Distributed Audio with GNU/Linux

by Daniel R. Warren

Abstract

This paper describes my implementation of *alsa_client* and *alsad*, client and server applications written for the GNU/Linux operating system, which give users the ability to create audio streams with user-defined sources and sinks. The current implementation allows the user to create sources and sinks that are sound-card hardware interfaces or clients running *alsad*'s custom network protocol. *Alsad* transfers sound data from a source to all of the existing sinks. *Alsa_client* and *alsad* transfer uncompressed audio data over the network. As a result, these applications are most suited for use on local area networks.

Users of *alsad* and *alsa_client* can create an audio stream where the source is a file or a sound-card hardware interface, optionally stream the sound data to multiple *alsads* with additional sinks, and ultimately write the streamed audio data to a sound-card hardware interface or a file.

Preface

This project was inspired by my love of computers and music, the two dominant areas of study in my life for the past twenty years. I have always been a believer in technology and its application in everyday life, especially when the technology makes life easier and more enjoyable. As a result, I have always had several networked computers around the house to provide convenient Internet access and file sharing. It seemed only natural to give these technical marvels the ability to broadcast music throughout the house or function like an intercom. With this thought in mind, and no prior experience developing sound-related applications, I began the design and implementation of what is now *alsad* and *alsa_client*.

Overview

Figure 1 shows a scenario where a user has configured *alsad* and *alsa_client* to stream captured compact-disc data from the sound card on machine 1 to a file on machine 1, while also streaming the captured data to the sound card on machine 2. The sound card on machine 2 converts the data into an analog signal and sends it to an amplifier for

playback.

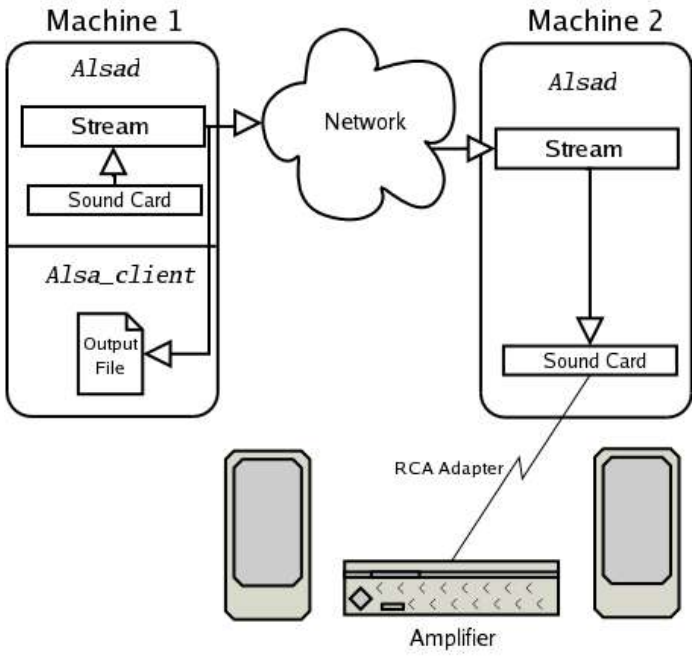

Fig. 1. An Example showing a potential use of *alsad* and *alsa_client*.

To create this scenario, a user must perform the following steps:

1. Start two instances of *alsad:* one on machine 1 and the other on machine 2.
2. Use *alsa_client*'s `add_stream` option to add streams configured for CD audio to both *alsad*s.
3. Use *alsa_client*'s `add_sink` option to create a sink for the output file on machine 1.
4. Use a mixer program to adjust sound card settings on machine 2 to the desired levels.
5. Use *alsa_client*'s `add_sink` option to create a sink for the sound card on machine 2.
6. Use *alsa_client*'s `add_src` option to create a source for the stream on machine 2. This source is the stream on machine 1 (the location of the source is `remote`).
7. Use a mixer program to configure the sound card on machine 1 so that it captures data from the compact disc player.
8. Use *alsa_client*'s `add_src` option to create a source from the sound card on machine 1.

Alsad allows connecting clients to add and delete streams, sources, and sinks. An **alsad stream**, known hereafter as a **stream**, is a chronologically ordered flow of sound data with a unique identifier, text description, and properties that define the type of sound data that flows through the stream. A stream may have a source and must have one or more sinks. *Alsad* places data received from a stream's source into a shared buffer for sinks to consume. As a result, all sinks consume the same data, but may be reading from different

parts of the shared buffer. A sink's position in the buffer depends upon the rate at which its destination asks for data.

When a user first adds a stream, it has no source and a sink that *alsad* automatically creates to discard the stream's data. Once a user creates a stream, he/she can then request that *alsad* add a source or additional sinks to this stream. If the user chooses to add a source first, the automatically generated sink removes the sources data as soon as it arrives. This action continues until the user adds a sink. If the user adds a sink to a stream before they add a source, the sink waits until the stream contains data to read. Once data is available, the sink reads from the buffer and sends the data to its defined destination. If a user requests that *alsad* remove a source, any data received by *alsad* is left for connected sinks to consume.

Sources and sinks for a stream can be sound-card hardware interfaces or clients running *alsad*'s custom network protocol. **Sound-card hardware interfaces** are the hardware input and output devices located on the sound card. They connect the sound card with external components that attach to the card. Microphones, headphones and low-level input and output devices (both in digital and analog format) use sound-card hardware interfaces to connect with the sound card. *Alsad* uses the Advanced Linux Sound Architecture (ALSA) libraries to configure and interact with the sound card when a stream has a source or a sink that is a sound-card hardware interface.

A user has three options when determining the rate sound data should flow though a stream. In the first option, *alsad* calculates the rate of flow based on user-defined stream properties that describe the real-time rate of the stream data. *Alsad* only allows sources to send data at this rate. All sinks are in turn forced to read at this rate or receive an underrun. With the second option, *alsad* allows a source to send data as fast as the slowest sink can consume it (if there are no sinks the source is allowed unlimited write permission). This strategy prevents any sinks from receiving an underrun. In the final option, *alsad* combines the last two strategies by choosing the fastest of: the real-time rate of option one and the slowest-sink rate of option two. This strategy gives a underrun to clients reading slower than the real-time rate and allows sinks to consume faster than real time if there are no real-time sinks.

Clients connected to *alsad* can also request stored information about existing streams, sources, and sinks. The client can obtain this information by making two different requests. The first request provides clients with a list of general information about defined streams on *alsad*. This information includes the stream's unique identifier, the textual description of the stream, and properties that define the sound data that passes through the stream. The second request allows clients that specify an existing stream to receive all relevant source and sink information. *Alsad* sends the unique identifier of the sources and sinks for the requested stream. If the source or sink is a sound-card hardware

interface, *alsad* also includes settings to configure the hardware. In addition to this information, *alsad* also sends address information for every connected client.

When a client adds a source or a sink that is not a sound-card hardware interface, it must also send a message containing the hostname of the machine on which it is executing. If the connecting client is an *alsad*, it sends the TCP port on which it accepts connections. Otherwise, the client sends 0 as the TCP port. Storing this information allows intelligent clients to query *alsad*s, using the queries described in the last paragraph, to find alternative locations for a desired stream. Finding an alternative location might be important in situations where clients are consuming a large percentage of an *alsad*'s network bandwidth. This lack of bandwidth may prevent clients from creating a sustainable sink on an *alsad*. If the desired stream has an existing sink that is an *alsad*, an intelligent client could try to connect with this alternative. If the alternative *alsad* is on a network segment with enough bandwidth to handle transporting the desired stream, the client can create a sink for this stream.

Introduction

Distributing audio data across a network is not a new concept. Today, there are several applications available that transport sound data over a network. RealNetworks, Inc. provides a streaming media application, RealPlayer, which is a popular example of these applications [1]. This application allows a users to listen to live audio broadcasts from a RealNetworks server. The server application is not readily available to the public because of its inflated price. This fact leaves most people without a means to create their own audio stream, thus limiting the usefulness of the client application. *Alsad* and *alsa_client* remove these limitations by allowing users to create both sources and sinks.

Background

Choosing an Operating System

Windows is the most common desktop operating system in the world. In order to allow the largest group of people to use my application, I would have developed for the Windows platform. I did not have much experience with Windows application development and had a general distaste for Microsoft and Microsoft's products. At this point in time, I had the most experience programming for Sun Solaris/Unix and GNU/Linux. Limited access to Solaris machines and two Linux machines at home made

the GNU/Linux platform the practical choice.

Sound and GNU/Linux

In years past, finding drivers for sound cards and applications that use the drivers was a headache. Hardware manufacturers did not want to release the specifications for their sound cards to the public, and there were few individuals with the time or resources to reverse-engineer drivers for all of the sound cards on the market. Two different groups responded by delivering quality drivers.

4Front Technologies was the first company to realize the benefits of developing sound-card drivers for Linux. They started the Open Sound System (OSS) project in the hopes of generating profit for their company by licensing their sound-card drivers. Eventually they released their basic drivers to the public, leaving the full-featured drivers under the licensing program. This release brought both joy and anger to the Linux community. People were happy that they could now listen to audio files on Linux but at the same time were irritated that they could not get the low-level drivers without paying a licensing fee. In an effort to meet the needs of the public, Jaroslav Kysela spearheaded the Advanced Linux Sound Architecture (ALSA) project. Their website shows the group's first posted comments on August 13,1998 [2].

Over the years, Jaroslav and a small group of programmers have developed quality full-featured drivers for virtually every major sound card. The drivers are so well crafted that they are now available in version 2.6 of the Linux kernel.

ALSA Libraries

Having completed my research, I decided to use the ALSA drivers. When I first started this project, the libraries were only available from the ALSA project website. The installation required downloading the kernel, lib and utils packages. These libraries are now integrated with the Linux kernel, so the user doesn't have to install them.

Alsad requires some adjustments to the sound-card mixer settings for normal operation. When *alsad* creates a source or sink that is a sound-card hardware interface, the user must insure that the interface in question is set to allow capture, not muted, and is at the desired volume. A user must manually modify these sound card settings with a mixer program. *Alsamixer*, a graphical console-based program that comes packaged with the ALSA libraries, is very helpful when a user needs to adjust the volume of the sound card from the console. The user may also use the default mixers of KDE and GNOME, because they come equipped with OSS and ALSA support. Using the OSS controls is also

acceptable because ALSA has an OSS compatibility layer. This layer allows applications that use OSS drivers to function normally.

Programming ALSA Applications

The documentation for ALSA is not extensive but does have a few sample pieces of code to get you started. I had hoped to make *alsad* flexible enough to allow multiple streams to play through the sound card simultaneously. After extensive research, I found that there is a plugin for ALSA that allows multiple processes to write to a specified device simultaneously. The ALSA libraries merge the sound data in software so that the two sources are played simultaneously. ALSA comes with a small command line program called *aplay* capable of playing back many different sound file formats. I used multiple instances of *aplay* to determine how well the software mixing plug-in operated. I was not impressed with my findings. The sound of the combined streams was not near the quality of the two played separately. There are two reasons why the ALSA engineers have probably not spent a great deal of time with this approach. First, mixing the streams efficiently, so as not to induce large amounts of latency in the audio pipeline, is nearly impossible. For real-time audio applications, low latency is critical. Second, most sound cards made today mix simultaneous streams via hardware. ALSA supports hardware-mixing sound cards, which are common in machines made in the last five years.

In the implementation of *alsad*, I chose to ignore the software-mixing plugin and let the sound card handle mixing connections. Making this decision means users with older sound cards won't have the option of sharing the sound card for simultaneous playback when running *alsad*.

Understanding the Basics

In order to understand the design of *alsad*, it is important to have an understanding of how sound data is stored, played back and captured on a computer. In most cases sound data is stored in files. The two primary formats are compressed and uncompressed data files. The compressed audio data is created by taking raw sound data from the sound card and reducing it in size with a compression algorithm. There are many different algorithms that can substantially reduce the size of sound files. This project does not use compressed data files. I leave further exploration of this topic to the reader.

Non-compressed audio formats are partially defined by the size and format of the samples they contain. A **sample** is a digital snapshot of the analog signal produced by the sound card. Larger sample sizes allow for a more accurate representation of the analog signal. The main drawback is increased storage size, which in this application, equates to more

data transfered over the network. In order to manipulate samples effectively in programs, samples are most commonly stored in signed and unsigned numeric data types. The size and type of data structure used for storing sound data is called the **sample type**.

These different samples are then compiled in chronological order into sound files. The rate at which these samples are pulled from the sound card is defined as the **sample rate**. This rate is normally measured in kilohertz, which is defined as thousands of samples per second. For example, compact-disc quality sound is recorded at 44.1kHz or 44,100 samples per second. The sound card produces a large amount of data when it is set to capture at this rate. This amount of data is multiplied if the sound file (like a compact disc) contains more than one audio channel. An **audio channel** is a digital representation of sound waves captured from one source in chronological order. An additional audio channel normally requires twice the number of bytes. This additional channel makes an hour of compact-disc quality sound data fill a 700MB compact disc.

Sound files stored on compact discs contain concurrent audio channels that have different sources. Most files have two audio channels, commonly defined as right and left. These audio channels were originally meant to replicate a stage layout accurately to a listener. For example, if sound samples are taken at the left and right side of a stage during a concert, upon playback through similarly placed speakers, the listener's brain will blend the two sounds together creating a feeling of immersion in the originally recorded environment. Movie makers try to enhance this feeling by capturing more audio channels. Most theaters support the playback of these extra audio channels to provide a more compelling movie experience.

In order to handle multiple audio channels in one file, it is necessary to come up with an ordering convention. The interleaved convention states that the samples are ordered in the file by the time of the sample, then by channel.

A two-channel interleaved file with A and B denoting single samples from each audio channel looks like this:

AB AB AB AB AB

Non-interleaved channels order the file by channel, then by time. A two-channel non-interleaved file with A and B denoting single samples from each audio channel looks like this:

A A A A B B B B

Most sound cards are not capable of capturing data in a non-interleaved format, so this storage strategy is not used in this project.

Configuring the Sound Card

A programmer needs to configure all of the audio-channel properties discussed in the previous section before using the sound card. The ALSA libraries have function calls that allow each of the parameters discussed to be separately configured. If the card is not capable of using a certain format or ordering convention, the function calls return with an error. These simple functions make the ALSA libraries very easy to use.

Each sound card has a circular memory buffer in hardware that holds sound data to be processed. The ALSA libraries provide a function that sets the amount of this buffer to use; this amount is defined as the **buffer size**. The function to set this parameter takes the number of microseconds worth of sound data that is required to fit in the buffer, the sample format, sample rate and number of audio channels. A larger buffer gives the program accessing the sound card more time between write or read calls. In addition to the buffer size, the period size on the sound card must be set. The **period size** is a standard data size sent to or received from the circular buffer on the sound card when a read or write command is issued. ALSA provides a function to set the period size based on the number of microseconds required to consume or produce this amount of sound data. It does seem a bit odd at first that both functions use time to set buffer size and period size. This design decision was made because the time allowed between read or write calls is required to be the same regardless of the sound format used. The programmer needs to ensure that read or write calls occur frequently enough to keep up with the rate of production/consumption of the sound card. If this goal is achieved, the sound card should not run out or have too much data to process.

Playing/Capturing Audio

The function call to open a connection to the sound card requires a character string specifying a sound-card hardware interface to open and an open mode. The mode determines whether the call to open the sound card is blocking or non-blocking. If the sound card is used by another process in a way that makes the device unable to be shared, the process will block until the resource is released. Programs using the non-block option will return immediately if the device is held by another process. The device handle returned by this function is used in other ALSA functions to communicate with the desired sound-card hardware interface. There are two sets of read and write function calls that use the device handle. One set handles interleaved ordering and another set handles non-interleaved ordering. After the the user configures the card and opens the sound-card hardware interface, capturing/playing sound data is as simple as calling the appropriate read/write functions with a specified device handle. This simplified interface makes the

ALSA libraries very easy to use.

Design and Implementation

Designing a large-scale application can be a challenge. Dividing the application into stand-alone components with generic interfaces is the key to success. Components allow programmers to focus work on one logical area of the application and prevent them from being overwhelmed by the complexity of the whole project. Generic interfaces make the components reusable in other applications. Keeping these principles in mind allows a programmer to create an application that is easy to use and maintain.

The ALSA developer team wrote the ALSA libraries in the C programming language. This development decision required that *alsad* be written in C or C++. I chose to develop in C because I had more experience with this language.

Helper Components

I developed the components that follow to provide the underlying framework for *alsad* and *alsa_client*. The libraries provide a programmer with tools to perform common tasks.

Socket Library

Many of today's applications require sending and receiving messages over a network. In order to make this task more programmer-friendly, I developed a component that provides an interface for the most common network functions. The component sends and receives data over network sockets using TCP in the transport layer. TCP is commonly used among network applications and provides the reliability and speed required for most applications.

One of the more common network tasks is creating a server socket that binds and listens for activity on a specified port. To accomplish this task a programmer uses the following function:

```
int create_serv_socket(int port, int queue_len);
```

The `port` parameter is the port to bind and listen on, and `queue_len` is the maximum

number of client connections queued while waiting to be accepted. By default this function binds the port for all local network addresses. If the function creates the socket successfully, the function returns the new socket descriptor; otherwise it returns -1.

A programmer can use the following function to accept client connections on the new server port:

```
int accept_connect(int sock);
```

This function uses the same algorithm as the `accept()` system call, but it has additional logging that records the connecting client. If the socket is accepted successfully, the function returns the new socket descriptor; otherwise it returns -1.

A programmer writing a client process also has helpful functions available for use in this component. To establish a TCP connection to a server listening on a specified port, a programmer can use the following function:

```
int create_connect(struct in_addr *ipaddr, short port);
```

`ipaddr` and `port` represent the IP address and port number of the server with which to establish a connection. If the connection is established, the function returns the new socket descriptor; otherwise it returns -1.

In order to send and receive data over an established TCP connection, it is important that the data be transfered in a manner that is safe for the calling process and ensures the transfer occurred. When a programmer uses the normal network `send()` function, the calling process is terminated if the process tries to send data over a socket that is closed. Problems can also occur when a programmer uses the normal network `recv()` function. This function causes the user to make multiple function calls or wait indefinitely. The following functions provide protection from these errors by blocking signals and creating timeouts:

```
int safe_sock_send(int sock,void *send_buff,
                   unsigned int send_len, unsigned int timeout);

int safe_sock_recv(int sock,void *recv_buff,
                   unsigned int recv_len, unsigned int timeout);
```

`sock` is the socket descriptor to send or receive data over. `send_buff` is a previously allocated buffer of length `send_len` that is filled with data to send over the socket. `recv_buff` is a previously allocated buffer of length `recv_len` that is filled on success with data to received over the socket. `timeout` is the maximum time in seconds that the send or receive functions will wait until data is sent or received. These functions return

-1 on failure or the number of bytes sent or received.

Configuration-File Library

It is often important for a program to input configuration data at runtime. This task is often achieved by using a configuration file. A function provided in this library allows a programmer to pull key-value pairs from a configuration file and put them into memory. Once they are in memory, a method in this library searches the list of key-value pairs by key to locate a value. If the calling application modifies the list, methods from this library can write the new key-value pair list to a configuration file. Most configuration files use the format `<key>=<value>`. This library provides the same basic format but is less flexible than other applications in that extra spaces, tabs, and new line characters will cause the key-value pairs to be loaded improperly. A programmer should strictly adhere to the following format when creating key-value pair files:

```
<beginning of file><key_1>=<value_1><new line character>
<key_2>=<value_2><new line character>
.
.
.
<key_n>=<value_n><end of file>
```

example file:

```
I=Tom
time of meeting=13:15:34
place_to_meet=Taco Bell
counter=1
```

Any string of characters (except strings than contain an equals sign) with a length greater or equal to 1 can be used as a key or value.

The following function parses a configuration file and puts the result in memory:

```
key_value_pair_t *config_file_read(char *filename,
                                   int *pairs_loaded);
```

`filename` is the full path to the configuration file. `pairs_loaded` is a pointer to a previously allocated integer that, upon successful return, holds the number of key-value pairs pulled from the configuration file. The function returns a pointer to the list of `key_value_pair_t` structures allocated by `config_file_read` or `NULL` on failure.

Once the key-value pairs are loaded into memory, a user performs a query by supplying a search key, and the method returns the value associated with the key. The following

function performs the query:

```
char *config_file_get_value(key_value_pair_t *pairs, char *key,
                            int pairs_loaded);
```

pairs is a pointer to a list of key_value_pair_t structures that is of length pairs_loaded. The function searches for key in the list pairs. If there are duplicate keys in the list, the function uses the first key reached in the file. The function returns a pointer to the value associated with the key. NULL is returned on failure.

As stated previously, it is also possible to create a configuration file from key-value pairs stored in memory. The following function performs this task:

```
int config_file_write(char *filename, key_value_pair_t *pairs,
                      int pairs_loaded);
```

As with the previous function, filename is the full path to the configuration file to write. If the file exists, then it is overwritten. pairs is a pointer to a list of key_value_pair_t structures. This list is of length pairs_loaded. The function returns 0 on success or -1 on failure.

The following function deallocates any memory used by the previous functions to store key-value pairs:

```
void config_file_destroy(key_value_pair_t *pairs, int
pairs_loaded);
```

pairs is a pointer to a list of key_value_pair_t structures that is of length pairs_loaded.

Authentication Library

The process of creating a daemon requires a developer to make many difficult design decisions. One of these decisions is the choice of authentication methods. Most daemons are required to use authentication because they provide data that should only be accessible to select users. There are two major categories of authentication, symmetric and asymmetric. Symmetric authentication bases authentication on a secret shared between client and server. Suppling a password to an FTP server is an example of this. Both the server and the client have to keep track of the client's password, which is the secret. Asymmetric authentication allows the server to authenticate the client without sharing secrets. Public/private key authentication uses this method.

Public/private key encryption is based on the principle that data encrypted with a client's public key can only be decrypted (in a reasonable amount of time) by that client's private key, and vice versa. This process also assumes that the private key is known by the client only.

In order to send a private message to a server, the client should encrypt the message with the server's public key, and then the server decrypts it with its private key.

To authenticate a client, a server sends it a challenge: a string of random bytes. The client then encrypts the challenge using its private key and sends the result back to the server. The server uses the client's public key to decrypt the message. If the resulting message matches the challenge; the client is authenticated.

Alsad provides an implementation of the process described above in the authentication library. The library is divided into two main functions, a server function and a client function. These functions are called by the client and server processes after a TCP connection has been established. The server function is defined as:

```
int auth_server(int sock, char *pub_key_dir);
```

`sock` is the socket descriptor used in communicating with the client process over the network. The `pub_key_dir` parameter is the path to the directory where all of the public keys for *alsad* are stored. The function returns 0 on success and -1 on failure.

Similarly a client process calls:

```
int auth_client(int sock, char *user_name, char *priv_key_file);
```

`sock` is the socket descriptor used in communicating with the server process over the network. The `user_name` parameter is the user name that will be used by the server to determine which public key file to use. `priv_key_file` is the path to the client process's private key file. The function returns 0 on success and -1 on failure.

For these two functions to work properly there is some initial setup that must occur. First, any client that the server needs to authenticate must have a public key file in `pub_key_dir` on the server that corresponds to the `user_name` supplied. The public key file on the server should be called:

`<user_name>.pub`

The naming convention for `priv_key_file` is not important because the parameter specifies the full path.

The public and private keys used in this library are read using the OpenSSL libraries. These libraries must be in place for the authentication library to compile. The library also expects the use of PEM format keys with no pass phrase. These keys are generated by typing the following commands at the shell:

```
openssl genrsa -out output_file 1024
```

This command creates a private key file `output_file` in PEM format that does not require a pass phrase and has a 1024-bit modulus.

In order to create the public key in PEM format, use the following command at the shell:

```
openssl rsa -in private_key_file -out output_file -pubout
```

This command creates the public key file `output_file` in PEM format from the private key found in `private_key_file`.

Logging Library

Over the years, I have come to realize that logging is an extremely important part of an application. Inevitably, a situation arises when not every piece of the process is configured properly to work with its environment. Log files are invaluable in situations of this nature. When a catastrophe occurs, the log gives you the information needed to avoid future problems. With major processes in Linux, it is helpful to have this information presented chronologically in the same file. In Linux, this functionality is provided by the syslog daemon (*syslogd*).

In the early stages of development it is also very common to log activities. Most developers use the occasional print statement that allows them to view the current progress or state of the application. Capturing all of the data that defines the state of the program and then printing it, can often be tedious.

There are also situations where a developer might want to print the log message to the screen and write to a file. To perform this task by hand would take twice as many lines of code.

To help solve both of these problems I created the logging library. This module makes logging messages easy while providing helpful information. The library uses a `#define` macro to make the function calls short.

log_lib.h defines the logging function as:

```
void logmessage(char *message, int unsigned location,
                char *process_name, char *log_file_name,
                char *filename, int line_num, char *function);
```

message is the message to be logged. location is where the message should be sent (any combination of LOGSYSLOG|LOGSTDOUT|LOGSTDERR|LOGFILE). process_name is the name of the calling process. log_file_name is the name and path of the log file the process should write to. filename is the file name that the call was generated from. line_num is the line number of the file that the call was generated from. function is the parent function that the call was generated from.

The **#define** macro in the program that is using the module:

```
#define LOGMESSAGE(string, location) logmessage(string, location,\
                    ALSAD_PROC_NAME, ALSAD_LOG_FILE, __FILE__,\
                   __LINE__, __FUNCTION__);
```

It is also necessary to #define ALSAD_PROC_NAME and ALSAD_LOG_FILE.

These two variables are the name of the calling process and the name and path of the log file the process should write to, respectively.

A call such as:

```
LOGMESSAGE("Hello World!", LOGSTDERR | LOGSYSLOG);
```

logs and formats 'Hello World!' along with the other function parameters to stderr and *syslogd*.

Circular-Buffer Library

The methods in this library allow a programmer to create an efficient buffer between one producer and many parallel consumers. The buffer is implemented by a circular memory region with three specialized functions. Two of the functions, the **serial-read** function and **serial-write** function, must acquire a separate reading or writing lock to execute. This requirement keeps read functions from executing simultaneously with other read functions and write functions from executing simultaneously with other write functions. The reads and write functions do not depend on one another, so their execution is permitted simultaneously. The third function is a read function that does not require any locking because it does not update any variables in the circular-buffer structure. This

parallel-read function can be called simultaneously by any number of threads.

A **overrun** occurs when a thread calls the serial-write function with not enough space available in the circular buffer. A buffer **underrun** occurs when a thread calls the serial or parallel-read function with not enough data in the circular buffer. The functions in this library come with the following guarantees. Any overruns or underruns in the circular buffer are guaranteed to be found by the serial read and write functions. The parallel-read function only guarantees finding underruns; the function might not detect overruns caused by the serial-write function.

The serial read and write functions adjust a current size parameter that is associated with each buffer. This parameter represents the difference between how much data has been written to the buffer and how much data has been read. The functions look at the current size parameter to control reads and writes so that unread data is not overwritten and there is no buffer underrun. The parallel-read function reads data from the buffer but does not modify the current size. This design decision means that the rate in which data passes through the buffer only depends on the rate of serial reads and writes.

Each circular buffer has a variable called the **global byte id**, which is incremented by one for every byte that is written by the serial-write function. This method gives an address, or byte id, to each byte written to the buffer. Each thread that calls the parallel-read function must store the byte index of the next byte to read from the buffer. The parallel-read function compares the next byte index to the current value of the global byte id. An overrun occurs if the difference between the two values is greater than the size of the buffer.

The parallel-read function guarantees the detection of an underrun as long as the global byte index stays below $2^{64} - 1$. *Alsad* would pass this defined limit after streaming approximately 3.3 million years of CD-quality stereo audio.

Before using the read and write calls provided with this structure, a thread must call the following function to initialize the circular buffer structure:

```
int circ_buff_init(circ_buff_t **new_buff, long buff_size);
```

`new_buff` is a double pointer of type `circ_buff_t`. This function allocates a buffer of size `buff_size` and initializes all of the variables and mutexes to prepare them for use by the other functions. The function returns 0 on success and -1 on failure.

Once the `circ_buff_t` structure has been initialized, data can be written to the circular buffer. The only way for a thread to write data to the circular buffer is to use the serial-write function discussed earlier. This function has the following interface:

```
long circ_buff_write(circ_buff_t *new_buff, void * write_data,
                     long bytes_to_write,
                     struct timespec *timeout);
```

In this function and all the functions to follow, `new_buff` is a pointer to an initialized `circ_buff_t` structure. This function writes the number of bytes specified in `bytes_to_write` from `write_data` to the circular buffer. To add data into the circular buffer the write pointer, current size and the global byte index are all incremented. Because these variables are shared between writing threads, this function blocks other threads that may also want to write. If the function is not able to lock the function by `timeout`, the function returns in error. `timeout` is defined as the amount of seconds and nanoseconds to wait until timeout. The following functions all rely on this argument because they must wait for a lock or data to read from the buffer.

This function also broadcasts a signal using a condition variable in an effort to inform other threads that unread data exists in the circular buffer. This function returns the number of bytes written to the circular buffer or -1 on failure.

Once a circular buffer contains data, a thread can read from the buffer by using the serial-read function. This function is almost identical to the serial-write function and as a result has a very similar interface.

```
long circ_buff_read(circ_buff_t *new_buff, void *read_data,
                    long bytes_to_read, struct timespec *timeout);
```

This function reads the number of bytes specified in `bytes_to_read` from the circular buffer and places them in the previously initialized buffer `read_data`. `read_data` must be at least `bytes_to_read` in size. This function decrements the current size variable in the circular-buffer structure. Calling this function is the only way to decrease the current size of the buffer. Because these variables are shared, it is necessary to block other threads calling `circ_buff_read`. If there is no data to read from the buffer when this function is called, the function will wait on a write condition variable until the `timeout` specified. This function returns the number of bytes read from the circular buffer or -1 on failure.

A user must initialize local thread variables before calling the parallel-read function. This initialization insures that the thread reads from the correct area of the circular buffer. To initialize these variables the following function must be called:

```
int circ_buff_init_thread(circ_buff_t *new_buff,
                          long *thread_read_pnt,
                          unsigned long *thread_byte_id,
```

```
                                struct timespec *timeout);
```

thread_read_pnt and thread_byte_id are pointers to local variables to keep track of the last read location from the buffer. They are initially set to point to the last byte written to the buffer. This initialization helps prevent an immediate underrun when a thread writes data to the circular buffer before the current thread calls the parallel-read function. The user must supply these parameters with each call to the parallel-read function. thread_read_pnt is a pointer to the location in the circular buffer to start reading from. thread_byte_id is the global byte index of the byte pointed to by thread_read_pnt. If this function is able to properly initialize these two variables it returns 0. The function returns -1 on failure.

The parallel-read function that was discussed earlier has the same interface as the serial-read function with the addition of two variables that keep track of where the thread should read data from in the circular buffer. This function has the following interface:

```
long circ_buff_read_thread(circ_buff_t *new_buff,
                           void* read_data,
                           long bytes_to_read,
                           long *thread_read_pnt,
                           unsigned long *thread_byte_id,
                           struct timespec *timeout);
```

read_data and bytes_to_read are defined exactly the same as they were in the serial-read function. thread_read_pnt and thread_byte_id are defined exactly the same as they are in circ_buff_init_thread. This function also waits for a signal in the write function if the buffer is empty. This function returns the number of bytes read from the circular buffer or -1 on failure.

It is also necessary for a thread that is trying to write to or read from the circular buffer to know when it can write or read data without constantly calling the serial write or read functions.

```
int circ_buff_wait_for_read(circ_buff_t *new_buff,
                            struct timespec *timeout);

int circ_buff_wait_for_write(circ_buff_t *new_buff,
                            struct timespec *timeout);
```

These functions solve this problem by waiting for the serial read and write condition variables belonging to new_buff. These functions return 0 on success and -1 when a timeout occurs.

When the user is finished with a circular buffer, they should use the following function to

deallocate memory used by the structure:

```
int circ_buff_destroy(circ_buff_t *new_buff);
```

This function returns 0 on success and -1 on failure.

Linked-List Library

Quite often a programmer needs to keep a list of objects for organizational purposes. This library provides common methods to operate on a generic list. The current methods allow inserting, removing, and searching for objects in the list. The list is not currently sorted but does allow the user to insert an unbounded number of objects. The time complexity of the search algorithms are therefore O(*n*), where *n* is the number of objects in the list.

Before calling any other functions in the library, the user must initialize the `linked_list_t` structure. The user can perform this task by calling the following function:

```
int linked_list_init(linked_list_t **new_linked_list,
                     size_t elements);
```

`new_linked_list` is a pointer that points to another pointer. This pointer will reference the newly created `linked_list_t` upon successful termination of the function. The initialization function will set the maximum number of objects that can be held in the list to `elements`. If `elements` is zero, then there is no limit on the number of objects that can be inserted. The function returns 0 on success and -1 on failure.

A programmer uses the following function in order to add an object to an initialized linked list:

```
int linked_list_insert(linked_list_t *new_linked_list,
                       int (generic_search_w_key(),
                       int *search_key,
                       void *data_to_add);
```

For this and the rest of the functions, `new_linked_list` is a pointer to an initialized `linked_list_t` structure. The function inserts the object pointed to by `data_to_add`. `generic_search_w_key` is a user defined function that compares `search_key` with the key located in the object `data_to_add`. This function operates on shared data and therefore must acquire a locked to insure that no other function in this library is being executed. Once the lock is obtained, the function compares the number of objects in the

list with the maximum that the user defined in `linked_list_init`. If the list is at its maximum size, the function returns -1. If the function successfully adds the object, it returns 0.

A programmer uses the following function in order to remove an object from an initialized linked list:

```
void * linked_list_remove(linked_list_t *new_linked_list,
                          int (generic_search_w_key(),
                          void *search_key);
```

This function is very similar to `linked_list_insert`. `generic_search_w_key` is a user defined function that compares `search_key` with the key located on the object in the list. This function operates on shared data and therefore must acquire a lock to insure that no other function in this library is being executed. The function returns -1 on error and 0 on success.

A programmer uses `linked_list_search` to find and retrieve an object that was inserted into a `linked_list_t` structure.

```
void *linked_list_search(linked_list_t *new_linked_list,
                         int (generic_search_w_key)(),
                         void *search_key);
```

`generic_search_w_key` is a user defined function that compares `search_key` with the key located on the object in the list. On success, this function returns a pointer to the object that matches the criteria provided in `generic_search_w_key`. If no matching objects can be found in the list `NULL` is returned.

A programmer can perform a user-defined operation on an object in the list by calling `linked_list_process_node`. This function keeps the programmer from removing an object from the list, performing some process on the object, and then inserting it back into the list.

```
int linked_list_process_node(linked_list_t *new_linked_list,
                             int (generic_search_w_key)(),
                             void *search_key,
                             int (generic_processing)(),
                             void *args);
```

This function is similar to `linked_list_search`. `generic_search_w_key` is a user-defined function that compares `search_key` with the key located on the object in the list. This function operates on shared data and therefore must acquire a locked to insure that

no other function in this library is being executed. generic_processing is the user-defined function that takes the desired object in the list and args as parameters. The function returns -1 on error and 0 on success.

A programmer uses linked_list_process_all to perform an operation on every object in a linked_list_t structure.

```
int linked_list_process_all(linked_list_t *new_linked_list,
                            int (generic_processing)(),
                            void *args);
```

This function is similar to linked_list_process_node. This function operates on shared data and therefore must acquire a locked to insure that no other function in this library is being executed. generic_processing is the user defined function that takes the desired object in the list and args as parameters. The function returns -1 on error and 0 on success.

A programmer uses linked_list_compare_all to compare all of the objects in a linked_list_t structure. The method compares the first two nodes of the list. Based on the return value of the generic_compare_nodes function, the method compares the first and third or the second and third nodes in the list. This process continues until the method processes all of the nodes in the list. A programmer can use this function to find the maximum or minimum value of data in the list.

```
void *linked_list_compare_all(linked_list_t *new_linked_list,
                              int (generic_compare_nodes)(),
                              void *args);
```

generic_compare_nodes needs to take three arguments. The first two arguments are pointers to the data stored inside the node. If the first node meets the desired criteria when comparing the two data samples, the user should write the function to return 0. Otherwise the function should return -1. If the developer requires any additional arguments, he can pass them in the third parameter, the args structure.

A programmer uses linked_list_destroy to deallocate all of the memory that linked_list_init allocated.

```
int linked_list_destroy(linked_list_t *list_to_destroy);
```

list_to_destroy is a pointer to the linked_list_t to be deallocated. If there are any objects remaining in the list they are not destroyed.

Application-Specific Libraries

The application-specific libraries that follow are designed specifically for *alsad*. They are meant to be useful for this program but are most likely not usable elsewhere. As a result, I will not define their public interfaces. All of the libraries described here are built from the libraries defined in the previous section. The purpose of the following section is to show the application of the Helper functions and describe the higher level functionality of the *alsad* process.

Communications Library

The Communications Library is a standardized method for transferring all application-specific structures across the network. *Alsad* needs six structures to transfer all of the data required for normal operation. Each structure has its own send and receive functions that use the socket library to transfer data with TCP. The socket library ensures that *alsad* transfers data on a successful send or receive. It also ensures that *alsad* will not terminate abnormally because of a send to a closed socket.

Alsad and *alsa_client* use the control structure most frequently. The **control structure** carries defined signals between the client and server. This structure most often precedes other structures in the communications library. The receiver of this communication structure has knowledge of all defined *alsad* structures and reads the required amount of data from the socket to fill the new structure. The communications library transfers all of *alsad*'s structures in a similar fashion. *Alsad* uses the control structure to define the type of service requested by the client and to send status messages back to the client.

Several required parameters must be defined in order to configure a sound card. *Alsad* gives full control to a client to define these parameters. These parameters require the development of a transport mechanism to share this information between client and server. The **hardware parameters structure** and the control structure provide this functionality. The structure also contains all of the information required to configure the sound card, open a device, and begin capture or playback.

As discussed before, *alsad* provides a mechanism for making streams available over a network. Streams contain a unique identifier, a textual description of the stream and other properties that define the stream. Any client creating or inquiring about this stream needs a mechanism to transport this information. In order to meet this need, the communications library provides send and receive functions that ensure the safe delivery of the **stream-information structure** between client and server.

When connecting with *alsad*, each client must report a network hostname or IP address and a port. This information is used to determine whether or not the attaching client is another *alsad* process capable of replicating the stream. A later section on *alsad* provides more details on why *alsad* needs this information. A programmer should use the **address information structure** and its associated send and receive functions to send this address information over a socket.

To request or send information related to a source or a sink, the client uses the **data-pipe structure** as a transport mechanism. The term "data pipe" is meant to encapsulate both source and a sink. Each pipe has a unique identifier to distinguish it from other pipes. The data-pipe structure is special in that it actually contains two previously mentioned structures: the address information structure and the hardware parameters structure. The addition of these two structures simplifies the implementation of the transmission functions by transferring the data all at once.

The final structure defined in the communications library is the audio-header structure. The **audio-header structure** contains a size variable that frames the sound data. This variable allows a receiving process to determine how much sound data follows the audio header structure. This method of transfer allows *alsad* to easily transfer sound data to/from a client.

Shared Library

The shared library contains functions that both *alsad* and *alsa_client* use to accomplish common tasks. The library currently contains functions that read the configuration files, read/write data from/to audio files, captures/sends sound data from/to the sound card, and performs standardized communication between *alsad* and *alsa_client*. In the current implementation, only *alsa_client* reads/writes data from/to audio files and only *alsad* captures/sends sound data from/to the sound card. Future development may allow both applications to use these functions.

Two functions in the shared library communicate with the sound card. One function captures data from the sound card, and the other plays back sound data using the sound card. Programs must call the sound card configuration function found in this library before calling these two functions. Both functions use the interlaced read and write functions provided by ALSA. When there is not enough data to read from the sound card's circular buffer or when data cannot be written to the sound card's circular buffer because it is full, the interlaced read and write functions return errors. The ALSA libraries provide a wait function that estimates when there will be enough data to read or

enough space to write. This implementation uses the wait function to avoid using too much of the CPU to check the status of the sound card.

There are two functions in the shared library that read and write sound data to files. These functions guarantee the requested reads and writes occur by performing the same type of checking that is performed in the socket library.

The third group of functions allow *alsad* and *alsa_client* to transfer structures between themselves. These functions provide a standardized way to move data between these two processes. The structures are transferred using the communications library developed for *alsad*.

Stream-List Library

The stream-list library is an unsorted linked-list structure that provides a common storage location for all streams available on *alsad*. Each node in the list represents a stream and contains every structure that has an association with a particular stream. Each node contains a circular buffer, a linked list for sources, a linked list for sinks, and a stream-information structure. The functions included in the stream-list library use the linked-list library functions that allow *alsad* to insert, delete, and search for nodes. There is also a specialized function created for the stream linked list that sends the stream-information structure for every stream over a defined socket. This function also uses some of the methods defined in the communications library.

Data-Pipe List Library

The data-pipe list library is almost identical to the stream-list library except for the fact that this list stores data-pipe structures instead of stream-information structures. The stream-list library uses this list to store sources and sinks for the stream. This library also uses the linked-list library to store the data-pipe structures and has a special function to transfer the entire list of data-pipe structures over the network.

The Applications

The ALSA Stream Daemon (*alsad*)

Alsad is a daemon that accepts connections over TCP on a port defined at run time.

Immediately after a client establishes a connection, *alsad* creates a worker thread to handle the client's requests and then continues to accept connections. First, the worker thread reads the configuration parameters from `alsad.conf` using the functions defined in the configuration-file library. Next the worker thread authenticates the client process using function calls from the authentication library. The worker thread then determines what type of service the client would like the server to perform by checking a control structure passed by the client. The client can request one of eight services:

Add a Stream
Delete a Stream
List Streams
Add a Source to a Stream
Delete a Source from a Stream
Add a Sink to a Stream
Delete a Sink from a Stream
List Sources and Sinks of a Stream

Upon receiving the request the thread then calls the appropriate function to handle the request. The thread terminates at the request of the client or upon completion of the task.

Add a Stream

Alsad uses the communications function from the shared library to request required configuration information from the client. This function verifies the data to make sure that the client has sent properly formatted structures. After the data verification, the function then initializes stream-information structure and creates a new circular buffer structure to hold audio data. *Alsad* then confirms that a stream with the the same unique identifier does not already exist in the global-stream list. The **global-stream list** is a linked list created from the stream-list library that contains all the streams for *alsad*. If the stream does not currently exist in the list, the stream-information structure and the circular-buffer structure are moved into a container node and then inserted into the list. In the final step, the function creates another worker thread that will monitor the amount of data in the circular buffer.

Alsad uses the monitor thread to grant permission to write to a portion of the circular buffer. The monitor thread grants permission to write data by calling the serial-read function. The serial-read function decreases the byte count of the circular buffer therefore allowing other threads to fill the empty space. Granting this permission indirectly allows sources (remote clients) to send more data to *alsad*. When creating a stream, the client can request one of three methods to determine the amount of write permission to grant.

The first method calls for the monitor thread to read data from the circular buffer at the rate defined in the stream-information structure. This rate represents real-time consumption of the audio data. The second method requires that the thread remove data from the circular buffer only when all of the sinks attached to the stream have read the data. To fulfill this requirement, the thread queries the stream's list of sinks to determine which sink thread has the lowest global byte index. The worker thread uses this query to determine the appropriate amount of data to remove from the circular buffer. The third method of granting write permission combines the rules defined in the first two methods. The monitor thread waits for all sink threads to read data from the buffer unless they are reading slower than real time. Threads reading slower than real time eventually receive an underrun error when reading from the circular buffer. These threads must call the `circ_buff_init_thread` function in the circular-buffer library to start reading from a valid point in the circular buffer. This action causes the reading thread to lose streamed audio data. Sink threads are allowed to read at any rate greater than or equal to real time. When there are no sink threads, *alsad* gives unlimited permission to write to the circular buffer.

Delete a Stream

A connecting client can request that *alsad* delete a previously created stream. First, *alsad* requests a stream-information structure from the client. This structure contains the unique identifier of stream the client wants deleted. If the stream is found in the global-stream list, *alsad* sets a closing flag on the stream structure. This raised flag causes all threads associated with the stream to terminate immediately. When all the threads have been terminated, *alsad* removes the stream-information from the list, destroys all unread data in the circular buffer, and sends the client a success message.

List Streams

Upon request, *alsad* sends every stream-information structure in the global-stream list to the client. *Alsad* signals the completion of the request by sending a success message.

Add a Source to a Stream

As with previous requests, *alsad* first asks the client for structures required to add the type of source requested. These structures differ depending on whether the client requests the source originate from the current socket, a remote socket (a remote *alsad* with this implementation), or a local sound-card hardware interface.

When the stream originates from the current socket, *alsad* requests an address information structure and a stream-information structure. *Alsad* uses the stream-information structure to determine to which stream to add the source. The address information structure allows *alsad* to store the address of the connected client. If the connecting client is not an *alsad*, the client sends 0 for the port. *Alsad*s send the port on which they receives connections. This rule allows other connecting clients that request a list of sources and sinks to determine if the sinks are *alsa_client*s or *alsad*s. If the sink is an *alsad*, the client can attempt to connect to the sink alsad to receive the same sound data. Clients can use this information to create a tree-structured peer-to-peer network to transport streamed sound data.

Alsad requires the client to send more information when adding sources that originate from remote hosts and sound-card hardware interfaces. In the case of the remote hosts, *alsad* needs an address-information structure and two stream-information structures. The address information structure is the hostname and port of the remote client *alsad* should connect with. The first stream-information structure identifies the local stream to add the source to and the other is sent to the remote *alsad* to identify the newly established sink. The remote *alsad* runs the same protocol as it would with *alsa_client*. The client, in this case *alsad*, requests that the data from the remote stream be sent over the current socket. When a client requests a source originating from a sound-card hardware interface, *alsad* requests a hardware information structure to locate and properly configure the sound card.

After this step, *alsad* checks for the existence of the stream to add the source to in the global-stream list. Once this step is complete, *alsad* adds the information gathered from the client into the source data-pipe list and prepares the sources for data transfer. If the source is a remote socket, *alsad* connects with the remote *alsad* and sends the required structures to establish a sink connection over the current socket. If the source is a sound-card hardware interface, *alsad* configures the sound card using the hardware-information structure.

After initialization of the the sources, *alsad* starts requesting data from the sources. The amount of data requested depends upon the permissions granted indirectly by the monitor thread. *Alsad* only asks for sound data when there is room in the circular-buffer structure. *Alsad* checks flags at defined intervals to insure that the stream and the source are not closing. If the source is not a sound-card hardware interface, clients can close the source by closing their end of the socket or by sending a close connection message to *alsad*. After the disconnection of the source, *alsad* removes the data pipe from the source data-pipe list.

Delete a Source from a Stream

A connecting client can request that *alsad* delete a previously created source. First, *alsad* requests a stream-information structure from the client. This structure contains the unique identifier of stream that has the source to be deleted. The client then must send a data-pipe structure to identify the source to be deleted. This extra request was established for future versions of *alsad* that might allow more than one source. If *alsad* finds the source in global-stream list, it sets a closing flag on the source structure. This flag causes the thread associated with the source to terminate. *Alsad* then removes the data-pipe structure and sends a success message to the client.

Add a Sink to a Stream

Fulfilling a request to add a sink is very similar to adding a source. Again, *alsad* requests the required structures from the client. Sinks can send data to any location that can be used as a source. Because of this fact, *alsad* uses the same protocol for transferring structures when adding sinks and sources. After gathering the required structures, *alsad* inserts the data into the sink data-pipe linked list. *Alsad* then connects with the sinks and prepares them for data transfer. After this task is complete, *alsad* calls the `circ_buff_init_thread` method on the circular buffer to prepare for reading.

When a sound-card hardware interface is the sink, *alsad* starts pulling data from the circular buffer and writing to the sound card without delay. In the case where the sink is a remote socket or a current socket, *alsad* waits for a request for data from the client. When these requests come, *alsad* sends the requested amount to the destination. If the circular buffer is empty, *alsad* sends a keep-alive message so the sinks won't drop their connection with the server.

If the sink is not a sound-card hardware interface, clients can remove the sink by closing their end of the socket or by sending a close-connection message to *alsad*. After the disconnection of the sink, *alsad* removes the data pipe from the sink data-pipe list.

Delete a Sink from a Stream

Alsad deletes a sink from a stream in the same fashion that it removes a source from the list. The only difference between the two requests is the list *alsad* queries. Each stream has a list to store sources and a list to store sinks.

List Sources and Sinks of a Streams

Fulfilling the request to list the sources and sinks is very similar to the request to list the

*stream*s. *Alsad* requests a stream-information structure to identify the stream to process and then iterates through the list of sources and sinks to send the data-pipe structures to the client. The control message preceding each of these data-pipe structures contains a flag to help the client distinguish between sources and sinks. On completion of this task, *alsad* sends a success message to the client.

Error Messages

When *alsad* cannot process a request, it returns an error code to the client. A developer can use the `alsad_strerror` function found in the shared library to give a text description of the error. Below is the list of errors and their meanings.

`ALSAD_ERR_MAX_PIPES`: *Alsad* cannot add another source or a sink to the stream because the pipe list contains the maximum number of elements.

`ALSAD_ERR_INSUF_INFO`: The client did not send enough information for *alsad* to perform the request.

`ALSAD_ERR_STREAM_FIND`: *Alsad* cannot find the stream in the global-stream list.

`ALSAD_ERR_STREAM_REM`: *Alsad* cannot remove the requested stream from the global-stream list.

`ALSAD_ERR_STREAM_INS`: *Alsad* cannot insert the stream into the global-stream list because a stream with an identical unique identifier already exists.

`ALSAD_ERR_INVLD_FRMT`: The audio format the client requested is not valid.

`ALSAD_ERR_INTERNAL`: *Alsad* has encountered a fatal error. An administrator should restart *Alsad* before further use.

`ALSAD_ERR_NO_PATH`: The client did not specify the path of the source or sink. The options are: `CURRENT, REMOTE, HW_IFACE`.

`ALSAD_ERR_PIPE_INS`: *Alsad* cannot insert another pipe because the pipe list contains the maximum number of elements.

`ALSAD_ERR_PIPE_REM`: *Alsad* cannot remove the pipe from the pipe list.

`ALSAD_ERR_PIPE_FIND`: *Alsad* cannot find the pipe in the pipe list.

`ALSAD_ERR_LOCAL_ADDR`: *Alsad* cannot retrieve its local address using the `getsockname()` function.

`ALSAD_ERR_RMTE_HOST`: *Alsad* cannot connect to the host that the client specified.

`ALSAD_ERR_IFACE_CFG`: *Alsad* cannot configure the sound-card hardware interface the client supplied. This error represents a misconfiguration of the hardware-parameters structure.

`ALSAD_ERR_RECV_CTRL`: *Alsad* was unable to receive a control structure. This error occurs when a client doesn't send keep-alive messages or sound data before the socket timeout occurs.

`ALSAD_ERR_INVLD_CTRL`: The client sent an invalid control message to *alsad*.

`ALSAD_ERR_RECV_AUDIOHD`: *Alsad* could not receive the audio-header structure.

`ALSAD_ERR_MAX_TRANS`: The client exceeded the requested or maximum transmission size.

`ALSAD_ERR_FRAME_DIV`: The client sent audio data that was not divisible by the frame size defined by the stream.

`ALSAD_ERR_RECV_AUDIO`: The client sent audio data less than the amount specified in the audio-header structure.

`ALSAD_ERR_PIPE_CLOSE`: *Alsad* closed the client's pipe.

`ALSAD_ERR_STREAM_CLOSE`: *Alsad* closed the client's stream.

`ALSAD_ERR_SHUTDOWN`: *Alsad* received a shutdown message and is closing all connections.

`ALSAD_ERR_SRC_PERM`: The client did not specify which source-permission strategy to use for the stream.

Configuration File

We have mentioned that *alsad* reads from a configuration file directly after the creation of a new worker thread. *Alsad* uses the configuration-file library to read this file. *Alsad* uses the following keys and default values for its configuration file, `alsad.conf`:

```
username=alsad
private_key_file=~/.ssh/alsad.priv
public_key_dir=/etc/alsad/pub/
```

```
max_stream_sinks=0
max_circ_buff_size=4194304
```

The user does not have to include any of these keys. If the keys do not exist, *alsad* will use the default values shown.

The `username` value is the user name to give another *alsad* for authentication.

The `private_key_file` value is the path to the private key that decrypts encrypted messages received by another *alsad*.

The `public_key_dir` value is the path to the directory that contains all of the public keys that are needed for the application.

The `max_stream_sinks` value is the maximum number of sinks that can connect to a single *stream*.

The `max_circ_buff_size` value is the maximum size to create for a stream's circular buffer.

The ALSA Client Application (*alsa_client*)

Alsa_client is a console-based application that adheres to the client protocol defined by *alsad*. *Alsa_client* allows the following options at the command line:

`-h, --help` Prints help to screen
`-v, --version` Prints the version to the screen
`-V, --verbose` Increases printed output
`-s, --server <server>` Hostname/IP address of *alsad* to connect with
`-p, --server_port <server port>` Port of the *alsad* to connect with
`-c, --command <command>` Command that defines the type of connection with *alsad*
`-C, --channels <channels>` Number of channels in the raw sound data
`-w, --hw_iface <hw_iface>` Name of the hardware interface to use
`-d, --data_pipe_id <data_pipe_id>` The unique identifier of the local data pipe
`-D, --remote_data_pipe_id <remote_data_pipe_id>` The unique identifier of the remote data pipe
`-l, --pipe_location <pipe_location>` The location for a source or sink
`-f, --file <file>` Full path of a file to playback or record to
`-F, --config_file <config_file>` Full path of a configuration file to use
`-r, --rate <rate>` Rate of capture or playback for a stream
`-i, --stream_id <stream_id>` The unique identifier of the local stream

`-I, --remote_stream <remote_stream_id>` The unique identifier of the remote stream

`-t, --text_desc <text_desc>` Textual description of the stream

`-W, --src_permission <src_permission>` Source permission strategy used by *alsad*

`-S, --remote_server <remote_server>` Remote server hostname/IP address for *alsad* to connect with

`-P, --remote_port <remote_port>` Remote server port for *alsad* to connect with

Alsa_client accepts the following values for the `<command>` parameter:

```
add_stream
del_stream
add_src
del_src
add_sink
del_sink
list_stream
list_pipe
```

Valid values for the `<pipe_location>` parameter:

```
hw_iface
current
remote
```

Valid values for the `<src_permission>` parameter:

```
calc_rate
sink_rate
calc_sink
```

The environment variable `ALSAD_SOUND` can be set to define the `<server>` and `<port>`.

```
ALSAD_SOUND=<server>:<port>; export ALSAD_SOUND
```

The environment variable does not take precedence over the command line argument.

Configuration File

Alsa_client uses the configuration-file library to read its configuration file. *Alsa_client* uses the following keys and default values for its configuration file,

`alsa_client.conf`:

```
username=alsad
private_key_file=~/.ssh/alsad.priv
pcm_format=S16_LE
number_pcm_channels=1
card_open_mode=0
circ_buff_size=1048576
card_buffer_time=500000
card_period_time=125000
card_device_name=default
alsad_port=5001
```

If the configuration file does not contain a parameter, the default value shown will be used.

The `username` value is the user name to give *alsad* when authenticating.

The `private_key_file` value is the path to the private key that decrypts encrypted messages received by *alsad*.

The `pcm_format` value is the sample type *alsa_client* will use. The list of valid sample types is available from *aplay*.

The `number_pcm_channels` value is the number of audio channels *alsa_client* will use.

The `card_open_mode` value is the open mode to use when opening the sound card. The default mode is a blocking call that waits for the device to be freed (`card_open_mode=0`). If card_open_mode is equal to 1, *alsad* will return immediately from the open call if it is in use.

The `circ_buff_size` value is a suggestion for the size for the stream's circular buffer size in bytes. *Alsad* looks compares this value to the maximum value found in its configuration file. It uses the lesser of these two numbers. The user should base the value of this key on how much delay should be present in the stream. Sinks will not start to empty data from the circular buffer until the buffer is half full. This means that the delay between source and sink is equal to (`circ_buff_size` / 2) / (sample size * 8 * channels * samples per second).

The `card_buffer_time` value is the number of milliseconds of sound data the circular buffer on the sound card should hold. This value is used to determine the sound-card buffer size. If the sound card does not support the supplied value, *alsad* uses the closest buffer time available.

The `card_period_time` value is the number of milliseconds of sound data a period of the circular buffer on the sound card should hold. This value is used to determine the period size. If the sound card does not support the supplied value, *alsad* uses the closest period time available.

The `card_device_name` value is the name of the sound-card hardware interface that *alsad* uses to capture data from or send data to.

Conclusions

It became clear in the early stages of the project that this application would not operate effectively over wide-area networks. The data transfer rate required by one compact-disc quality stream of music is approximately 1.35 Mb/s (rate * sample size in bits * number of channels). This transfer rate is too high for common Internet connections and borderline for 802.11b wireless-access points. For high-speed local area networks (LANs) this application does provide some useful features.

In a few experiments, I used *alsad* as a mechanism to transport sound while displaying DVD movies. This experiment resulted in high quality sound almost a second behind the video displayed. This lag in sound produced a Kung-Fu overdub effect. The reason for the delay is the number of sound buffers that reside in the data path. Each sample has to go through both the *alsad* buffer and the buffer on the sound card. This path creates a great deal of latency in the audio pipeline.

Overall, *alsad* provides an effective way of transferring high-quality sound samples over the network when timeliness is not a factor.

Future Work

The most important improvement that can be made to this application is the addition of common compressed file formats to the *alsad* protocol. In the current implementation, *alsad* can only process raw sound data. The design of *alsad* allows a developer to make these improvements with relative ease. The addition of this capability would solve many of the bandwidth issues associated with *alsad*.

Currently, *alsa_client* cannot easily create a stream from several different files. The fact that the *alsa_client* cannot change the properties of a stream after its creation means that

files with different audio formats require different streams. Solving this problem requires *alsad* to send changes in the stream properties to all of the connected sinks. Sinks that write to the sound card would have to keep track of the changing formats and reconfigure the sound card at the exact moment they begin processing the new data format.

In its current state, the user of *alsa_client* must know the address of an *alsad*. Future development of *alsad* might center around the creation of a directory service to advertise the existence of a public *alsad*. The directory could be centralized or peer-to-peer. The centralized design would have designated master servers that have knowledge of all existing *alsad*s on the network. The peer-to-peer design would allow a client that knows the address of one *alsad* to recursively find more. The *alsa_client* would send a request to *alsad* to return a list of *alsad*s connected as sources or sinks. To find more *alsad*s, *alsa_client* would perform the same procedure on the returned list.

Future developers might also want to consider developing a graphical interface for *alsa_client*. A new user-friendly interface could present the information gathered from *alsad*s in a more effective manner.

Lessons Learned

This project has been the largest technology-related endeavor of my life. It caused me to study many areas not even related to the project. I studied topics like the Linux Terminal Server Project, wireless network cards for laptops (specifically their configuration in Fedora Core Linux), Vorbis and mp3 audio encoding, and Linux firewall configuration.

There have also been many skills that I have learned that are directly related to this project. I was introduced to many new topics related to the C programming language such as threads, condition variables, mutexes, shared memory, using `getopt()` for command-line arguments, and process signaling. I have also learned how to design, implement, and document a large scale application. Most of this application was created within the Eclipse integrated development environment [3]. This application is normally used with the Java programming language but has plug-ins for C development. It also has plugins for Doxygen [4], the document generation tool I used to create on-line documentation [http://sweb.uky.edu/~drwarr0/doxygen/index.html] .

In addition to learning new tools and programming concepts, I also gained the necessary technical writing skills to produce this document. There is no question that I have gained a tremendous amount of knowledge from this project.

Glossary

address-information structure. A structure in the *alsad* communications library that contains address information about connecting clients.

alsa_client. An application provided to authenticate with *alsad* and generate all requests that the server is capable of handling.

stream. One or more audio channels having a common format. Each stream has a unique numeric identifier and a textual description.

alsamixer. A graphical console-based program that comes packaged with the ALSA libraries.

aplay. A small command-line program capable of playing back many different sound file formats.

audio channel. A digital representation of sound waves captured from one source in chronological order. The representation of sound waves can be in any format including analog, digital and compressed.

audio-header structure. A structure in the *alsad* communications library that contains the size of a the sound data transmission. This information is used as a framing mechanism to communicate how much sound data to send or receive.

buffer size. The amount of the hardware buffer on the sound card to be used.

control structure. A structure in the *alsad* communications library that contains defined control values.

data-pipe structure. A structure used to request or send information related to a source or a sink, the client uses the data-pipe structure as a transport mechanism. The term “data pipe” is meant to encapsulate both sources and sinks.

global byte index. A variable in the `circ_buff_t` structure that gives a unique identifier to every byte that is written to the buffer.

global-stream list. A linked list created from the stream-list library that contains all the available streams on *alsad*.

hardware-parameters structure. A structure in the *alsad* communications library that contains all of the information required to configure the sound card, open a device and begin capture or playback.

overrun. A state in which a write is called with more data than there is space available in the circular buffer.

parallel-read function. Function that read data from a circular buffer that multiple threads can execute simultaneously.

period. A standard data size sent to or received from the hardware buffer on the sound card when a user issues a read or write command.

sample. A digital snapshot of the analog signal produced by the sound card.

sample rate. The rate at which the sound card reads samples from a sound-card hardware interface.

sample type. The size and type of data structure used for storing a sample.

serial functions. Functions that use locking mechanisms to prevent execution in parallel by multiple threads.

sound-card hardware interfaces. The hardware input and output devices located on the sound card. They are interfaces from the sound card to components that connect with the card. These interfaces allow microphones, headphones and low-level input and output devices (both in digital and analog format) to connect with the sound card.

stream-information structure. A structure in the *alsad* communications library that contains the unique identifier and textual description of a stream.

underrun. A state in which a read is called and there is not enough data available in the circular buffer.

References

[1] Real Networks Inc. http://www.realnetworks.com/.

[2] ALSA: Advanced Linux Sound Architecture. http://www.alsa-project.org/.

[3] Eclipse. http://www.eclipse.org/.

[4] Dimitri van Heesch. Doxygen. http://www.doxygen.org/.