Media Capture from iOS Devices Using Distributed, Remote Control

# Abstract

iOS devices have at least one camera and a microphone as well as media control frameworks available to app developers. The main goal of this project was to investigate the possibility of remotely initiating video feeds from iOS devices and of capturing the resultant streams at the server for later playback. Part of this remote-sensing study was to find out if this can even work in principle. In addition to answering the basic questions of device controllability and video capture, the end goal was to deliver a working prototype that can be demonstrated and used as the basis for further investigations. This document details the findings of the study and describes an integrated system of command, control, and data capture. The source code for all elements of this architecture is included as part of the project deliverable. The delivered documentation includes instructions for setting up a SW development environment for builds, code deployment and system testing. The test configurations include both automated unit tests and UI-based end to end tests.

# Findings

The following concrete results were discovered and/or demonstrated:

* Audio/Video remote monitoring is possible and is demonstrated by the delivered code.
* Video and audio capture are programmatically configurable in iOS apps based on the MonoTouch framework and iOS 5.x
* The media feeds can be segmented into small slices and the resulting capture buffers can be sequentially uploaded to a REST-based web service for reassembly by a monitoring process.
* A central monitor can be hosted in IIS 7.5 running on Windows 7 that can act in three separate but critical roles:
  1. Web Service endpoint for ***sensors***. The monitor process accepts web service calls from the sensors and uses these calls to accept stream fragment uploads as well as command polling calls.
  2. Web Service endpoint for ***controllers***. The monitor process can deliver sensor status information to controllers and can accept start/stop commands for any running sensor.
  3. Media Controller. The monitor coordinates stream fragment uploads and command dispatches to the sensors. The stream processing module aggregates fragments into 1 minute ‘sensor movies’ and stores them for later playback.
* A distributed network of controller nodes and sensors can coordinate with the central web service. This architecture provides maximum flexibility in building future sensor-management solutions.
* The resulting media streams suffer from splicing artifacts due to the iOS capture limitations. The fragments cannot be captured from the iOS media frameworks in a ‘frame accurate’ manner. There is always a small lag between one fragment ending and another fragment recording starting up. Additionally, the captured data fragments are in H.264 video format and the pieces cannot simply be concatenated at the server. An open source tool called FFMPEG is used to do the fragment splicing and it works well for video but leaves the audio slightly out of sync. These limitations arise from the fact the .mov files created by iOS cannot be pre-allocated so you cannot employ a rotating window approach to grabbing ‘frame accurate’ samples.
* Future investigations might consider the following alternative capture/reassembly strategies:
  1. Use the iOS device as an HTTP streaming source. Apple has an RFC for HTTP streaming and that protocol would have to be implemented with the iOS device as a web server. The web service monitor would have to accept a sensor’s HTTP host/port/scheme tuple in an API call and then become the client of the device.
  2. Use the current approach but consider using very long capture intervals for the fragments. This would cause greater latency for finally viewing the feed, i.e., this approach does not lend itself to real time monitoring.
* The MonoDevelop and MonoTouch tools have issues with high thread counts and high-resolution media capture. Many lockups occurred. This result coupled with the need to stream over 3G links from phones makes medium resolution capture the better choice.

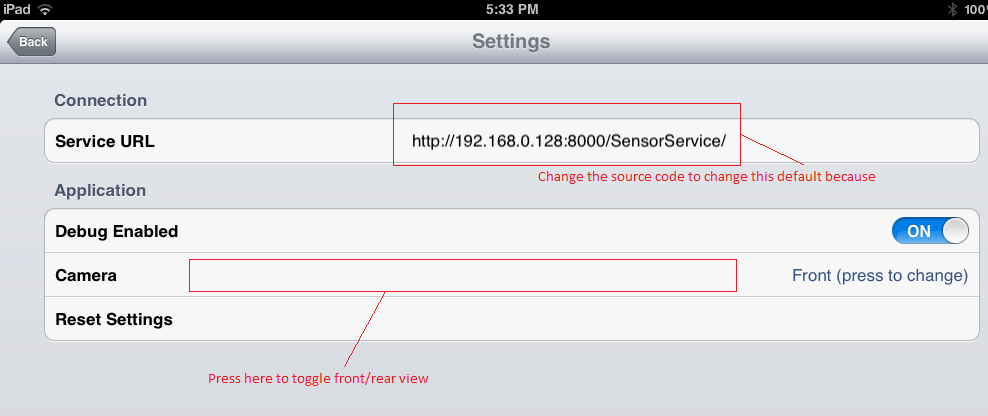
# Deliverables

1. Source code and documentation in a zip file
2. Documentation on setting up build and deploy environments for the client and the server components
3. A MonoTouch .sln file for the iOS client APP
4. A VS 2010 .sln file for the web service and sensor manager
5. Automated (non-UI) unit tests for the server side components
6. A test UI for sensor control (based on WinForms)

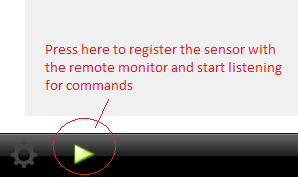
# Usage

* Using the accompanying documentation, compile and deploy the iOS app to an iPad or iPhone.
* Using the accompanying documentation, compile and deploy the web service monitor on IIS 7.5. Use the ‘publish’ step described in the deployment document so that the service runs in IIS on port 8000.
* *Note: there are a few code changes required to point to file locations for the FFMPEG tool and to set the default web service URL for the app settings. These are described in the deployment doc.*
* Once the app and the service are deployed the user interaction scenario is as follows:
  + Run the iOS app
  + Go to the settings screen and enter the web service URL (it should be defaulted to the service if you made the code changes discussed above)
  + Press the green ‘play’ button on the toolbar. This makes the sensor connect to the service and register itself for remote command receipt. It also starts a background command listener loop.
  + Run the server TestUI. You should see the connected sensor show up in the sensor list view.
  + Select the sensor and click start. This tells the sensor to start streaming.
  + Press the stream monitoring button to see the .mpg files that are uploaded. Note: the sensor feeds show up in small parts and get reassembled into one minute long recordings that are kept.
  + The screen shots below show the usage scenarios with annotations:

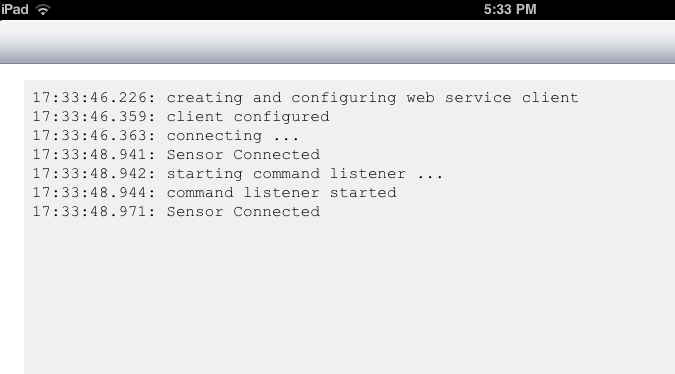
## Change the Settings



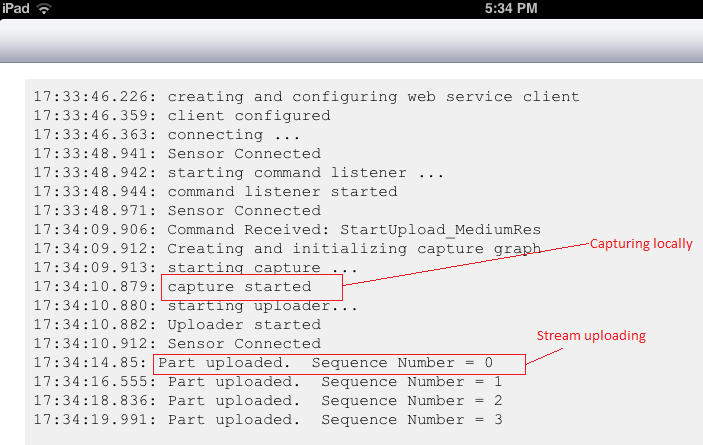
## Press the Start button (it has a slight lag so wait for it …)



## Notice the debug messages. These can be turned off in settings.

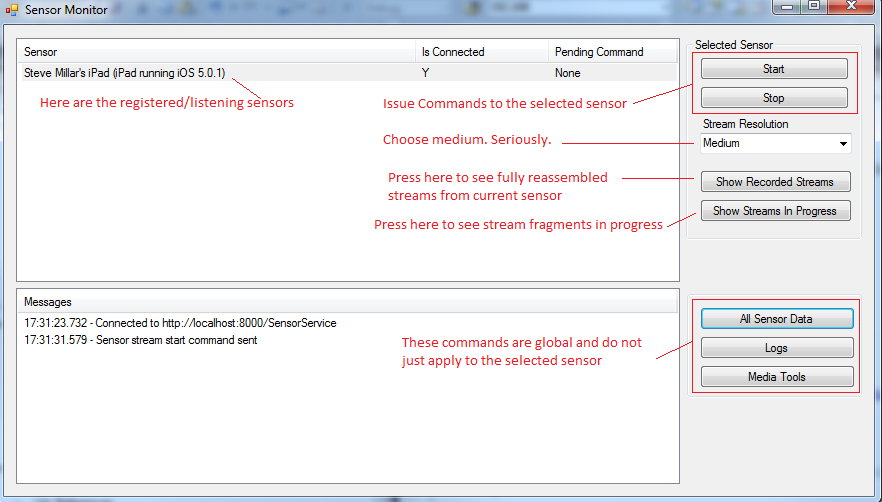


## You can tell when the capture is running and the stream fragments are uploading

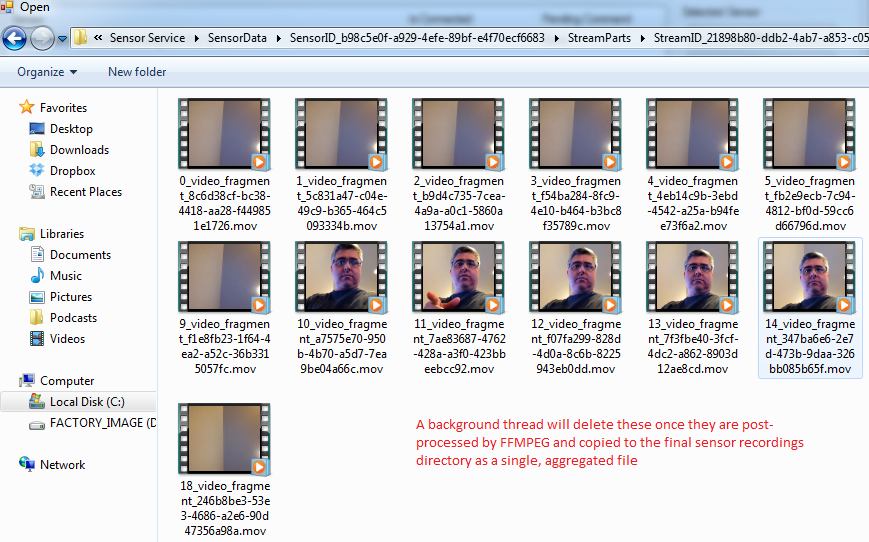


## Server Test UI.

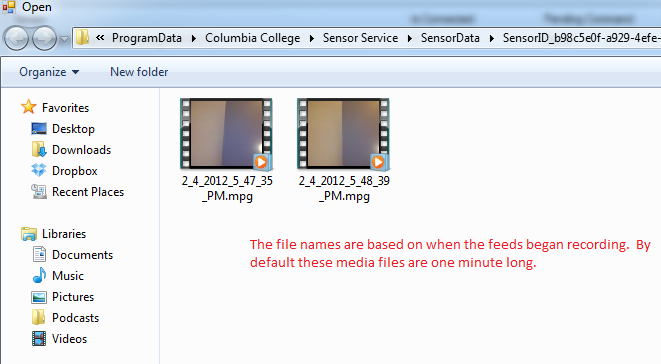
### Everything it does is via the web service so this can just as easily become a web page control UI based on AJAX or can be a background timer-based process that runs on a schedule with no UI.



## Stream Fragments View



## Final Recordings View



# Architecture

The diagram that follows shows three distinct layers to the system architecture.

### Sensors

The iOS clients act as sensors that provide audio and video data on demand. There is no fixed limit to the number of sensors supported by the system but the server will impose some resource-based limits based on the CPU overhead of centrally aggregating multiple media streams. Sensors announce themselves to the server via connection calls over the web service channel. Once registered in this manner, the server provides sensor data to registered controllers. Sensors receive commands from the server asynchronously by polling over the web service channel.

### Server Monitor

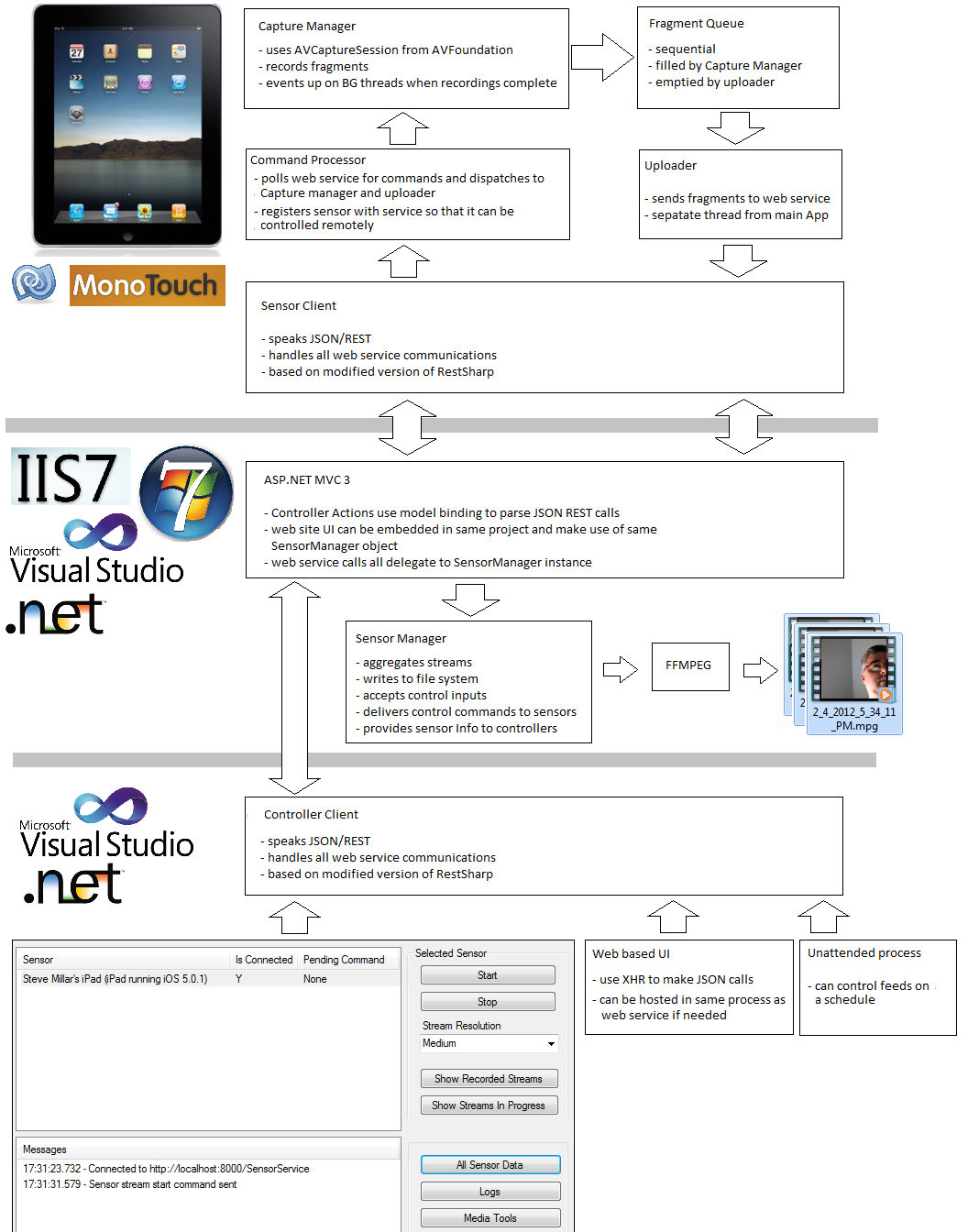
The server provides web service API channels over http so that the sensors can upload media streams and also poll for commands. The server stores and forwards commands from remote controller agents. These commands are forwarded to sensors during command polling (initiated by the sensors). The server aggregates media streams from individual sensors and converts them into MPG files which it stores in sensor-specific folders. The server also provides web-service channels for remote controllers to use to get a list of available sensors and to send commands intended for those sensors. The server does not explicitly limit how many sensors or remote controllers can coexist. All controllers can currently see and control all sensors. Future additions to the system may include role-based security. The current web server can easily be extended using ASP.NET’s membership provider components which would allow for http-based authentication and server-based authorization. It would be a simple matter to redirect initial web service calls through a login page (assuming web-based controller UI).

### Remote Controllers

Remote controllers use the controller client component to get provisioned sensor data from the server. These controllers can send commands to specific sensors via the web server which acts as a dispatcher/forwarder for routing commands. The delivered source code contains a WinForms-based UI that uses the controller client API to detect, visualize and control sensors. There is no built-in limit to how many controller instances can be running.

### Distributed Control

It should be emphasized that the controller UI does not have to run on the same machine as the web server. The provided sample UI in the VS 2010 solution and the unit tests all run on the same machine that hosts the web server but this is only for convenience of delivery and testing. Remote sensors and remote controllers all communicate via REST-based (JSON) web service calls and only need to be properly configured with an alternate URL to communicate with other physical nodes in the network.



# Client Components

The client application (App) is an iOS 5.x project built exclusively is C# using MonoTouch and MonoDevelop. Some of the shared data types dealing with web service calls and the JSON/REST clients are also present in the server side solution. This application is multi-threaded and highly concurrent.

### There are three projects in the solution:

* MonoTouch.Dialog: This project provides UI settings views.
* RemoteSensor: This is the client app
* RestSharp: This is a modified version of the popular RestSharp library used for JSON-based communication with RESTful web services.

### The RemoteSensor project has the following components

|  |  |
| --- | --- |
| Root View Controller | Contains the buttons and the debug text view. The view is rotatable. |
| CaptureManager | Constructs the capture graph using low-level AVCapture devices from the AV foundation. This singleton runs multiple threads and exposes start/stop functionality as well as media capture events. The eventing and control follow idiomatic .NET coding patterns so this component adapts the iOS delegate and controller pattern to be more ‘.NETish’ |
| Movie Writer Delegate | Processes capture buffers that contain media samples and multiplexes the audio and video into a single .mov fragment which will be uploaded to the server. |
| Video Frame Sampler Delegate | This component is not used but was useful in early debugging. It captures still images from the live stream and would be helpful if a ‘sensor snapshot’ feature was needed. The capture graph demonstrated some oddities when this output was connected at the same time as the movie writer. Either one will run individually but the movie capture delegate seems to turn off the frame capturer if you run them together. Just FYI. I left the code in the project in case of future need. |
| App Data | XML file containing the self-generated sensor ID Guid. |
| Settings | Strongly typed object that encapsulates user configuration and makes it accessible to the UI dialog. |
| Uploader | Provides a media fragment queue that the capture manager deposits fragments into. A background thread sequentially uploads the fragments from the shared queue. The uploader makes all service calls through the SensorClient component. |
| CommandMonitor | Polls the server on a background thread. Raises .Net events to abstract commands into a proper asynchronous model. The main UI view subscribes to the commands and manipulates the other components as needed. The command monitor makes all service calls through the SensorClient component. |
| SensorClient | Translates strongly types API calls into JSON for delivery to the service using HTTP. Binds the return values to .NET types as well. |
| iPad xib | UI for iPad |
| iPhone xib | UI for iPhone |

# Server and Controller Components

The web service and controller logic lives in a single IIS-hosted process that runs on any PC that can host IIS 7+. The Visual Studio solution contains 7 projects:

|  |  |
| --- | --- |
| Clients | * BaseClient handles sensor to service connection and pinging * SensorClient handles command fetching and stream part uploading * ControllerClient handles command setting and sensor data fetching * This project is used in both client and server solutions. The server solution only creates test sensors for unit test purposes. |
| RestSharp | Modified version of the open source library. This library handles JSON to .NET class translation well in most cases. I made one significant change to the JSON encoding of post data so the source code needs to be built in the client and server solutions |
| Utilities | Logging, testing, byte manipulation and App Data utilities. |
| SensorCommandTestUI | UI for controlling sensors and viewing resulting feed files in file browser. |
| SensorService | * Provides API via controller action methods * Contains singleton sensor manager that tracks commands and streams in progress * Sanitizes dropped streams and abandoned/disconnected sensors * Provides sensor lists to controllers * Provides commands to sensors * Accepts stream uploads from sensors * Accepts commands from controllers * Shells out to FFMPEG to convert aggregated files * Provides home page as a place-holder for future web UIs and role-based security integration |
| SensorServiceTests | Tests JSON APIs, uploading, file aggregation, command delivery. |
| SensorSharedTypes | Command, sensor, and stream data classes that get passé over the wire in JSON format and translated back and forth to .NET classes by RestSharp and ASP.NET MVC model binding |

# Reusability and Extensibility points

### Diverse Sensors

The three tier topology was designed to allow for easy extension of the existing system. Even though the iOS devices are the sole sensors right now, there is a well-defined uploader and command interface to the server which means that Android devices, windows phones or web-cam-based PC apps would be equally viable as sensors - perhaps with better frame splicing accuracy. No changes would be needed to the server or controllers.

### Diverse Controllers

The same principle of interface-based communication applies to controllers. There is no reason you cannot create mobile device-based controllers, web-based UIs or daemons that run scheduled capture tasks. Again, no sensor or server changes would be implied.

### Server Enhancements

The existing feed aggregator uses an external process to invoke FFMPEG to do video splicing and time rebasing. Given that this service runs on windows it makes more sense to pull that processing logic into the SensorManager where the same transforms can be applied via DirectShow (or the Media Foundation). Other types of transforms could be done on the feeds in the server as well. Some ideas include: motion detection based on image processing, time overlays, speech to text parsing. These are just a few ideas but the main point is that DirectShow is modular and the transforms would be self-contained and would not adversely affect the remainder of the system.

### Reusable Code

The MVC Controller classes all use a similar pattern of wrapping http calls into strongly-typed input to the SensorManager. The SensorClient and ControllerClient modules derive from a BaseClient class which makes adding APIs to the service very simple and straight-forward. The existing unit tests are factored in such a way as to allow ‘F5 debugging’ in an instance of IIS Express that runs on port 8888. Adding an API to the client or controller channel involves this simple sequence:

1. Add API to SensorAPIController.cs
2. Add the same API to the client class
3. Add a test method to the unit test class that makes the new API call
4. Press F5 and watch the client and server spin up and test the new call

This test-driven approach works because of the RestSharp and MVC model-binding features.

### General Thoughts

The delivered system is highly flexible, extensible and offers the ability to construct experiments based on new sensor and controller types. The command and control routing infrastructure will work equally well with future sensor types that may not have anything to do with media streaming. Rather than being just a remote video app, *this system is an experimentation platform that can easily be built upon in a research setting*. The components were constructed based on modern technologies, minimum coupling of components and test-driven principles.