The design of Movino - S60 phone client, OS X components and video server

Martin Storsjö

March 29, 2007

Contents

1 **Overview** **1**

2 **The S60 client** **2**
2.1 Video APIs on S60 2
2.2 Video buffering 4
2.3 Audio buffering 4

3 **The core library** **4**
3.1 Audio/video synchronization 5
3.2 Video stream format assumptions 5

4 **The OS X application** **6**
4.1 The QuickTime component 7

5 **The video server** **7**

1 Overview

The Movino application suite consists of four different components: a client for S60, a GUI application for OS X, a QuickTime component for OS X and a video server for Linux. A J2ME client can also be used instead of the S60 client.

The dataflow between these components is depicted in figure 1, giving a rough illustration of how they can be connected.

The main connectivity properties of the componens are as follows:

- The S60 client can send one stream of data over either Bluetooth or TCP/IP (over WLAN, 3G or GPRS).
- The OS X GUI application can receive one stream over Bluetooth or TCP/IP and relay it over TCP/IP. The stream could thus be chained between many computers running the OS X application.
- The QuickTime components receives one data stream each from the running Movino GUI application over unix domain sockets.
- The video server can receive many streams at the same time, over TCP/IP.

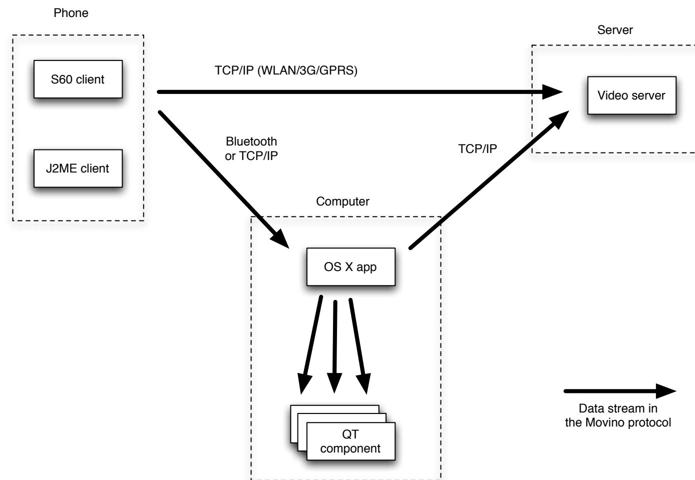

Figure 1: The dataflow in Movino

All these components communicate using the movino protocol. Even though the same content stream can be forwarded over many connections, each connection in the chain is still considered a completely separate session. Only the actual content is forwarded, but e.g. authentication data and other non-content packets are local to the session. All the necessary details are available in the movino protocol documentation.

All components are designed to be as usable as possible without requiring the other components. E.g., the S60 client can send data straight to the video server if it has access to the internet. The OS X GUI application and QuickTime component can be used even though no video server is available. The video server can be used without any drupal web front end.

This document is a collection of notes on the implementation of these components; the main design decisions, trade-offs and their rationale etc.

2 The S60 client

An overview of the data flow routes and encoding alternatives is given in figure 2.

2.1 Video APIs on S60

The initial intention was to use the built-in video codecs, which could utilize the various accelerators on the phone. However, the APIs proved troublesome. In order to stream the video in real time, one would want to encode individual video frames (or small bundles of frames) and receive the encoded bitstream in small packets suitable from the codec. The APIs only seemed to support recording of complete videos into files, and the functionality was not well documented. In addition, one could not assume the existence of any particular codec. The level of detail offered by the API was that recording of video could be started

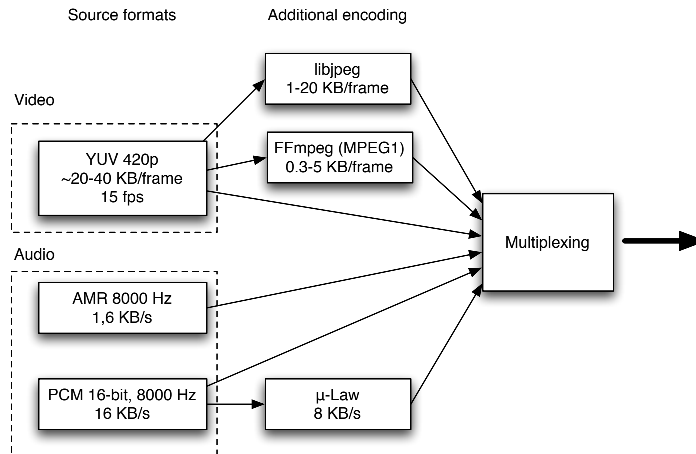

Figure 2: Overview of the data flow possibilities in the S60 client. The formats in the "source formats" column are provided by the underlying APIs (possibly utilizing hardware accelerators), the conversions in the "additional encoding" column are done in software within the Movino S60 client.

(specifying a file name to store the video in), stopped, and recording parameters could be set. The actual codecs available also varies between different S60 phones, especially between different generations of them.

On the other hand, the low-level camera API worked very well, and was well documented. Using this API, the application gets a callback with the image data for every frame captured from the camera (e.g. 15 times per second). The image data can be retrieved in a few different formats (with support varying between phones); we chose to use YUV420p, which seemed to be well supported on all tested phones.

We did a test implementation using the built-in codecs by recording a few seconds of video into a file, stopping the recording and immediately starting recording into another file, and then sending the whole recorded file. If more effort would have been spent on this solution, we might have achieved something relatively usable, but we chose to focus on the low-level camera API instead, which provided a better trade-off between functionality and flexibility.

Since the low-level camera API provides uncompressed data, we had to use completely software based compression without any support from the built-in accelerators. We mainly use libjpeg[1], which compresses the frames fairly well and fairly quickly. (On a Nokia 6630, it seems to manage to compress 15 frames per second in 176x144, where 15 is the total amount of frames provided by the camera.)

We also invested quite some effort in compiling FFmpeg[2] for Symbian, in order to get an MPEG1 encoder, which we also achieved. The encoder is relatively slow (capable of encoding about 5-10 frames per second in 176x144 on a Nokia

[1]libjpeg is open source, and is compileable for Symbian with little effort

[2]FFmpeg is an open source library consisting of codecs and muxers/demuxers for lots of different formats

6630), but can be usable in some cases, e.g. when the available bandwidth is very limited.

2.2 Video buffering

When encoding a video stream without considering the transport network, and the output exceeds the available bandwidth, there will inevitably be increasing delays in the video in the receiving end. Therefore, one would have to constantly adjust the stream bitrate in order to match the available transport capacity.

When manually encoding the frames, one can easily overcome this limitation, by adjusting the framerate instead of the encoding quality.

On Symbian, the libraries provides very little socket buffering; while one block is being sent, one can't send another. This is handled through the normal Symbian framework for asynchronous events. Therefore, one have to implement the socket buffering on top of this. This gives the opportunity to get a lot of information about the sending buffer.

Therefore, when an encoded frame is sent, the send buffer position is stored. When the camera provides the next frame, the send buffer is queried. If the previous frame has been sent, this frame is encoded and buffered for sending, otherwise the new frame is just discarded. In this way, only one single frame can be in transport, and the frame rate automatically adapts to the available network.

If the encoding of the frame takes too long, the camera API just skips the frames which were captured during the callback call. Therefore, this method also adapts well to different levels of processing power.

2.3 Audio buffering

Due to the nature of audio, the effect of a dropped packet of audio is much more disturbing than the variable framerate. Therefore, all recorded audio data is buffered, as long as the total send buffer length is below a threshold value. Since the audio bitrate is relatively low (below 2 KB/s for AMR encoded audio and 8 KB/s for μlaw encoded audio), this still works quite well, as long as there is enough bandwidth. If the send buffer grows too big, there isn't any particularly good solution except just skipping some audio data - otherwise the buffer would grow endlessly.

3 The core library

The same core library is shared between the OS X GUI application, the QuickTime component and the video server. It consists of a main demultiplexing module, which unpacks the received data stream and gives the image and audio data to different classes which manage the decompression and buffring of that data.

Many parts of the core library can be enabled/disabled at compile time, mainly components interfacing with different external libraries, e.g. FFmpeg, Ogg/Vorbis/Theora, the reference AMR codec from 3GPP, OS X built-in AMR codec.

Additionally, the core library contains different utilities shared by the components using the library, but which doesn't belong to the stream unpacking chain, e.g. a simple HTTP server (for serving video streams and XML info feeds).

3.1 Audio/video synchronization

Audio and video can be synchronized, if both of them are used at the same time. This requires both that audio is provided from the client in the data stream, and that the receiving application also uses the received audio (either by playing it back or consumes it from the audio buffer). This automatically is the case in the GUI application and the video server, but no synchronization is done in the QuickTime component if only video is used.

The video decoding class contains a buffer of a fixed number of previous decoded video frames, with their respective timestamps. Code using it can request the latest video frame or request the frame closest to a given timestamp.

The audio decoding class contains a circular buffer containing the received audio data which hasn't been consumed yet. If data is requested from the audio class but the buffer doesn't contain enough data, the class notices that the buffer has been underrun, and thereafter won't return any of the received data until more data has been received, exceeding a threshold value (500 bytes of uncompressed audio data). This is done to try to avoid subsequent buffer underruns, if the consuming of buffered audio data starts immediately when the first packet has been received.

In addition to the circular buffer, the audio decoding class also contains a queue of timestamps and their corresponding stream positions. When a chunk of data is consumed from the buffer, a range of timestamps is returned, rougly corresponding to the range of audio data returned. This range can then be used for requesting a video frame corresponding to that audio data block.

If the client isn't providing any audio data at all, no synchronization is done.

3.2 Video stream format assumptions

Construction of streams using a stored header and skipping earlier data packets

The usage of routines for encoding and multiplexing a video stream in a more widely supported format could permit usage of almost any container format and codec, but one important assumption is made. This assumption is that when a multiplexed video stream is created, it consists of one consecutive block of header data, and the rest of the stream is some kind of data blocks. It is assumed that a usable video stream can be constructed by taking the header and directly appending later consecutive data blocks.

As an example, the original multiplexed video stream is "H0123456789", where each letter refers to a block, H for the header block and the numbered blocks for the sequential data blocks. In this case, it is assumed that a decoder should be able to decode and synchronize to a stream consisting of "H56789". That is, any number of data packets can initially be skipped, but every data packet following the first included one must be present.

For audio/video codecs where the decoding depends on earlier frames or packets, this implies that the decoder can manage to skip data until it manages to sync with the received data. It also implies that some potential sequential

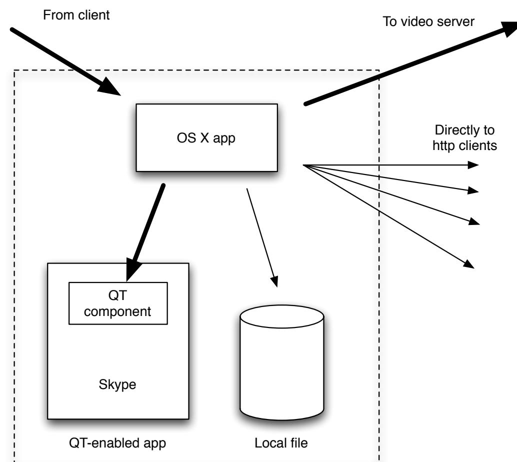

Figure 3: Overview of the OS X application

numbering in packets should be allowed to start from any number. Additionally, it requires that the encoding/multiplexing code only provides whole, complete parseable packets to the upper layers.

These assumptions are due to the way the serving of live streams is done. When the video server receives a data stream, it starts multiplexing a video stream out of the contents. It stores the header block, and if the stream is set to be archived, starts storing everything into a file. Whenever a client connects to view the stream, the client is sent the stored header, and all new data blocks encoded after that. This way, the server only needs to encode and multiplex the stream once, no matter how many viewers there are.

The same principle is also used if the archiving initially is disabled. The stream is encoded, but perhaps not really stored to disk. When archiving is enabled, the stored header block is written to a file, and the following data blocks are written there, too.

(As a possible improvement, one perhaps would like to remultiplex the stream for every receiver/destination, but still only do the encoding once.)

This is currently implemented for both MPEG1 (although not currently used anywhere) and for Ogg/Vorbis/Theora. The header block for streams containing Ogg/Vorbis/Theora actually contains several ogg pages, containing several ogg packets. But to the rest of the framework, it still is presented as one single opaque header block.

4 The OS X application

The OS X application both works as a simple previewer of the streamed content, has some server functionality (sharing some code with the real video server), can be used to forward the data stream to a video server (or to the OS X application on another machine) and acts as a server for the QuickTime components. An overview of it is provided in figure 3.

The main use case is that it decodes the received data and displays the video (and plays back the audio). It listens for connections from clients on both a TCP/IP port and on Bluetooth. While a client is connected, no more connections are accepted.

It also listens for connections on a unix domain socket, which is used for forwarding data to the QuickTime components. This makes the QuickTime components dependant on the gui application, but decouples the client connection from the host application using the QuickTime component. If the QuickTime component would handle the connection from the client itself, handling the connection smoothly would be much more difficult.

There is a trade-off between the amount of data to pass around between processes and the amount of decoding needed in both the application and the component. Currently, all video formats except JPEG and all audio formats are forwarded uncompressed.

If the user selects to forward the content stream to another machine, all data is forwarded in the initial compressed form. A potential improvement would be to re-encode the data into some more efficient format, but that might also degrade the quality slightly even more.

The application also can store the stream into a file, or act as a simple http server, serving the stream live to viewers (e.g. VLC). In this case, the server URL and port has to be manually communicated to the viewers.

4.1 The QuickTime component

Since the video stream can change size without prior notice, this isn't really well suited for QuickTime video devices, which can't change size spontaneously. The quick and dirty solution to this problem is that multiple video input devices are provided, one for each video size. Only the device with the correct size actually return any data, the other ones simply remain empty.

Initially three sizes are provided, 128x96, 176x144 and 352x288. If the GUI application received video frames of a size currently not in the list of sizes, it is added. This should be transparent to the user, as long as some video data of the needed size has been received before the QuickTime host application enumerates the video devices. (The GUI application provides controls for adding sizes not yet received and for removing unnecessary sizes.)

This list of video sizes is stored in the GUI application's property list file, which the QuickTime component reads when opened.

Examples on creating a video device has been taken from the Macam project. No good examples or documentation on creating a sound input device were found; it is implemented mainly by trial and error. Both parts of the component are very minimally implemented, many component interface functions still are unimplemented, but it seems to be enough to work sufficiently in the applications it was tested in.

5 The video server

The video server functionality is drafted in figure 4.

For each connected client, it spawns a new thread, handling that client. The server thread writes the stream content into a file in the archive (unless

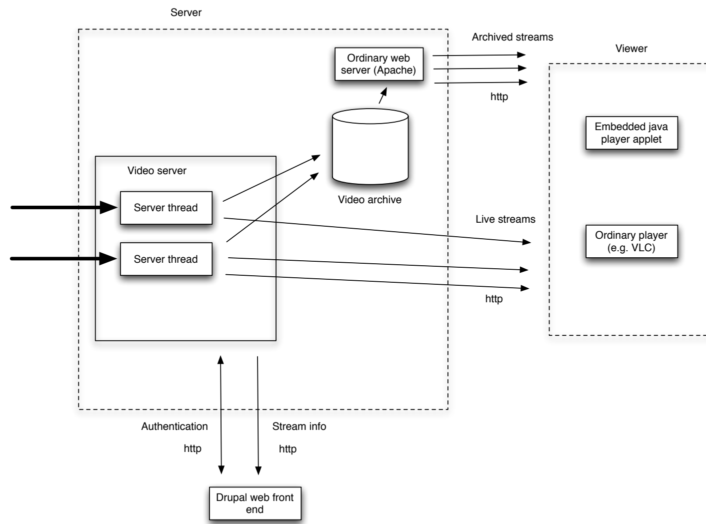

Figure 4: Overview of the video server data flow

archiving explicitly is disabled by the client), and runs a http server providing a live stream of the content from this client. If the stream is archived, a preview image is taken a few seconds after the video stream has started, and stored in the archive directory.

The live streams are served by a built-in simple http server, but the archived content is stored in a directory, which is assumed to be available online through an ordinary webserver. (This decision was made to allow for a much simpler implementation of the internal http server.)

The video server also has another http server running all the time, which serves xml feeds with information about the current live and archived streams. The drupal web front end uses this, but it is human-readable and easy to parse (if one would want to implement another front end).

For client authentication, one can either user a plaintext password list or set the server to contact the movio drupal module. The authentication is a challenge/response scheme based on the MD5 hash algorithm; the password is never transmitted in plaintext. (MD5 was chosen since the drupal user database stores its passwords as MD5 hashes, one would need an MD5 algorithm in the code base anyway.) The details on the authentication scheme can be found in the movino protocol documentation.

One slight drawback in the authentication scheme is that the MD5 hashes in the drupal database become critical information, a rogue client with access to those hashes could get access to the video server without actually knowing the plaintext password.

To overcome this limitation, the plaintext password would need to be transported to the drupal authentication module in some way, which probably would be much easier to intercept (unless it's e.g. SSL-encrypted) than getting access to the drupal user database.