Writing network multimedia applications with MAS

7. Juni 2004

Note légale

Dieser Beitrag ist lizensiert unter der Creative Commons License.

Zusammenfassung

MAS provides media support for the desktop. We will briefly describe the MAS architecture and the core set of devices for audio manipulation. We will see how most clients end up being ”thin”because they consist mainly of code that invokes assemblages (collections) of devices (plug-ins) in the server.

Assemblages can be pre-defined in MAS for use by GNOME for example. We will explain how MAS can provide media server functionality for GNOME, adding MAS's network audio capability to existing GNOME apps. We will show how to write a simple MAS client.

1 1 Introduction

This chapter should be a general introduction to MAS. I figure about a page or two to describe what the goals and philosophy are, what is being done with MAS, maybe history, etc.

...

At the end, talk about the organization of this paper:

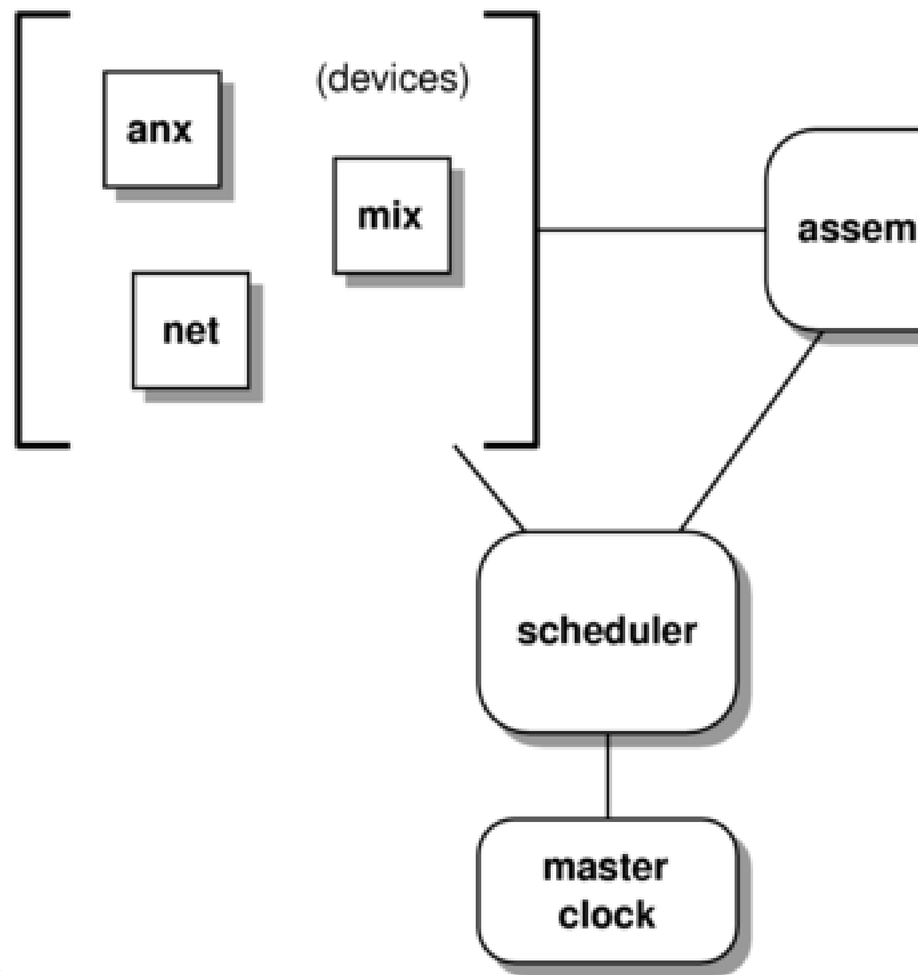
Sections 2, 3 and 4 give a relatively detailed description of MAS programming. The target audience for this overview are programmers who wish to extend MAS with new devices, contribute to MAS or write applications that require the fine grained control that the core MAS APIs afford.

We are working on a simplified API for programs that do not require such exquisite control over the process. Many 'desktop' style programs will simply want to play a song, trigger a sound or record from the microphone. For these use cases we are working on default assemblages (setups of MAS devices, explained below) which will be accessed through conventional library calls and which will automate many common audio tasks. For example, given a sound file to play, MAS should know what codec and transform devices to use, automatically set up a pathway into its mixer and play the file. [more on this in the GNOME section?]

2 2 MAS Architecture.

This section could be called Basic MAS concepts. Or it could be called Overview. What should it be called? It's supposed to teach the important concepts in MAS (not THE important stuff, rather the stuff that's important for someone who's going to ptrogram using MAS)...

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.

2.1 2.1 Assembler

All audio processing in MAS is performed on small chunks of data. Such chunks consist of buffers of data (see `mas_data` struct) typically a few hundreds of bytes long, along with some metadata and are passed from device to device. All the action is in the devices; they manipulate, generate, or consume data. Devices are discussed in ch \ref.

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 a 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 that will be used to plug in data cables to patch together different devices. There is a queue on each port in which data packets waiting for processing are held. Ports must be either sink or source ports; most devices will have static ports labeled ßinkänd ßource”.

2.2 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 the connecting of two devices by specified ports is an action of the assembler device. 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 scheduler coordinates all actions by maintaining a queue of *events* .

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 about this later \ref). 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 each action and keeps statistics of these action times. You can ask it for the time (global mean, minimum and maximum, windowed average and statndard deviation) spent in each action for all devices. This information can also be used by a future scheduler to anticipate how long certain actions will take and adjust scheduling accordingly.

2.3 2.3 Master Clock

MAS contains a master clock, essentially your computer's system clock. MAS then simulates clocks running at different speeds. You can tie events and actions to a suitable clock or ask for your own clock with a special rate. For example, one clock is synchronized with the (continually estimated) tick rate of your sound card's internal clock. Some sound card clocks are quite inaccurate. By driving your MAS data flow with this special clock, you can be sure you are sending data at the right rate.

2.4 2.4 Base Devices, Channels

MAS is a networked media server. Any network activity is handled by the net device. Therefore this device will almost always be used. A description of the net device is outside the scope of this article. Suffice it to say that MAS data is sent in RTP (include link) packets bla bla change this...

It is unlikely that a MAS programmer will use the net device directly. MAS being a peer-to-peer system, a MAS client always communicates with a MAS server on another host through the local MAS server. Most of the time this will be automatic: the client will default to have the local server as its target server or know the target server from environment variables. You can use *channels* to denote specific servers.

There are data and control channels. Most assembler actions have an on_channel equivalent action which 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.

At start time, a default änx assemblageïs also loaded unless MAS is started with the -s (silent) option. 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, and channels are passed using handles of öpaque 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.

3 3 Devices

A MAS device is a dynamically loaded library that defines certain conventional symbols located in the *device profile* . You may want to look at the source code for a simple device while you read this section. I suggest the endian device, which converts data between big- and little-endian formats (in devices/endian/) because it is very basic. 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 and, if found, loads it with dlopen . The assembler then resolves the profile_xxx symbols 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 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. Further details on this process appear in the section on Devices.

3.2 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 either big- or little-endian values on its sink port and convert to either on its source port.

3.3 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 3.3 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, `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 much less frequent case 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. Get the data characteristic for you port and use it to configure your device (in many cases the specifics of operation of a device depend on the type of its in- and output data). After both ports are configured and the device is ready, you will want 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 to "react"to actions called by the scheduler by triggering other actions and events. The action or event you specified 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 events or actions in it on the appropriate device. Thus, in the endian device, we schedule the `mas_endian_convert` action with the special priority `MAS_PRIORITY_DATAFLOW` . Every time data becomes available on the device's sink, this action will be called. 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 here (independent of any data availability) and tie them to a specific MAS clock. See `mas/reaction.c` for the details.

`mas_dev_disconnect_port` represents a chance to undo anything you did when configuring a port. The next time the port is configured you start with a clean slate. `mas_dev_exit_instance` is used to free the state associated with the device.

3.5 3.4 mas_get, mas_set and MAS packages

Each device performs its distinct, specialized task. Many devices have parameters that can be tweaked at runtime 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 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. Before I describe the mas_get and mas_set functions, you need to have an idea of MAS 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 whose value may be yet another package. This nesting allows you to describe tree structures. MAS provides mechanisms that can serialize and de-serialize such 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 case of both `mas_get` and `mas_set` , a key identifies a task for the device. The associated value is a package containing all the ärguments"to the call. A `list` mas_get call gives you a list of all the supported queries on the device.

3.6 3.5 Core Devices

The table shows a few devices that are part of the MAS core set of devices and that are frequently used. Please refer the their respective `README` files to find out how to use them.

Tabelle 1: Some MAS devices.

net	
anx	Audio nexus (interface to the native audi
mix	Software mixer. This mixer uses 20 bit/sample accuracy for all operations internally. Dithering with pseud
sbuf	
srate	Sample rate
channelconv	
squant	

4 4 Clients

To show how to pull all this together, we'll write a small client. It will be a 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 . I suggest you have a copy of this source code handy as you are reading here.

Traditionally (with native audio interfaces or with some other sound servers), a wav file player might be implemented as follows. The client opens the wav file and performs any conversions on the audio to match the file format with the format required by the audio interface (this might be done with the help of a library). Then the client would use blocking I/O as a way to ensure that just 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. This 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? And while you certainly can send audio data from your client to the MAS server (see `clients/maswavplay` for example), we won't do that here.

As it turns out, MAS already has a device that can read wav files, and send out little packets of raw pcm audio. This device is one of the `source_` devices. That is, it implements the source interface, just like the mp3 source device and the `sbuf` buffer device.

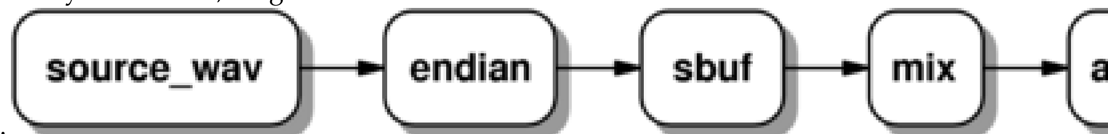
By moving as much work as possible into the server we make life easier for the client and give MAS more control over things; the `source_wav` device frees us from having to interpret the wav file format, and the MAS scheduler can do all the timing for us.

Let's get started looking at the source code. The first interesting thing in `main()` is a call to `mas_init()`. It authenticates us to the MAS server and makes a control channel. Authentication is very basic at this point; MAS just checks whether the connecting version of maslib is compatible, and does some simple `xhost` style access control. An actual authentication scheme will be dropped in here in the future.

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

All MAS function calls return a 32 bit integer error code. There are bit masks with different meanings reserved for different sets of bits in the `int32`. See `common/mas_error.h` for more information. A negative value always means that an error occured, so the above idiom is quite common in MAS.

If a connection was successfully established, we go on to instantiate all the devices we will need. The flow of audio data will be as follows:



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

The `source_wav` device needs to be instantiated in the MAS server on the same machine that our client runs on. After all, that's where the wav file is that we are supposed to read. However, this doesn't happen by default.

During `mas_init()`, maslib looked at your environment. If there was a `MAS_HOST` variable defined to a host name or IP address, or in lieu of that a `DISPLAY` variable, then MAS took *that* to be the machine with the server you are targeting. This is how the audio magically follows your `DISPLAY` variable when using MAS with X. In our case we need to advise MAS that the wav source device really has to be sitting on the local host, while all the other devices need not. To do this, we obtain a channel to the local MAS server and then instantiate the wav source device ön 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 );
```

The endian and sbuf devices will sit on the target host, so we can use the default mas_asm_instantiate_device() function for these. The mix device is already present in the server, so we only need to ask for a handle to it using mas_asm_get_device_by_name() .

Now we have all the devices we need. Next, we need to "wire them upïnto an assemblage. First, 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" );
```

There are several things to note here. Firstly, we are not specifying a data characteristic for the ports. The reason is that the source device itself sets the data characteristic on its output port to linear 44.1kHz signed short little endian (this fixed format is 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.

Secondly, the source and endian device may or may not reside in the same MAS server! If they are in the same server process, then the connection of ports just means that the scheduler gets informed to shuffle pointers to mas_data buffers from the source_wav 's source port to the endian 's sink port. If they are on separate hosts, however, MAS will automatically handle sending and receiving packets between these ports over the network. In this case you'll see the net device proxying between your ports. The point is that this process 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 (as explained above, this is 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 from the data characteristic above is in the endianness–it has changed to "hostëndianness, which means the native endianness of the machine where the endian device is running. This is in general how we tell converter style devices what conversions to perform. The sink and source ports get configured to specific formats, which lets the device know what we expect it to do.

Note the masc_strike_dc() function. MAS uses function names containing ßetupänd ßtrike"for working with many of its data structures. Generally, setup means to allocate memory and initialize, and strike means clean up and deallocate. In the above case, masc_make_audio_basic_dc() is a convenience function that in turn will call masc_setup_dc() .

After that, we connect the sbuf 's source to the mix device's default sink. The mix device will replicate this port named default_mix_sink as soon as you connect to it. In other words, an arbitrary number of clients can connect to the mixer's default_mix_sink . The mixer will just change the name label of the port as you are connecting. So, don't rely on port names; use mas_port_t instead.

The mix device is by default already connected to the anx device. we are therefore finished setting up our assemblage. Next, we get to tell the source device the file name of a wav file to play. Actually, the source device

can deal with more than one song; it has the concept of a playlist and knows how to make smooth noiseless segways between songs. The way to set this device's playlist is through the `mas_set()` mechanism briefly described above:

```
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 );
```

First, we set up a package. We give null arguments for the package's buffer, size and package flags arguments. This means that a default amout of memory will be allocated for the package. Packages can grow dynamically as needed. There are ways to have packages use static memory–that's what the buffer and flag arguments are for. You can read about this in the MAS common API documentation at http://mediaapplicationserver.net.

After setting up an empty package, we stuff the information expected by the device into the package. In this case, we give a starting position (in terms of track numbers), followed by the file names of all the wav files in the playlist. Finalizing the package gets the package ready for transmission. Then we use the `mas_set()` call to send this argument package with key ”playlist”to the endian device.

Now we are all set to start playback. We tell both the wav source device and teh sbuf to play:

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

After this, our task is done. MAS is playing the requested files now and there's nothing left for us to do. In an actual player application, we might return to an event loop awaiting user input. In our 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 stopped, we exit our client too. Note that if our client had exited right away, 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, which is not what we wanted.

5 5 Cool stuff

sound shim, in-line insertion of codecs and effects

what's being done. Mention conferencing.

6 6 GNOME Integration

??

7 7 Conclusion

7.1 7.1 Further Reading

There is some reference documentation on the various MAS APIs available at http://mediaapplicationserver.net under ”documentation”.

However, at some point you will want to start looking at the source code itself. Obtain the MAS source code at http://mediaapplicationserver.net. The top level MAS directory contains a `README` file that explains the source tree layout, among many other things. Most of the devices and clients also have `README` files in their respective directories that give a high level overview.

refer to web site, mailing lists

refer to API, DPI PDF reference

refer to source code of the sample client

MasClientExample