used API, IP tells the application only what IP address it is talking to, not what user is on the other end. So it ends up providing only host-level security. The network layer does not know which process packets belong to, all packets to the same host are encrypted the same way. To support user-level security changes have to be made to the application layer.
SSL/TLS on the other hand share the philosophy that it is easier to deploy something if you don't have to change the Operating System. SSL/TLS try to mimic the popular "Berkley Sockets" API and require the applications to interface to SSL instead of TCP. So although applications have to be modified (albeit minimally), the Operating System, which includes TCP and layers below it, does not need to be modified. SSL/TLS also support user level security in that packets to different processes are encrypted differently. On the downside, SSL/TLS assumes that the underlying packet delivery mechanism is reliable, so it cannot run over UDP. Also in case of SSL, since the TCP packet will not be participating in the cryptography, it will have no way of noticing if malicious data is inserted into the packet stream, as long as the bogus data passes the (non-cryptographic) checksum. TCP will acknowledge such data and sent it up to SSL. SSL will discard it because the integrity check will indicate the data is bogus, but there is no way for SSL to tell TCP to accept real data at this point. When the real data arrives, TCP will assume it is duplicate data and discard it. An attacker could launch a successful denial-of-service attack by inserting a single packet into the data stream.

Why prefer TLS to SSL?

The following list describes the security improvements that TLS provides over SSL:

- Key-Hashing for Message Authentication: TLS uses Key-Hashing for Message Authentication Code (HMAC), which ensures that a record cannot be altered while traveling over an open network such as the Internet. SSL Version 3.0 also provides keyed message authentication, but HMAC is considered more secure than the (Message Authentication Code) MAC function that SSL Version 3.0 uses.
- Enhanced Pseudo random Function (PRF): PRF is used for generating key data. In TLS, the PRF is defined with the HMAC. The PRF uses two hash algorithms in a way, which guarantees its security. If either algorithm is exposed then the data will remain secure as long as the second algorithm is not exposed.
- Improved finished message verification: Both TLS Version 1.0 and SSL Version 3.0 provide a finished message to both end points that authenticates that the exchanged messages were not altered. However, TLS bases this finished message on the PRF and HMAC values, which again is more secure than SSL Version 3.0.
- Consistent certificate handling: Unlike SSL Version 3.0, TLS attempts specify the type of certificate, which must be exchanged between TLS implementations.
- Specific alert messages: TLS provides more specific and additional alerts to indicate problems that either session end point detects. TLS also documents when certain alerts should be sent.

In our case, TLS providing both session key establishment based on unilateral and optionally mutual public key based authentication seemed to be a good approach due to following reasons

- Since we were developing application from scratch, it would be easier for us to incorporate TLS/SSL kind of security to protect signaling messages. Also since CASP proposes using TCP/SCTP for exchanging signaling content, TLS would be a good choice for providing
- Flexible security levels. TLS can support privacy, integrity, authentication or some combination of all of these. This allows clients and servers to dynamically, during a session, decide on the level of security required for a particular data transfer, It is possible to use X.509 certificates to authenticate client users and not just client hosts.
- SSL/TLS provides support for session resumption. Once a per-session master secret is established using expensive public key cryptography, multiple connections can be cheaply derived from that master secret.
- In case of first peer communication where First-peer communication refers to the communication between the originator of the signaling message and the edge router in the attached network. In most mobility scenarios the signaling messages are initiated by (or terminate at) the mobile node. Support for IPSec-based protection with IKE or similar to establish an IPSec SA to protect signaling messages between the CASP peers requires interaction between the CASP implementation and the key management daemon. First there needs to be an interface to trigger the dynamic creation and modification of IPSec security associations (such as provided by PF_KEY [52]). Since protection is necessary for the CASP signaling messages only the IPSec security policy database should be able to install traffic selectors with a granularity at protocol type (TCP, SCTP) and specific port numbers. Additionally there is a need to fetch the credentials used at the key management protocol for the purpose of policy based admission control and accounting. Using TLS these tasks are easy since these APIs are available and widely used for example comparing the identity used in a certificate and the URL at a TLS-supporting web browser.

For our initial implementation we decided to target Linux systems. For a session layer implementation, the language of choice was obviously ‘C’.

3.2 Separate ports Vs Upward Negotiation:

Since each CASP node is supposed to support both secure and insecure versions, another design issue was whether to use separate-ports strategy where server listens for connections on two different ports(secure and non-secure) or to use upward negotiation where an extra message has to be added indicating one side would upgrade to SSL.

Since the upward-negotiation strategy could be vulnerable to downgrading attacks if not implemented carefully, we chose to use the separate ports policy for the sake of simplicity.

4. Programming Setup

4.1 CASP daemon:

As per the CASP draft, each CASP node may or may not support all types of clients like QoS, Firewall etc, but every CASP supporting node must run a messaging layer daemon. An M-layer daemon keeps listening for new incoming connections, keeps track of what client applications are running on the current host, forwards the incoming messages to appropriate client layers or forwards them to next peers if no client layers are registered with the messaging layer. For generality we will refer to M-layer daemon as CASP daemon. In our implementation the functionality of CASP daemon is mainly contained in files caspd.c and mLayer.c. The CASP daemon can be started in both secure and non-secure modes based on the command line arguments. The default is to start both secure and non-secure versions.

If started in un-secure mode, the messaging layer daemon starts listening on the un-secure port and blocks at the poll statement waiting for incoming connections.

The CASP daemon does a little more work when the secure version is also started. First it calls the `init_OpenSSL()` function. This function in turn calls `SSL_library_init()` and `THREAD_setup()`.

According to openssl documentation the `SSL_library_init()` function must be called before we do anything else with the library. This function registers the available ciphers and digests. We decided to add thread support and compile the whole package with pthreads library keeping in view the future addition of threads to support M-layer functionality. The next call is to function `seed_prng` which seeds the OpenSSL's Pseudo Random Number Generator. The RAND package in openssl provides a cryptographically strong pseudorandom number generator (PRNG). This means that the "random" data it produces isn't truly random, but it is computationally difficult to predict. Cryptographically secure PRNGs, including those of the RAND package, require a seed. In our case we use the openssl's `RAND_load_file` function to read 1024 random bytes from /dev/urandom. Improper seeding of PRNG function is a common pitfall while using openssl library.

```
RAND_load_file("/dev/urandom", 1024);
```

4.1.1 Setting up the server context:

The CASP daemon then calls the setup_server_ctx() function, which sets up a server context that will be used by the system. The context object could then be used to create new connection objects for each new SSL connections. The connection objects are in turn used to do SSL handshakes, reads and writes. This approach has two advantages. First, the context object allows many structures to be initialized only once, saving time. When we wish to create a new connection we can simply point that connection to the context object. The second advantage of having a single context object is that it allows multiple SSL connections to share data, which is relevant for session resumption. Session data can be stored in context object so that you can automatically resume a session merely by creating a new connection with same context object.

Setting up a new Context: As discussed above, an `SSL_CTX` object will be a factory for producing `SSL` connection objects. This context allows us to set connection configuration parameters before the connection is made, such as protocol version, certificate information, and verification requirements. It is easiest to think of `SSL_CTX` objects as the containers for default values for the SSL connections to be made by a program. Objects of this type are created with the function `SSL_CTX_new`. This function takes only one argument, generally supplied from the return value of one of the SSL_METHOD functions in Table 1.
An `SSL_METHOD` represents an implementation of SSL functionality. In other words, it specifies a protocol version. OpenSSL provides populated `SSL_METHOD` objects and some accessory methods for them. They are listed in Table 1.

Table 1. Functions to retrieve pointers to SSL_METHOD objects

Function	Comments
SSLv2_method	Returns a pointer to SSL_METHOD for generic SSL Version 2
SSLv2_client_method	Returns a pointer to SSL_METHOD for an SSL Version 2 client
SSLv2_server_method	Returns a pointer to SSL_METHOD for an SSL Version 2 server
SSLv3_method	Returns a pointer to SSL_METHOD for generic SSL Version 3
SSLv3_client_method	Returns a pointer to SSL_METHOD for an SSL Version 3 client
SSLv3_server_method	Returns a pointer to SSL_METHOD for an SSL Version 3 server
TLSv1_method	Returns a pointer to SSL_METHOD for generic TLS Version 1
TLSv1_client_method	Returns a pointer to SSL_METHOD for a TLS Version 1 client
TLSv1_server_method	Returns a pointer to SSL_METHOD for a TLS Version 1 server
SSLv23_method	Returns a pointer to SSL_METHOD for generic SSL/TLS
SSLv23_client_method	Returns a pointer to SSL_METHOD for an SSL/TLS client
SSLv23_server_method	Returns a pointer to SSL_METHOD for an SSL/TLS server

OpenSSL provides implementations for SSL Version 2, SSL Version 3, and TLS Version 1. Also, some SSLv23 functions don't indicate a specific protocol version but rather a compatibility mode. In such a mode, a connection will report that it can handle any of the three SSL/TLS protocol versions.

In our CASP implementation, we have used the SSLv23_method(), which return a pointer to generic SSL/TLS method. However we do not want to support SSL version 2 because it is un-secure. For this reason we later make a call to SSL_CTX_set_options to remove SSLv2 from the list of acceptable protocols.

```
        SSL_CTX_set_options(ctx, SSL_OP_ALL | SSL_OP_NO_SSLv2 |
SSL_OP_SINGLE_DH_USE);
```

After creating an SSL_CTX object, we can use this SSL_CTX object to create an SSL type object using the SSL_new function. This function causes the newly created SSL object to inherit all of the parameters set forth in the context.

The next thing that the setup_server_ctx function does is to load the appropriate certificate, which a connecting client may look at to determine if the server is authentic and can be trusted. A peer validates a certificate through verification of its chain of signers. In our CASP implementation we are storing the

certificates in a **certs/** directory. I have used the openssl kit to generate X.509 certificates for each host. The certificates are named as `host_name.pem`, so that a program could automatically load them on startup.
The `SSL_CTX_use_certificate_chain_file` loads the chain of certificates from the appropriate certificate file. The file should contain the certificate chain in order, starting with the certificate for the application and ending with the root CA certificate. Each of these entries must be in PEM format.

In addition to loading the certificate chain, the `SSL_CTX` object must have the application's private key. This key must correspond to the public key embedded within the certificate. The easiest way that we can supply this key to the context is through the `SSL_CTX_use_PrivateKey_file` function.

```
if (SSL_CTX_use_PrivateKey_file(ctx, certName, SSL_FILETYPE_PEM) != 1)
        debug(0, "Error loading private key from file");
```

where `certName` is the variable containing the path of the certificate file.

Since this private key is stored in an encrypted PEM format on the disc, we need to tell openssl the passphrase to decode it. OpenSSL collects passphrases through a callback function.

```
SSL_CTX_set_default_passwd_cb(ctx, password_cb);
```

The function call `SSL_CTX_load_verify_locations` tells our application the list of CA certificates that it can trust.

The `SSL_CTX_set_cipher_list` function allows us to set the list of cipher suites that we authorize our `SSL` objects to use. The list of ciphers is specified by a specially formatted string. This string is a colon-delimited list of algorithms. Given the number of possible combinations, specifying all the acceptable ones explicitly would be quite cumbersome. OpenSSL allows for several keywords in the list, which are shortcuts for sets of ciphers. For instance, "ALL" is a shortcut for every available combination. Additionally, we can precede a keyword with the "!" operator to remove all ciphers associated with the keyword from the list. Using this, we will create a string to define our custom cipher list

```
#define CIPHER_LIST "ALL:!ADH:!LOW:!EXP:!MD5:@STRENGTH"
if (SSL_CTX_set_cipher_list(ctx, CIPHER_LIST) != 1)
    debug(0, "Error setting cipher list (no valid ciphers)");
```

The special keyword " @STRENGTH" indicates that the list of cipher suites should be sorted by their strength (their key size) in order of highest to lowest. Employing this keyword causes our SSL connections to attempt to select the most secure suite possible, and if necessary, back off to the next most secure, and so on down the list.

New Connections:

After doing all these the secure server also blocks on poll system call. When it receives a new connection request on the secure port, it calls SSL_new to create a SSL object.

```
ssl = SSL_new(ctx);
```

This SSL object derives its parameters from the SSL_CTX object we created earlier. For a secure connection the accept() function is replaced by SSL_accept() function.

4.1.2 Certificate Verification and Peer Authentication:

Certificate verification entails checking the cryptographic signatures on a certificate to be sure an entity we trust has signed that certificate. It also involves checking the certificate's `notBefore` and `notAfter` dates, trust settings, purpose, and revocation status. Verification of a certificate takes place during the SSL handshake (during the call to `SSL_connect` or `SSL_accept`).
Once we've properly loaded trusted certificates into the `SSL_CTX` object, OpenSSL has a built-in function to verify the peer's certificate chain automatically. The routine used to verify the certificate chain could be changed from the default via a call to `SSL_CTX_set_cert_verify_callback`, but under almost all circumstances, this is undesirable since the default routine for signature verification is amply complete and

robust. Instead, the developer can specify a different callback that filters the return status of the default verification and returns the new verification status. The function to perform this task is **SSL_CTX_set_verify**.

After connecting the `SSL` object, we need to assert that some assumed properties about the connection are indeed true. OpenSSL provides several functions that allow us to create a post-connection verification routine to make sure that we haven't been fooled by a malicious peer. This post-connection verification routine is very important because it allows for much finer grained control over the certificate that is presented by the peer, beyond the certificate verification that is required by the SSL protocol proper.

The function `SSL_get_peer_certificate` will return a pointer to an `X509` object that contains the peer's certificate. While the handshake is complete and, presumably, the verification completed correctly, we must still use this function. Consider the case in which the peer presents no certificate when one is requested but not required. The certificate verification routines—both the built-in and the filter—will not return errors since there was nothing wrong with the **NULL** certificate. Thus, to prevent this condition, we must call this function and check that the return value is not NULL. If this function returns a non-NULL value, the reference count of the return object is increased. In order to prevent memory leaks, we must call `X509_free` to decrement the count after we're done using the object.

Our application will be vulnerable if we do not check the peer certificate beyond verification of the chain. For example, let's say that we're making a web browsing application. To keep it simple, we'll allow just one trusted CA. When we do this, any SSL peer with a certificate signed by the same CA will be verified correctly. This isn't secure. Nothing prevents an attacker from getting his own certificate signed by the CA and then hijacking all your sessions. We thwart this kind of masquerade by tying the certificate to some piece of information unique to the machine. For purposes of SSL, this piece of information is the entity's *fully qualified domain name* (FQDN), also called the DNS name.

The common practice with X.509v1 certificates was to put the FQDN in the certificate's `commonName` field of the `subjectName` field. This practice is no longer recommended for new applications since X.509v3 allows certificate extensions to hold the FQDN as well as other identifying information, such as IP address. The proper place for the FQDN is in the `dNSName` field of the `subjectAltName` extension.

The function `post_connection_check` is used to perform these checks for us. We always check for the `dNSName` field first, and if it isn't present, we can check the `commonName` field. Checking the `commonName` field is strictly for backward compatibility, so if this isn't a concern, it can safely be omitted.

`SSL_get_verify_result` is another API function that we will employ in our post-connection check. This function returns the error code last generated by the verification routines. If no error has occurred, `X509_V_OK` is returned. We should call this function and make sure the returned value equals `X509_V_OK`.

When browsing the example application, it is obvious that robust error handling has been left out for clarity. For example, the programs simply exit when an error occurs. In most cases, we will want to do something better to handle errors in some application-specific way. Checking the verify result is always a good idea. It makes an assertion that no matter what error handling occurred up to this point, if the result isn't OK now, we should disconnect.

```
long post_connection_check(SSL *ssl, char *host)
{
   X509     *cert;
   X509_NAME *subj;
   char      data[256];
   int       extcount;
   int       ok = 0;

   if (!(cert = SSL_get_peer_certificate(ssl)) || !host)
      goto err_occured;
   if ((extcount = X509_get_ext_count(cert)) > 0)
   {
      int i;

      for (i = 0;  i < extcount;  i++)
```

```
      {
         char              *extstr;
         X509_EXTENSION    *ext;

         ext = X509_get_ext(cert, i);
         extstr = (char*) OBJ_nid2sn(OBJ_obj2nid(X509_EXTENSION_get_object(ext)));

         if (!strcmp(extstr, "subjectAltName"))
         {
            int                 j;
            unsigned char       *data;
            STACK_OF(CONF_VALUE) *val;
            CONF_VALUE          *nval;
            X509V3_EXT_METHOD    *meth;
            void                *ext_str = NULL;

            if (!(meth = X509V3_EXT_get(ext)))
               break;
            data = ext->value->data;
#if (OPENSSL_VERSION_NUMBER > 0x00907000L)
            if (meth->it)
              ext_str = ASN1_item_d2i(NULL, &data, ext->value->length,
                                ASN1_ITEM_ptr(meth->it));
            else
              ext_str = meth->d2i(NULL, &data, ext->value->length);
#else
            ext_str = meth->d2i(NULL, &data, ext->value->length);
#endif
            val = meth->i2v(meth, ext_str, NULL);
            for (j = 0;  j < sk_CONF_VALUE_num(val);  j++)
            {
               nval = sk_CONF_VALUE_value(val, j);
               printf("value = %s\n host = %s\n",nval->value,host);
               if (!strcmp(nval->name, "DNS") && !strcmp(nval->value, host))
               {
                  ok = 1;
                  break;
               }
            }
         }
         if (ok)
            break;
      }
   }

   if (!ok && (subj = X509_get_subject_name(cert)) &&
      X509_NAME_get_text_by_NID(subj, NID_commonName, data, 256) > 0)
   {
      data[255] = 0;
      if (strcasecmp(data, host) != 0)
         goto err_occured;
   }

   X509_free(cert);
   return SSL_get_verify_result(ssl);

err_occured:
   if (cert)
```

```
    X509_free(cert);
  return X509_V_ERR_APPLICATION_VERIFICATION;
}
```

Figure 9: post_connection_check() routine

At a high level, the function `post_connection_check` is implemented as a wrapper around **SSL_get_verify_result**, which performs our extra peer certificate checks. It uses the reserved error code `X509_V_ERR_APPLICATION_VERIFICATION` to indicate errors where there is no peer certificate present or the certificate presented does not match the expected FQDN. This function will return an error in the following circumstances:

- If no peer certificate is found
- If it is called with a `NULL` second argument, i.e., if no FQDN is specified to compare against
- If the `dNSName` fields found (if any) do not match the host argument and the `commonName` also doesn't match the host argument (if found)
- Any time the `SSL_get_verify_result` routine returns an error

As long as none of the above errors occurs, the value `X509_V_OK` will be returned.

4.2 Client Library Functions:

Since the functionality was split into different layers, separate functions had to be provided to enable messaging layer and scout discovery part to establish secure and un-secure connections. The client library functions accomplish exactly this task. The following are the client library functions with a brief description for each of them.

`caspConnect`: This function could be called either by scout layer or the messaging layer. It takes the Address structure as the argument. If first tries to make a make a secure connection and in case of failure sets up an un-secure connection. A CASP node would call the scout function before calling this function in order to determine the next CASP node address. This function also adds the resulting connection to the list of socket descriptors being monitored by the CASP daemon for incoming data. It returns the resulting new socket descriptor.

`caspSend`: The caspSend function takes socket descriptor, data buffer and buffer length as arguments and appropriately calls the write() or the SSL_write() functions.

`caspClose`: This function closes the socket descriptor and in case of secure sockets calls SSL_free to free the ssl object.

`sockType`: This is a utility function that helps messaging layer to determine if the socket that it is sending data upon is a secure connection or an un-secure connection. The M-layer plugs in this information in its header.

4.3 Messaging Layer Functionality

The messaging layer functionality was developed jointly by Jignesh and me. The following section describes how the messaging layer code is organized and the data structures used by messaging layer. The messaging layer predominantly has two interfaces, one with the client layer and second with the CASP daemon. The CASP daemon hands any incoming messages to the messaging layer and the messaging layer calls the client library functions if it needs to send any messaging. Some of the important mLayer functions are

`generateSessID (unsigned char sessID[16])` – This function generates a unique random session ID by reading 16 bytes from /dev/urandom.

`createSession` – This takes a session ID as a parameter and creates state table entries for the current node. It calls scout to determine the nextCASP node's IP address. It then sends a message to its next CASP node, so that the next CASP node can also add entries for this session ID in its state table.

`void parseMsg(char *buffer, int inSD)` – This is the first function called when any incoming message arrives. Its main job is to parse the message contained with in the buffer into mLayer format. After this it calls the `processMsg` function.

`void processMsg(struct caspMsg *cMsg, int inSD)` – As its name suggests this function is mainly responsible for deciding the next course of action based upon the entries it already has in its State table. If it is the first message for this session ID it calls the `createSession` function. If any client layers for this message type are registered then it hands over the message to appropriate client layer. If no appropriate client layers are registered then the messaging layer just forwards the incoming message on the appropriate outgoing socket descriptor, again based on its state table entries. It also refreshes the session expiration timer for the M-Layer state table.

4.3.1 M-Layer State Table:

Session ID	Timeout	Client ID	Source Address	Destination Address	Client Buff ptr	Incoming socket	Outgoing socket	Next CASP node	Pending Message

Data Structures:

State Table: As mentioned above, the state table maintains information about on current ongoing sessions. Entries in this table are looked up by the messaging layer every time a new message arrives or when a message is being sent to identify the next course of action. The entries in this table are hashed based on session ID's. We used the hcreate() function to create and maintain this hash. (I will later speak about how we had to split the hash table.)

```
typedef struct {                        // Address structure;
  enum which {V4, V6} type;             // To identify type of address.
  union {
    struct in_addr v4;   // IPv4 address, s_addr.
    struct in6_addr v6; // IPv6 address, s6_addr.
  } ip;
} Address;

struct state {
  unsigned char sessID[16];             // Session ID. Key for the main hash table.
  unsigned int timeOut;                 // Time out value for this session.
  unsigned char clientID;               // Client ID of this session.
  Address srcAddr;                      // Source Address for this session.
  Address desAddr;                      // Destination Address for this session.
  void* clientBuffer;                   // For Client to use for this session.
  int inSD;                             // Incoming socket descriptor for this session,
                                        // used to send messages in reverse direction
  int outSD;                            // Outgoing Socket descriptor for this session
                                        // used while forwarding messages that belong to this session
  Address nextCASP;                     // IP address of next CASP node, provided by scout
                                        // It is also key for another hash table containing socket and
```

```
                                    // list of sessions using this socket. When this list gets empty
                                    // remove the connection to nextCASP node.
  struct pendingMsg* pMsg;          // Pending messages are added here until discovery is done
};
```

Connection table:
Since existing connections to a next CASP node can be shared, this table helps us to keep track of all the existing sessions which are reusing the same connection. This table is indexed by next CASP node's IP address. When the list is empty we can safely close down the connection by making a call to caspClose().

```
struct connectionState {            // Connection state table.
  unsigned char nextCASPkey[16];    // Key for this state table;
  int soc;                          // Connected through this socket.
  struct sessIDNode* sList;         // List of sessions using this connection.
};
```

5. Un-foreseen Problems:

5.1 Multiplexed I/O with openssl: Apparently the usage of "poll" or "select" type multiplexed I/O with OpenSSL library is far from clean due to the record oriented nature of SSL. The following paragraph describes the problem we were facing and how we fixed it.

The basic problem we were facing was that SSL is a record-oriented protocol. Thus, even if we want to read only one byte from the SSL connection, we still need to read the entire record containing that byte into memory. Without the entire record in hand, OpenSSL can't check the record MAC and so can't safely deliver the data. Unfortunately, this behavior interacts poorly with `poll()`, as shown in Figure 8.

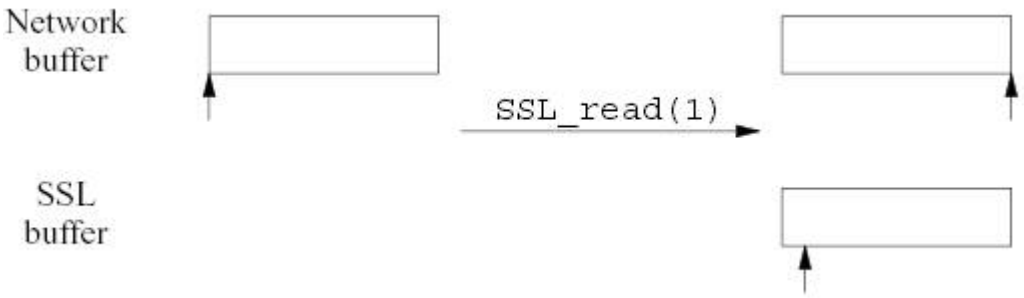

Figure 8: Select/Poll problem with openssl [9]

The left-hand side of Figure 8 shows the situation when the machine has received a record but it's still waiting in the network buffers. The arrow represents the read pointer, which is set at the beginning of the buffer. The bottom row represents data decoded by OpenSSL but not yet read by the program (the *SSL buffer*). This buffer is currently empty so we haven't shown a box. When the program calls `select()` at this point, it will return immediately indicating that a call to `read()` will succeed. Now, if we call `SSL_read()` requesting one byte. This takes us to the situation at the right side of the figure. As we said earlier, the OpenSSL has to read the entire record in order to deliver even a single byte to the program. In general, the application does not know the size of records and so its reads will not match the records. Thus, the box in the upper right-hand corner shows that the read pointer has moved to the end of the record. We've read all the data in the network buffer.
When the implementation decrypts and verifies the record, it places the data in the SSL buffer. Then it delivers the one byte that the program asked for in `SSL_read()`. We show the SSL buffer in the lower right-hand corner. The read pointer points somewhere in the buffer, indicating that some of the data is available for reading but some has already been read.
Consider what happens if we call `select()` at this point. `select()` is concerned solely with the contents of the network buffer, and that's empty. Thus, as far as `select()` is concerned there's no data to read.

Depending on the exact arguments it's passed, it will either return saying that there's nothing to read or wait for some more network data to become available. In either case we wouldn't read the data in the SSL buffer. Note that if another record arrived then `select()` would indicate that the socket was ready to read and we'd have an opportunity to read more data. Thus, `select()` is an unreliable guide to whether there is SSL data ready to read. We need some way to determine the status of the SSL buffer. This can't be provided by the operating system because it has no access to the SSL buffers. After a bit researching I found that OpenSSL provides exactly such a function. The function `SSL_pending()` tells us whether there is data in the SSL buffer for a given socket. `SSL_pending()` returns the number of bytes which are available inside **SSL** for immediate read.

5.2 Problem with hcreate allowing only one hash per process: Another interesting problem that we faced was with the `hcreate` function, which we were using to create hash tables. We later needed to create more than one hash table to index entries by session ID, next node's IP address etc. But the `hcreate` function has a limitation that it only allows us to create one hash table per program. Dr. Schulzrinne suggested the fix for this and we ended up prefixing the hash table keys with different prefixes for each hash table that we needed to create. For example we used '0' as the prefix for session ID, '1' for ip address etc.

6. Further Improvements:

Since we were basically concentrating on getting a simple prototype working, there is a lot that could be done to improve this package before we could reach our goal of making it an open-source version of CASP.

1. One of the things that I had initially discussed with Dr. Schulzrinne but could not work on because of time constraints was the Client layer protection using CMS (Cryptographic Message Syntax).
2. The messaging layer code could be changed from a single process poll based model to a multi-threaded model, which could more efficiently take care of incoming and outgoing requests.
3. For intra domain and corporate networks another useful feature to add would be Kerberos based authentication and key exchange.
4. A key component of CASP is the Authentication, Authorization and Accounting module which is currently being developed by Bannerjee. This would be integrated soon.
5. Right now the messaging layer initializes all the client layers by calling standard functions like QOS_init etc. In future we plan to add support for dynamic loading of client layers using modules for each client type.

7. Learning Outcome:

This project gave me an invaluable experience of working in a team. It made me realize the difficulties you come across in a real time development environment where everyone is working on his module independently until it comes time for integrating. The interfacing requirements were constantly changing and we had to keep everyone updated about the changes we were making. Dr. Schulzrinne was again very generous in promptly guiding us through various issues we had regarding CASP.

I also learned some of the ongoing programming practices from Dr. Schulzrinne. I learned how to use the OpenSSL library for providing security and also learned about the pitfalls it can cause if not used properly. Adding TLS/SSL security is not just about replacing the calls to "sockets API" with the calls to "OpenSSL API", but each application's unique requirements must be considered, and the best decision must be made for both security and functionality.

Being part of CASP mailing list also helped me in understanding how protocol issues are discussed.

8. Acknowledgements:

I am grateful to Dr. Kenneth Calvert and Dr. Henning Schulzrinne for giving me an opportunity to do my Master's project under their joint guidance. I appreciate the time and effort put in by both to discuss various issues and bring this project to a successful conclusion.

I would also like to thank Dr. Manivannan and Dr. Fei for serving on the faculty committee for my defense.

I would also like to thank all my teachers in the Computer Science department at the University of Kentucky who have trained me in the field of Computer Science.

9. References:

1. Design of CASP – a Technology Independent Lightweight Signaling Protocol, H. Schulzrinne, X. Fu, C. Pampu, C. Kappler
2. IETF draft draft-schulzrinne-nsis-casp-01.txt, H. Schulzrinne, H. Tschofenig, X. Fu, A. McDonald
3. Braden., R. Ed., et. al., "Resource ReSerVation Protocol (RSVP) -- Version1 Functional Specification", RFC 2205, September 1997.
4. IETF Next Steps in Signaling (NSIS) working group, http://www.ietf.org/html.charters/nsis-charter.html
5. The CASP development team at UK comprised of following people with their respective development areas.
    a. Jignesh Doshi – Part of M-Layer functionality
    b. Harsha Vardhan – Client QoS functionality
    c. Raju Manchala – Scout Discovery protocol
    d. Satish Ashok – Firewall Client
    e. D. Bannerjee – Authorization and Accounting
6. OpenSSL is a free (BSD-style license) implementation of SSL/TLS based on Eric Young's SSLeay package. Web site is at www.openssl.org
7. Unix Network Programming, Volume1, Richard Stevans
8. Network Security – Private Communication in a Public World, Charlie Kaufman, Radia Perlman, Mike Speciner.
9. SSL and TLS, Designing and Building Secure Systems, Eric Rescorla
10. The Pocket Guide to TCP/IP Sockets, Michael J. Donahoo and Kenneth L. Calvert
11. Applied Cryptography, Bruce Schneier
12. RFC 2246, "The TLS Protocol Version 1"
13. Network Security with OpenSSL, Pravir Chandra, Matt Messier, John Veiga