the user a choice depending on his/her connection goals. One unforeseen result of this experiment is the observation that DCCP can be even faster than UDP in Wi-Fi connections, for example.

As observed in our results, other factors besides the transport protocol choice can influence latency, such as the machine setup, the number of network devices and the physical connection. Since repeating these tests is feasible in a short period of time, it would be interesting to add them as a performance measurement feature in a network music tool. With this feature users would be able to test their given connections and to choose the best protocol for the specific scenario at hand.

As future work we intend to test and compare these protocols over Internet connections.

6. ACKNOWLEDGMENTS

A special thanks to Beraldo Leal, Daniel Batista and Stephen Sinclair who helped with ideas, feedback and reviews. The authors would like also to thank the support of the funding agencies CNPq (grant no 141730/2010-2), FAPESP - São Paulo Research Foundation (grant no 2008/08632-8) and CAPES (grant no BEEX 1194/12-7).

7. REFERENCES


ensures that ensemble can be maintained in a work that
eschews traditional notions of rhythm and tempo. [10]

Figure 2. Excerpt from In the Cut (2010), by Cat Hope

LAPTOP ISSUES

There are a great number of advantages to using a tablet over a laptop, especially in a performance environment. For a start, tablets are far more discreet. Since they fit easily into a music stand, there is no need for a laptop that would aesthetically create such a noticeable barrier between the performer and the audience. Additionally they have a much longer battery life, which means that they can safely remain unplugged for the duration of a concert, eliminating the need for any cabling on stage that isn't otherwise required for audio processing. This lack of cabling and their reduced bulk makes it easier for performers to move between stations during a concert if need be. In fact, the only drawback of a tablet compared to a laptop in terms of hardware is the reduced screen real estate that's available. But as long as scores are designed in such a way as to make efficient use of the available screen space, this issue can be overcome.

In addition to these hardware related benefits, there were some other compelling reasons for Decibel to look towards developing purpose built software. Firstly, the limits of using Max/MSP were becoming evident. As the images being used in the scores were increasing in size, animation related performance issues began to arise, such as jittery proceedings of scrolling scores, or crashes. This was mitigated by using a jitter matrix rather than an epic object to store the images, but the result was still not as smooth as what could be achieved using lower level languages, such as Objective C. Networking had also been somewhat problematic in terms of its reliability, although this had been solved to a large extent by the creation of a network monitoring patch programmed by Stuart James. While this allowed for network drop outs to be quickly identified and fixed, it tied the scores even more closely to our specific network configuration. This meant that any other ensemble that wanted to perform these scores would have to either duplicate our network set up exactly, or they would have to manually edit the Max patch. This is far from an ideal situation.

FROM LAPTOP TO TABLET

The process of moving to a tablet computing environment presented quite a few challenges in terms of design and implementation. What Max had lacked in performance it more than made up for with flexibility. This not only made it the ideal prototyping tool, but it meant that every score could be subtly different, as dictated by the needs of the piece or the desires of the composer. But retaining this level of flexibility would have required us to develop our own interpreted language, which is no mean feat. Instead, we selected to organise extant scores into several major categories, with more specific behaviours being controlled by options stored within the score files themselves. This provided the flexibility that we needed, but without the huge complication or performance hit of having to interpret code.

To allow for a wide range of score styles to be implemented, and to allow for the easy implementation of new styles in the future, a modular framework was developed that split the basic player functions from the more specific rendering code. The basic player code was responsible for the user interface and networking, as well as providing a network synchronised, low resolution, split second timer and canvas that could be used by the rendering module. The rendering module was responsible for drawing the necessary images on the canvases, and reacting appropriately to various events such as (play, reset, and seek commands) sent by the player. The interface between the rendering and player modules also provided a conduit for score specific data to be sent and received across the network by the rendering module. This allows for any necessary initialisation data or real time information to be sent across the network for scores that are designed to have a random element to them.

Networking has been one area in particular that has seen a great deal of improvement with our move to the iPad, at least in terms of ease of usage. By using the Bonjour protocol for service discovery, the app is programmed to automatically find and connect to any iPad on the same wireless network that is running the same score. [1] In this way the end user doesn’t need to know anything about the underlying network configuration, which instantly increases the usability of the player for a non-technical audience. Additionally, there are improved levels of redundancy coded into our solution. While the first iPad to load a score becomes the server, the first iPad to connect to this server is configured as a backup server. So if the primary server fails for any reason, the backup server is takes over, and a new client is promoted to take the place of the backup. In this way, it isn’t possible for a single iPad to bring down the entire network.

It was an important design feature of the Decibel Score Player to have the ability to upload new scores to the application’s library. This is achieved using the file sharing feature in iOS, which allows for application specific files to be copied onto the device via iTunes. (This is a slightly more cumbersome task than it really should be at the moment, but unfortunately it’s a limitation of the device at this time.) The score files themselves are simply zip files with their extension altered to .dsz, which contain any image resources needed by the score and an XML file (opus.xml) to describe the parameters of the score. Multiple scores can be described within the same opus.xml file, so scores can be distributed separately or bundled together as a collection of works. When these files are copied to the device, the app extracts the contents of the file to the appropriate internal directory and updates the list of available scores. The advantage of using such a format is that anyone who understands XML syntax is able to easily modify or create new scores without the need for any special software.

CURRENT SCORE STYLES

The initial release of the app ships with four scores, all of which are scrolling scores. However other modules have already been developed and used internally, showing the potential of what can be achieved under the current application framework. While the scrolling score remains the most basic style of score, the code responsible for it has become increasingly flexible and sophisticated as the needs of the ensemble have evolved. It was the first style of score to allow for the display of individual parts, and has also been extended to accommodate incredibly long scores (over 20000 pixels) by loading them as a series of smaller image tiles. There is also some capacity for randomisation. The scores Juana Neilsen (2012) and Limitum (2012) by Cat Hope both contain middle sections where a sequence of short fragments from random locations in the score are presented to the performer. [1]

Figure 4. A screenshot of the scrolling score Longing (2011), by Cat Hope. (The vertical orange line to the left of screen is the playback.)

Despite this possibility of randomisation, scrolling scores remain, for the most part, linear in nature. In an attempt to address this limitation, Hope and Vickery developed a number of other styles of animated scores, and a couple of these have been ported (at least partially) to the score player. Talking Board (2011) takes a large background image, with dimensions several times the size of the available screen resolution, and sets it in motion randomly across the canvas. A number of circles, colour coded for each performer, then move over it as the performers improvise based on the textures framed by their particular circle. Currently the circles move independently of one another, although the player will eventually implement several different behaviour patterns for the circles that were developed in the score’s initial Max patch. These include having the circles move in formation, having them converge on a selected circle, and having one circle increase in size to indicate that the corresponding performer should embrace a more soloistic role. [9]

Figure 5. Talking Board (2011), by Cat Hope and Lindsay Vickery.

The first score to be developed entirely for the score player, Ubahn c. 1985 (2012) by Lindsay Vickery has the players represented by trains that traverse the Berlin U-Bahn network as it was before the fall of the Iron Curtain. The piece ends when all of the players end up

---

[1] A demonstration video of this can be seen at [7]
ensures that ensemble can be maintained in a work that eschews traditional notions of rhythm and tempo. [10]

Figure 2. Excerpt from In the Cut (2010), by Cat Hope

LAPTOP ISSUES
There are a great number of advantages to using a tablet over a laptop, especially in a performance environment. For a start, tablets are far more discreet. Since they fit easily into a music stand and can be used the same way that printed music would, they aesthetically don’t create such a noticeable barrier between the performer and the audience. Additionally they have a much longer battery life, which means that they can safely remain unplugged for the duration of a concert, eliminating the need for any cabling on stage that isn’t otherwise required for audio processing. This lack of cabling and their reduced bulk also makes it easier for performers to move between stations during a concert if need be. In fact, the only draw back of a tablet compared to a laptop in terms of hardware is the reduced screen real estate that’s available. But as long as scores are designed in such a way as to make efficient use of the available screen space, this issue can be overcome.

In addition to these hardware related benefits, there were some other compelling reasons for Decibel to look towards developing purpose built software. Firstly, the limits of using Max/MSP were becoming evident. As the images being used in the scores were increasing in size, animation related performance issues began to arise, such as jittery proceedings of scrolling scores, or crashes. This was mitigated by using a jitter matrix rather than an epic object to store the images, but the result was still not as smooth as what could be achieved using lower level languages, such as Objective C. Networking had also been somewhat problematic in terms of its reliability, although this had been solved to a large extent by the creation of a network monitoring patch programmed by Stuart James. While this allowed for network drop outs to be quickly identified and fixed, it tied the scores even more closely to our specific network configuration. This meant that any other ensemble that wanted to perform these works would have to either duplicate our network setup exactly, or they would have to manually edit the Max patch. This is far from an ideal situation.

FROM LAPTOP TO TABLET
The process of moving to a tablet computing environment presented quite a few challenges in terms of design and implementation. What Max had lacked in performance it more than made up for with flexibility. This not only made it the ideal prototyping tool, but it meant that every score could be subtly different, as dictated by the needs of the piece or the desires of the composer. But retaining this level of flexibility would have required us to develop our own interpreted language, which is no mean feat. Instead, we were able to organise existing scores into several major categories, with more specific behaviours being controlled by options stored within the score files themselves. This provided the flexibility that we needed, but without the huge complication or performance hit of having to interpret code.

To allow for a wide range of score styles to be implemented, and to allow for the easy implementation of new styles in the future, a modular framework was developed that split the basic player functions from the more specific rendering code. The basic player code was responsible for the user interface and networking, as well as providing a network synchronised, low resolution, split second timer to maintain canvas that could be used by the rendering module. The rendering module was responsible for drawing the necessary images on the canvas, and reacting appropriately to various events (such as play, reset, and seek commands) sent by the player. The interface between the rendering and player modules also provided a conduit for score specific data to be sent and received across the network by the rendering module. This allows for any necessary initialisation data or real time information to be sent across the network for scores that are designed to have a random element to them.

Networking has been one area in particular that has seen a great deal of improvement with our move to the iPad, at least in terms of ease of usage. Since the Bonjour protocol for service discovery, the app is programmed to automatically find and connect to any iPad on the same wireless network that is running the same score. [1] In this way the end user doesn’t need to know anything about the underlying network configuration, which instantly increases the usability of the player for a non-technical audience. Additionally, there are improved levels of redundancy coded into our configuration, which instantly increases the usability of the player for a non-technical audience. Additionally, there are improved levels of redundancy coded into our solution. While the first iPad to load a score becomes the server, the first iPad to connect to this server is configured as a backup server. So if the primary server fails for any reason, the backup server is takes over, and a new client is promoted to take the place of the backup. In this way, it isn’t possible for a single iPad to bring down the entire network.

It was an important design feature of the Decibel Score Player to have the ability to upload new scores to the application’s library. This is achieved using the file sharing feature in iOS, which allows for application specific files to be copied onto the device via iTunes. (This is a slightly more cumbersome task than it really should be at the moment, but unfortunately it’s a limitation of the device architecture.) The score files themselves are simply zip files with their extension altered to .dsz, which contain any image resources needed by the score and an XML file (opus.xml) to describe the parameters of the score. Multiple scores can be described within the same opus.xml file, so scores can be distributed separately or bundled together as a collection of works. When these files are copied to the device, the app extracts the contents of the file to the appropriate internal directory and updates the list of available scores. The advantage of using such a format is that anyone who understands XML syntax is able to easily modify or create new scores without the need for any special software.

Figure 3. The opening screen of the app, showing the score library.

Unfortunately, while new scores of extant styles are fairly trivial to create, adding new styles of score to the app is beyond the ability of the end user. While the modular framework of the score player does allow for new styles to be coded in a rather straightforward way, the iOS platform does not allow for code to be dynamically linked at runtime. This means that the code for any new score style has to be compiled into the app itself by Decibel and released as an upgrade to the player.

CURRENT SCORE STYLES
The initial release of the app ships with four scores, all of which are scrolling scores. However other modules have already been developed and used internally, showing the potential of what can be achieved under the current application framework. And while the scrolling score remains the most basic style of score, the code responsible for it has become increasingly flexible and sophisticated as the needs of the ensemble have evolved. It was the first style of score to allow for the display of individual parts, and has also been extended to accommodate incredibly long scores (over 20000 pixels) by loading them as a series of smaller image files. There is also some capacity for randomisation. The scores Juanita Neilsen (2012) and Liminum (2012) by Cat Hope both contain middle sections where a sequence of short fragments from random locations in the score are presented to the performer.

1 A demonstration video of this can be seen at [7]
trapped in East Berlin, unable to return to the west once they’ve made the one way journey over the Berlin Wall. When this happens, the music gradually transitions into a much of the difficult processing work was implemented in real time at the touch of a button on the screen. Since this is randomly generated (ie Talking Board). The way this is achieved, the user can't causes the player to hide unnecessary user interface elements such as the play button and location slider. Finally, the badPreferencesFile method is used to inform the player that a preference file associated with the score is corrupt or incorrectly formatted. Since the player can’t have any way of knowing about the specific requirements of the player to free any manually allocated memory, and to release any goodPreferencesFile method is called to inform the player that a score is essentially unplayable, and that the user should generates a specific amount of playback data based on the length of the score. The rotate method is called to notify the renderer that the user has rotated the iPad. This allows the rendering code to scale any images as required by the change of resolution that accompanies a change in orientation. The final methods used to relay user interaction to the rendering code are the swipeUp and swipeDown methods. These notify the renderer that the user has made a swipe gesture across the canvas. Their most common use currently is to change which part is displayed where a score style allows for the display of individual parts in addition to a full score. The remaining optional methods are for timing and network purposes. The tick method is called whenever the timer of the genericClock fires. While most rendering classes will generally want to implement their own high resolution timers for animation purposes, this method provides access to a single, split second clock that is shared by both the control and rendering code. This is useful as a means of correcting any timer drift and ensuring that no synchronisation issues occur between different parts of the code. The receiveMessage method is called to notify the renderer that the player has received network data for it. This method can then be used to react to that data in an appropriate manner. Finally, the becomeMaster function notifies a renderer that the current iPad has been promoted to be the primary server. This need only be implemented by scores where the server is responsible for generating real time data to be used by the clients.

The RendererMessaging Protocol

```
@protocol RendererMessaging <NSObject>
@required
- (BOOL)sendData:(NSMutableArray *) message;
- (void)setStaticScore;
- (void)badPreferencesFile;
@end
```

While the RendererDelegate protocol allows for the player code to communicate with the renderer class, the RendererMessaging protocol allows for communication in the other direction. The sendData method allows the renderer to send data across the network, and is the complementary method to the receiveMessage method of the RendererDelegate protocol. The setStaticScore method is used by a renderer to notify the player that the particular style of score that it implements is static. This causes the player to hide unnecessary user interface elements such as the play button and location slider. Finally, the badPreferencesFile method is used to inform the player that a preference file associated with the score is corrupt or incorrectly formatted. Since the player can’t have any way of knowing about the specific requirements of the player to free any manually allocated memory, and to release any goodPreferencesFile method is called to inform the player that a score is essentially unplayable, and that the user should
The network is defined by an XML file containing a map of all the relevant coordinates as they relate to the supplied image file. And work is being undertaken to allow for different conditions to be used to determine when and how the score should end. Work has also commenced on moving Decibel's digital adaptations of the John Cage Variations from Max to the iPad using the score player's framework. The first code to be ported to the platform was that for *Variation III* (1962). In its original form, the performer would create the “score” for a performance by dropping 42 transparency circles onto a surface. The largest group of touching circles would become the score, with intersections between the circles representing events as the performer traversed it at their discretion. [3] The computerised version mimics this process, but the score is generated in real time at the touch of a button on the screen. Since much of the difficult processing work was implemented in the original Max patch as a Java extension, it was a relatively straightforward task to translate it to Objective C. [14]

The RendererDelegate Protocol
@protocol RendererDelegate <NSObject>
@required
- (id)initRenderer:(Score *) scoreData :
- (void)close;
- (void)reset:(BOOL ) master;
@optional
- (void)play;
- (void)seek:(CGFloat) location ;
- (void)changeDuration:(CGFloat) location : (BOOL) atLoad;
- (void)tick:(int) progress of : (int) progress tock: (int) splitSecond;
- (void)swipeUp;
- (void)swipeDown;
@end

Any class that implements a specific style of score must conform to the RendererDelegate protocol. At a minimum, the class must have an initialisation method, a close method and a reset method. (These can be seen under the @required section of the protocol declaration.) When the player first loads a score, it checks to see which renderer class is required to animate it and creates an instance of the relevant class. When it does this, the inifitRenderer method is called. As can be seen by the method declaration, three object pointers are passed to this method: scoreData is an object that contains the parameters of the current score, PlayerCanvas is a reference to the low resolution image file that is used to render the score, and genericClock is a reference to the low resolution timer provided by the player. The close method is called when the player shuts down, and is used by the renderer to free any temporary allocated memory, and to release any object references. The reset method is called whenever the player receives a reset message over the network. (This is generated in response to the user hitting the reset button on the score player.) It is also called when the score is first loaded to draw the initial image on screen. As can be seen, this method takes one parameter: the boolean master, which informs the rendering class as to whether the iPad is the server or a client on the network. This way the renderer class can decide whether to generate new data by the score, or to send a request for such information over the network.

The rest of the methods are optional and can be implemented either not based on the requirements of the specific score style. The `play` and `seek` methods are also called as a result of user interaction with the score player, and these are usually implemented unless a score is static by nature (ie some of the Cage Variations), or unless seeking is a relatively meaningless concept where a score is randomly generated (ie Talking Board). The `changeDuration` method is called, as the name suggests, when the user changes the duration of a score. This generally only needs to be implemented by a renderer that generates a specific amount of playback data based on the length of the score. The `rotate` method is called to notify the renderer that the user has rotated the iPad. This allows the rendering code to scale any images as required by the change of resolution that accompanies a change in orientation. The final methods used to relay user interaction to the rendering code are the `swipeUp` and `swipeDown` methods. These notify the renderer that the user has made a swipe gesture across the canvas. Their most common use currently is to change which part is displayed where a score style allows for the display of individual parts in addition to a full score.

The remaining optional methods are for timing and network purposes. The `tick` method is called whenever the timer of the `genericClock` fires. While most rendering classes will generally want to implement their own high resolution timers for animation purposes, this method provides access to a single, split second clock that is shared by both the control and rendering code. This is useful as a means of correcting any timer drift and ensuring that no synchronisation issues occur between different parts of the code. The `receiveMessage` method is called to notify the renderer that the player has received network data for it. This method can then be reused to react to that data in an appropriate manner. Finally, the `becomeMaster` function notifies a renderer that the current iPad has been promoted to be the primary server. This need only be implemented by scores where the server is responsible for generating real time data to be used by the clients.

THE FRAMEWORK
As mentioned earlier, different rendering modules can be added to the app to allow for a wide range of score styles to be played. This is made possible in code by a feature of the Objective C programming language known as protocols. Protocols define a basic template that classes can implement and extend upon. To conform to a given protocol, a class must implement code for all of the required methods, and can choose to provide code for any of the optional methods listed. [2] While the app uses a number of custom protocols to facilitate communication between various classes, there are two main protocols which allow for the relatively clean separation of the control and rendering code. These are detailed below.

The RendererDelegate Protocol
@protocol RendererDelegate <NSObject>
@required
- (id)initRenderer:(Score *) scoreData :
@end

So that the code can be reused to create other scores that involve the movement of players across a network, the network is defined by an XML file containing a map of all

![Figure 6. Ubahn c. 1985 (2012), by Lindsay Vickery. This is the map seen by the audience.](image)

The players, on the other hand, each see a zoomed in version of the map centred on their particular “train.” As can be seen in the screenshot below, the train lines have become musical staves, with various notes or effects to be played at each stop along the way. (The current location is pinpointed by the red dot in the centre of the screen.) When the train arrives at a junction, it pauses for a longer time than usual, and a fragment of musical material appears on screen. This is written to be far more solocistic than the textual material that exists at the regular stations, and so different instruments are brought to the fore at random by the journey of the trains across the network.

![Figure 7. Ubahn c.1985 as seen by the performers.](image)

![Figure 8. Variation III (1962), by John Cage. The clear circles are the largest touching group. The blue circles are discareded and fade out after a few seconds.](image)

204 | 2013 ICMC Idea | LONH FIFERS
be informed. This method should be called by the renderer at initialisation if any preference file fails to load properly.

CREATING A SCORE FILE

As previously mentioned, a score file is little more than a renamed zip file. Importantly though, as well as any necessary graphic resources, it must contain an opus.xml file. The specific format of this XML file is defined by a custom Document Type Definition, which is attached at the end of this paper. (See Appendix A – opus.dtd). Below is the opus.xml file for a sample score.

A Sample Score

```xml
<?xml version="1.0" encoding="UTF-8"?>
<opus>
  <filename>IntheCut.png</filename>
  <duration>480</duration>
  <type>ScrollScore</type>
  <composer>Cat Hope</composer>
  <name>In the Cut</name>
  <startoffset>1500</startoffset>
  <filename>IntheCut.png</filename>
</opus>
```

Figure 9. An early version of the ScoreCreator OS X application.

The biggest additional feature would be to release a port for the Android platform to allow the app to run on a greater variety of tablet devices. This would also be the biggest programming endeavour, since it will require the code to be translated into Java, with Apple specific frameworks being replaced with the relevant Android ones. Currently, the network protocol used by the app is entirely dependent on Apple specific data structures, so this would also have to be reworked to be platform neutral. And the diversity of Android hardware means that we will have to test the app on a far greater number of devices.

APPENDIX A – OPUS/DTD

This is the Document Type Definition that should be followed in the creation of an opus.xml file:

```xml
<?xml version="1.0" encoding="UTF-8"?>
<opus>
  <name>In the Cut</name>
  <composer>Cat Hope</composer>
  <filename>IntheCut.png</filename>
  <duration>480</duration>
  <type>ScrollScore</type>
  <startoffset>1500</startoffset>
</opus>
```

REFERENCES


be informed. This method should be called by the renderer at initialisation if any preference file fails to load properly.

Creating a Score File

As previously mentioned, a score file is little more than a renamed zip file. Importantly though, as well as any necessary graphic resources, it must contain an opus.xml file. The specific format of this XML file is defined by a custom Document Type Definition, which is attached at the end of this paper. (See Appendix A – opus.dtd). Below is the opus.xml file for a sample score.

A Sample Score

```xml
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE opus SYSTEM "opus.dtd">
<opus>
  <score><name>In the Cut</name><composer>Cat Hope</composer><type>ScrollScore</type><filename>InTheCut.png</filename><startoffset>1500</startoffset><duration>480</duration></score>
</opus>
```

Most of the parameters listed in the opus.xml file are fairly self explanatory. The name and composer fields contain the title of the score and the composer’s name in exactly the way in which they are to appear within the app. The type specifies the style of score, and in this case the score is a basic scrolling score, while the duration determines how long the score should run for in seconds. For a scrolling score, this affects the rate at which the image is moved across the screen. The final two parameters are related to the image itself. The filename is the name of the image to be used, while the startoffset specifies the size of the lead in area in pixels. This allows for instructions to be placed at the left hand edge of the score without them interfering with playback.

The final score file can then be created with the following terminal command, executed in the appropriate working directory (containing both the XML and image file):

```
zip "In the Cut.dsz" opus.xml IntheCut.png
```

Further Developments

While the app has shown itself to be generally stable, and has performed well in concerts to date, there are still plenty of features that are slated for future releases. Some of these should be quite trivial to implement, while others are much more involved and will require significant development time. Currently the score player deals purely with the task of animating the scores, but in the future it will hopefully have the ability to communicate with other devices and applications on the network using OSC, so that electronic processing and effects can be automatically triggered where appropriate. The main drawback to this is that some of the advantages of zero config networking will be lost, as the user would need to set the IP address or hostname of the machine doing the processing. One workaround for this, at least where Max/MSP is being used, would be to create a Max external that advertises its presence via Bonjour. This could then be used within the audio processing patch to alert the score player of the appropriate network address to send data to.

There are two features in particular that would make the app much more extensible for the end user. The first is a companion Score Creator application, which would automate the creation of score files from parameters and images supplied by the user, reducing the level of technical knowledge required to produce a working score. An early alpha version of this currently exists for Mac OS X (see screenshot below), although at present this is limited to creating basic scrolling scores. The creation of an in-app store would also provide a central repository from which additional scores could be easily purchased. This would then become a convenient distribution channel for any other composers who decided to embrace it.

Figure 9. An early version of the ScoreCreator OS X application.

The biggest additional feature would be to release a port for the Android platform to allow the app to run on a greater variety of tablet devices. This would also be the biggest programming endeavour, since it will require the code to be translated into Java, with Apple specific frameworks being replaced with the relevant Android ones. Currently, the network protocol used by the app is entirely dependent on Apple specific data structures, so this would also have to be reworked to be platform neutral. And the diversity of Android hardware means that we will have to test the app on a far greater number of devices.

Appendix A – Opus/DTD

This is the Document Type Definition that should be followed in the creation of an opus.xml file:

```xml
<ELEMENT opus(score+)

<ELEMENT score (name, composer, type, variation?, duration, startoffset?, readoffset?,

References


[13] Vickery, L. Personal communication with the composer.

