

Cmpt433 Final Group Project Submission

Fall 2011

Nap Time Sound Appliance

Benny Yan

Cal Woodruff

Thomas Betz

System Overview

Our system is a multi-function sound appliance that is network aware. This means that if you have more than one they can be made aware of each other and be used to share mp3 files, memos and intercom connections. We achieve this by making use of a local multi-cast network that any node can use to find out what other nodes are available on the network.

Nodes are able to play mp3 files, record memos in wav format, send memos to other nodes and share their mp3 files with other nodes. In addition, any node can initiate an intercom session with another node on the network.

The components needed to make a nap player node work are divided into layers. At the top layer there is the web interface and touch screen gui. These interact with an intermediate application layer that does the work of finding other nodes on the network, advertising a node's presence on the network, playing media, recording memos, connecting and breaking down intercom sessions and so on. The diagram below shows the basic architecture of the system.

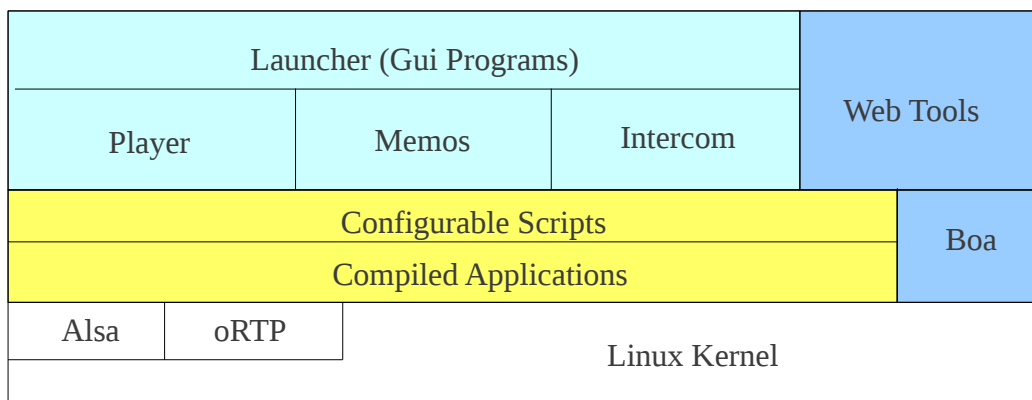


Fig. 1 System Diagram

Component	Installed location
Gui/ Scripts/ Applications	/bin or {remote fs}/bin
Web Tools (main directory for the website and shared data)	{remote fs}/p2p/shared
Other scripts	{remote fs}/p2p
Non-shared data from web config, memos (defined in /etc/nap.conf)	{remote fs}/napdata

A *remote fs* is used because storage requirements needed exceed the on board NAND flash.

The Launcher App

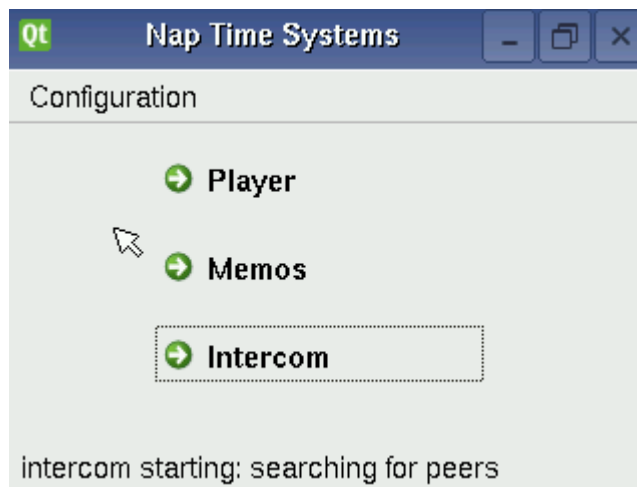


Fig. 2 The launcher app

The launcher gui program is the front end of the appliance. From there an end user can select the player, memos or intercom gui applications. These gui applications do not interact with compiled applications directly. Instead the application start up logic is encapsulated in configurable scripts. This architecture was chosen because there were many unknowns in terms of what application software would work on the EM2440 and how the application software would need to be configured to work best on the hardware. Similarly, a web interface using the boa web server is used to configure various aspects of the system. What is fairly easy for an end user to do in a web browser would be prohibitively difficult in a small touch screen gui. A simple extensible CGI web interface was developed that can be easily extended to provide new configuration features and functionality. This web interface is also used for various tasks which involve communications between nodes in the network such as the transfer of memos between nodes in the network and connecting and disconnecting from an intercom session.

The launcher is meant to start with the system automatically. It takes care of starting up long term processes such as those that enable the node to be found on the network by other nodes (see below) and other periodic tasks that need to run all the time such as checking for new memo messages from other nodes.

The Player App



Figure 3a: The Player Control

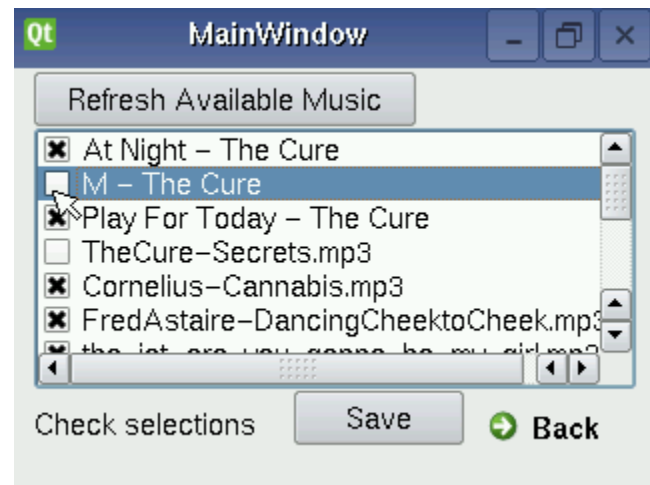


Figure 3b: The Playlist Editor

The player is comprised of two interface components: player control and playlist editor. The player control is initiated by the launcher. The player control provides a playlist of songs from local and remote sources, and the gui buttons to control playing, stopping, and movement up and down through the list. The gui also allows for double tapping a song name as the means to start it playing. The 'list all songs' button brings up the playlist editor, where users add or remove songs from the playlist by checking off their selections and clicking the save button, whereupon they are brought back to player control again. The 'refresh available music' button updates the list of available songs as well as the nodes from which they are served, (implicitly). QT QProcess calls to the madplay music player underlie the gui play control buttons, while song meta-data, (title and artist), are likewise obtained with calls to madplay for local mp3 files.

The player application relies upon the napping / naplistener node discovery method, (described below), to identify peers. Additionally, a simple client/server model serves the task of sending lists of shared music generated by a shell script on the remote node (the server) upon request from the player (the client) which then populates the list of available music. Music added to the playlist is transferred to local SD card storage when played. The *musicscan* client/server and appropriate configurations of BOA allow for remote nodes which are either ARM boards or x86 PCs.

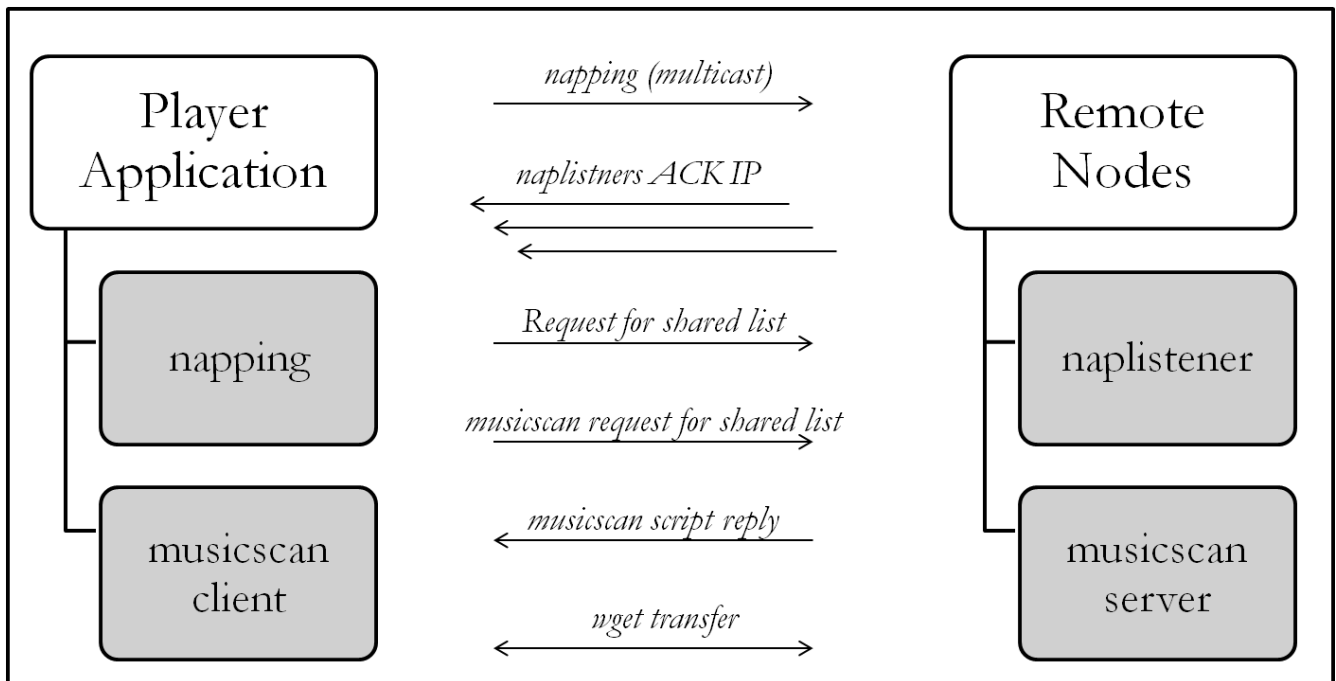


Figure 4: Player Application Process

Madplay Control

Madplay is a command-line MPEG audio decoder and player based on the MAD library(libmad). we tried to use madplay to control the music play operation at first, but madplay could only be controlled from keyboard. Our boards do not come with a keyboard, so basically we have to find another way to replace keyboard.

We looked into the madplay source code and decided to modify the source code so that madplay gets its control command from a named pipe. This way any other process has a way to pause, stop, forward, back, increase and decrease volume, and quit madplay.

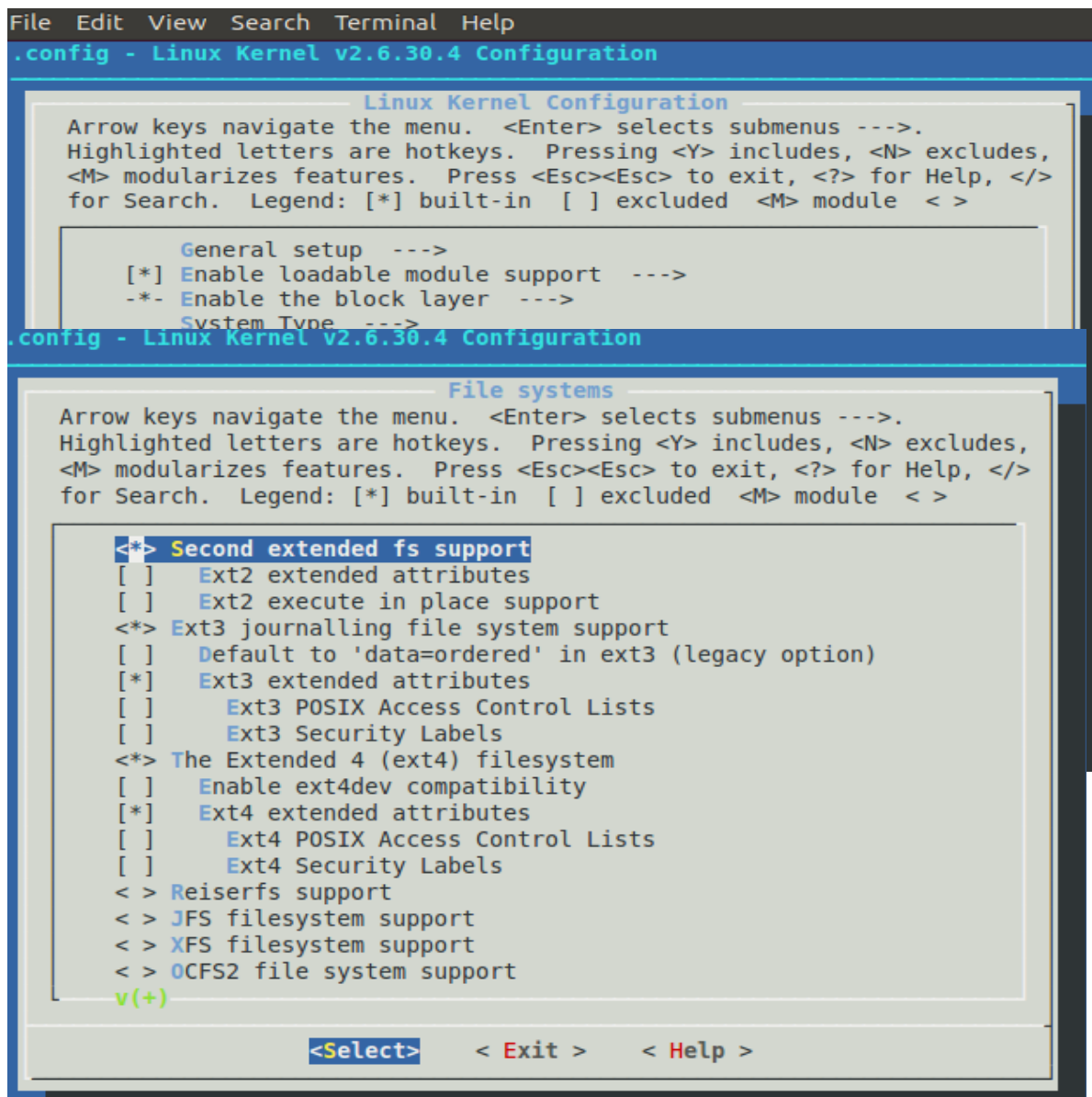
We also programmed the button press to actually write to the named pipe such that pressing left button actually backs the music play, pressing right button forwards the play, pressing up button increases volume, pressing down button reduces volume, pressing up and down button simultaneously pauses the play, pressing up and down button once more would resume the play once the madplay is in

pause mode, pressing left and right button simultaneously quits madplay.

USB Auto Detection

Inserting a USB flash drive into a USB port, only FAT16 and FAT32 file systems can be recognized. One has to either open message buffer of the kernel by running dmesg or read through partition table manipulator by running fdisk -l so that he/she can mount the USB flash drive somewhere manually.

We decided to have our boards to be able to read any file systems of USB flash drive and also are able to auto mount this USB whenever it is being inserted. We configured linux kernel as follows:



```
File Edit View Search Terminal Help
.config - Linux Kernel v2.6.30.4 Configuration

Linux Kernel Configuration
Arrow keys navigate the menu. <Enter> selects submenus --->.
Highlighted letters are hotkeys. Pressing <Y> includes, <N> excludes,
<M> modularizes features. Press <Esc><Esc> to exit, <?> for Help, </>
for Search. Legend: [*] built-in [ ] excluded <M> module < >

General setup --->
[*] Enable loadable module support --->
-* Enable the block layer --->
System type --->

.config - Linux Kernel v2.6.30.4 Configuration

File systems
Arrow keys navigate the menu. <Enter> selects submenus --->.
Highlighted letters are hotkeys. Pressing <Y> includes, <N> excludes,
<M> modularizes features. Press <Esc><Esc> to exit, <?> for Help, </>
for Search. Legend: [*] built-in [ ] excluded <M> module < >

<Y> Second extended fs support
[ ] Ext2 extended attributes
[ ] Ext2 execute in place support
<Y> Ext3 journalling file system support
[ ] Default to 'data=ordered' in ext3 (legacy option)
[*] Ext3 extended attributes
[ ] Ext3 POSIX Access Control Lists
[ ] Ext3 Security Labels
<Y> The Extended 4 (ext4) filesystem
[ ] Enable ext4dev compatibility
[*] Ext4 extended attributes
[ ] Ext4 POSIX Access Control Lists
[ ] Ext4 Security Labels
< > Reiserfs support
< > JFS filesystem support
< > XFS filesystem support
< > OCFS2 file system support
v(+)

<Select> < Exit > < Help >
```

After compiling the kernel and loading it in, our boards now can recognize any file system. We still need to be able to auto mount the usb flash drive. After investigating online, we found two solutions: autofs and udev rule. However, both requires extra services installed. We are really short of space. So we wrote a program which reads partition table, auto mount the flash drive when it is inserted and auto unmount when it is removed.

The Memos App

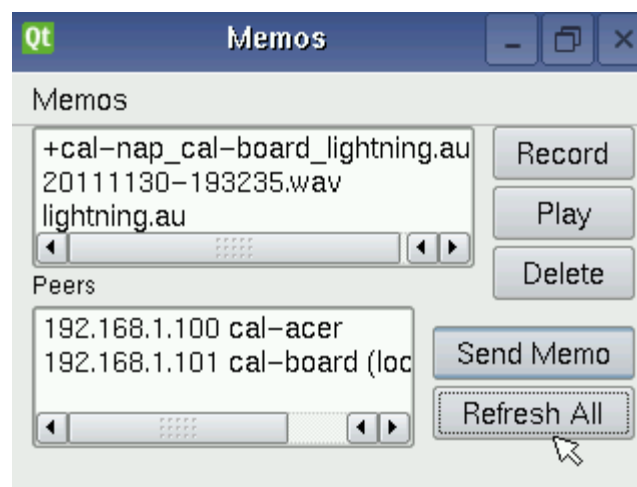


Fig. 4 The memos app

The memos app allows an end user to save, listen to and exchange short sound files called memos that are stored on the appliance itself. These files are named with a timestamp and saved in a node specific location and can be sent to other nodes. When a memo is sent to another node the leds on the target node blink with a series of short blinks equal to the number of memos received by that node. Memos from other nodes are indicated by the name of the node followed by the date and time the memo was recorded.

Node Discovery

How do apps find out about other nodes? We devised a very simple system that uses network

multicast to allow a node to advertise its existence on the network. This system consists of two c applications (which can be found in the napscan directory in the code repository). The first *naplistener* is a multicast server it listens on a configurable port for multicast requests from other nodes. Nodes who wish to find out who else is in their multicast group use the *napping* program to find other nodes on the network. The script *getpeers* encapsulates this behaviour and returns a list of found peers. These peers can then interact with each other via the web interface or other interfaces for various tasks.

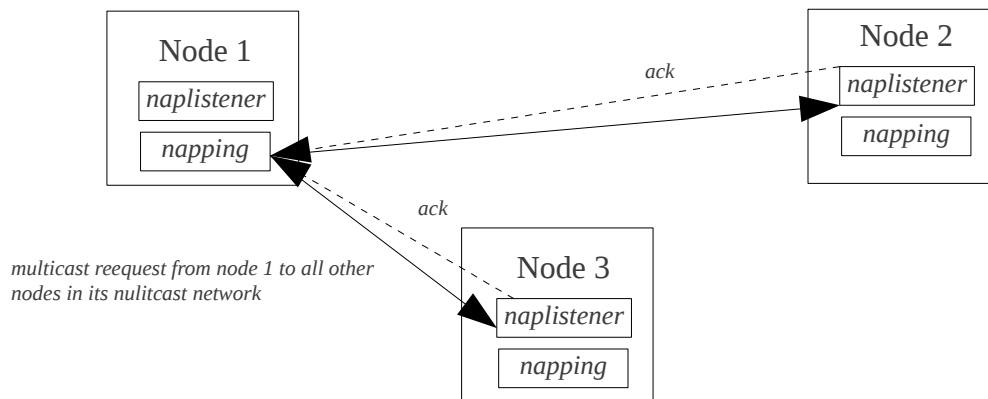


Fig. 5 The node discovery process

By default all nodes use the same multicast port and address but via the web configuration it is possible to partition a large network into smaller subnets by providing custom port and multicast ip addresses for specific groups of nodes. By default the multicast network can only communicate within its local subnet.

The Intercom App

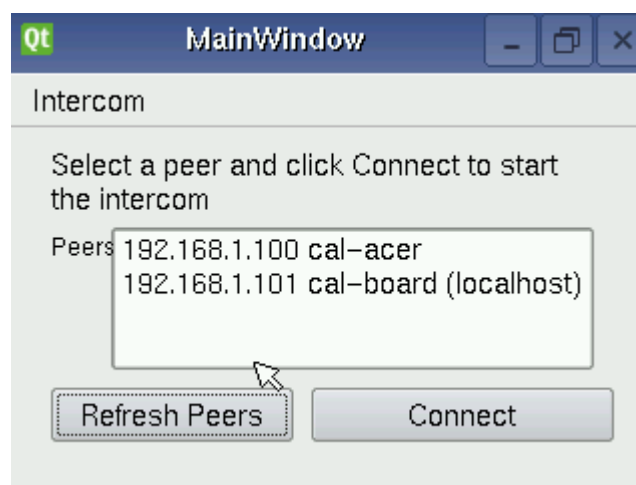


Fig. 6 The intercom app

The intercom app lets an end user select another node initiate an audio conversation with another end user at that node via the network. They would select a peer from the gui form in the figure above and click connect. The target node will then beep to alert anyone nearby that they are in audio communication with the initiating node. The initiating user will then shut down the connection.

The connection is initiated via a web request tied to the “Connect” button in the gui. The request logic is encapsulated in the *naprtptest* script and the *connect.cgi* web script that is invoked on the target node from the initiating node. These, if successful start an RTP server on each node. RTP handles the network latency issues and quality of service of the audio network connections. The output from the RTP servers is then piped through the ALSA aplay program to send the audio to the sound card on the EM2440 board. At the same time audio sender processes on each node takes raw data generated by the ALSA aplay program from the sound card's microphone interface and pipes the raw data and pipe it through an RTP sender process that sends the data to the other node via the network.

The diagram below illustrates the intercom call process:

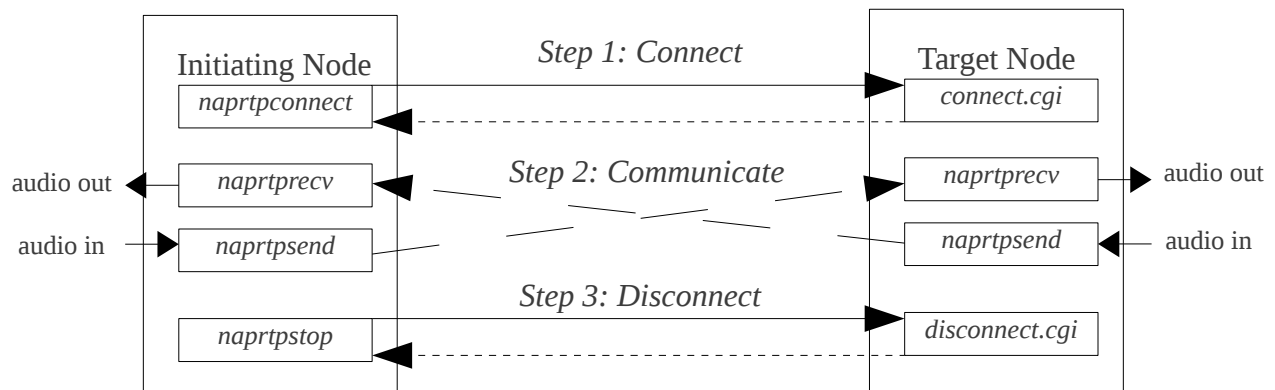


Fig. 7 The intercom process

Extra Hardware & Software Used

Alsa, TCL, oRTP

Explanation of Challenges

Qt build
ALSA problems
Storage issues – broken sd cards etc.

Some difficulty was encountered in the attempt to use SHA1SUM identification of songs as part of an effort to avoid transfer of duplicate songs. An as yet unexplained segmentation fault arose which unfortunately was not successfully resolved—and as a consequence, this functionality is not present in the player. Along with it was lost the meta-data mp3 information from remote node mp3s, which thus are now shown in the player using only filename identifiers.

Acknowledgements

Brian, various online guides.

In addition to the abundant referencing of qt developer information at the nokia website, stackoverflow and hackchina.com provided helpful guidance for coding. Thomas also acknowledges the assistance of Jiahui Xu, a companion and frequent pair programmer – whatever small merit my effort may be credited with must be shared with her, while responsibility for errors and omissions is of course fully my responsibility.