# CS297 Report

# Media Applets and Tunneling Applets

Xunyan Yang (xunyanyang@hotmail.com)

**Advisor:** Dr. Chris Pollett

## 1. Introduction

Presently, there are three major media formats: Windows Media format, Real Media format and QuickTime Media format. To play the media streams supported by different media formats in browser, we need to install plugin software such as Windows Media player, Real Time player and QuickTime player, etc. There are two problems of this approach. First, it's not convenient for users since they need to install different media plugin. Second, the security implementations of different player plugin software are different. There is no unified mechanism to ensure the security of downloading and playing media streams. The goal of this master's project is to develop a Java media applet and server which have the ability to play multiple media stream formats in a secure way. The Java media applet will work with server side software which implements Secure Sockets Layer (SSL) protocol to ensure the security. There are several challenges in this project which will be explored. One is that the Java's ability to play different media formats. The second difficulty is to set the media to streams on top of SSL.

The original proposal above was modified in some respects after the study of this semester. First, the structures of Real Media and Windows Media files were not available, so therewais no means to develop an RTSP server that can read frames from media files in Real Media and Windows Media formats. Adding a function to the RTSP server to convert all other media formats to that of known such as Microsoft's AVI or QuickTime's MOV might be a solution. Second, in the proposal above, the concept of tunneling applets is not explained in any way. The Applet here is in fact an SSL client that can not only play media streams but also provide an optional function to tunnel the decrypted media streams to the local media players. Through secure channels built by tunneling applets, the secure downloading is guaranteed.

This report is organized as follows. Section 2 mainly explores the working mechanism of the Java Media Framework. Section 3 introduces the Real Time Stream Protocol. Section 4 discusses media formats. Section 5 introduces the Secure Socket Layer Protocol. Section 6 summarizes some important part in the previous sections and Section 7 is a brief overview about of the CS297 study. All deliverables are discussed in corresponding sections.

## 2. Java Media Framework

Java Media Framework (JMF) [SUN98] is a Java API which provides a convenient way to deal with time-based media data. With JMF, programmers are able to add functions such as playing back, capturing live data and manipulating media streams to applications and applets. Many types of media can be processed and manipulated through JMF.

### 2.1 How Does JMF Relate to the CS298 Project?

Since Java is the target language of the CS298 project, its ability to play media streams is crucial.

### 2.2 JMF Working Mechanism

JMF uses four manager interfaces to control the behavior of capture, process and present time-based media. The interface `PlugInManager` is used to plugin new JMF player plug-ins. The interface `Manager` is used to produce `Player` object. `PackageManager` interface contains all the registered `Player` classes. The `CaptureDeviceManager` interface contains all the capture devices like voice recorder.

JMF-based programs must be informed when the state of the media system is changed. JMF uses the publisher-subscriber design pattern to implement the event handling structure. The event model of JMF is shown in Figure 1.1.

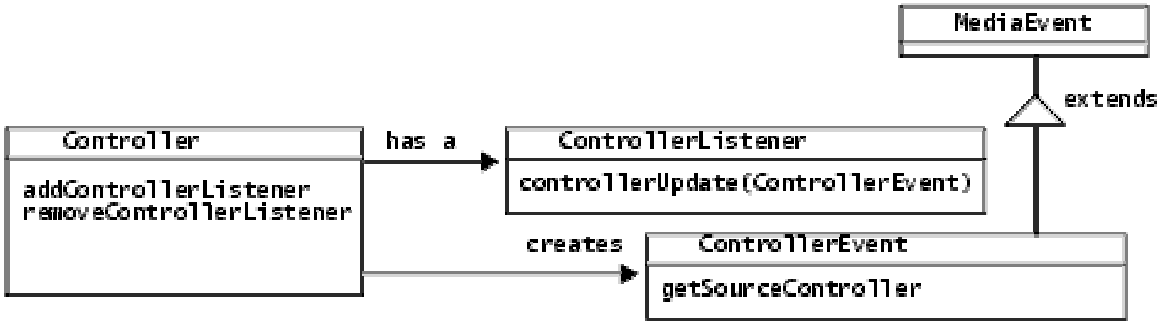


***Figure 1.1*** *JMF event model*
*Source: [SUN98]*

`Controller` objects are usually `Players` and `Processors`. Each `controller` maintains a list to contain all listeners that want to receive the `MediaEvent` posted by this controller. A `controllerListener` must have itself registered with the corresponding `Controller` objects. A variety of listener interfaces are defined in JMF. An appropriate listener interface must be implemented if a class wants to be notified when a `Controller` posts a `MediaEvent`. Once a `controllerListener` object receives an event, an update is done automatically.

`Players` and `Processors` are used to process and present the incoming media streams. They mainly work with the low-level hardware such as sound card and video card. Each `Player` or `Process` is only responsible for one particular type of `DataSource`. If a new `DataSource` needs to be processed, the old `player` or `processor` must be destroyed and a new one must be created. The only difference between a `Player` and `Processor` is that the `Processor` can output the processed media streams to a `DataSource` used for many other purposes.

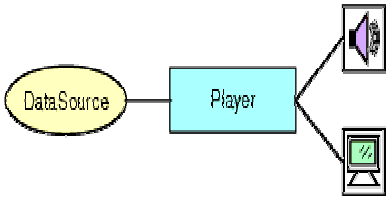


***Figure 1.2*** *JMF player model*
*Source: [SUN98]*

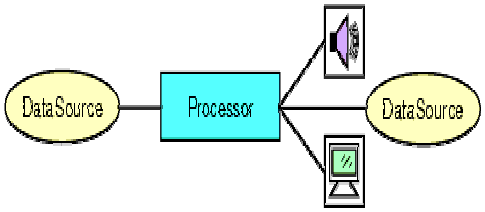


***Figure 1.3*** *JMF processor model*
*Source: [SUN98]*

A `Player` maintains a variable to indicate the working status of the `Player`. They are six states: unrealized, realizing, realized, prefetching, prefetched and started. Under normal situation, a `Player` will go through each state till the last state. Otherwise, an error message will be reported by the `Player`.

In order to set and query attributes of an object, JMF defines a `Control` interface. By implementing the corresponding `Controls` interface, a JMF object can provide an access to its `Control` objects. For example, if the `DataSink` class implements the `StreamWriterControl` interface, by calling the `getStramWriterComponent` method in `DataSink` class, users can get the interface component to control the `streamWriter`.

The detailed information about the format of data stream is contained in the `Format` class. The Format class has two subclasses: `autoformat` and `videoformat`.

### 2.3 Deliverable 1: an Experiment with JMF Playing Functionality

Deliverable 1 is a simple Java program to experiment with functionalities of the JMF API. JMF, as mentioned before, provides ability of adding audio and video to Java applications and applets. JMF has build-in features to integrate media with Java Swing Components. In this program, the class Manager is used to create an instance of the Player interface which is created and used only for the given media file.

Figure 1.4 is a screenshot of the output of deliverable 1:


*Figure 1.4 A screenshot* of Deliverable 1

# 3. Real Time Stream Protocol

## 3.1 RTSP Introduction

The Real-time Streaming Protocol (RTSP) [SRL98] is an application level protocol used to control the time-based media data such as audio and video. We may get a rough picture about what the RTSP acts as and how it works by thinking about a remote control panel for a DVD player. Suppose that we are sitting before a TV set and watching a DVD for Harry Potter. Once the DVD player is properly connected to a TV set, we can press the "play" button on the remote control panel to start our movie. We can stop the presentation anytime during the presentation by pressing the "pause" button and resume it by pressing the "play" button again. We can use the "forward" and the "backward" buttons to locate a new start point. Here, the DVD player acts as a multimedia server and the TV set acts as a client. The difference between this example and the real RTSP client-server application is that, instead of using a separate remote control panel to send requests to the DVD player, the real RTSP client send requests directly to the RTSP server. If a client application implements RTSP, it is provided a "network remote control" [SRL98] for RTSP servers and therefore, a control for the real-time media presentation.

RTSP is a standard that both RTSP clients and servers must follow during their communication. In syntax and operation, RTSP is similar to the Hypertext transfer Protocol (HTTP). This similarity is made for easy extending the RTSP as the HTTP is extended time after time. There are two significantly different aspects between the RTSP and HTTP. First, both the RTSP server and client must maintain a state in order to correlate an RTSP request to a stream. For example, if the current state of an RTSP session is "playing", a request of the type PAUSE or TEARDOWN are executable by the RTSP server. However, a request of the type PLAY will carry no meaning to the RTSP server. Usually there are four states: init, ready, playing and recording. Table 2.1 and 2.2 interpret the meaning of each state. Second, with the HTTP, only clients can issue requests to servers, not vice versa. With RTSP, requests can be issued by either side.

| State | Condition |
|---|---|
| *Init* | Waiting for the reply from the server after the request SETUP has been sent. |
| *Ready* | Received the SETUP or PAUSE reply from the server. |
| *Playing* | Received the PLAY reply from the server. |
| *Recording* | Received the RECORD reply from the server. |

***Table 2.1*** *States of an RTSP client*

| State | Condition |
|---|---|
| *Init* | Waiting for the SETUP request from the client. |
| *Ready* | Received the SETUP or PAUSE request from the client. |
| *Playing* | Received the PLAY request from the server. |
| *Recording* | Received the RECORD request from the client. |

***Table 2.2*** *States of an RTSP server*

### 3.2 RTSP in the CS298 Project

In the final project, both the server-side and client-side programs will implement the RTSP.

### 3.3 Deliverable 2: a RTSP Server

Deliverable 2 was originally planned to be a server-side program which implements RTSP. Due to some practical difficulties, this goal was not achieved. Only a few types of RTSP requests worked correctly in my RTSP server. These requests are PREDESCRIBE and SETUP. At this stage of study, due to the lack in the knowledge of encoding and decoding, a ready-for-use RTSP client, JMFStudio, was used to communicate with my RTSP server for the debugging purpose. Although JMFStudio is an open-source application written in Java, due to the limited time, I was not able to make another three important requests, PLAY, PAUSE and TEARDOWN, go through a full RTSP session. An RTSP session is a complete transaction which typically consists of a sequent requests from a client. For example, Listing 3.1 is the content in the log file of JMFStudio which recorded both outgoing and incoming message of JMFStudio as an RTSP client.

```
## outgoing msg:
## DESCRIBE rtsp://localhost/test.avi RTSP/1.0
CSeq: 16
Accept: application/sdp
User-Agent: JMF RTSP Player Version 2.1.1e

## incoming msg:
## RTSP/1.0 200 OK
CSeq: null
Content-type: application/sdp
Content-length: 100

v=0
m=audio 3456 RTP/AVP 0
a=control:rtsp://localhost/test.avi
m=video 2232 RTP/AVP 31
```

```
a=control
!! [SDP Parser] Token missing: o=
!! [SDP Parser] Token missing: s=
!! [SDP Parser] Token missing: t=
!! [SDP Parser] Token missing: c=
!! [SDP Parser] Token missing: c=
## incoming msg:
## :rtsp://localhost/test.vid

## outgoing msg:
## SETUP rtsp://localhost/test.avi RTSP/1.0
CSeq: 17
Transport: RTP/AVP;unicast;client_port=61870-61871
User-Agent: JMF RTSP Player Version 2.1.1e

## incoming msg:
## RTSP/1.0 200 OK
CSeq: 17
Session: 10001
Transport: RTP/AVP/UDP;unicast;client_port=39498-39499;server_port=5000-
5001
```

***Listing 3.1*** *Comunication records*

# 4. Media Formats

## 4.1 AVI Format Introduction

AVI (Audio Video Interleave) [MJ96] is an audio/video format defined by Microsoft. It is a special form of RIFF. RIFF [MC98] stands for Resource Interchange File Format which is a general purpose format defined by Microsoft and IBM for exchanging their media data types. To understand the format of AVI, it's necessary to understand the building blocks and structure of RIFF files. There are three types of storage units from which RIFF files are built: RIFF form header, chunk and list. Listing 4.1 is showing these structures.

```
(1) RIFF Form Header

    'RIFF' (4 byte file size) 'xxxx' (data)

        Where 'xxxx' identifies the specialization (or form) of RIFF.
        'AVI ' for AVI files.

        Where the data is the rest of the file.  The data is comprised of
        chunks and lists.  Chunks and lists are defined immediately below.

(2) A Chunk

    (4 byte identifier) (4 byte chunk size) (data)

        The 4 byte identifier is a human readable sequence of four
        characters such as 'JUNK' or 'idx1'
```

```
(3) A List

    'LIST' (4 byte list size) (4 byte list identifier) (data)

       Where the 4 byte identifier is a human readable sequence of four
       characters such as 'rec ' or 'movi'

       Where the data is comprised of LISTS or CHUNKS.
```

***Listing 4.1*** *Storage units of RIFF files*
*Source: [MJ96]*

The general structure of RIFF files is described in Listing 4.2.

```
RIFF Form Header (List|Chunk);
List(List|Chunk);
```

***Listing 4.2*** *The structure of RIFF files*

An AVI file has an RIFF form header with the identifier "AVI ". The data part of the RIFF form header consists of two lists, a `hdrl` list and a `movi` list, and an optional `idx1` chunk as shown in Listing 4.3. The `Hdrl` list contains an AVI header chunk, and two lists that represent audio and video stream headers and stream formats. The `movi` list is where the actual audio and video data are stored. The media data can be contained in `rec` lists or chucks. Usually, an AVI file that contains both audio and video data uses `rec` lists. An audio or uncompressed video AVI file may use only chunks. A `rec` list contains a single frame. In a real AVI file, the `##` in the identifier of `##wb`, `##dc` and `##db` will be replaced by the stream number. For example, if an AVI file has two streams; 00 represents the video stream and 01 represents the audio stream, `##wb` will be `01wb`, `##dc` will be `00dc` and `##db` will be `00db`.

```
'RIFF' (4 byte file length) 'AVI '
      --- 'LIST' (4 byte list length) 'hdrl'
            --- 'avih' (4 byte chunk size) (data)
            --- 'LIST' (4 byte list length) 'strl'
                  --- 'strh' (4 byte chunk size) (data)
                  --- 'strf' (4 byte chunk size) (data)
            --- 'LIST' (4 byte list length) 'strl'
                  --- 'strh' (4 byte chunk size) (data)
                  --- 'strf' (4 byte chunk size) (data)
      --- 'LIST' (4 byte list length) 'movi`
            --- 'LIST' (4 byte list length) 'rec '
                  --- '##wb' (4 byte chunk size) (data)
                  --- '##dc' (4 byte chunk size) (data)
                  --- '##db' (4 byte chunk size) (data)
            --- '##wb' (4 byte chunk size) (data)
            --- '##dc' (4 byte chunk size) (data)
            --- '##db' (4 byte chunk size) (data)
      --- 'idx1' (4 byte chunk size) (index data)
```

***Listing 4.3*** *The structure of AVI files*

The information contained in the `hdrl` list is crucial for programs to process an AVI file. In the final project, the RTSP server need to know how many frames and streams in this AVI file, which one is the initial frame, and what is maximum size among these frames, etc. The `avih` chunk (AVI header) usually answers these questions. Other important information about the audio and video streams, such as what codec is used for data compression and how many frames played per second, can be found in `strl` lists. In a word, the `avih` chunk specifies the properties of the AVI file and the `strl` list specifies the stream-specific properties. One stream has a corresponding `strl` list.

The limitation of AVI files is that there is no time stamp or any other way to name the frames. Audio and video data are organized in time sequence, so if a frame wrapped in a package is missing in the networking, there is no way for a server to resend that package since no one knows which frame it is.

### 4.2 Microsoft Windows Media Format

Windows media files are stored with different file name extensions. The extension of a file can be used to identify the content and purpose of this file. The most common Windows media files are these with file name extensions WMV, WMA and ASF. During my studying, no documentation that specifies the formats of these Windows media file types is found. It is said that the WMV and WMA are indeed stored in ASF 1.0, but so far I have not found any convincing evidence (either from the Microsoft's official documentation or from the raw media files) showing that this is true. Since Microsoft has the ASF [MICROSOFT03] specification released on its website, it is possibly helpful for my further study about Windows media formats, it is worth spending some time to talk about it.

ASF stands for Advanced System Format and is designed by Microsoft. The design goals of ASF as stated in the previously mentioned document were:

- To support efficient playback from digital media servers, HTTP servers, and local storage devices.
- To support scalable digital media types such as audio and video.
- To permit a single digital media composition to be presented over a wide range of bandwidths.
- To allow authoring control over digital media stream relationships, especially in constrained-bandwidth scenarios.
- To be independent of any particular digital media composition system, computer operating system, or data communications protocol.

The given circumstance determines features of ASF. In turn, features of ASF determine that the ASF is an ideal file format for the digital media presentation over varied types of networks. Like AVI files, the ASF media file structure has tree parts: header part, data part (one or more media streams) and index part. However, ASF files are constructed from a different type of building blocks called the ASF object. Recall that in AVI files, the building blocks are chunks and lists. The structure of an ASF object is shown in Figure 4.1:

| Object GUID (16 bytes) |
|---|
| Object Size (8 bytes) |
| Object Data (N >= 0) |

***Figure 4.1*** *The structure of an ASF object*
*Source: [MICROSOFT03]*

There are many types of ASF objects but only three are called top-level objects. These three objects are: the header object, the data object and the index object. Calling them top-level objects is because they are the out most layer of an ASF file; no other object can contain them.

Every ASF file has exactly one header object, one data object and zero or more index object(s). The header object must be at the beginning of the file and followed by the data object. If there is an index object, it must be placed after the data object.

Among all objects of an ASF file, the only one that can contain other objects is the header object. All the important information that is needed to interpret the content in the data object is included in the header object. Table 4.1 lists most common objects that are contained in a header object. Of these objects, a file Properties object, a header extension object and a stream properties object (at least one) must not be absent in the header object. Otherwise, this header object is invalid.

| Object Name | Functions |
|---|---|
| File Properties Object | Contains globe file attributes. |
| Stream Properties Object | Defines a digital media stream and its characteristics. |
| Header Extension Object | Allows extending the ASF file with backward compatibility. |
| Content Description Object | Contains bibliographic information. |
| Script Command Object | Contains commands executed on the playback timeline. |
| Marker Object | Marks named jump points in the ASF file. |

***Table 4.1*** *Objects inside a header object*

The Data Object is where all media data of an ASF file resides. Inside the Data Object, the data is stored in a unit called the ASF Data Packet. Data for one or more streams (interleaved data) can be stored in a Data Packet. A Data Object usually contains many Data Packets and each packet is with a fixed length. Data Packets are stored based on their send times. Unlike AVI files, adding information about the presentation time stamp for each Data Packet is an option in ASF files.

# 5. The Secure Sockets Layer Protocol

## 5.1 SSL Introduction

SSL [NETSCAPE98], short for Secure Sockets Layer protocol, was developed by Netscape Communications for Internet security. As illustrated in Figure 5.1, SSL is a network level protocol and runs on the top of TCP/IP.

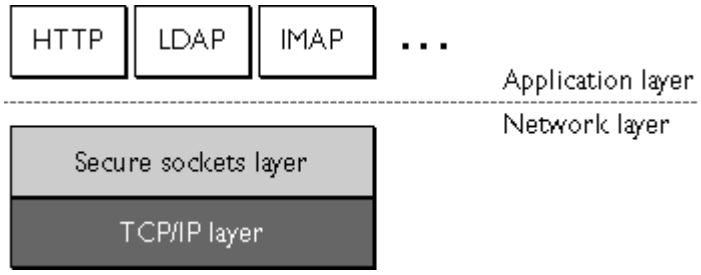


***Figure 5.1*** *The SSL layer*
*Source: [NETSCAPE98]*

With the SSL protocol, a private and authenticated communication based on TCP/IP connection can be achieved.

An SSL session always starts with an SSL handshake. The SSL handshake is actually a short but complicated communication between a client and server. During this communication, the client and server make an agreement on the level of security and exchange the information needed in the SSL session, such as the cipher settings and SSL version number. The SSL protocol provides both SSL client and server authentications. The side which is required the authentication must send its certificate to the other. A certificate is used to authenticate the identity of a client or server. It's usually issued by a trusted certificate agent (CA). A certificate is just like your driver's license and a CA functions as the DMV.

After a successful handshake, an encrypted SSL connection has been established and the transfer of real data strings begins. Encryption techniques are used based on the pre-selected cryptographic algorithms.

## 5.2 SSL in the CS298 Project

In the final project, both the RTSP server and client are SSL-enabled. The SSL applet is not only a media player but also has a function to tunnel decrypted media streams to the local media players of the client [NETSCAPE97]. The working mechanism is illustrated in Figure 5.2.

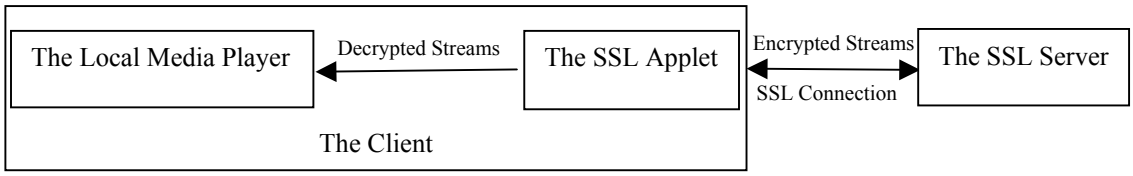


***Figure 5.2*** *The SSL tunneling applet*

### 5.3 Deliverable 3: a Secure Tunneling Program

Deliverable 3 is a simple application that is written on top of JSSE (Java Secure Socket Extension) [SUN03] API. The goal of Deliverable 3 is to experiment with the functionalities that provided by the JSSE and get ready to build a SSL applet and server in the final project.

Deliverable 3 is designed to simulate the working model of the final project, so it includes three programs: a regular client, a SSL client and a SSL server. The regular client acts as a local media player; the SSL client plays the role of the media/tunneling applet and the SSL server works as the SSL/RTSP streaming server.

In this application, the regular client can only request a connection to the SSL server through the SSL client. The SSL client opens an SSL connection based on the host name and the port number provided by the regular client. The communication between the SSL client and server is in a secure way: traveling data in both directions is encrypted. Once the SSL connection is established, the SSL client encrypted data it received from the regular client and send it to the server and decrypted data it received from the SSL server and send it to the regular client.

## 6. Summary

Experimenting with the JMF's functionalities was very helpful in writing the server-side program. The server-side program takes advantages since it can use any API provided by Java. By directly using the functionality provided by a variety of JMF classes, we can shorten the time needed to build the server.

The client-side program in the final project will not be allowed to use JMF since we do not suppose that client machines have JMF library preinstalled. But this does not mean JMF is not worth studying for the client's sake. In fact, JMF provides an application example called JMFStudio, which is not only a local media player but also an RTSP client. Moreover, the source code of JMFStudio is freely available. By looking inside the source code, the full procedure of building an RTSP session is disclosed. The answer for the question of how the JMFStudio decompresses the media streams sent by the server is possibly obtained by studying the source code.

The client-side program, a media applet, in fact has a build-in mini media player which is not build on the top of JMF, so a self-developed architecture for this player will be desired in the final project. Studying the JMF's working mechanism helps achieve this goal.

Since the RTSP will be implemented in both sides of the final project, fully understanding contents of the RTSP is very important. After working with deliverable 2, some hands-on experiences are gained. Also, problems that may arise in the final project have been predicted.

## 7. Challenges and Overview

There were two main challenges encountered during the CS297 study:

- Trying to find out how the JMF deals with different media file formats.
- Trying to let my RTSP server talking with JMFStudio.

In the final project, some problems will disappear and some will remain. Problems that involved communications between the RTSP client and server will become easier since I will develop programs for both sides. The problems related to interpreting media files will need further study. More research on how to implement a media player needs to be done in CS298.

## Reference


[MC98] RIFF/WAV File Specification. http://www.music-center.com.br/spec_rif.htm. 1998.

[MICROSOFT03] Advanced Systems Format (ASF) Specification. http://www.microsoft.com/windows/windowsmedia/format/asfspec.aspx. Microsoft. 2003.

[MJ96] AVI Overview, John F. McGowan. http://www.jmcgowan.com/avi.html. 1996.

[NETSCAPE97] SSL Tunneling and the Proxy. http://developer.netscape.come/docs/manuals/proxy/ProxyUnx/SSL-TUNL>HTM. Netscape Communications. 1997.

[NETSCAPE98] Introduction to SSL. http://developer.netscape.com/docs/manuals/security/sslin/contents.htm. Netscape Communications. 1998.

[SRL98] Real Time Streaming Protocol (RTSP). H. Schulzrinne, A. Rao, R. Lanphier http://www.cs.columbia.edu/~hgs/rtsp/draft/draft-ietf-mmusic-rtsp-09.pdf. Columbia Univ./Netscape/RealNetworks. 1998.

[SUN98] Understanding JMF. http://java.sun.com/products/java-media/jmf/2.1.1/guide/JMF Architecture.html. Sum Microsystems. 1998.

[SUN03] Java Secure Socket Extension (JSSE). http://java.sun.com/products/jsse. 2003.