JTor Hidden Services

Kory Kirk

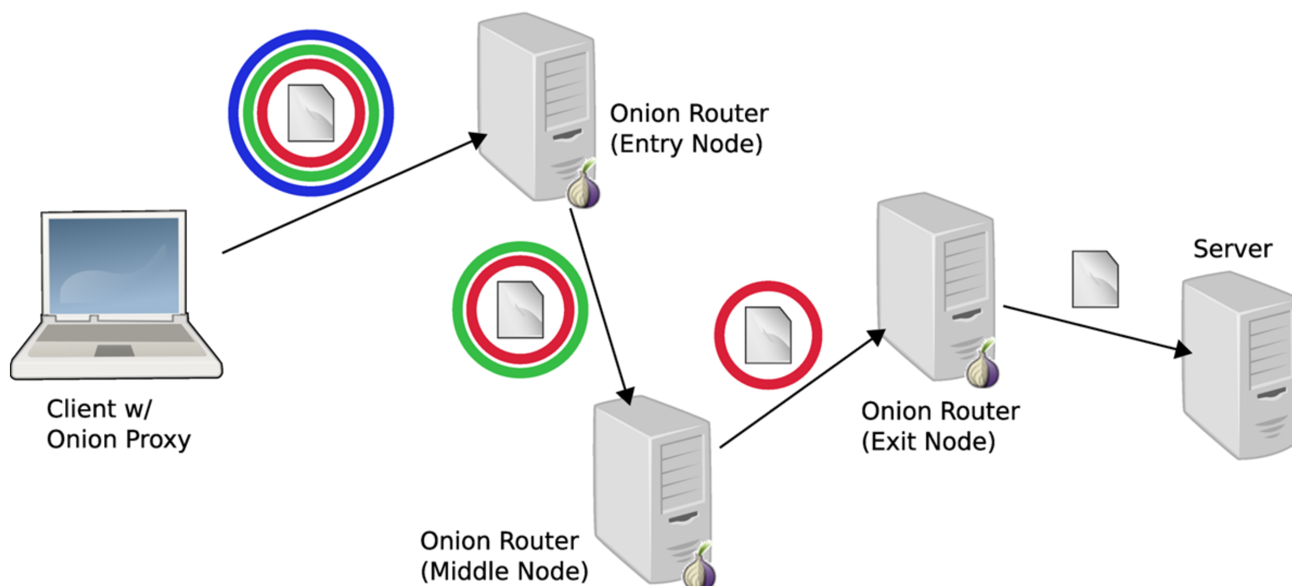
Fall 2009

CSC8530 Distributed Systems

Dr. Schragger

I. Overview

The need for JTor became apparent when the first Tor app hit the Android App Store. The app was built off of OnionCoffee. A quick audit of the OnionCoffee code deemed it unfit for supporting the newest version of the Tor protocol. Thus the need for a well written library to connect to the Tor network became a good idea. I was asked personally to be involved by a few full time Tor developers. Bruce Leidl was contracted for the project, and I became the first volunteer developer. Initially, JTor was nothing, so I asked what I could do that was not dependent upon the foundational code (which was yet to be written). So my project became JTor Hidden Services.



Here are the definitions of some of the terms used in this write up:

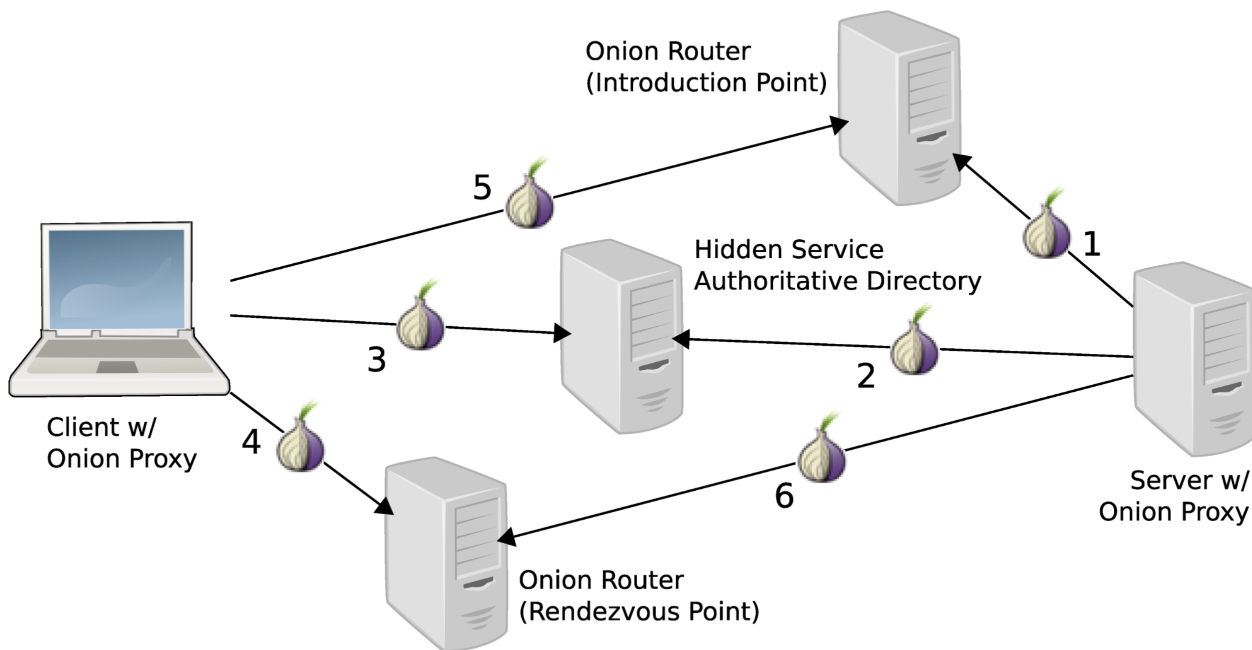
Tor: a secure open source anonymity network. Above is a diagram of a circuit through the Tor network. This diagram shows the sending of a message/packet through the Tor network. The message is the file icon, and each colored circle around it represents a layer of encryption. When the client sends the message into the tor network, it has three levels of encryption, and on each hop through the network, one of those layers get stripped off, and becomes decrypted when it exits the Tor network.

Circuit: A path through the Tor network (Usually consists of 3 relay nodes, including a guard node and an exit node)

Onion Proxy(OP): A client node in the Tor network

Onion Relay(OR): A relay node in the Tor network

Hidden Service: A hidden service is a way to offer a server on the Tor network anonymously (by not broadcasting IP address). Instead the server is identified by its public key, and traffic is routed through circuits in the Tor network.



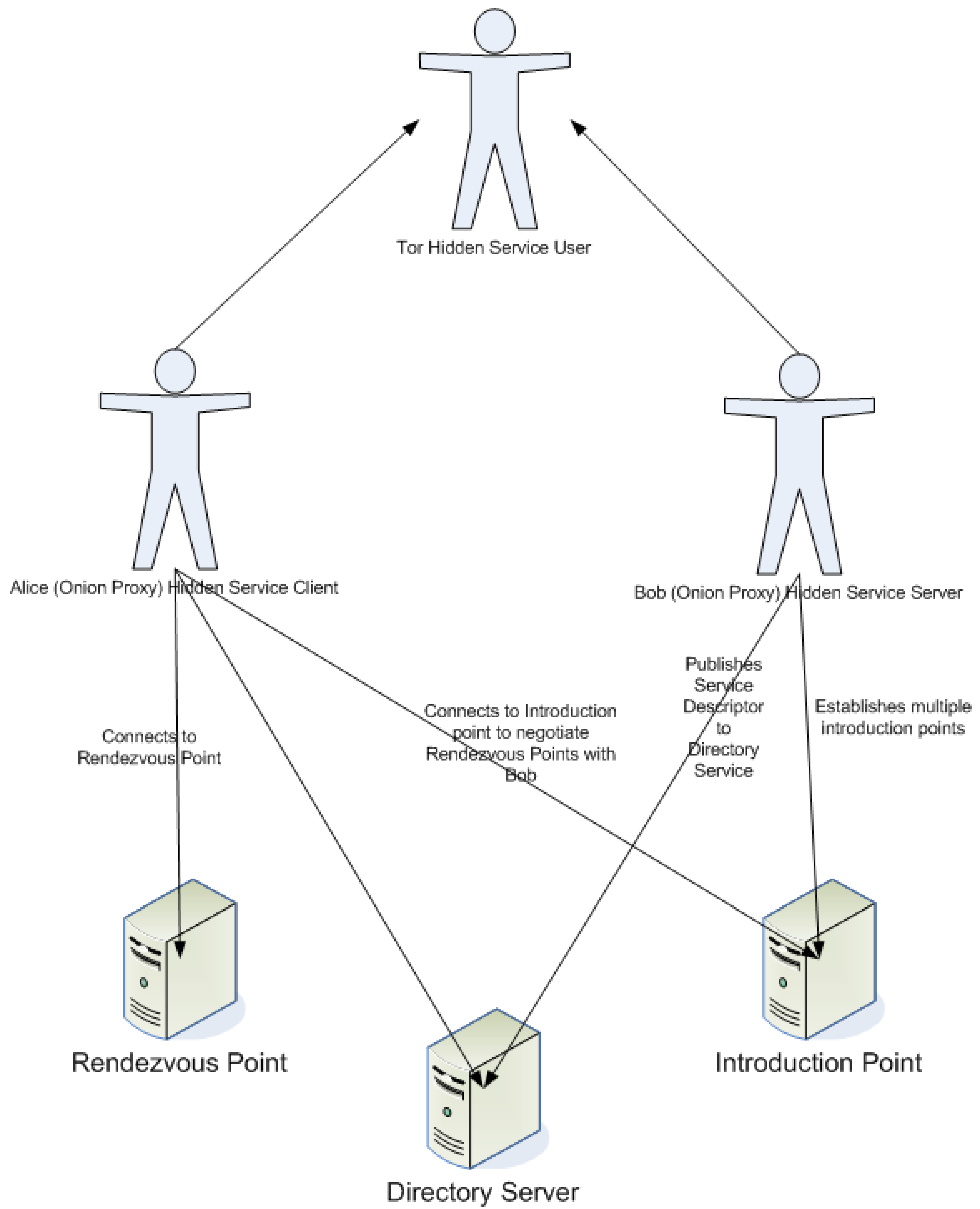
A little more overview on how Hidden Services work; meet Alice and Bob, Bob is hosting a hidden service on his OP, and Alice wants to connect to his hidden service via her OP, here are the steps.

1. Bob's OP hosts the Hidden Service (service name and ports)
2. Bob's OP generates keypair and rendezvous service descriptor.
3. Bob's OP talks to the Introduction points via Tor, to negotiate keys to use for communication (one pair per intro point).
4. Bob's OP->directory service via Tor: publishes Bob's service descriptor [advertisement]
5. Alice receives a [x.y.]z.onion:port address. She opens a SOCKS connection to her OP, and requests x.y.z.onion:port. (x and y are optional and usually associated with further authentication methods).
6. Alice's OP gets Bob's descriptor
7. Alice's OP chooses a rendezvous point, opens a circuit to that rendezvous point, and establishes a rendezvous circuit.
8. Alice connects to the Introduction point via Tor, and tells it about her rendezvous point and

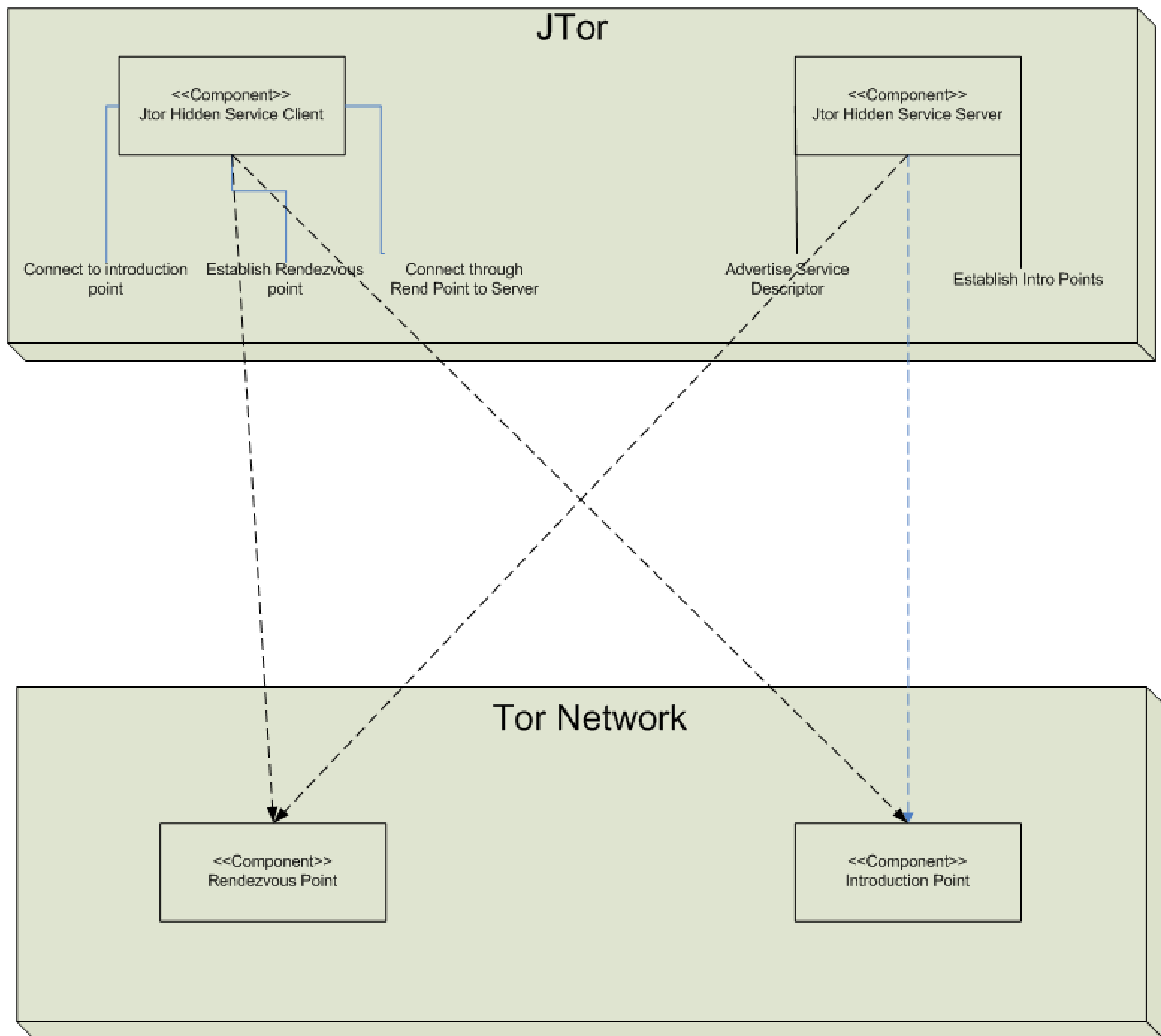
optional authentication/authorization information.

9. The Introduction point passes this on to Bob's OP via Tor, along the introduction circuit.
10. Bob's OP decides whether to connect to Alice, and if so, creates a
circuit to Alice's RP via Tor. Establishes a shared circuit.
11. Alice’s OP sends BEGIN cells to Bob’s OP, initializing the communication.

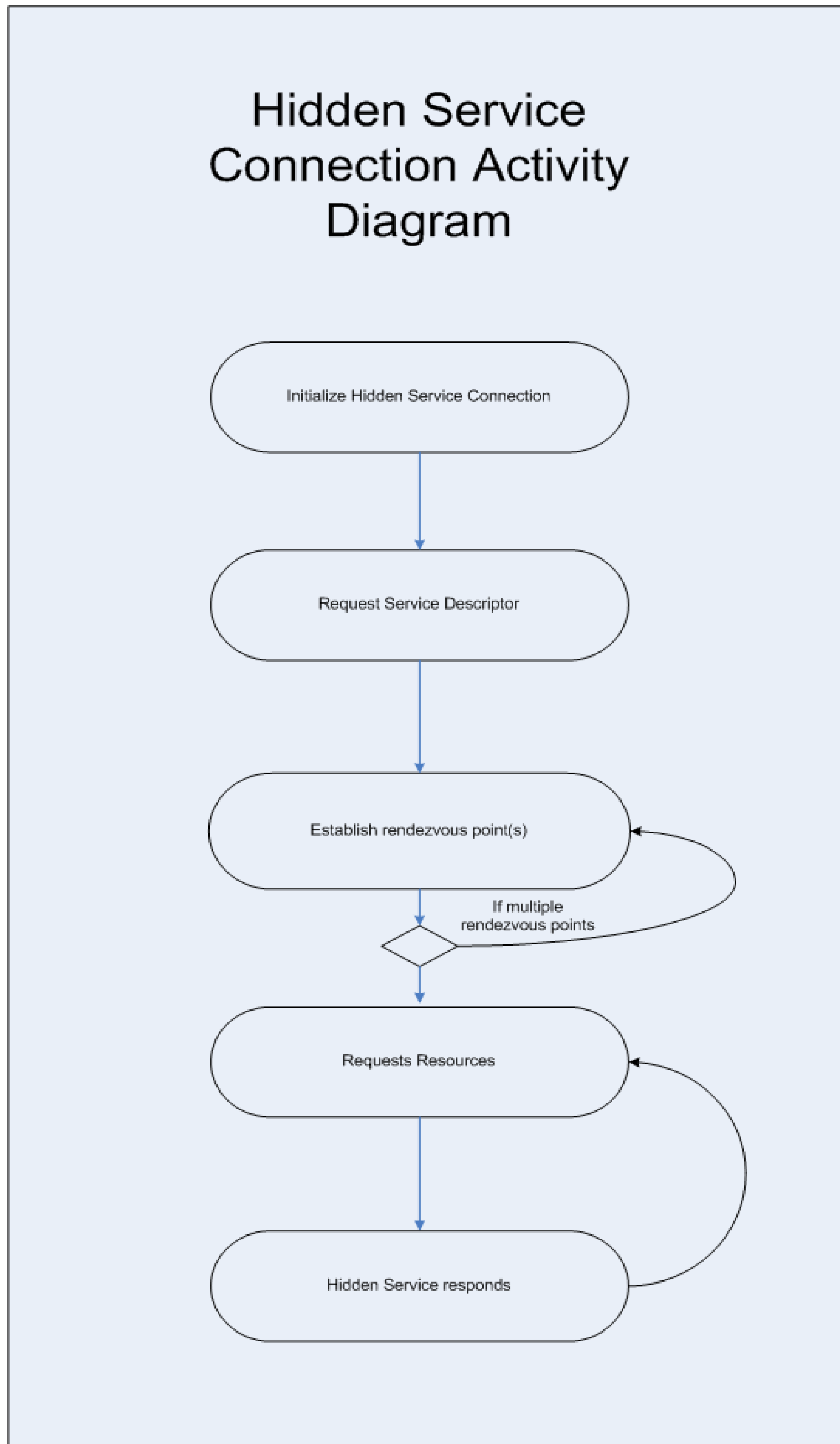
II. Use Case Diagram

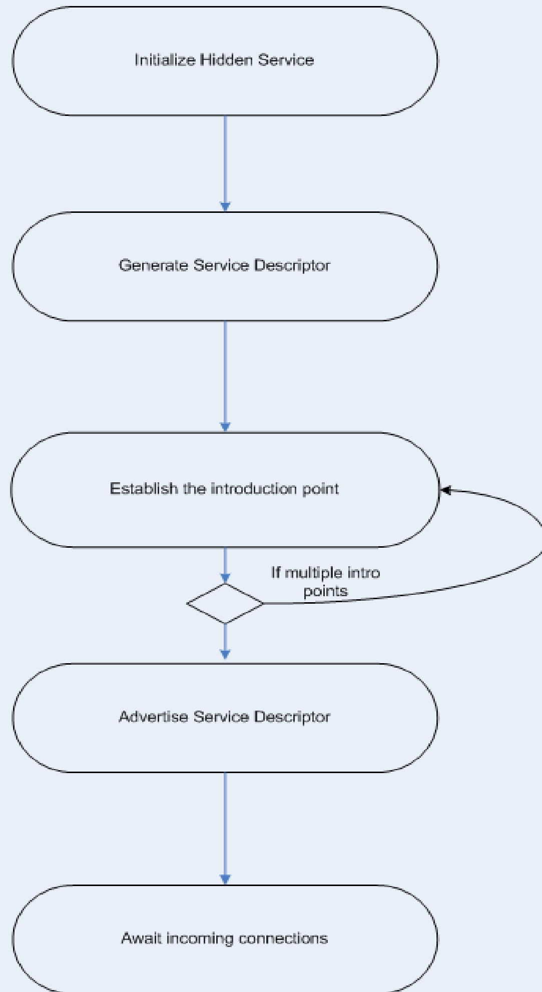


III. Communications Diagram



IV. Activity Diagram





V. Class overview

Please see javadocs for class heirarchy

http://korykirk.com/CSC8530/doc/

VI. Running Code/Test Description

Test 1: Correct encoding/encryption

I arranged tests to check certain working parts of the Hidden Service code. First of all, it is important that I was generating the Hidden Service descriptor id correctly, so I created a Hidden Service using the current Windows Tor Client, Vidalia, and used the same public/private key pair to generate the same service descriptor (same service name, keys, and ports) to check their encoding. The results for this test were successful – Vidalia generated the same service id as my code. However, the full body of the service descriptor was not the same, due to supporting different protocol versions, and using different rendezvous points (the encrypted part and the signature were different)

Test 2: advertising the service descriptors. So after the V2 service descriptor is encoded, it is sent to the primary directory authorities via an http POST request. To test this, I made sure that each post request was responded to by the directory server. Using wireshark, I watched the POST packet and the response. I also compared this to the POST and response that I observed when Vidalia was advertising its service descriptor and found them to be similar (not the same due to content).

Test 3: Encoding/Decoding Service descriptor. It is important for my client to be able to decode the service descriptor that I created, so I wrote a part to decode it to test if I was doing it properly and because I needed to do that for the client part as well.

VII. Conclusion

One of the most important parts of this project was writing good code. The goal was not necessarily to finish all of the Hidden Service code – especially because it was not as modular as we earlier thought – there are a lot of special cases that need to be put in for Hidden Service implementation, therefore those changes need to be added to the foundational part of the code. So my goal was to write as much

as I could as well as I could – with usability in mind. I wanted to provide the data for a HiddenService so that it could be easily used in the rest of the code, or in a Tor client implementation using JTor.

I learned a lot about how the Tor network works, especially in reference to Distributed Systems. I also learned a lot about cryptography in Java, especially BouncyCastle, which is a great java crypto library. Using github as version control and working on this project from the ground up was a great experience in software engineering. My code was dependent on a lot of things that were not written initially (that are still incomplete). So it was neat to see my code evolve along with the other code in the library.

In the end there are still things that need to be done before this part of the library can be finished. The possibility of establishing rendezvous circuits from introduction points is the obvious next step. This part needs to be coded into the main circuit handling code – something that I will have to work on closely with the other developer. This is because Hidden Service traffic needs to have dedicated circuits that are not to be used by anything else. Once that part is done, the client will be able to establish a circuit to the hidden service. After this, the support for optional client-authentication methods should be added. After that, the hidden service client part of the code needs to be finished. The code for fetching and decrypting the service descriptor already exists; therefore the code responsible for connecting to introduction points and establishing rendezvous points needs to be written. This part is also not modular, so I need to add some code to the existing foundational code.

I agreed to work on JTor, and I plan on continuing to integrate this code into the code base. I know a lot about the code already, especially the parts I did not write! So I plan on continuing to donate my time towards this project, and am grateful for a chance to do this kind of work as a part of my education.

VIII. Bibliography/Citations

The Tor and Hidden service diagrams are from a presentation done by Karsten Loesing -

http://www.uni-bamberg.de/fileadmin/uni/fakultaeten/wiai_lehrstuehle/praktische_informatik/Dateien/Forschung/Tor/loesing-hidden-service-activity.pdf

A lot of this information comes from the Tor hidden services spec.

https://svn.torproject.org/cgi-bin/viewvc.cgi/tor/branches/tor-0_0_6-patches/doc/rend-spec.txt?revision=1773&view=markup

IX. Appendix A: Relevant Source Code

Most of the code that I wrote (besides a few additions I made to the rest of the code base) is in the hiddenservice package, the source code can be found here:

http://github.com/koryk/JTor/tree/master/src/org/torproject/jtor/hiddenservice/

or my full repository can be cloned with the git url: git://github.com/koryk/JTor.git

below are snippits of important code.

```
//The code used to generate the descriptor id in a service descriptor, as
//well as the hidden id
public void generateDescriptorID(){
        generateSecretID();
        TorMessageDigest digest = new TorMessageDigest();
          BigInteger result = new BigInteger(secretID);
      result = new BigInteger(digest.getDigestBytes()).or(new
BigInteger(permanentID));
          digest.update(result.toByteArray());
       descriptorID = digest.getDigestBytes();
}
 public void generateSecretID(){
   generateTimePeriod();
     TorMessageDigest digest = new TorMessageDigest();
         BigInteger result = BigInteger.valueOf(timePeriod);
       if (hasDescriptorCookie())
                result = result.or(new BigInteger(descriptorCookie));
     result = result.or(BigInteger.valueOf(replica));
          digest.update(result.toByteArray());
      secretID = digest.getDigestBytes();
}

public void generateTimePeriod() {
        long currentTime = (new Date()).getTime();
        timePeriod = currentTime + ((int)(permanentID[0] * 86400 / 256)) /
86400;
 }
```

//the static constructor (only way of initializing) of a service descriptor

```
public static ServiceDescriptor generateServiceDescriptor(TorPrivateKey
privKey){
         TorMessageDigest digest = new TorMessageDigest();
         TorPublicKey publicKey = privKey.getPublicKey();
          digest.update(publicKey.getRSAPublicKey().getEncoded());
          byte[] permanentID = new byte[PERMANENT_ID_SIZE];
         System.arraycopy(digest.getDigestBytes(),0,permanentID,0,PERMANENT_I
D_SIZE);
      ServiceDescriptor ret = new ServiceDescriptor(permanentID);
```

```
            ret.setPrivateKey(privKey);
            ret.setPermanentKey(publicKey);
        return ret;
    }
```

//This is the method called before the Service descriptor is being sent

```
        public void generateDescriptorString() throws TorException{
             generateDescriptorID();
                descriptorString = "rendezvous-service-descriptor " +
formatDescriptorID() + "\n";
              descriptorString += "version " + VERSION + "\n";
                descriptorString += "permanent-key \n" + permanentKey.toPEMFormat()
+ "\n";
             descriptorString += "secret-id-part " + formatSecretID() + "\n";
                descriptorString += "publication-time " + getPublicationTime() +
"\n";
          //supported versions - not sure about this yet
          descriptorString += "protocol-versions V2 \n";
          descriptorString += "introduction-points\n";
            descriptorString += "-----BEGIN MESSAGE-----\n";
                /**all the introduction points base64 encoded if descriptor cookie
is present, then list is encrypted with AES in CTR mode with a random
               initialization vector of 128 bits that is written to
            the beginning of the encrypted string, and the "descriptor-cookie" as
           secret key of 128 bits length. **/
               byte[] introPoints = null;
              if (!hasDescriptorCookie())
                     introPoints = getIntroductionPointString().getBytes();
          else {//encrypt it
                      byte[] ivBytes = getRandomKey();
                        byte[] keyBytes = getDescriptorCookie();
                        IvParameterSpec ivSpec = new IvParameterSpec(ivBytes);
                  SecretKeySpec key = new SecretKeySpec(keyBytes, "AES");
                 try{
                             Cipher cipher = Cipher.getInstance("AES/CTR/NoPadding",
"BC");
                           cipher.init(Cipher.ENCRYPT_MODE, key, ivSpec);
                      ByteArrayInputStream bIn = new
ByteArrayInputStream(getIntroductionPointString().getBytes());
                            CipherInputStream cIn = new CipherInputStream(bIn,
cipher);
                     ByteArrayOutputStream bOut = new ByteArrayOutputStream();
                       bOut.write(ivBytes);
                             int ch;
                         while ((ch = cIn.read()) >= 0) {
                          bOut.write(ch);
                       }

                      introPoints  = bOut.toByteArray();
                  } catch (Exception e) {
                         throw new TorException(e);
                      }

                      //getIntroductionString.getBytes();
```

```
      }
         descriptorString += Base64.encode(introPoints);
  descriptorString += "\n-----END MESSAGE-----\n";
         descriptorString += "signature \n";
      //add signature string using private key.
}
```