9P For Embedded Devices

Bruce Ellis
Tiger Ellis

Club Birriga
Bellevue Hill, NSW, Australia
brucee@chunder.com

ABSTRACT

9P has proved over the years to be a valuable and malleable file system protocol. Furthermore, as is it embraced by Plan9, it is more than a convenient protocol for interaction between disparate devices. Indeed Plan9 relies on it.

The protocol can be used to encapsulate control of an embedded device, which simply serves a 9P file system. However, even though 9P is very lightweight, it can be adapted to be more frugal on device resources. This is important on very small devices (FPGAs) where a full 9P implementation can consume most of the available gates.

We address this issue as a filesystem (embedfs) on the embedded machine's gateway Plan9 machine. We provide implementation and configuration details targeted at the *Casella* Digital Audio device.

1. Introduction

9P filesystems are used for diverse and often unexpected purposes. You need only look at upas [ref], fossil [ref], and ftpfs(1). Most are served by user-level processes, the kernel providing the necessary multiplexing and presenting physical devices as 9p servers. Remote devices are accessed seamlessly via whatever connection protocol is appropriate to the target. Typically this a common service, like 9fs, using a TCP connection. It can easily be a specialized server on an embedded device connecting via USB, serial, raw ether, etc.

A small embedded device may not have enough resources to provide a full 9P service. The resources that may be lacking include buffer space, outstanding request queue space; and of major concern sufficient silicon for handling the full protocol. Our intention is to provide a a file system which acts as an interface to a device implementing a (configurable) subset of 9P, seamlessly – respecting the integrity of the model.

Arguably a filesystem tailored to a specific device with a custom protocol is a more efficient use of cycles. We instead embrace a reuseable, respectable, configurable model and existing code – a more efficient use of brain cycles.

2. An Embedded File System Interface

The interface is implemented using lib9p [ref], which provides some clear optimizations. (Familiarity with the 9P protocol is assumed in this paper for brevity.) It is well structured and malleable.

Given the disclaimer we will state a result for a small embedded device, which has a very fixed structure and limited resources. This could easily be the conclusion – except there is more to tell.

This is what Casella looks like:

```
% cd /n/casella; ls -l
--rw-rw-rw- M 324 casella casella 0 Aug 26 22:02 audioctl
---w--w--w- M 324 casella casella 0 Aug 26 22:02 audioin
--r--r--r-- M 324 casella casella 0 Aug 26 22:02 audioout
--rw-rw-rw- M 324 casella casella 0 Aug 26 22:02 ctl
--rw-rw-rw- M 324 casella casella 0 Aug 26 22:02 irom
--rw-rw-rw- M 324 casella casella 0 Aug 26 22:02 midictl
---w--w--w- M 324 casella casella 0 Aug 26 22:02 midiin
--r--r--r-- M 324 casella casella 0 Aug 26 22:02 midiout
```

The directory served is flat with a constant map between name and `stat` info (including `Qids`). This information is loaded by `embedfs` from a configuration file.

Enumerating the `9P` `Tmesgs` served by `embedfs`:

Tversion
: `lib9p` handles this message.

Tauth
: `lib9p` user `auth()` function handles this. Usually no authentication is required, access is managed by permissions on the `srv` file. It seems unnecessary to replicate the natural `plan9` access mechanism.

Tflush
: Passed onto the device, held by the server, or even discarded.

Tattach
: Returns the root `Qid`.

Twalk
: Returns the appropriate `Qid`.

Topen
: Returns the appropriate `Qid`, and a suitable `iounit`. Informs the device if appropriate.

Tcreate
: `Eperm`.

Tread, **Twrite**
: Passed onto the device.

Tclunk
: `lib9p` handles this message. User function `destroyfid()` informs the device if appropriate.

Tremove
: `Eperm`.

Tstat
: `lib9p` user `stat()` function handles this (based on configuration data).

Twstat
: `Eperm`.

Note that the communication with the device can (and does) use a subset of `9p` (specifically: `open`, `clunk`, `read`, and `write`). In fact the device need only support `read` and `write`.

3. A Closer Look

The result presented above is readily implemented using `9pfile(2)` – the `Tree` and the collection of `Files` are fixed once the configuration is loaded, the communication with the device uses `fcall(2)`. The device requirements are small – storage and logic fall into "a small chunk of the device" category. So what's up? First we'll look at improvements to this implementation for a small, simple, device (`casella`) and then examine enhancements for more capable devices.

3.1. `iounit` Bottleneck

The high bandwidth files, `audioin`, `audioout`, and `irom`, have small on-chip buffers, so the obvious thing is to reflect this in the returned `iounit`. This has a very adverse effect upstream as a read of 8K will generate an enormous amount of host to host traffic. If these files are configured as "buffered" we can advertise a large iounit and handle the large transaction in the server with multiple (local speed) transactions with the device.

Example: The server receives a `Tread` request with size of 4K. The device has a 32 byte buffer. The server sends multiple 32 byte `Tread` requests to the device until one of a) the 4K buffer is full, b) a short read, or c) an `Rerror`. Similarly for `Twrite`.

3.2. Outstanding Requests

The chip has limited resources for storing outstanding requests. The device architecture is such that a restriction of a single request per file is natural and adequate. The server could simply queue requests per file. It may also wish to gate file opens to effectively make each file "exclusive–open with wait rather than error", allowing reads/writes of an open file to overtake waiting opens. This is particularly handy for control files. `Fids` and `Tags` are handled in the server, translated to device file number for communication with the device.

3.3. The Result

With these modifications the silicon footprint on the device is bounded (always good) and small in both storage and logic.

4. Enhancements

`Casella` has strict real–time constraints. Audio input and output are both 176KB/sec. Midi is much slower but still must not overflow/underflow. A program using `embedfs` to control a `casella` must use multiple outstanding reads and writes to meet these constraints. A library is provided to encapsulate this. The server uses `edf` [ref] to guarantee the device data rates specified in the configuration file.

5. Example Configuration

The configuration file for `casella` is listed below.

```
#
# casella.conf
#
downlink   2M
uplink     2M
iounit  32
buffer  8K
file   audioctl  666
file   audioin   222    buffered 176K
file   audioout  444    buffered 176K
file   ctl       666
file   irom      666    buffered
file   midictl   666
file   midiin    222    buffered 3125
file   midiout   444    buffered 3125
```