So You Want To Write A SAM Library

One of the best features of I2P, in my opinion, is it's SAM API, which can be used to build a bridge between I2P and your application or language of choice. Currently, dozens of SAM libraries exist for a variety of languages, including:

- i2psam, for c++
- libsam3, for C
- txi2p for Python
- i2plib for Python
- i2p.socket for Python
- leaflet for Python
- gosam, for Go
- sam3 for Go
- node-i2p for nodejs
- haskell-network-anonymous-i2p
- i2pdotnet for .Net languages
- rust-i2p
- and i2p.rb for ruby

If you're using any of these languages, you may be able to port your application to I2P already, using an existing library. That's not what this tutorial is about, though. This tutorial is about what to do if you want to create a SAM library in a new language. In this tutorial, I will implement a new SAM library in Java. I chose Java because there isn't a Java library that connects you to SAM yet, because of Java's use in Android, and because it's a language almost everybody has at least a *little* experience with, so hopefully you can translate it into a language of your choice.

Creating your library

How you set up your own library will vary depending on the language you wish to use. For this example library, we'll be using java so we can create a library like this:

```
mkdir jsam
cd jsam
gradle init --type java-library
```

Or, if you are using gradle 5 or greater:

```
gradle init --type java-library --project-name jsam
```

Setting up the Library

There are a few pieces of data that almost any SAM library should probably manage. It will at least need to store the address of the SAM Bridge you intend to use and the signature type you wish to use.

Storing the SAM address

I prefer to store the SAM address as a String and an Integer, and re-combine them in a function at runtime.

```
    public String SAMHost = "127.0.0.1";
    public int SAMPort = "7656";
    public String SAMAddress(){
        return SAMHost + ":" + SAMPort;
    }
```

Storing the Signature Type

The valid signature types for an I2P Tunnel are DSA_SHA1, ECDSA_SHA256_P256, ECDSA_SHA384_P384, ECDSA_SHA512_P521, EdDSA_SHA512_Ed25519, but it is strongly recommended that you use EdDSA_SHA512_Ed25519 as a default if you implement at least SAM 3.1. In Java, the 'enum' datastructure lends itself to this task, as it is intended to contain a group of constants. Add the enum, and an instance of the enum, to your Java class definition.

```
    enum SIGNATURE_TYPE {
        DSA_SHA1,
        ECDSA_SHA256_P256,
        ECDSA_SHA384_P384,
        ECDSA_SHA512_P521,
        EdDSA_SHA512_Ed25519;
    }
    public SIGNATURE_TYPE SigType = SIGNATURE_TYPE.EdDSA_SHA512_Ed25519;
```

Retrieving the signature type:

That takes care of reliably storing the signature type in use by the SAM connection, but you've still got to retrieve it as a string to communicate it to the bridge.

```
    public String SignatureType() {
        switch (SigType) {
            case DSA_SHA1:
                return "SIGNATURE_TYPE=DSA_SHA1";
            case ECDSA_SHA256_P256:
                return "SIGNATURE_TYPE=ECDSA_SHA256_P256";
            case ECDSA_SHA384_P384:
                return "SIGNATURE_TYPE=ECDSA_SHA384_P384";
            case ECDSA_SHA512_P521:
                return "SIGNATURE_TYPE=ECDSA_SHA512_P521";
            case EdDSA_SHA512_Ed25519:
                return "SIGNATURE_TYPE=EdDSA_SHA512_Ed25519";
        }
        return "";
    }
```

It's important to test things, so let's write some tests:

```
    @Test public void testValidDefaultSAMAddress() {
        Jsam classUnderTest = new Jsam();
        assertEquals("127.0.0.1:7656", classUnderTest.SAMAddress());
    }
    @Test public void testValidDefaultSignatureType() {
        Jsam classUnderTest = new Jsam();
        assertEquals("EdDSA_SHA512_Ed25519", classUnderTest.SignatureType());
    }
```

Once that's done, begin creating your constructor. Note that we've given our library defaults which will be useful in default situations on all existing I2P routers so far.

```
    public Jsam(String host, int port, SIGNATURE_TYPE sig) {
        SAMHost = host;
        SAMPort = port;
        SigType = sig;
    }
```

Establishing a SAM Connection

Finally, the good part. Interaction with the SAM bridge is done by sending a "command" to the address of the SAM bridge, and you can parse the result of the command as a set of string-based key-value pairs. So bearing that in mind, let's estabish a read-write connection to the SAM Address we defined before, then write a "CommandSAM" Function and a reply parser.

Connecting to the SAM Port

We're communicating with SAM via a Socket, so in order to connect to, read from, and write to the socket, you'll need to create the following private variables in the Jsam class:

```
    private Socket socket;
    private PrintWriter writer;
    private BufferedReader reader;
```

You will also want to instantiate those variables in your Constructors by creating a function to do so.

```
    public Jsam(String host, int port, SIGNATURE_TYPE sig) {
        SAMHost = host;
        SAMPort = port;
        SigType = sig;
        startConnection();
    }
    public void startConnection() {
        try {
            socket = new Socket(SAMHost, SAMPort);
            writer = new PrintWriter(socket.getOutputStream(), true);
            reader = new BufferedReader(new InputStreamReader(socket.getInputStream()));
        } catch (Exception e) {
            //omitted for brevity
        }
    }
```

Sending a Command to SAM

Now you're all set up to finally start talking to SAM. In order to keep things nicely organized, let's create a function which sends a single command to SAM, terminated by a newline, and which returns a Reply object, which we will create in the next step:

```
    public Reply CommandSAM(String args) {
        writer.println(args + "\n");
        try {
            String repl = reader.readLine();
            return new Reply(repl);
        } catch (Exception e) {
            //omitted for brevity
        }
    }
```

Note that we are using the writer and reader we created from the socket in the previous step as our inputs and outputs to the socket. When we get a reply from the reader, we pass the string to the Reply constructor, which parses it and returns the Reply object.

Parsing a reply and creating a Reply object.

In order to more easily handle replies, we'll use a Reply object to automatically parse the results we get from the SAM bridge. A reply has at least a topic, a type, and a result, as well as an arbitrary number of key-value pairs.

```
public class Reply {
    String topic;
    String type;
    REPLY_TYPES result;
    Map<String, String> replyMap = new HashMap<String, String>();
```

As you can see, we will be storing the "result" as an enum, REPLY_TYPES. This enum contains all the possible reply results which the SAM bridge might respond with.

```
    enum REPLY_TYPES {
        OK,
        CANT_REACH_PEER,
        DUPLICATED_ID,
        DUPLICATED_DEST,
        I2P_ERROR,
        INVALID_KEY,
        KEY_NOT_FOUND,
        PEER_NOT_FOUND,
        TIMEOUT;
        public static REPLY_TYPES set(String type) {
            String temp = type.trim();
            switch (temp) {
            case "RESULT=OK":
                return OK;
            case "RESULT=CANT_REACH_PEER":
                return CANT_REACH_PEER;
            case "RESULT=DUPLICATED_ID":
                return DUPLICATED_ID;
            case "RESULT=DUPLICATED_DEST":
                return DUPLICATED_DEST;
            case "RESULT=I2P_ERROR":
                return I2P_ERROR;
            case "RESULT=INVALID_KEY":
                return INVALID_KEY;
            case "RESULT=KEY_NOT_FOUND":
                return KEY_NOT_FOUND;
            case "RESULT=PEER_NOT_FOUND":
                return PEER_NOT_FOUND;
            case "RESULT=TIMEOUT":
                return TIMEOUT;
            }
            return I2P_ERROR;
        }
        public static String get(REPLY_TYPES type) {
            switch (type) {
            case OK:
                return "RESULT=OK";
            case CANT_REACH_PEER:
                return "RESULT=CANT_REACH_PEER";
            case DUPLICATED_ID:
                return "RESULT=DUPLICATED_ID";
            case DUPLICATED_DEST:
                return "RESULT=DUPLICATED_DEST";
            case I2P_ERROR:
                return "RESULT=I2P_ERROR";
            case INVALID_KEY:
                return "RESULT=INVALID_KEY";
            case KEY_NOT_FOUND:
                return "RESULT=KEY_NOT_FOUND";
            case PEER_NOT_FOUND:
                return "RESULT=PEER_NOT_FOUND";
            case TIMEOUT:
                return "RESULT=TIMEOUT";
            }
            return "RESULT=I2P_ERROR";
        }
    };
```

Now let's create our constructor, which takes the reply string recieved from the socket as a parameter, parses it, and uses the information to set up the reply object. The reply is space-delimited, with key-value pairs joined by an equal sign and terminated by a newline.

```
    public Reply(String reply) {
        String trimmed = reply.trim();
        String[] replyvalues = reply.split(" ");
        if (replyvalues.length < 2) {
            //omitted for brevity
        }
        topic = replyvalues[0];
        type = replyvalues[1];
        result = REPLY_TYPES.set(replyvalues[2]);

        String[] replyLast = Arrays.copyOfRange(replyvalues, 2, replyvalues.length);
        for (int x = 0; x < replyLast.length; x++) {
            String[] kv = replyLast[x].split("=", 2);
            if (kv.length != 2) {

            }
            replyMap.put(kv[0], kv[1]);
        }
    }
```

Lastly, for the sake of convenience, let's give the reply object a toString() function which returns a string representation of the Reply object.

```
    public String toString() {
        return topic + " " + type + " " + REPLY_TYPES.get(result) + " " + replyMap.toString();
    }
}
```

Saying "HELLO" to SAM

Now we're ready to establish communication with SAM by sending a "Hello" message. If you're writing a new SAM library, you should probably target at least SAM 3.1, since it's available in both I2P and i2pd and introduces support for the SIGNATURE_TYPE parameter.

```
    public boolean HelloSAM() {
        Reply repl = CommandSAM("HELLO VERSION MIN=3.0 MAX=3.1 \n");
        if (repl.result == Reply.REPLY_TYPES.OK) {
            return true;
        }
        System.out.println(repl.String());
        return false;
    }
```

As you can see, we use the CommandSAM function we created before to send the newline-terminated command `HELLO VERSION MIN=3.0 MAX=3.1 \n`. This tells SAM that you want to start communicating with the API, and that you know how to speak SAM version 3.0 and 3.1. The router, in turn, will respond with like `HELLO REPLY RESULT=OK VERSION=3.1` which is a string you can pass to the Reply constructor to get a valid Reply object. From now on, we can use our CommandSAM function and Reply object to deal with all our communication across the SAM bridge.

Finally, let's add a test for our "HelloSAM" function.

```
    @Test public void testHelloSAM() {
        Jsam classUnderTest = new Jsam();
        assertTrue("HelloSAM should return 'true' in the presence of an alive SAM bridge", classUnderTest.HelloSAM());
    }
```

Creating a "Session" for your application

Now that you've negotiated your connection to SAM and agreed on a SAM version you both speak, you can set up peer-to-peer connections for your application to connect to other i2p applications. You do this by sending a "SESSION CREATE" command to the SAM Bridge. To do that, we'll use a CreateSession function that accepts a session ID and a destination type parameter.

```
    public String CreateSession(String id, String destination ) {
        if (destination == "") {
            destination = "TRANSIENT";
        }
        Reply repl = CommandSAM("SESSION CREATE STYLE=STREAM ID=" + ID + " DESTINATION=" + destination);
        if (repl.result == Reply.REPLY_TYPES.OK) {
            return id;
        }
        return "";
    }
```

That was easy, right? All we had to do was adapt the pattern we used in our HelloSAM function to the `SESSION CREATE` command. A good reply from the bridge will still return OK, and in that case we return the ID of the newly created SAM connection. Otherwise, we return an empty string because that's an invalid ID anyway and it failed, so it's easy to check. Let's see if this function works by writing a test for it:

```
    @Test public void testCreateSession() {
        Jsam classUnderTest = new Jsam();
        assertTrue("HelloSAM should return 'true' in the presence of an alive SAM bridge", classUnderTest.HelloSAM());
        assertEquals("test", classUnderTest.CreateSession("test", ""));
    }
```

Note that in this test, we *must* call HelloSAM first to establish communication with SAM before starting our session. If not, the bridge will reply with an error and the test will fail.

Looking up Hosts by name or .b32

Now you have your session established and your local destination, and need to decide what you want to do with them. Your session can now be commanded to connect to a remote service over I2P, or to wait for incoming connections to respond to. However, before you can connect to a remote destination, you may need to obtain the base64 of the destination, which is what the API expects. In order to do this, we'll create a LookupName function, which will return the base64 in a usable form.

```
    public String LookupName(String name) {
        String cmd = "NAMING LOOKUP NAME=" + name + "\n";
        Reply repl = CommandSAM(cmd);
        if (repl.result == Reply.REPLY_TYPES.OK) {
            System.out.println(repl.replyMap.get("VALUE"));
            return repl.replyMap.get("VALUE");
        }
        return "";
    }
```

Again, this is almost the same as our HelloSAM and CreateSession functions, with one difference. Since we're looking for the VALUE specifically and the NAME field will be the same as the `name` argument, it simply returns the base64 string of the destination requested.

Now that we have our LookupName function, let's test it:

```
    @Test public void testLookupName() {
        Jsam classUnderTest = new Jsam();
        assertTrue("HelloSAM should return 'true' in the presence of an alive SAM bridge", classUnderTest.HelloSAM());
        assertEquals("8ZAW~KzGFMUEj0pdchy6GQOOZbuzbqpWtiApEj8LHy2~O~58XKxRrA43cA23a9oDpNZDqWhRWEtehSnX5NoCwJcXWWdO1ksKEUim6cQLP-
 VpQyuZTIIqwSADwgoe6ikxZG0NGvy5FijgxF4EW9zg39nhUNKRejYNHhOBZKIX38qYyXoB8XCVJybKg89aMMPsCT884F0CLBKbHeYhpYGmhE4YW~aV21c5pebivvxeJPWuTBAOmYxAIgJE3fF
 fucQn9YyGUFa8F3t-0Vco-9qVNSEWfgrdXOdKT6orr3sfssiKo3ybRWdTpxycZ6wB4qHWgTSU5A-gOA3ACTCMZBsASN3W5cz6GRZCspQ0HNu~R~nJ8V06Mmw~iVYOu5lDvipmG6-
 dJky6XRxCedczxMM1GWFoieQ8Ysfuxq-j8keEtaYmyUQme6TcviCEvQsxyVirr~dTC-F8aZ~y2AlG5IJz5KD02nO6TRkI2fgjHhv9OZ9nskh-I2jxAzFP6Is1kyAAAA",
 classUnderTest.LookupName("i2p-projekt.i2p"));

    }
```

Sending and Recieving Information

At last, we are going to establish a connection to another service with our new library. This part confused me a bit at first, but the most astute Java developers were probably wondering why we didn't extend the socket class instead of creating a Socket variable inside of the Jsam class. That's because until now, we've been communicating with the "Control Socket" and we need to create a new socket to do the actual communication. So we've waited to extend the the Socket class with the Jsam class until now:

```
public class Jsam extends Socket {
```

Also, let's alter our startConnection function so that we can use it to switch over from the control socket to the socket we'll be using in our application. It will now take a Socket argument.

```
    public void startConnection(Socket socket) {
        try {
            socket.connect(new InetSocketAddress(SAMHost, SAMPort), 600 );
        } catch (Exception e) {
            System.out.println(e);
        }
        try {
            writer = new PrintWriter(socket.getOutputStream(), true);
        } catch (Exception e) {
            System.out.println(e);
        }
        try {
            reader = new BufferedReader(new InputStreamReader(socket.getInputStream()));
        } catch (Exception e) {
            System.out.println(e);
        }
    }
```

This allows us to quickly and easily open a new socket to communicate over, perform the "Hello SAM" handshake over again, and connect the stream.

```
    public String ConnectSession(String id, String destination) {
        startConnection(this);
        HelloSAM();
        if (destination.endsWith(".i2p")) {
            destination = LookupName(destination);
        }
        String cmd = "STREAM CONNECT ID=" + id + " DESTINATION=" + destination + " SILENT=false";
        Reply repl = CommandSAM(cmd);
        if (repl.result == Reply.REPLY_TYPES.OK) {
            System.out.println(repl.String());
            return id;
        }
        System.out.println(repl.String());
        return "";
    }
```

And now you have a new Socket for communicating over SAM! Let's do the same thing for Accepting remote connections:

```
    public String AcceptSession(String id) {
        startConnection(this);
        HelloSAM();
        String cmd = "STREAM ACCEPT ID=" + id  + " SILENT=false";
        Reply repl = CommandSAM(cmd);
        if (repl.result == Reply.REPLY_TYPES.OK) {
            System.out.println(repl.String());
            return id;
        }
        System.out.println(repl.String());
        return "";
    }
```

There you have it. That's how you build a SAM library, step-by-step. In the future, I will cross-reference this with the working version of the library, Jsam, and the SAM v3 specification but for now I've got to get some other stuff done.