How To Send Discord Notifications with Webcam Image using a Node.js Server

By: Sukhpal Gosal, Sajan Toor, Jimmy Nimavat, Gurnoor Chahal

Last Updated: Apr 15, 2024

Guide has been tested on:

BeagleBone (Target):	Debian 11.8
PC OS (host):	Debian 11.8 (or higher)
Node.js	v20.11.1
FFmpeg	4.3.6-0+deb11u1

This document guides the user through:

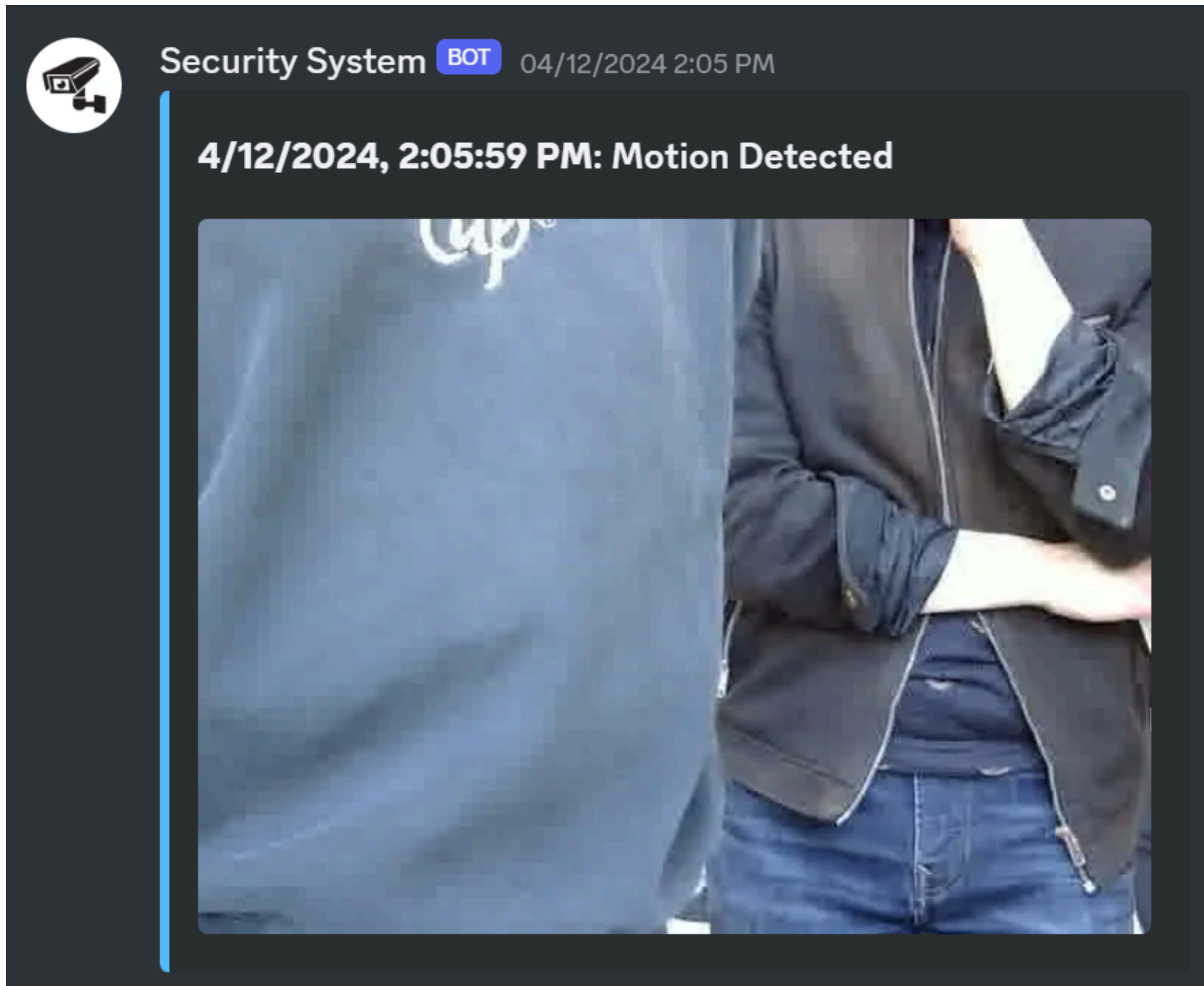
Introduction 1
1. WebCam Setup 2
2. UDP Socket Setup 2
3. Node.js Server Setup 5
4. Discord Webhook Notifications 7

Formatting

1. Commands for the host Linux's console are show as:
   `(host)$ echo "Hello PC world!"`
2. Commands for the target (BeagleBone) Linux's console are shown as:
   `(bbg)$ echo "Hello embedded world!"`
3. Almost all commands are case sensitive.

Introduction

This guide will provide a step by step guide of how to send Discord notifications including images from a webcam. For including the image, it is recommended to check the “Streaming Webcam from BeagleBone to NodeJS Server” guide as well, as it will give a step by step walkthrough of how to get the live stream running. If no image is needed, simply skip the steps of the stream as needed in the code.



Required hardware:
- BeagleBone Green
- Logitech C270 webcam

Required Software:
- Node.js
- FFmpeg

Note: Both FFmpeg and Node.js need to be installed where you want your server running. This could be on target, host, or the cloud. We will be showcasing it setup on the host.

Steps:
Note: If image not needed skip to Step 2

1. WebCam Setup

1. Refer to and follow [Streaming Webcam from BeagleBone to NodeJS Server guide] in order to get the live stream working.
2. You could skip step 1 as we have provided our code in the support files, to learn more about how the code works, please follow the guide.
   2.1. Ensure you have ffmpeg and Node.js installed
   2.2. Plug in the webcam via USB into the BBG and ensure its connected by running
   `(bbg)$ lsusb`
   2.3. For further troubleshooting, refer to the webcam guide, else follow this guide.

2. UDP Socket Setup

1. We will be using the following UDP socket module to make sending data and messages easier. Refer to supporting files for the code. The following code will be for socket.hpp, with definition for the functions in socket.cpp
2. Since we are sending streaming data and notifications separately, we will be using 2 different classes for UDP messages. We will have a base class and 2 sub classes to determine the message being sent:

```
class UdpPacket {
 protected:
   std::string ip;
   unsigned int port;

 public:
   UdpPacket(std::string ip, unsigned int port) : ip(ip),
port(port) {}
   std::string getIp(void) { return ip; }
   unsigned int getPort(void) { return port; }
   virtual const void* getData(void) = 0;
   virtual unsigned int getSize(void) = 0;
   virtual ~UdpPacket() {}  //Needed for delete
};

/**
* Represents a message that is sent or received from the UDP
socket.
*/
class UdpMessage : public UdpPacket {
 public:
   UdpMessage(std::string message, std::string ip, unsigned
int port) : UdpPacket(ip, port), message(message) {}
```

```
    const void* getData(void) override { return
message.c_str(); }
    std::string getMessage(void) { return message; }
    unsigned int getSize(void) override { return
message.size(); }
    void setMessage(std::string message) { this->message =
message; }

 private:
    std::string message;
};

/**
* Represents a stream of data that is sent or received from
the UDP socket.
*/
class UdpStream : public UdpPacket {
 public:
    UdpStream(const void* data, unsigned int size, std::string
ip, unsigned int port)
        : UdpPacket(ip, port), data(data), size(size) {}

    const void* getData(void) override { return data; }
    unsigned int getSize(void) { return size; }

 private:
    const void* data;
    unsigned int size;
};
```

3. To handle sending these messages, we will create a socket class inside socket.hpp:

```
class Socket {
 public:
    Socket(void);
    /**
     * CLoses the socket and stops the receiving thread.
     */
    void closeSocket(void);
    /**
```

```
    * Received a message from the UDP socket. UdpMessage is
allocated using new
    * and must be freed by the caller.
    */
   UdpMessage* receive(void);
   /**
    * Sends a message
    */
   void send(UdpPacket* message);
   /**
    * Returns true if the socket is currently receiving messages.
    */
   bool getIsReceiving(void);
   /**
    * Stops receiving messages from the socket.
    */
   void stopReceiving(void);

   /**
    * Sends a message to the web server
   */
   void sendToWebServer(std::string message);
   /**
    * Sends stream data to the web server
   */
   void sendDataToWebServer(const void* data, unsigned int size);

 private:
   int socketFd;
   bool isReceiving;
   static Socket* instance;
};
```

Note: There are 2 different ways to send to the server as one is for stream data only and the other is for all other messages.

4. Inside socket.hpp there are 4 important constants:

```
IP_ADDRESS_STREAMING = "192.168.6.1";
PORT_STREAMING = 1234;
```

These are for the webcam live streaming IP address and port, ensure you use the same IP address and port for the FFmpeg server in the server section of this guide.

```
IP_ADDRESS_SERVER = "192.168.6.1";
PORT_SERVER = 7070;
```

These are for sending regular messages using the `sendToWebServer` method in `Socket`, ensure you have the same address and port for the Node.js server in the server section of this guide.

Note that the IP address used is the IP address of the host, this is because we are running our FFmpeg server on the host, if you want to run the server locally on the target use “127.0.0.1” for localhost instead.

Troubleshooting:
If packets are failing to send to the server:

- Check your IP address. On Mac the IP address for the host is 192.168.7.1, not 192.168.6.1.
- Ensure you have followed the networking guide and networking is working with a simple ping:
  `(bbg)$ ping google.ca`

3. Node.js Server Setup

See the support files for index.js.

```
const UDP_SERVER_ADDRESS = '192.168.6.1';
const UDP_SERVER_PORT = 7070;

const udpServer = dgram.createSocket('udp4');
udpServer.bind(UDP_SERVER_PORT, UDP_SERVER_ADDRESS);
```

This creates a new UDP server used to receive UDP messages, ensuring this is the same as the `IP_ADDRESS_SERVER` and `PORT_SERVER` as the previous section.

Start the FFmpeg process using the `startFFMpegProcess();` function. This creates a new FFMpeg process as a child process. Options are passed in using `ffmpegArgs`. Ensure `UDP_BBG_ADDRESS` and `UDP_BBG_STREAMING_PORT` are the same as the previous section.

```
const UDP_BBG_ADDRESS = '192.168.6.2';
const UDP_BBG_STREAMING_PORT = 1234;

function startFFMpegProcess() {
```

```
  console.log("ffmpeg process started");

  const ffmpegArgs = [
    "-re",
    "-y",
    "-i",
   `udp://${UDP_BBG_ADDRESS}:${UDP_BBG_STREAMING_PORT}?overrun_nonfatal=1`,
    "-preset",
    "ultrafast",
    "-f",
    "mjpeg",
    "-tune",
    "zerolatency",
    "-omit_video_pes_length",
    "1",
    "pipe:1"
  ];

  return child.spawn("ffmpeg", ffmpegArgs);
}
```

```

ffmpegProcess.on("connect", function () {
  console.log("ffmpeg connected");
});

ffmpegProcess.on("error", function (err) {
  console.log(err);
  throw err;
});

ffmpegProcess.on("close", function (code) {
  console.log("ffmpeg exited with code " + code);
  ffmpegProcess = null;
  setTimeout(function () {
    ffmpegProcess = startFFMpegProcess();
  }, 50);
});

ffmpegProcess.stderr.on("data", function (data) {
  // Don't remove this
```

```
  // Child Process hangs when stderr exceed certain memory
});

ffmpegProcess.stdout.on("data", function (data) {
  frame = Buffer.from(data).toString("base64"); //convert raw data to string
});
```

These are events that happen in the FFmpeg child process, the most important event is the “data” event, where image data comes through.

Troubleshooting:
If the child process is not spawning or is returning errors ensure FFmpeg is installed.

```
(host)$ ffmpeg --version
ffmpeg version 4.3.6-0+deb11u1 Copyright (c) 2000-2023 the FFmpeg
developers ...
```

If you are not receiving UDP messages or live stream data, ensure the IP address and port are the same as those you set in step 2. If it is not working, ensure packets are being set by using a tool like tshark or tcpdump.

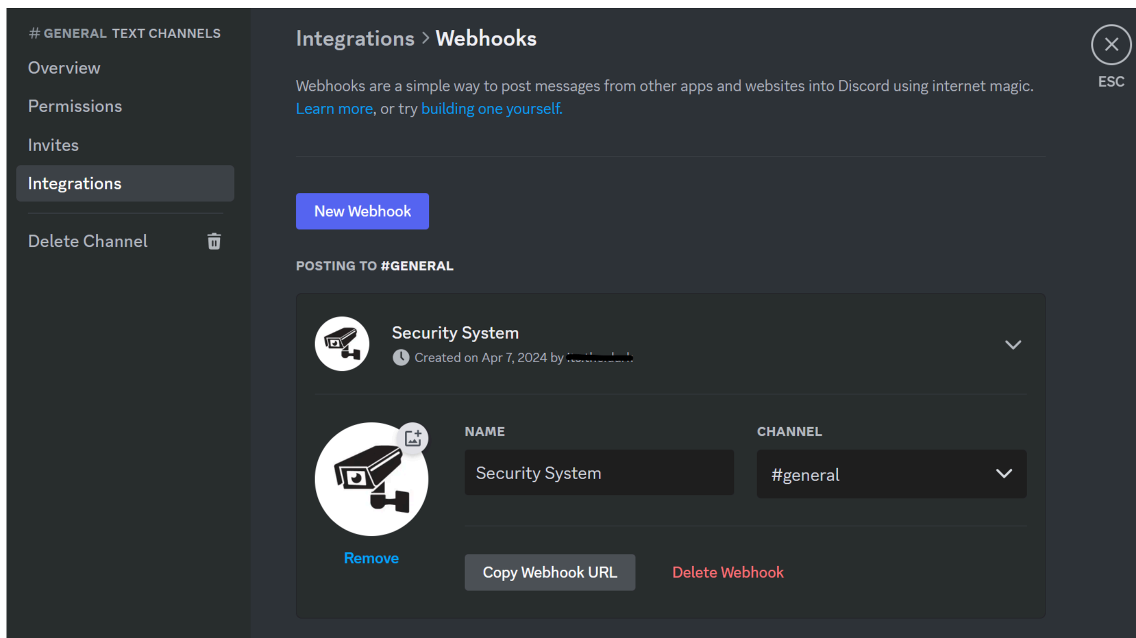
4. Discord Webhook Notifications

Pre-requisites:
Discord Webhooks allow users to post messages to a channel of their choice, without requiring a bot account or any authentication.

You need to create a Discord server or have sufficient admin access to an existing server to enable Webhooks.

To create a Discord server, follow:
https://support.discord.com/hc/en-us/articles/204849977-How-do-I-create-a-server

To get the Webhook URL for a server channel, follow the “Making A Webhook” section:
https://support.discord.com/hc/en-us/articles/228383668-Intro-to-Webhooks



To send a Discord Webhook notification, a POST request must be sent to the Webhook URL you would have received from the previous step. The most important data for the POST request is the JSON body, which contains parameters and attributes specified within the documentation: https://discord.com/developers/docs/resources/webhook#execute-webhook

In this use-case, the notification will be in the form of an embed, and contain title, coloured border (the colour code in the form of an integer), and an image. You may choose to also add a `content` attribute to send a text message alongside the image. We will employ the `axios` library to send the POST request. The following screenshot is an example of how to send a POST request to the Webhook, **WITHOUT** an image. This is the simplest use-case.

The `DISCORD_WEBHOOK` variable represents the URL that you previously copied from your Discord channel. You should reference it from an environment variable, as **anyone with access to your Discord Webhook URL will be able to send messages** to the given channel.

```
const DISCORD_EMBED_COLOR = 5814783;

const sendDiscordMessage = (message) => {
  axios
    .post(DISCORD_WEBHOOK, {
      username: 'Security System',
      embeds: [
        {
          title: message,
          color: DISCORD_EMBED_COLOR,
        },
      ],
    })
    .catch((error) => {
      console.error('Error:', error);
    });
```

The webcam frame received is in base64 format. As such, while it is easy to attach an image through a URL or with another format, using the base64 format adds some complications.

Adding an image makes the POST request much more complicated, as the Discord API has some limitations, such as a change to the JSON body. Keep in mind that there is also a default file (or image) upload size limit of 25 MiB. To upload an image, please read: https://discord.com/developers/docs/reference#uploading-files

To begin, create a function to convert the base64 image received from the webcam to a blob. A blob (binary large object) is a type of file object with raw and immutable data. The following sample code, referenced from a StackOverflow post, demonstrates a possible algorithm for this conversion.

https://stackoverflow.com/questions/16245767/creating-a-blob-from-a-base64-string-in-javascript

```
// https://stackoverflow.com/questions/16245767/creating-a-blob-from-a-base64-string-in-javascript
const b64toBlob = (b64Data, contentType = '', sliceSize = 512) => {
  const byteCharacters = atob(b64Data);
  const byteArrays = [];

  for (let offset = 0; offset < byteCharacters.length; offset += sliceSize) {
    const slice = byteCharacters.slice(offset, offset + sliceSize);

    const byteNumbers = new Array(slice.length);
    for (let i = 0; i < slice.length; i++) {
      byteNumbers[i] = slice.charCodeAt(i);
    }

    const byteArray = new Uint8Array(byteNumbers);
    byteArrays.push(byteArray);
  }

  const blob = new Blob(byteArrays, { type: contentType });
  return blob;
};
```

Next, you will need to modify the previous `sendDiscordMessage` function to accept a base64 string representing the Webcam image. There will also be modifications to support the Discord API requirements for attaching a file.

Most importantly, the change requires the `application/json` body to be replaced with a `multipart/form-data` body. The JSON body above will now be provided in the `payload_json` property. The image uploaded should contain a filename that is later referenced in the Discord embed as part of the payload.

```
// Send message to Discord through Webhook
const sendDiscordMessage = (message, base64String = '') => {
  const image = base64String ? b64toBlob(base64String.replace(/^data:image\/[a-z]+;base64,/, ''),
'image/jpeg') : null;
  const data = {
    username: 'Security System',
    embeds: [
      {
        title: message,
        color: DISCORD_EMBED_COLOR,
        image: image
          ? {
              url: `attachment://image.jpeg`,
            }
          : null,
      },
    ],
  };

  // https://discord.com/developers/docs/reference#uploading-files
  var headers = new Headers();
  const formData = new FormData();
  if (image) formData.append('file', image, 'image.jpeg');
  formData.append('payload_json', JSON.stringify(data));
  headers.append('Content-Type', 'multipart/form-data');
  axios.post(DISCORD_WEBHOOK, formData, { headers: headers }).catch((error) => console.error(error));
};
```

The above sample code receives a message (representing the post title) and an optional base64 string representing the image to be attached. The code works for both use cases of attaching an image, as well as sending a message without an image.

The formData interface is utilised to generate a pair of key values that will be sent with the request. FormData objects are used to capture data from submissions with HTML forms. In this use case, we are manually creating this object, as we are required to have a `multipart/form-data` body.

Convert the image to a blob which will be attached as a file in the formData. To convert the base64 string to a blob with the function provided previously, strip any prefix from the base64 string. The regex above, listed within the function call to `b64toBlob` removes the `data:image/jpeg;base64,/` prefix if it exists. Ensure that the file name that you provide, such as `image.jpeg` is the same name you use in the embed image URL. In this case, our image URL is an attachment object referencing the `image.jpeg` we attached in the formData body.

The rest of the implementation in the code block above is trivial.

Troubleshooting:

If messages are not being sent, check the console. It's possible you are getting a 400 or 403 HTTP status, ensure that the webhook is in the environment and defined properly.

- Try using postman and sending a request with the same format described in the above code to ensure your webhook is working properly

If the images are not being attached to the message, ensure the BBG app is running and has the correct port and ip address, defined in step 2 and 3.

- Ensure that the FFmpeg process is spawned and running
- Check by adding a console log in the “data” event, here:

```
ffmpegProcess.stdout.on("data", function (data) {
  frame = Buffer.from(data).toString("base64"); //convert raw data to string
});
```

- If no console log is received, there is likely an issue with your IP address and port or the FFmpeg process.