Writing network multimedia applications with MAS

Silvio Neef and Leon Shiman and Brian O'Brien

June 10, 2004

1 Introduction: MAS Overview

This paper describes MAS, The Media Application Server. The first part of the paper is a high-level introducion to some key ideas behind MAS. Later, we explain how to program with MAS's core APIs.

MAS is a sound and multimedia server, designed with three main goals: fundamental network integration, extensibility, and fine control over timing. Architecturally, it incorporates experience from the X Window system and works with X although it does not need X to function. MAS is network transparent, operable on many different platforms, and has an open core developed within a standards body (X.Org Foundation).

MAS more nearly resembles a peer-to-peer model than a client-server model: MAS servers on different hosts communicate with each other on behalf of their clients. MAS uses RTP, the Real Time Transport Protocol (http://www.cs.columbia.edu/~hgs/rtp/) for all network traffic. Use of RTP makes efficient transfer possible (for example, over UDP) and ensures that all data packets carry such metadata as precise time stamps and talk spurt information. Finally, MAS is network *transparent* in the same way that the X Window system is; most MAS clients run unchanged across the network.

Extensibility is key for any project that intends to grow beyond its original application focus. MAS extensibility begins with binary plug-ins, called devices. Individual devices can be combined into complex data processing assemblages. MAS can be extended to perform new tasks by writing new devices and combining them with existing ones. A MAS application is the union of a client process and its devices and assemblages inside one or more MAS servers. Because almost everything in MAS is a plug-in, MAS can be extensively customized and can scale from simple applications on small embedded systems to demanding tasks on large servers.

MAS keeps statistics for all actions in its devices, including network activity. Combined with the timing-related RTP metadata, such monitoring permits fine grained control in the scheduling of data processing and transmission.

MAS is intended to be a full-featured sound server for desktop applications. MAS is independent of desktop toolkits and windowing systems, but is designed to be easy to integrate into desktop programming environments like GNOME and KDE.

MAS comes with few dependencies and a permissive license. Most of MAS is written in ANSI C89. (The only exception is the RTP library, which is in C++.) The MAS core relies only on standard libraries and functions (e.g., POSIX, ANSI). MAS, therefore, compiles and runs on many different platforms. Usually the only changes required for porting are in the device that wraps the native audio interface on the new platform (the `anx` device).

The MAS core (server, core devices, and several clients) is free software under the MIT license (aka the X Window System license), a simple license compatible with many other free software licenses. This license also allows potential third-party MAS devices and clients to have different (i.e. proprietary) licensing terms.

MAS was designed with accessibility requirements in mind. Accessibility in this case means being usable by people with disabilities. In the audio domain, such requirements include relatively low latency, the ability to quickly stop and start utterances from a screen reader, and good synchronization of multiple streams.

The following sections (2, 3, and 4) present a relatively detailed description of basic MAS programming. The target audience is programmers who wish to extend MAS with new devices, contribute to MAS, or write applications requiring the fine grained control that the core MAS APIs afford. Section 5 talks about our plans for a simplified API, and section 6 offers advice for further study.

2 MAS Architecture: Basic Concepts

Almost everything in MAS is an extension. Processing of audio or video data is performed by so-called *devices*, which are shared objects (aka plug-ins) dynamically loaded into the server at run-time. The three main components of the server are an *assembler*, a *scheduler* and a *master clock*. These entities are compiled into the server but share so much syntax with the plug-in devices that you can consider them devices as well. A set of core devices is also usually loaded by default.

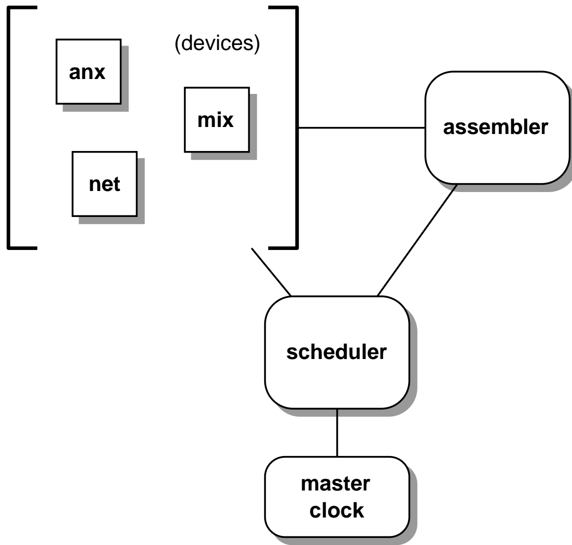

Figure 1: MAS overview.

2.1 Assembler

All audio processing in MAS is performed on small chunks of data. Such chunks consist of buffers of data (typically a few hundred bytes long, along with some metadata–implemented in the `mas_data` struct) that are passed from device to device. All the action is in the devices; they manipulate, generate, or consume data. Devices are discussed in section 3.

Processing is accomplished by sending `mas_data` from device to device. The assembler is responsible for both instantiating devices on behalf of a MAS client and connecting those devices on behalf of the client. The assembler keeps track of such arrangements of devices (assemblages) for each client. Loaded device libraries are reference counted. When the client exits, the assembler knows which device instances it can terminate and which shared libraries it can unload.

The assembler joins devices by connecting their *ports*. A device can have an arbitrary number of ports, and the ports may be dynamically allocated. Think of a port as a socket into which you plug a data cable to patch together different devices. There is a queue on each port in which data packets waiting for processing are held. Ports are either sink or source ports; most devices have static ports labeled “sink” and “source”.

2.2 Scheduler

A MAS client can, by making assembler calls, instantiate devices in the server and connect them by their ports. All the different things a device can do are called *actions*. Thus, taking a buffer full of 44.1kHz audio data and turning it into a buffer full of 8kHz data is an action of the `srate` sample rate converter device. The connecting of two devices by specified ports is an action of the assembler device. The scheduler coordinates all actions by maintaining a queue of *events* and triggers these at the appropriate time as determined by the scheduling algorithm. The scheduler always runs in a single thread.

MAS events are not like events in the GUI world. Think of a MAS event (`mas_event`) as a wrapper for an action that specifies when the action will be performed. Events can be periodic or unique. Their timing may also depend on the availability of data on given ports (more details in section 3.3). Events are prioritized. Low priority events can be delayed for the benefit of high priority events. We also say that an event *triggers* an action.

The scheduler automatically times and records each action and keeps statistics of the action times. You can query the scheduler for the time (global mean, minimum and maximum, windowed average and standard deviation) spent in each action for all devices. A future version of the scheduler will use this information to anticipate the length of certain actions and adjust scheduling accordingly.

2.3 Master Clock

The MAS master clock is essentially your computer’s system clock. MAS uses it to synthesize and synchronize clocks running at different speeds. With the master clock, you can bind events and actions to a suitable clock or you can create your own clock with a special rate. For example, some sound cards have clocks that are quite inaccurate. You may set your format to 44.1kHz but the audio hardware may actually play the data at around 43kHz. To resolve this problem, MAS contains a clock that ticks at the estimated rate of your sound card clock (the estimate being continually refined). By driving your MAS data flow with this special clock, you ensure that you are sending data at the correct rate.

2.4 Base Devices, Channels

MAS is a networked media server. Because the `net` device handles all network activity, it is almost always used. A MAS programmer will use the net device in a very indirect way most of the time. A MAS client always communicates with a MAS server on another host through the local MAS server. This proxying will be mostly transparent to the programmer. Either the client will default to have the local server as its target server or it will know the target server from environment variables. When you do need more explicit control, you can use *channels* in your programs to denote specific servers.

Channels consist of two types: data and control. Most assembler actions have an `on_channel` equivalent action that lets you specify where (i.e., on which channel) to perform the action. Think of channels as wormholes to the target server (local or remote); whatever action you specify will be carried out on the server to which the channel leads.

Because it is crucial for both local and remote connections, the net device is always loaded at server start.

Unless MAS is started with the `-s` (silent) option, a default "anx assemblage" is also loaded at start time. The anx assemblage contains the `mix` device (a software mixer ready to accept any number of connections) and the *audio nexus* or `anx` device, an abstraction of the native interface to the audio hardware. With this default assemblage, you're ready to play or record sound by attaching something to the sink port of the mixer and/or the source port of the anx device.

Ports, devices, as well as channels are passed using handles of "opaque type": `mas_port_t`, `mas_device_t` and `mas_channel_t`. The premise behind this abstraction is that you can use variables of these types just as you would use variables of fundamental types, without knowing about any internal structure. For example, you can talk to the server transparently about a device when that device is actually instantiated in a MAS server on a different machine. Think of values of these types as similar to unique integers that you can use to refer to specific devices or ports or channels.

2.5 Working With MAS

At this point it is a good idea to download the MAS source code if you haven't already. The next sections will refer to specific parts of the source, and you may want to follow some of the references. Navigation tools such as emacs or vi "tags" are of great help for looking up definitions. If you're unfamiliar with tags, try typing '`info emacs tags`' at your shell prompt.

The MAS source code can be downloaded from http://mediaapplicationserver.net. The source code is laid out in a straightforward way:

- `mas/` main server executable
- `maslib/` client library
- `common/` used by clients as well as devices (i.e., MAS package functions)
- `devices/` plug-in devices
- `clients/`, `devtools/`, `control-apps/` client programs for different uses

Read the `README` file in the top level directory to familiarize yourself with MAS. If you compile MAS from source, the `INSTALL` file has some helpful hints. Note that MAS uses imake, not autoconf, for build configuration. You can start a MAS daemon by typing `mas-launch` on the command line. If you can, run MAS as root user because it will allow MAS to obtain real time scheduling privilege from the Linux kernel.

Along with the source code, the MAS log file is very useful for understanding what happens. If MAS runs as a daemon, this log file goes into either `/usr/local/mas/log/`, or `/var/log/`, depending on your compile-time options. MAS logs in one of several *verbosity levels*. The level can be set through a command line option. Most useful debugging information is printed at level 50 (`MAS_VERBLVL_DEBUG`).

3 Devices

You may want to look at the source code for a simple device while you read this section. Because it is basic and useful, We suggest the `endian` device (in `devices/endian/`), which converts data between big- and little-endian formats.

A MAS device is a dynamically loaded library that defines certain conventional symbols located in the *device profile*.By convention, the file profile.h defines a number of symbols, all starting in `profile_`.

When you ask the assembler to instantiate a device called `endian`, the assembler looks for a shared object named `libmas_endian_device.so` (the extension depends on the operating system). If found, the library is loaded with `dlopen`. The assembler then resolves the `profile_xxx` symbols from `profile.h` and stores them in a device profile structure.

The major properties of a device defined in its device profile are *action names*, *characteristic matrices*, and *ports*.

3.1 Action Names

The `profile_action_names` array is a list of function names representing the actions your device makes available. The scheduler can then schedule events that trigger these actions. By convention, a few names are expected to be there. Others are defined by the device programmer and depend on the device's purpose.

3.2 Characteristic Matrices

Connecting two ports makes sense only if the data format of the source port is the one expected by the sink port. The mechanism that specifies a data format is the characteristic matrix, a relatively free-form two-dimensional array of strings: only the two devices involved need to understand the contents. (For the audio data conventions involved, look in `mas/mas_cmatrix.h`.) Each row of the matrix describes a format that the device understands. Asterisks in any of the fields denote any value. The endian device, for example, can accept big- or little-endian values on its sink port and convert to either on its source port.

3.3 Ports

The next step is the association of a characteristic matrix with each of the device's ports in the `profile_ports` array. When you ask the assembler to connect two devices, the assembler can see if there is a match between any two rows in the matrices of the two device ports involved. If there is a match, the assembler configures these ports by triggering a `mas_dev_configure_port` action on both devices (see below) with this common format, called a *data characteristic*. The other elements of the `profile_ports` array are a name for each port and its type (source or sink).

3.4 The Life of A Device

The minimum actions your device must define are:

- `mas_dev_init_instance`. Initializes the device. In object oriented terms, this is the constructor, which instantiates the device object, that is, allocates resources and initializes them.
- `mas_dev_exit_instance`. The opposite of init_instance.
- `mas_dev_configure_port`. Allows you to set up your device for its task, given a data characteristic for a given port.
- `mas_dev_disconnect_port`. The opposite of configure_port.

When the assembler instantiates your device, the action `mas_dev_init_instance` is called. This is the point at which you allocate and initialize the *state* structure that carries the complete state of an instance of your device. Note that there may be many instances of your device in the server, all operating concurrently. Each instance has its own independent state. For the less frequent occasions when you want to initialize global state in your shared library (thus affecting all devices of this "class") you can define a `mas_dev_init_library` action. This action is called immediately after the shared library is loaded but not when individual devices are instantiated.

With an initialized instance of the device in the server, the client may ask the assembler to connect the device to another device. Assuming there is a matching data characteristic, `mas_dev_configure_port` is then called in your device. At this time, you should get the data characteristic for your port and use it to configure your device. (In many cases, a device's operational specifics depend on the type of its input and output data.) After both ports are configured and the device is ready, you will need to schedule some action that actually performs the device's work. In the endian device, as well as in many other simple devices, this is a *dataflow dependent* action, meaning that whenever data is available on the specified port, the action is triggered.

To schedule an action from within the device, use the device's *reaction port*. The reaction port allows you, by triggering other actions and events, to respond to actions called by the scheduler. Your specified action or event is put in a queue on the device's reaction port. When control flow returns to the scheduler, the scheduler checks this queue and schedules any events or actions in it on the appropriate device.

Thus, in the endian device, if you schedule the device action `mas_endian_convert` with the special priority `MAS_PRIORITY_DATAFLOW`, the action will be called whenever data becomes available on the device's sink. The `mas_endian_convert` function gets a `mas_data` struct from the sink port, converts the associated buffer (also called the data's *segment*), and posts the converted data on the source port. You can also queue periodic events (independent of any data availability) and tie them to a specific MAS clock. See `mas/reaction.c` for the details.

Using `mas_dev_disconnect_port`, you should undo anything you did when configuring a port. As a result, the next time the port is configured you start with a clean slate. Use `mas_dev_exit_instance` to free the state associated with the device.

3.5 mas_get, mas_set, and MAS packages

Each device performs a distinct, specialized task. Many have parameters that can be tweaked at run-time to influence their tasks. Some devices have a special client-side API. The core MAS devices' APIs are part of maslib, the client side MAS API.

A non-core device can make its own library available for MAS clients to link with. Functions within that library can connect to the MAS server and influence actions within the device. Some devices share common interface APIs, constructed in the same way. For example, the wav and mp3 source devices, as well as the `sbuf` device, all implement the `source` interface with functions like `mas_source_play` and `mas_source_stop`.

The `mas_get` and `mas_set` calls provided by maslib optionally offer a way for clients to interact with a device without using a special API. They are a simple way to make remote function calls. Figure 2 illustrates these two ways available to a client to interact with a device. The figure might also be helpful to explain how the server, devices, clients, and maslib are related.

The mas_get and mas_set functions work with *packages*, which provide a flexible way to represent data. Think of packages as sacks of information. You can stuff arbitrary key-value pairs into a package. A value may be yet another package. Such nesting allows you to describe tree structures. MAS provides mechanisms that can serialize and de-serialize packages and send them over the network. (More detail is in the common API reference at http://mediaapplicationserver.net/mas-common-0.6.0.pdf.) In the cases of both `mas_get` and `mas_set`, a key identifies a task for the device. The associated value is a package containing all the "arguments" to the call. A `list` mas_get call returns a list of all the supported queries on the device.

3.6 Core Devices

Here are some frequently used devices that are part of the MAS core. Please refer to their respective `README` files to find out how to use them. A full list of devices that come with MAS is in the top level `README` file.

net

The network, abstracted into a MAS device. The net device, using the Real Time Protocol RTP on top of

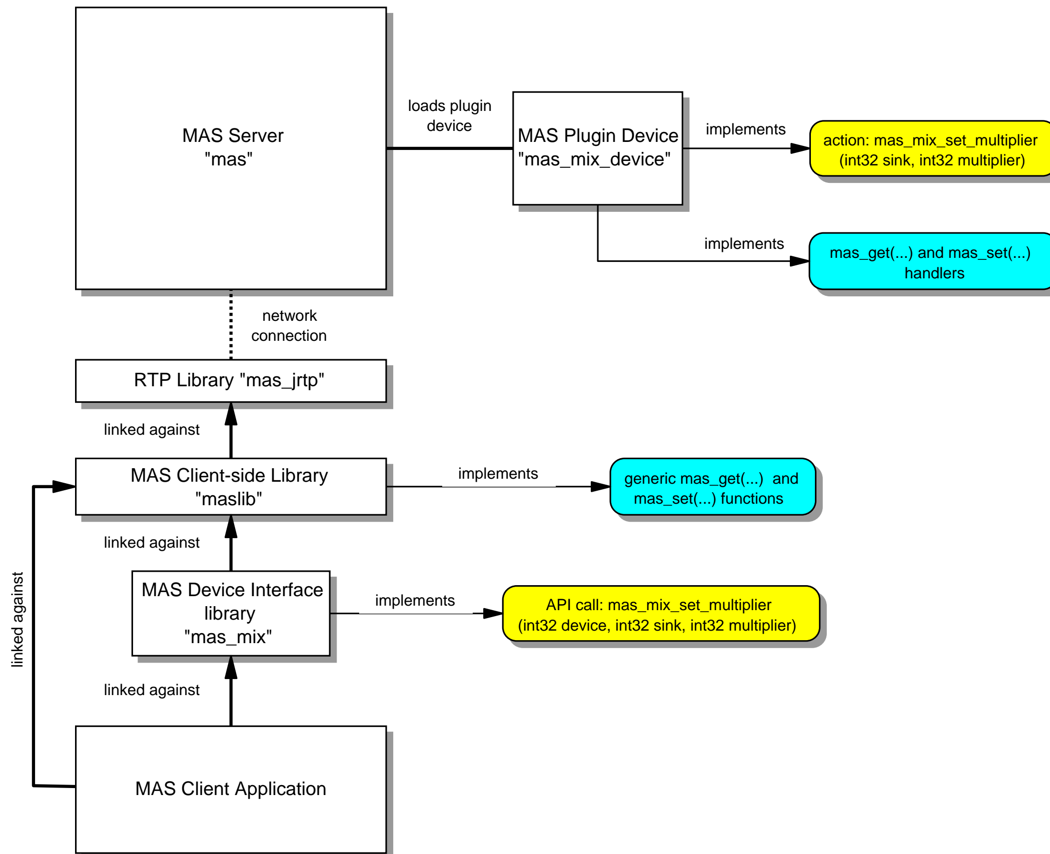

Figure 2: MAS libraries and device communication.

TCP or UDP, takes care of lower level networking. If necessary, the assembler will switch this device into your data pathway to proxy for a remote connection.

anx

Audio nexus (interface to the native audio system). Currently, we have support for Linux (OSS/ALSA), Solaris, and AIX. Porting is relatively straightforward because of modular design, so that only the truly platform-dependent parts need to be rewritten. The `anx` device has different capabilities on different platforms. The `list` query to `mas_get()`, the startup log messages, and the source code are helpful.

mix

Software mixer. This mixer uses 20-bit/sample accuracy for all internal operations. For high quality sub-bit encoding of mixed data, dithering with pseudo-random noise on the lowest bit is employed before downgrading to 16-bit resolution. On the other end of the scale, sample values that are too large for a given integer representation aren't merely clamped to the maximum value; the mixer employs a maximizing limiter with settable limiter knee softness. You can individually adjust volume levels in 128 steps for all incoming streams.

sbuf

A buffer on the data packet level. Use the sbuf device as insurance against network delays and jitter. You can `mas_set()` the buffer time in ms, and you can also specify how much of the buffered data will be released immediately after the buffer time has been reached.

srate

Sample rate conversion. Some distinguishing characteristics: You can adjust sample rates while conversion is going on without audible artifacts. You can set the sample rates smoothly in *fractions* of a Hertz, down to 1/100 Hz. (This device does not include any filters to remove aliasing. For that, you can use separate filters.)

channelconv

Mono to/from stereo conversion.

squant

Quantization (bit depth) changes.

4 Clients

To integrate all this information, we will now write a short client command line program that can play one or more 44.1kHz stereo .wav files using MAS.

The commented source code for this sample program is available at http://mediaapplicationserver.net/linuxtag. We suggest you have a copy of this source code handy as you read.

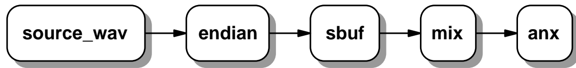

Figure 3: Data flow in the example client.

A wav file player is usually implemented in the following manner using native audio interfaces or some other sound servers. The client opens the wav file and performs whatever audio conversions are needed to match the file format required by the audio interface. (This might be done with the help of a library). The client then uses blocking I/O to ensure that the right amount of data is being read or written per time. For example, calls to write data to the `/dev/audio` device may simply stall until more data is needed by the audio device. Such behavior can be emulated in MAS using dataflow dependencies.

However, with all the clocks available in MAS, why not use them to send just as much data as the audio interface needs? While we certainly could send audio data from the client to the MAS server (see `clients/maswavplay`, for example), we aren't going do that now.

MAS already has a device that can read wav files and send out little packets of raw pcm audio. This `source_wav` device implements the source interface, just like the mp3 source device and the `sbuf` buffer device.

By moving as much work as possible to the server, we make life easier for the client and give MAS more control over things. The `source_wav` device frees us from having to interpret the wav file format, and the MAS scheduler can do all the timing for us.

Starting with the source code, the first thing to note in `main()` is a call to `mas_init()`, which authenticates the user to the MAS server and creates a control channel. MAS authentication is very basic at this point; MAS simply checks whether the connecting version of maslib is compatible, then does some simple `xhost` style access control. (In the future, MAS would implement a robust authentication scheme at this point.)

```
err = mas_init();
if (err < 0)
    ...
```

All MAS function calls return a 32-bit integer error code. In the `int32`, bit masks with different meanings are reserved for different types and origins of errors. (See `common/mas_error.h` for more information.) A negative value signals that an error occurred, so the above idiom is quite common in MAS.

If a connection is successfully established, we proceed to instantiate all the devices we will need. The flow of audio data is given in Figure 3.

The `source_wav` device reads the file and sends raw audio packets from its source port. The endian device converts this raw data to the correct byte order for the target machine. If the data travels over the network, the `sbuf` buffer device will be used to compensate for delays and jitter by adding a bit of latency. Then the data will be mixed with any other streams that are playing at the time, and the resulting mix will be played by the `anx` device.

We instantiate the `source_wav` device in the MAS server on the same machine our client runs on because that's where the wav file we're supposed to read resides. This instantiation, however, doesn't happen by default.

During `mas_init()`, maslib looked at our environment. If a `MAS_HOST` variable was defined to a host name or IP address, MAS took *that* to be the machine with our target server. In the absence of a `MAS_HOST` variable, MAS looks at the hostname part of the `DISPLAY` variable. This is how audio follows the display when you use MAS with X. If neither `MAS_HOST` nor `DISPLAY` are defined, MAS assumes the local host to be the target.

In our case, we have to advise MAS that the wav source device must reside on the local host, while all the other devices need not. Thus, we obtain a channel to the local MAS server, then instantiate the wav source device "on that channel":

```
mas_device_t source;
mas_channel_t local;
...
mas_get_local_control_channel( &local );
mas_asm_instantiate_device_on_channel(
    "source_wav", 0, 0, &source, local );
```

Because the `endian` and `sbuf` devices sit on the target host, we can use the default `mas_asm_instantiate_device()` function for them. The `mix` device is already present in the server; by using `mas_asm_get_device_by_name()`, we need only ask for a handle to it. We now have all our required devices.

Next, we need to "wire them up" into an assemblage. We connect the source port of the wav source device to the sink port of the endian converter.

```
err = mas_asm_connect_devices(
      source, endian, "source", "sink" );
```

Note also that we are not specifying a data characteristic for the ports. The source device itself sets the data characteristic on its output port to linear 44.1kHz signed short little endian (a current limitation of the wav source device). The above call will cause the sink of the endian device to be configured with the same data characteristic.

Note that the source and endian devices may or may not reside on the same MAS server. If they are in the same server process, the connection of ports means that the scheduler will simply shuttle pointers to `mas_data` buffers from the `source_wav`'s source port to the `endian`'s sink port. If, however, they are on separate hosts, MAS automatically handles

sending and receiving packets between these ports over the network. In such a case, we will see the net device proxying between our ports. The point is that the insertion of the net device is automatic and doesn't require much thought on the programmer's part. Simply using mas_device_t variables lets the right thing happen.

The next connection we set up explicitly:

```
dc = masc_make_audio_basic_dc(
  MAS_LINEAR_FMT, 44100, 16, 2, MAS_HOST_ENDIAN_FMT );

mas_asm_connect_devices_dc(
     endian, sbuf, "source", "sink", dc);

masc_strike_dc( dc );
```

First, we create a data characteristic (i.e., one row of a characteristic matrix). Then we connect the endian's source to the sbuf's sink port using this data characteristic. The only difference between this data characteristic and the one with which the endian's sink was configured is in the endian field: it has changed to "host", which means the native byte order of the machine where the endian device is running. This is how we tell converter devices what conversions to perform. The sink and source ports are configured to specific formats, which let the device know what we expect it to do.

Now examine the masc_strike_dc() function. MAS uses function names containing "setup" and "strike": to work with many of its data structures. In general, setup means allocate memory and initialize; strike means clean up and deallocate. In the example above, masc_make_audio_basic_dc() is a convenience function that will, in turn, call masc_setup_dc().

After that, we connect the sbuf's source to the mix device's default sink. The mix device replicates this port named default_mix_sink as soon as we connect to it. In other words, an arbitrary number of clients can connect to the mixer's default_mix_sink. The mixer simply changes the name label of the port as you are connecting. This renaming of ports by the device is the reason you should not rely on a port's name. Instead, refer to ports by variables of type mas_port_t.

The mix device is, by default, already connected to the anx device, so we have finished setting up our assemblage. Next, we tell the source device the name of a wav file to play. Actually, the source device can deal with more than one song; it maintains a playlist and knows how to make smooth noiseless segues between songs. We set this device's playlist through the mas_set() mechanism:

```
masc_setup_package( &nugget, NULL, 0, 0 );
masc_pushk_int16( &nugget, "pos", 0 );

for( i=1; i<argc; i++ )
    masc_push_string( &nugget, argv[i] );

masc_finalize_package( &nugget );
mas_set( source, "playlist", &nugget );
masc_strike_package( &nugget );
```

The first step is to set up a package. Providing null arguments for each of the package's buffer, size, and package flags arguments will allocate a default amount of memory for the package. Packages can grow dynamically as needed. We can require that packages use *static* memory through the buffer and flag arguments. (Learn how in the MAS common API documentation at http://mediaapplicationserver.net.)

After we have set up an empty package, we can stuff the information that the device expects into the package. In this case, we provide a starting track number, followed by the file names of all the wav files in the playlist. Finalizing the package gets it ready for transmission: we use the mas_set() call to send this argument package with key "playlist" to the source device.

We are now ready for playback. We tell both the wav source device and the sbuf to play:

```
mas_source_play( source );
mas_source_play( sbuf );
```

With that, our task is complete: MAS plays the requested files. In an actual player application, we might return to an event loop to await user input. In our current case, we'll just go to sleep(). Periodically, we wake up and ask the source device if it is still playing our wav files. If it stops, we exit our client too.

If, however, our client had exited right away, that is, without sleep(), MAS would have sensed the broken unix pipe and proceeded to "tear down" the assemblage associated with our client. Our devices would have been destroyed and cleaned up right away, not the result we want.

A Makefile, which shows how to compile this MAS client, is included with the example source code. Compile the player, then run it:

```
$ player some_file.wav
```

Make sure that the file is a CD quality stereo wav file, because this is the only format our simple client can play. Don't forget to start the MAS server before giving the above command.

Now try the following. Start MAS on a second machine on the network. Then set the environment variable MAS_HOST to that machine's host name. Assuming you are using the bash shell and the remote host name is remote.yourdomain.org, the command is

```
$ export MAS_HOST=remote.yourdomain.org
```

Then start the player again exactly as above. The wav file will now be streamed across the network and be played on the remote host. In the absence of a MAS_HOST variable, MAS will follow the DISPLAY variable in the same way, should it exist. Therefore, another way you might use the wav file player is to log into a remote machine through an X session

or with ssh and X port forwarding, and then start the player with a wav file sitting on that machine. The sound will, in this case, automatically play out of your local speakers.

Let us try one more thing. A MAS developer tool called masprobe lets you insert a device in-line into a data pathway. Obviously, this will only work if the inserted device has a sink and a source port and doesn't change the data format. The datalog device is very helpful in debugging–it logs the data that goes through it (or just the RTP headers) to the MAS log. For this example, we will use the visual device, which pops up a graphical spectrum analyzer in an X window. The following will only work if you have the FFTW library and have enabled it in MAS's imake configuration (config/host.def) before building MAS. Try this (again assuming the bash shell) while our example wav player is running:

```
sleep 1 && \
  masset -n visual do_work s do_work 1 & \
  masprobe -n visual -d 18 mix_source -d 17 audio_sink
```

This is not a very intuitive command line due to the ideosyncracies of the visual device and the masprobe client. However, it serves to prove the point: we've added the spectrum analyzer without even touching our wav file player.

5 Desktop Integration

We are working on a simplified API for programs that do not require such exquisite control over the process. Many 'desktop' style programs simply want to play a song, trigger a sound, or record from the microphone. For such cases, we are working on default assemblages, which are accessed through conventional library calls and which automate many common audio tasks. For example, given a sound file to play, MAS should know which codec and transform devices to use, automatically set up a pathway into the MAS mixer, and play the file.

To become involved in this effort and stay abreast of development, visit the MAS web site and follow the mas-devel mailing list.

6 Conclusion

We have discussed the basic architecture of MAS, introduced some core plug-in devices and showed how to write a MAS client.

We hope this paper shows that MAS is a solid foundation upon which multimedia applications can be built. We encourage developers to adopt MAS and develop new applications or extend existing applications to work with MAS. A simplified audio API for desktop applications would help in this adoption and is planned for the near future. Have a look at the project web site and join the effort!

6.1 Further Reading

The MAS web site is http://mediaapplicationserver.net. The source code for the sample client discussed in this paper can be obtained from http://mediaapplicationserver.net/linuxtag.

The site contains reference documentation on the various MAS APIs available under "documentation".

At some point you will want to start looking at the source code itself. Download the source directly from CVS or as a tarball as described on the web site. The top level MAS directory contains a README file that elaborates on the material in section 2.5 and should get you up and running with MAS. Most of the devices and clients also have README files in their respective directories that offer a high level overview over their specific functions.

There are two mailing lists, mas-devel and mas-bug, which you can subscribe to from the web site. These lists are archived on the site as well.