

BROWN UNIVERSITY
Department of Computer Science
Master's Thesis
CS-93-M16

“FutureFone”

by
Christopher Nuzum

This research project by Christopher J. Nuzum is accepted in its present form by the Department of Computer Science at Brown University in partial fulfillment of the requirements for the Degree of Master of Science.

Date: 5/10/93

A handwritten signature in black ink, appearing to read 'S. P. Reiss', written over a horizontal line.

Steven P. Reiss

FutureFone

A Splash of Multimedia

Christopher Nuzum

May 1993

Submitted in partial fulfillment of the requirements for the Degree of Master of Science in the Department of Computer Science at Brown University

Abstract

We present a framework for distributed multimedia applications with recording and synchronized playback, capable of supporting browsing of the time dimension.

FutureFone

SECTION 1	Introduction	7
1.1	Overview	7
SECTION 2	Network Architecture and Interprocess Communication	9
2.1	Requirements and Issues	9
2.2	Overall Design	10
2.2.1	The daq process	10
2.2.2	The audIO process	10
2.2.3	The ff process	10
2.2.4	Network specifics	10
2.3	Picture of the Network Architecture	11
2.3.1	Audio Lines	11
2.3.2	Control Lines	11
2.3.3	Event line	13
SECTION 3	Design of the Audio Substrate	14
3.1	daq design	14
3.2	audIO design	15
3.3	actrl: Using audIO without ff	15
3.4	Fault tolerance	15
SECTION 4	Design of <i>ff</i>	16
4.1	Devices	16
4.2	Startup	17
4.3	Establishing a Connection	18
4.4	Recording and Playback	19
4.4.1	Parallax	19

SECTION 5	A Sample Device: dd The Distributed Drawing Program	21
5.1	But First, A Closer Look at Devices, Events and the Dispatcher	22
5.1.1	Devices	22
5.1.2	Events	22
5.1.3	The Dispatcher	22
5.2	Deciding the Interaction Granularity	22
5.3	Compound Events	23
5.4	The Importance of Insulating State...	23
SECTION 6	Future Work	24
6.1	Time	24
6.1.1	Time Controllers, Time Devices, Devices, and The Dispatcher	24
6.1.2	Devices as Time Controllers	25
6.1.3	Multiple Time Controllers	26
6.1.4	Status of Time Controllers	26
6.2	Other Devices	26
6.2.1	A Ringer Device	27
6.2.2	A Marker Device	27
6.2.3	A Text Device	27
6.2.4	Other Devices	27
6.3	Messaging Systems	27
6.4	Shortfalls and Enhancements	28
6.4.1	ff Needs more GUI	28
6.4.2	Conference Calls	28
6.4.3	Timer Synchronization	28
6.4.4	Domain Address Server	29
SECTION 7	Comparison With Other Work and Influences	30
SECTION 8	Conclusion	31
8.1	Acknowledgments	31

Introduction

The *ff* system provides a mechanism for easily creating distributed multimedia applications. It is currently geared towards groupware applications, which fall under the rubric of Computer Support for Cooperative Work (CSCW).

ff began as an alternative to the telephone, capable of providing simple workstation-to-workstation audio communications, but the possibilities for developing interesting applications on top of such a transport were too numerous to be ignored.

A typical session with *ff* has an audio connection and some number of non-audio applications running. These applications, referred to as *devices*, need only generate and respond to *events*. *ff* manages routing the events over the network, dispatching them to the various devices, recording them and playing them back in real time. Currently, "real time" is provided by the underlying audio signal; during playback, all events are synchronized with the audio signal.

1.1 Overview

The paper is divided into the following sections:

Section 2: Network Architecture and Interprocess Communication

We begin with a description of the design of the *ff* system, from the outside in. *ff* consists of three communicating processes. In this section we discuss the issues surrounding this architecture.

Section 3: Design of the Audio Substrate

We next look more closely at the two processes which manage the audio portion of the system and present their internal design.

Section 4: Design of *ff*

Next we present the control process *ff* itself. All of the devices are bundled in this process. We present the device model, events and the dispatcher.

Section 5: A Sample Device: *dd*, The Distributed Drawing Program

We have implemented a shared, object-oriented drawing program. We will briefly discuss it in this section.

Section 7: Future Work

In this section, we describe how the *ff* system is capable of dealing with time, present designs for several unimplemented devices, describe various messaging systems which could be integrated into *ff* and discuss some of the shortfalls of and possible enhancements to the system.

Section 8: Comparison With Other Work and Influences

We briefly compare *ff* to some other systems in the CSCW literature, and we discuss some of the ideas which have appeared in other places which have surfaced in the design or implementation of *ff*.

Section 9: Conclusion

Network Architecture and Interprocess Communication

Although it would have been nice to have bundled all of the functionality of *ff* into one process, several factors intervened. While it is true that these factors derive, to a certain extent, from the Sun hardware and operating system on which we implemented *ff*, they are indicative of more general UNIX multimedia support problems.

First of all, the audio device `/dev/audio` must be continuously sampled in order to maintain a high level of audio quality. If a process doesn't sample the device continuously, data may be lost, which could result in unintelligible sound. Additionally, network throughput does not necessarily keep pace with the audio data rate, so the data which is sampled continuously must be buffered somewhere.

This buffered data must then be routed to its destination, incoming audio data must simultaneously be played, and we would like to be able to record all the audio data. Furthermore, we would like to do all this without affecting the performance of the rest of the *ff* system.

2.1 Requirements and Issues

- The process should be able to place and accept calls.
- The network behavior should be robust; it should only be necessary to run *ff* once per session.
- In order to maximize performance, each process should be an equal peer, rather than designating one process as a server. Thus, the communications paradigm should be peer-to-peer.
- Audio performance should not adversely affect the performance of the rest of the system.
- The network paradigm should support multiple participants, even though the initial implementation will support only point-to-point communications.

2.2 Overall Design

Our solution was to divide the work between three communicating processes. The *daq* process manages *data acquisition*, the *audioIO* process manages *audio input/output*, and the *ff* process deals with devices.

2.2.1 The *daq* process

The *daq* process continuously monitors `/dev/audio`, piping everything to the *audioIO* process. This generates approximately 8,000 bytes/second of u-law encoded audio data.

The pipe takes care of buffering the audio data, assuring that none will be missed unless the pipe overflows. If this actually happens, some data will be lost as the *daq* blocks during a write; we are willing to accept this, since in practice it happens *very* rarely.

2.2.2 The *audioIO* process

The *audioIO* process broadcasts all of the audio data from the *daq* to another connected *audioIO* process across the network, and it plays the audio data from the other *audioIO* process(es) to `/dev/audio`. It also manages the recording and playback of audio data. The *audioIO* process runs as a server on the network, but it can connect to another *audioIO* process as a client.

Additionally, *audioIO* gets control commands from a separate network client, a control process which connects to it. These control commands include placing a call to another *audioIO* process, activating or deactivating the *daq*, starting recording, etc. *audioIO* can run as a network daemon, much like the talk daemon *talkd*; we have implemented a very simple control process, much like *talk*, which allows audio-only point-to-point communications.

2.2.3 The *ff* process

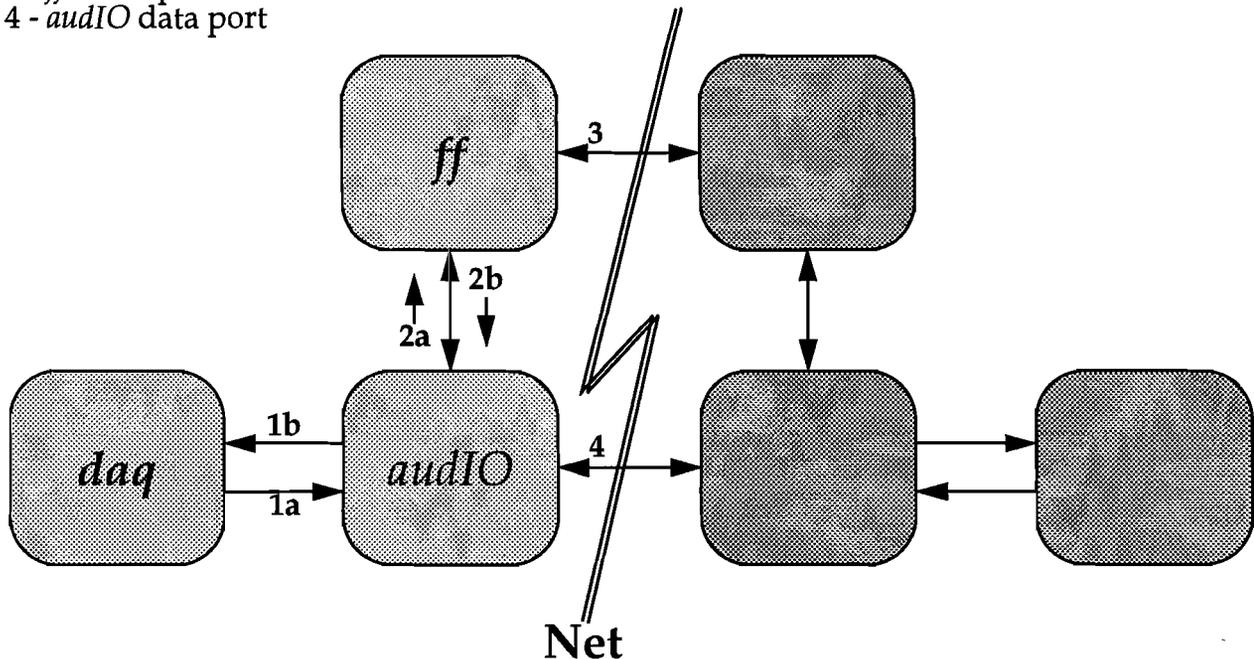
The third process is the *ff* process itself, which runs as a client of the *audioIO* process and as a server or a client of another *ff* process.

2.2.4 Network specifics

Both *ff* and *audioIO* run as non-blocking servers, awaiting a connection. When a user places a call, the caller connects as a client to the callee's server port. If either peer receives calls during the session, the called process forks, accepts the call, then terminates the connection by dying. This indicates that the process is busy.

2.3 Picture of the Network Architecture

- 1a - *daq* data pipe
- 1b - *daq* control pipe
- 2a - *ff* control port
- 2b - *audioIO* control port
- 3 - *ff* event port
- 4 - *audioIO* data port



If you are reading the color version of this document, notice that audio data flows along the red lines (1a, 4), control commands along the blue (1b, 2), and events along the magenta line (3). The numbering and colorings of these lines are consistent throughout this paper.

2.3.1 Audio Lines

The audio lines are pure; we do not modify the signal in any way. We experimented with subsampling, but found that the time necessary to compress and expand each sample obviated the decrease in network latency, unless such small samples were taken that intelligibility was sharply diminished.

2.3.2 Control Lines

The *audioIO* process sends only one type of control command to *ff*: a **time update** command. During recording, the time specified is used as the timestamp for the events being dispatched and recorded. During playback, *ff* responds to the time update command by dispatching all events up to the given time to the various devices.

ff sends several commands to *audIO*. The commands are:

Network Commands

- **call *address***. This command is used to tell an *audIO* process to connect as a client to another *audIO* process running on the node specified by address.

Time Commands

- **time on**. Activates the counter. When the counter is active, a time update command is sent to the control process (usually *ff*) after every [increment] samples.
- **time off**. Suspends the counter.
- **time reset**. Sets the counter to 0 and resets the increment to the default of 1000.
- **time *n***. Sets the counter to time *n*.
- **increment *n***. Sets the increment to *n*. Since audio data comes in at a rate of 8000 samples per second, using an increment of 1000 generates eight time updates per second. This means, in turn, that the synchronization granularity is 1/8 of a second.

Recording Commands

- **open *filename***. When this command is issued, the *filename* is opened for recording, and all of the audio data is stored in it. Currently, only the data from the local node's microphone is stored. We have experimented with various recording mechanisms.
- **close**. When it's time to stop recording, this command takes care of closing the file.

Playback Commands

- **play *filename***. This command sends all of the audio data in the file to the audio device.
- **playsynch *filename***. This is the same as **play**, except that it also generates time update events after every [increment] bytes of data and sends them over the control line (2a) to the control process.

Daq Commands

These commands are ignored by *audIO* and are forwarded to the *daq* process through the control pipe (1b) which connects *audIO* and *daq*.

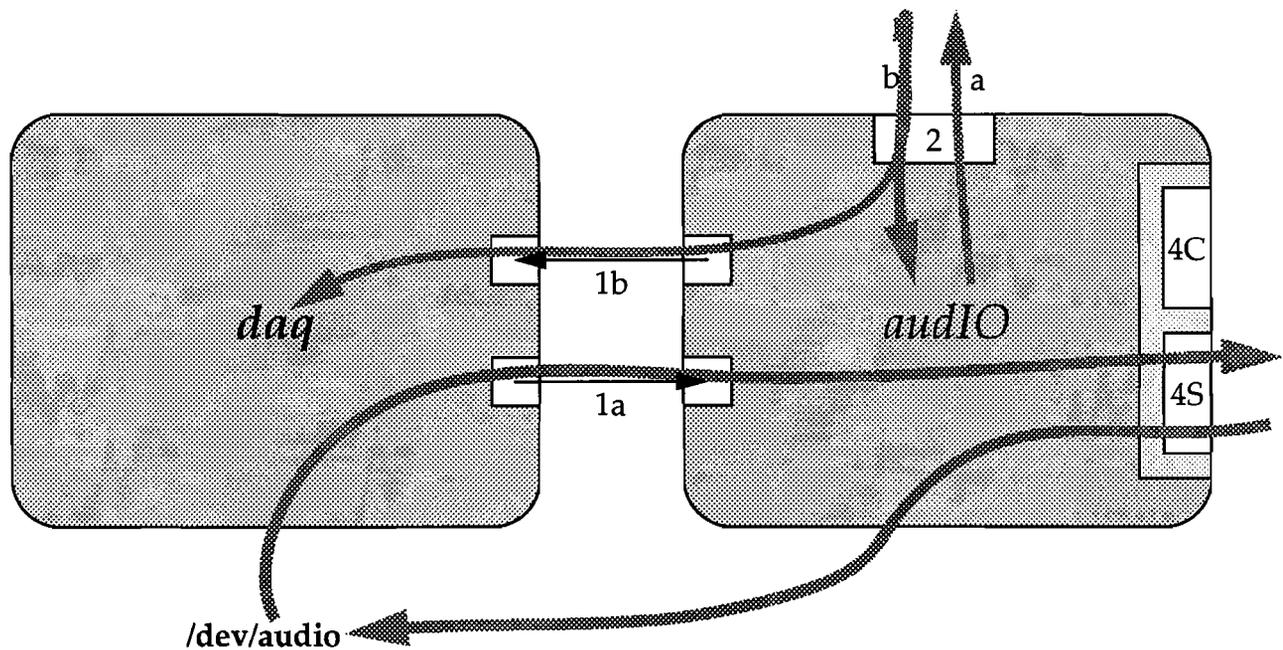
- **on**. This tells the *daq* to begin polling the audio device and sending the output to *audIO*.
- **off**. This tells the *daq* to stop polling the audio device.
- ***n***. An integer sets the current subsampling rate. 1 specifies that no samples are to be skipped, 2 specifies that every other sample should be taken, 3 specifies every third, etc.
- **die**. If this command is issued, the *daq* process exits. If *audIO* detects that the *daq* died, it creates a new *daq* and reestablishes the connection exactly as before.

2.3.3 Event line

The events which are generated, broadcasted, dispatched and responded to by the various devices pass over this line, insulated from the audio data. This mechanism is described in more detail below.

Design of the Audio Substrate

As we saw before, local microphone input is captured by the *daq*, piped to *audio* and then sent out to the network. Remote audio data is received by *audio* and sent to the audio device. Most control commands are responded to by *audio*, but some are forwarded to the *daq*.



When it starts up, *audio* creates pipes 1a and 1b and forks. The child process becomes the *daq*. When an *audio* process connects as a client to another *audio* process, data will travel through the first process' client socket (4C) to the second process' server socket (4S). Since it is irrelevant which socket is used, these sockets are referred to collectively as the *audio* data port.

3.1 *daq* design

The *daq*'s activity is centered in a loop, wherein the *daq* performs a `select` on pipe 1b, responds to any commands, then, if it is currently on, sends a sample through pipe 1a to *audio*.

3.2 *audIO* design

audIO is a somewhat more extensive implementation of the *daq*'s basic design; in its main loop, *audIO* selects on pipe 1a and sockets 2b and 4. During every pass through its main event loop, it checks:

- Whether a control connection is pending. If so, it is accepted, unless there was already a control connection, in which case *audIO* forks, the child process accepts the connection and dies, which is interpreted by the calling process as a busy signal.
- Whether a control command is pending. If so, it is handled.
- Whether a data connection is pending. It is handled the same way as a control connection request.
- Whether network data is incoming. If so, a buffer's worth is channeled to the audio device.
- Whether data is incoming from the *daq*. If so, it is sent out on the network.
- Finally, if a recording is being made, *audIO* writes the buffers containing the audio data read during this pass.

3.3 *actrl*: Using *audIO* without *ff*

It is possible to use *audIO* and *daq* without the *ff* process; the *actrl* process connects to *audIO* and allows the user to type control commands, which are relayed verbatim to *audIO*. A typical session would look like

```
actrl hostname-running-audio
```

```
call hostname-to-connect-to
```

At this point, the *audIO* connection should be made. *actrl* can then be used to start and stop recordings, test the *daq*, etc.

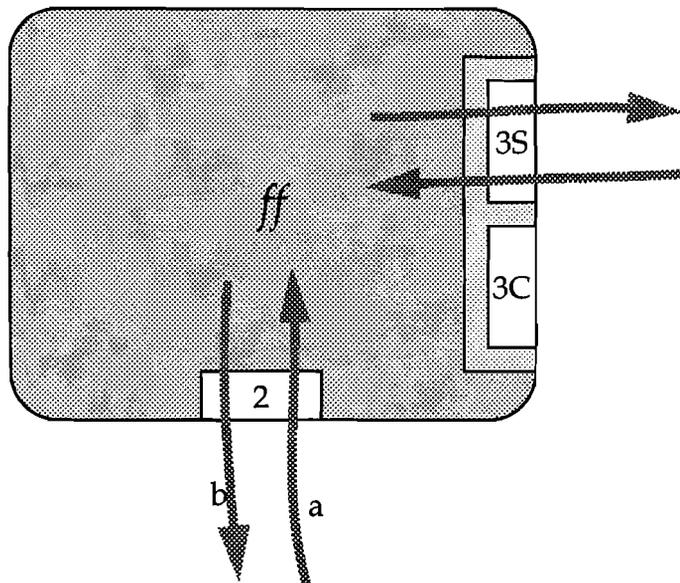
3.4 Fault tolerance

Special care was taken to insure that the network behavior of each of these processes is as robust as possible. If a connection is broken, the processes go back into a server state, awaiting a new connection. Thus, these processes are long-lived; they typically stay alive for as long as a user is logged in. If they are run as *setuid root*, *audIO* and *daq* are analogous to *talkd* and can stay alive as long as the system.

ff tries to start up *audIO* when it is run. However, if an *audIO* process is already running, the new process will exit immediately (when it can't bind a socket) and *ff* will connect to the extant *audIO* process.

Design of *ff*

Like *audio*, *ff* always runs as a server, and may run as a client as well, depending on which process initiates the connection. Once the connection is established, the distinction is irrelevant. The figure below depicts *ff* running as a server.



Unlike the invisible *daq* and *audio* processes, *ff* is a graphical application. We implemented *ff* using Motif (facilitated by the Baum encapsulation), using the `XtAppAddInput` interface to select for notification of incoming commands or events.

4.1 Devices

The fundamental paradigm of the *ff* system is that of event-generating-and-handling devices clustered around an event dispatcher which is connected to the network.

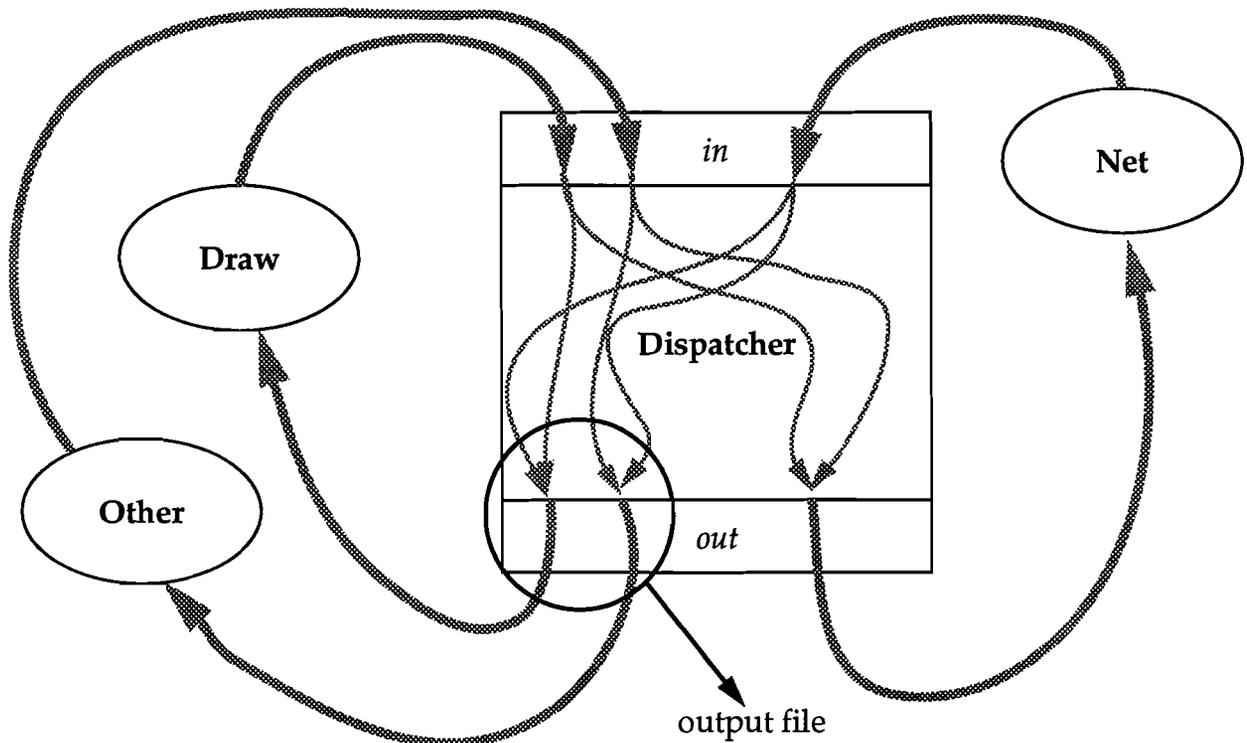
In order to facilitate network transparency, devices should be constructed in two insulated parts: the event-generation part and the event-handling part. A device should be able to respond to any valid event from devices of the same type, at any time, regardless of where the event originated.

All events are broadcast before being handled. For example, if a drawing program wishes to draw a rectangle, the rectangle should first be specified (via rubber-banding, but without leaving any imprint once the specification is complete). Next, the device should

create an event which describes the rectangle and broadcast it by sending it to the dispatcher. The device should then relax.

As events arrive for the device, the device will be called to handle them. In order to maintain high interactive performance, events broadcasted by a device will be immediately echoed to the network and dispatched back to the device. It is possible that other events will arrive and need to be handled while an event its being specified. This is why it is important that the event-generation part and the event-handling part not rely on each others' state.

Here is an illustration of how the dispatcher functions.



4.2 Startup

When *ff* is run, it forks and execs an *audio* process. It then connects to the running *audio* process as a client. While awaiting connections, Motif is in control and the user can interact with any available devices or use the UI to place a call, start recording, etc.

4.3 Establishing a Connection

When the user specifies “call” in the “session” menu, he is prompted for an address to which to connect. The port numbers used by *ff* are currently hardcoded; using these well-known ports, a network address suffices for specifying a connection.

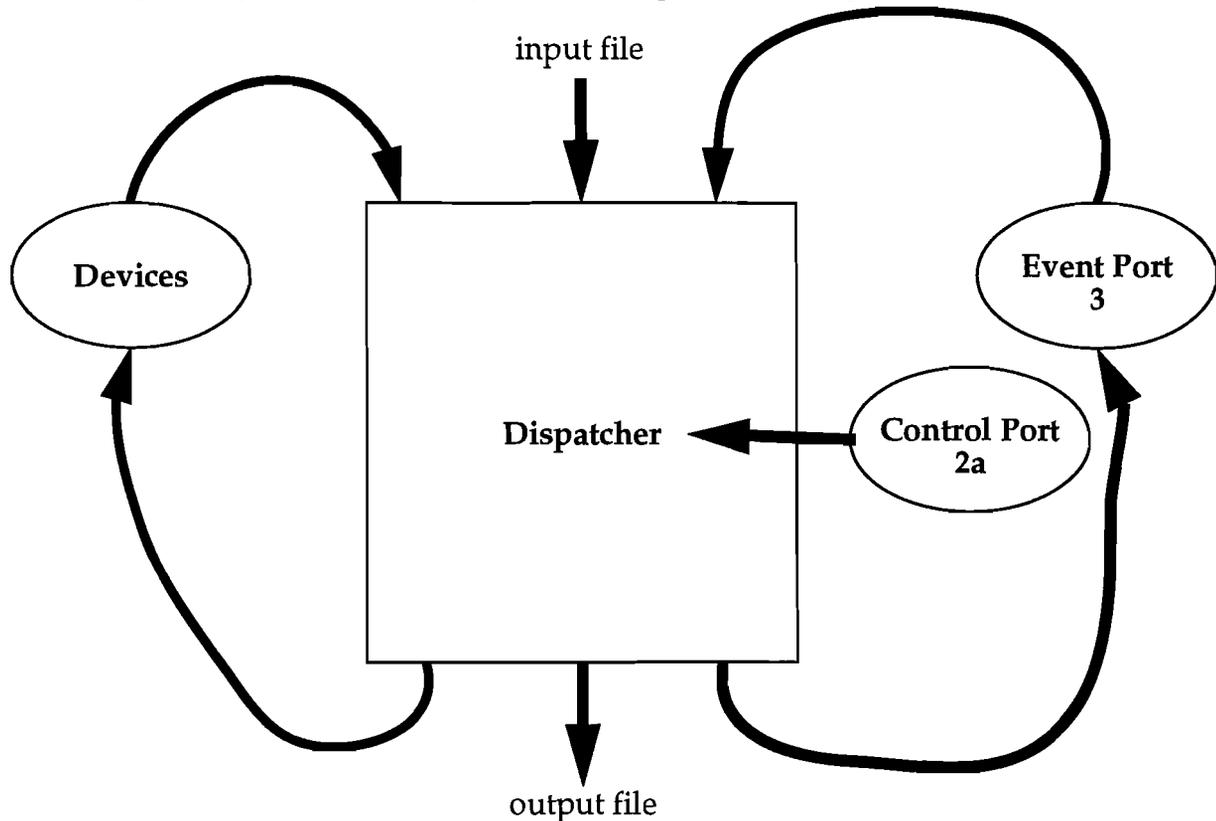
Before it actually tries to call the other *ff* process, *ff* tells *audio* to call the same address. When both processes have connected to their peers on another node, the connection is complete. Although not currently implemented, timeouts should be used to detect connection problems.

As an alternative to having the connection happen at two levels, we could have waited to fork the *audio* process until after all the connections were made. Several factors influenced our choice:

- */dev/audio* is not a shared resource. If *audio* is run *setuid root*, or if another user owns */dev/audio* and is running an *audio* process, the running *audio* process will be able to service any user’s requests, whereas an *audio* process forked by *ff* could fail to obtain access to the audio device.
- We wanted to be able to use *audio* separately, such as with *actrl*.

4.4 Recording and Playback

This figure depicts the control port's time input to the dispatcher.



During recording, two or more files are written: *ff* writes the events to an event file, and *audioIO* writes one or more audio files, which may later be mixed using the *mix* program to form one audio file. The events are timestamped and recorded immediately before being dispatched so the recording is as close as possible to what the user actually sees.

To begin playback, *ff* issues the **playsynch** command on the audio file. To handle each time update command, *ff* reads and dispatches the events with smaller timestamps from the event file. The resulting synchronization is excellent for everything we have seen so far except for drawing device move events, which are jerky. Adding a timer to *ff* would be trivial and would allow for synchronization with a much finer granularity, independent of the audio stream.

4.4.1 Parallax

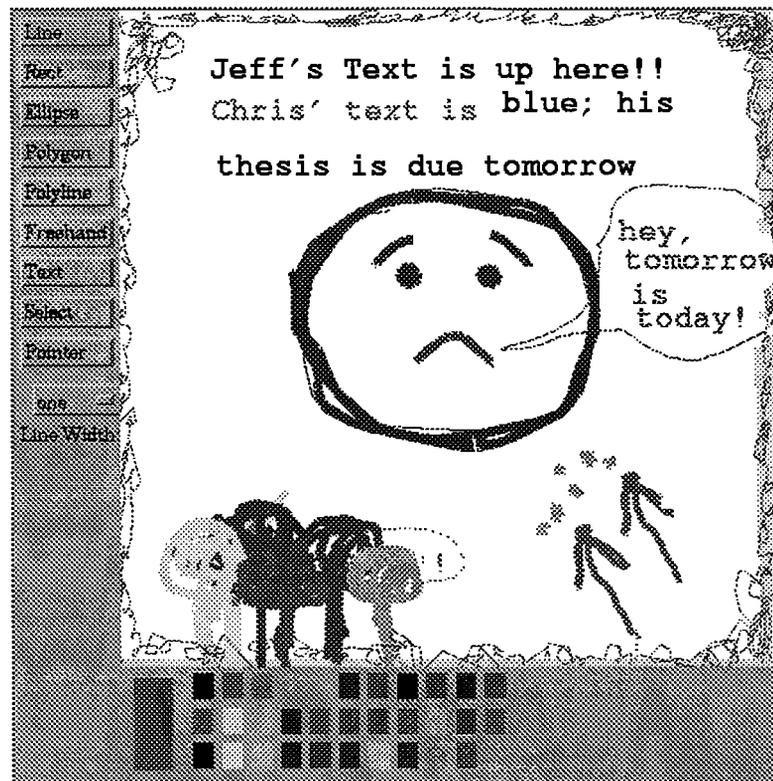
An interesting feature of this system is that each user on the network will handle (and record) events in a slightly different order, since a user's own events are received much more quickly than those from remote hosts. By recording events in exactly the order they are

handled, a user's recording will reflect the user's own perception of a session, which is fine, since there is no one "true" session which can be recorded.

A Sample Device: *dd*

The Distributed Drawing Program

The *dd* device functions very well within the *ff* framework, confirming that the model is indeed viable and the performance entirely acceptable for interactive applications.



dd supports a useful set of drawing primitives, including: lines, rectangles, freehand (polylines), text, various colors and line widths, selection and motion. Objects may be created and manipulated by any user connected to the system.

For a complete description of *dd*'s architecture, please refer to Jeff Stamm's *dd* project summary. Here we will use *dd* as an example of a typical device in order to illustrate some of the issues related to designing devices for use with *ff*.

5.1 But First, A Closer Look at Devices, Events and the Dispatcher

5.1.1 Devices

The C++ class which implements devices defines several methods as well as the protocol for handling device responsibilities.

Each device which is derived from class `Device` inherits an event queue, with enqueue and dequeue operators, and a constructor which registers the device with the dispatcher.

5.1.2 Events

Events are fairly simple constructs in *ff*. They contain fields describing their creator, device type, receipt time, how much data is associated and a pointer to the data itself. Events know how to create themselves from and write themselves to a network connection or file.

5.1.3 The Dispatcher

To broadcast an event, the dispatcher requests that the event write itself to the network stream. To dispatch an event, the event is enqueued on the appropriate device's event queue.

5.2 Deciding the Interaction Granularity

While writing *dd*, it became necessary to ask, "what should be an event?" One possibility was to make events out of every user action, cursor movement, rubberbanding motion, color selection, etc. The problem with this approach is that the number of events skyrockets, but the advantage is that every user sees *exactly* what every other user is doing. Another disadvantage is that certain things, like color selection, tend to want to modify the device's state, which is a bad idea, since the states of various users' devices are likely to become inconsistent very quickly.

At the other extreme, events could correspond only to completely specified objects, so that once a user finished drawing a box, the finished box would appear to everyone. This is fine for lines and boxes, but suffers when dealing with long freehand objects and movement commands — things it is interesting to talk about while specifying.

5.3 Compound Events

If all we have available to us are events, we must break down objects like polylines into several events. When this is the case, we must be careful in how we deal with incoming events; some may pertain to the polyline one user is drawing, while another event may specify a totally unrelated move or even part of another, different polyline. To deal with these problems, the event handler may need to maintain a certain minimal amount of state. In the drawing program, it sufficed to keep a certain amount of state for each user during polyline specification.

5.4 The Importance of Insulating State...

CANNOT BE OVERSTATED!!! (no pun intended) During our initial experiments, polylines frequently changed color while being specified because another event, which arrived from the net, changed — and forget to change back — the state of the graphics context.

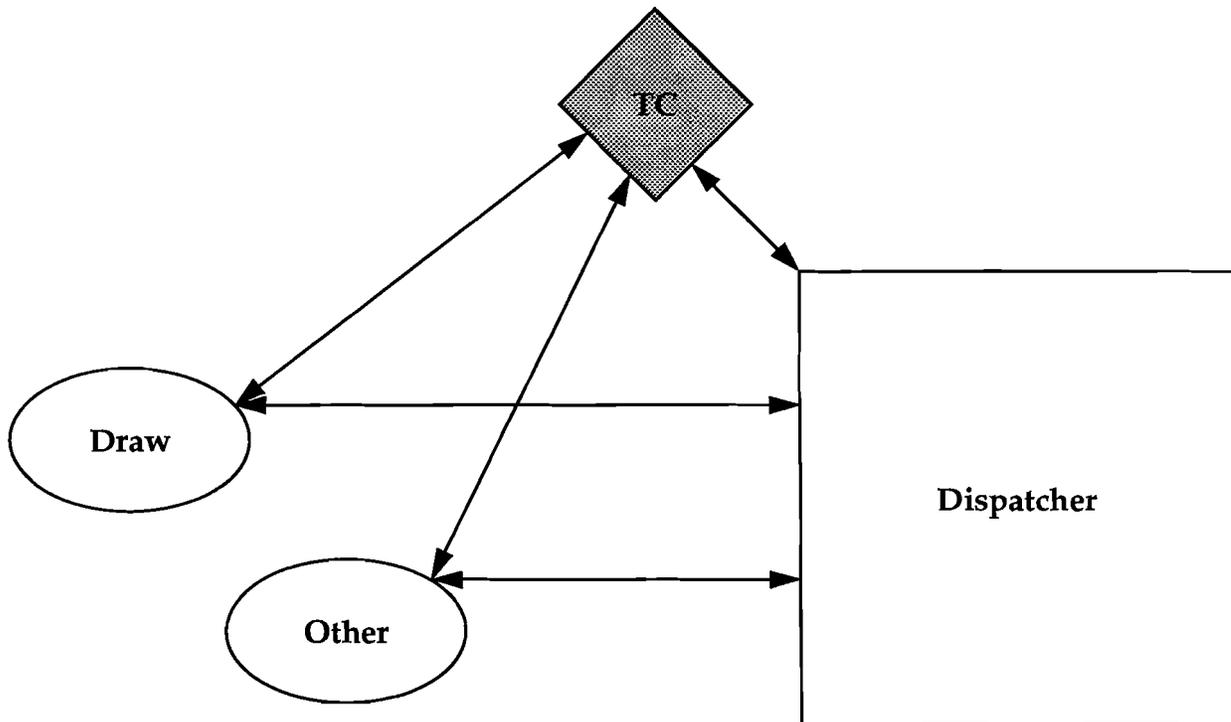
6.1 Time

The *ff* framework was designed to allow browsing of the time dimension of recordings as well as simple playback. *ff*'s notion of time is maintained by Time Controllers.

6.1.1 Time Controllers, Time Devices, Devices, and The Dispatcher

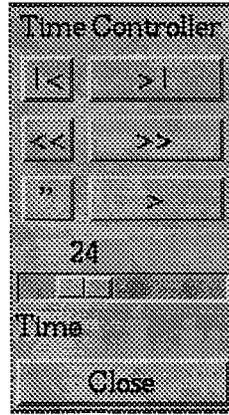
Both the `Dispatcher` and `Device` classes inherit from the `TimeDevice` class. This is a very simple class which defines the protocol for how derived classes should handle time requests and contains a pointer to a `TimeController` object.

Normally, each `TimeDevice` points to the same `TimeController`, the Universal Time Controller, which is always associated with the `Dispatcher`. This is depicted below. In fact, when the `Dispatcher` responds to a time update command, it actually sets the time value of the Universal Time Controller.



The primary methods declared by the `TimeDevice` class are `jumpTo` and `playTo`. A `TimeController` will call the `jumpTo` methods of each device it controls to tell the device to update its screen so that it looks like it did at the given time.

The `TimeController` class has a time data member, and it also has a method `sendAndSetTime`, which tells each device associated with the Time Controller to jump to the specified time.



This allows the user to use a graphical user interface to a Time Controller, such as the one depicted above, to jump to any point in time in a recording.

Once there, the user may press play (depicted as `>`) in order to commence playback at the current time. There are two ways of handling this; either the device can keep track of all of its data structures over time, so that it is capable of moving forward in time as well as backward, or the dispatcher can fetch all of the events from the events file and re-dispatch them to the device as necessary; this allows the device to throw away all data that occurs after the current time. The second approach is simpler for the device implementor, but it suffers from decreased performance.

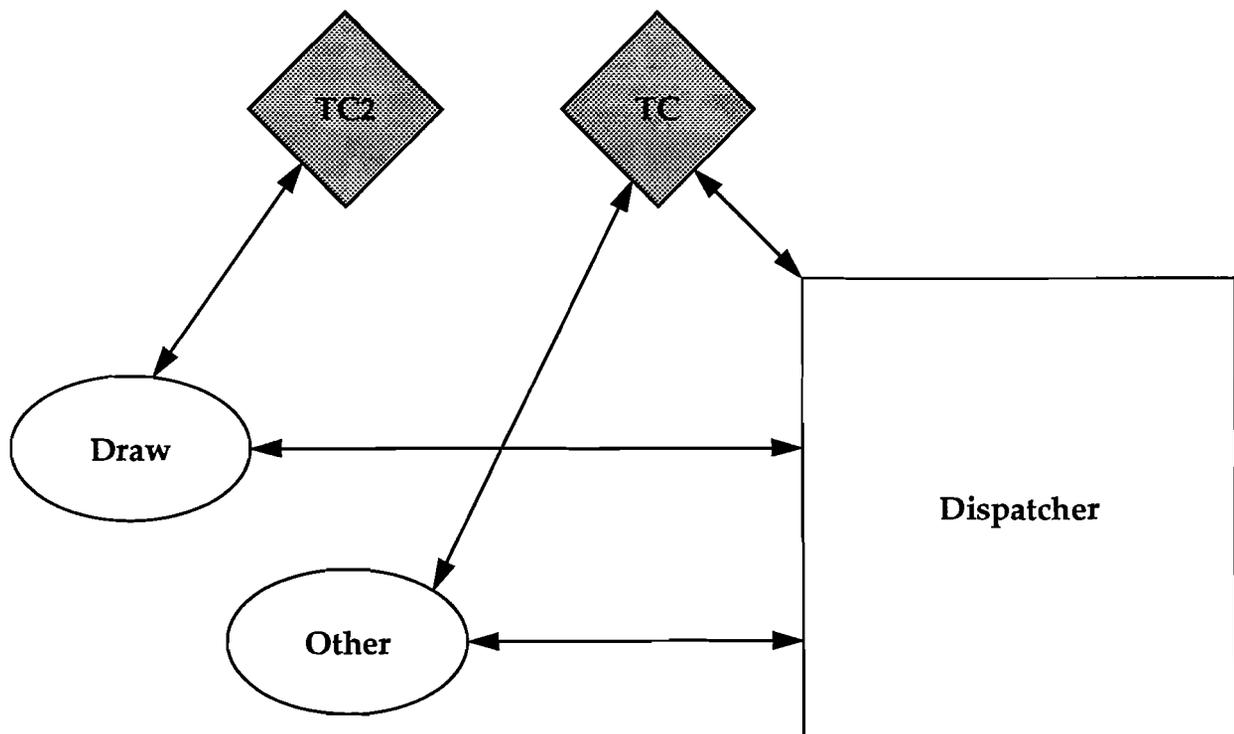
6.1.2 Devices as Time Controllers

Devices also inherit from the `TimeDevice` class, which means that they also have an associated Time Controller, and are therefore capable of using the `sendAndSetTime` method. This allows many time control tools to be written as devices, and it allows other devices to implement certain interesting functions.

For example, one possible function would be to allow a user of *dd* to click on an object in the graphics window, jump to when in time it was defined, and play the recording forward from that point. This would make it very simple to review an interesting part of a discussion.

6.1.3 Multiple Time Controllers

Occasionally, it is desirable to de-couple the notion of time associated with a certain device from that controlled by the dispatcher. For example, imagine a text device which operates like a talk window. While listening to the audio portion of a conversation, a user may wish to have the entire text available to scroll through. He would accomplish this by creating a separate time controller for the text device and positioning it at the end of time. This would allow him to scroll through the full temporal extent of the text while he listens to the recording in real-time.



6.1.4 Status of Time Controllers

The Time Controller work was the next step in the implementation of *ff*. It was

6.2 Other Devices

While the number of devices currently implemented in *ff* is small, the number of devices imaginable is very large. Furthermore, the design, implementation and functionality of these devices is easily understood. Here are some examples:

6.2.1 A Ringer Device

A ringer device could be built which sends a user's icon and signature audio file. This would allow the callee to know to whom he is talking. The data could be stored in a preferences file. The device could be implemented with one event for each type of ring, and each event could be broadcast. When the data is received, the audio file could be stored in the /tmp directory and the device could request *audio* to play the sound file.

6.2.2 A Marker Device

It would be useful to be able to "dog ear" moments in a conversation like pages in a book, so that they can be easily returned to at a later time. A marker device would broadcast events which contain a short textual description of what is currently happening. These descriptions would be displayed in a scrolled window. During playback, a user could click on any marker and jump to the point in time specified by that marker.

6.2.3 A Text Device

A text device could look like *talk*, except that each user would have a scrolled window of text. Every letter typed could be an event, and clicking on any letter could jump to when in time it was defined.

6.2.4 Other Devices

One can also imagine a memo device which would take advantage of the feature that events may be marked as private, in which case they will be recorded but not broadcast to other users. The user could click anywhere in the memo and jump to that point in time.

Another possibility would be a screen grab device, or a device which customizes an existing device, or a device which can send X text events to someone else's Emacs process.

6.3 Messaging Systems

Given that we have the ability to play and record files of audio and events, it's almost possible to add just a nice GUI and call it multimedia voicemail. With the time-browsing functionality added, such messages could become a very useful resource. However, there are a few other, subtler issues which must be addressed before *ff* would be a convenient voicemail facility.

- Messages should be consolidated into one file. There are many possible ways of doing this. In order to support rapid browsing of the time dimension, an index which cross references event times with offsets in the audio data might also be bundled into this file.
- A transport must be chosen for the message. If smtp mail is used, it may be necessary to segment long messages into several mailings and reconstruct them

later. If a shared filesystem and file-based messaging is used, messages will be limited to the reach of the shared file system.

- A mechanism would need to be developed which would allow access to outgoing messages, even if *ff* is not running. One possibility would be an OGM server which either knows everyone's OGM or is capable of peeking into private places in the user's account where OGM's are stored. Another would be to keep OGM's in a well-known place, like `~/ogm`. This would limit OGM's to working on a shared filesystem, and would make it difficult to keep different OGM files private, but would be easy.

6.4 Shortfalls and Enhancements

The current proof-of-concept implementation is rough carpentry; you can lean on it, but you might get a splinter if you sit on it. Some of the following refinements are analogous to sanding, others to building a sun deck.

6.4.1 *ff* Needs more GUI

In our rush to complete the proof-of-concept implementation, we neglected to build dialog boxes for fetching addresses or selecting files; although a pull down menu is used to choose to place a call, the address must still be entered on the command line.

6.4.2 Conference Calls

The current implementation is restricted to point-to-point communications. It would be a simple matter to support multi-way event dispatch by connecting every pair of participants in a conference call.

The audio data represents a more formidable challenge; while it is a simple matter to mix all the audio data coming in on multiple pipes, it is unusual for data to arrive simultaneously. It would be necessary to buffer a quantity of data from all parties (waiting, if necessary, until enough has arrived, to prevent dropouts), mix it, and then send it to the audio device. While not impossible, this is non-trivial, and a certain amount of "tuning" would probably be necessary to produce acceptable performance.

Alternatively, broadcast could be used instead of a completely connected graph, but reliability could suffer.

6.4.3 Timer Synchronization

Although the initial motivation for *ff* was partially to provide an alternative to the telephone, *audio* and *daq* tend to use a disproportionate amount of system resources. When these processes aren't desired, they should not be required. In their absence, timer callbacks could be used for timestamping and playback synchronization.

6.4.4 Domain Address Server

It is currently necessary to know what machine a friend is logged on to in order to establish a connection. An address server could simplify this process and allow calls to be specified using the recipient's name or userid instead of the network address of the machine where he is sitting.

Comparison With Other Work and Influences

We intentionally designed the original *ff* system before doing any research on other systems, hoping that this would allow us to dream up a clean model. Nonetheless, the literature on other systems later proved useful for how it helped to elaborate the design and to warn against certain implementation choices.

The architecture of the MMConf¹ system is similar to that of *ff*. It also uses events. However, it does not make any attempt to record conferences, and therefore has no built in time support. It did identify and expound upon various relevant issues, including centralized vs. replicated architectures, misordered inputs, non-determinism and late comers. Where they implement no solution to the latecomers problem, once the multi-party network support is installed, *ff* could easily be extended to pause while a late joiner reads in the contents of an event file.

Degen et al introduce a primitive version of time markers in their "Working with Audio"² paper. Since they use a personal tape recorder, their markers are limited to two tones which can later be identified and annotated in a graphical representation of the recording. They do have a sound browser utility, but we question the utility of actually displaying the sound waveform in a system like ours; it makes more sense for us just to display the semantic markers and provide a time controller for browsing the time dimension. The waveform display is most useful for editing sound, which is a dangerous idea when the sound is used for synchronization.

Goldfarb's HyTime article³ and the current draft of the ISO HyTime standard were useful while thinking about how to accomplish and represent the synchronization.

Finally, the network support in the *axross* project in the Brown University Computer Graphics group evolved along the same lines as *ff*'s. *Axross*' limitations motivated much of *dd*.

-
1. Crowley, Milazzo, Baker, Forsdick and Tomlinson, "MMConf: An Infrastructure for Building Shared Multimedia Applications", in Proceedings of the Conference on Computer-Supported Cooperative Work, October 1990, page 329.
 2. Degen, Mander and Saloman, Apple Computer Advanced Technology Human Interface Group, "Working with Audio: Integrating Personal Tape Recorders and Desktop Computers" in CHI '92 Conference Proceedings, page 413.
 3. Charles F. Goldfarb, IBM Almaden Research Center, "HyTime: A Standard for Structured Hypermedia Interchange", IEEE Computer, August, 1991.

Conclusion

The *ff* system allowed us to explore many issues related to shared multimedia applications and synchronization in a UNIX environment. Although the implementation of *ff* could continue for many months, we are satisfied that we have demonstrated that the system's model, design and foundation are sound. Furthermore, the current working implementation is usable, useful and fun.

We would like to see the integration of time controllers and synchronized multimedia messages into popular messaging applications; our experience with the system suggests that recording such messages is simple and natural; although the result is often similar to a video recording of the screen, the user doesn't suffer from the feeling of being in front of a camera.

As the hardware for multimedia support becomes available, we look forward to the applications that people will create, and expect that a simple, uniform system architecture like *ff* will facilitate the implementation process.

8.1 Acknowledgments

Thanks to Steve Reiss for his advice and sponsorship of the *ff* project; Jeff Stamm for *dd*, his input on the *ff* architecture and his patience, enthusiasm and companionship in the face of what all-too-occasionally seemed like a dangerously over-ambitious undertaking; Jeremy Gaffney for the initial Connection class which we stretched, patched and cajoled into supporting non-blocking IO; Andy van Dam, who provided us with a valuable mental model of the system's most skeptical potential user; Mike Anderson and Chris Brown, for sharing their experience from *axross*; and to my family and friends, for not understanding, but caring anyway.

