Developer manual

TOR connectivity

The store application provides an API for developers to retrieve the details of its TOR proxy instance. With these details, developers can send and receive data for their applications across the TOR network.

Receiving TOR proxy updates

The store application sends and receives information about the TOR proxy instance using Android's broadcast system. As a developer, you application should subscribe to broadcast updates that have the following action "me.tombailey.store.PROXY_STATUS_UPDATE".

These broadcast updates will include a String intent extra which indicates if the TOR proxy instance is currently running or has stopped. In the case that the TOR proxy instance is running, the following data is provided:

Name	Type	Description
status	String	The current status of the TOR proxy instance. In this case when the TOR proxy instance is running, this will have the value "running".
host	String	The IP address or hostname which the TOR proxy instance is bound to.
port	int	The port which the TOR proxy instance is bound to.

It is important to note that the TOR proxy instance will bind to any available port when it is started. This means if the TOR proxy instance is stopped and started again it could be available on a different port. As a developer, your application should also take note of when the TOR proxy instance is no longer running to avoid data being sent to a port that maybe in use by another application. When the TOR proxy instance stops running, the following data is provided:

Name	Type	Description
status	String	The current status of the TOR proxy instance. In this case when the TOR proxy instance is not running, this will have the value "not

		running”.

Query the TOR proxy status

Before your application is started, the TOR proxy instance may already be running. If this is the case, a broadcast with the status of the TOR proxy instance will have already been sent so your application will not be aware of the current status. For this reason, your application can query the status of the TOR proxy instance.

To query the status of TOR proxy instance, your application simply needs to start the TorConnection background service. The following code can be used to do this:

```
Intent proxyQuery = new Intent();
ComponentName componentName = new ComponentName("me.tombailey.store",
"me.tombailey.store.service.TorConnectionService");
proxyQuery.setComponent(componentName);
proxyQuery.setAction("status");
startService(proxyQuery);
```

You can find more information on starting services on Android here.

Starting the TOR proxy

If your application is informed the TOR proxy instance is not running, you can request that the TOR proxy instance is started. If your application requests to start the TOR proxy instance but it is already running, a broadcast of the current proxy is sent which should inform your application that the TOR proxy instance is already running.

To start the TOR proxy instance, your application simply needs to start the TorConnection background service. The following code can be used to do this:

```
Intent startProxy = new Intent();
ComponentName componentName = new ComponentName("me.tombailey.store",
"me.tombailey.store.service.TorConnectionService");
startProxy.setComponent(componentName);
startProxy.setAction("start");
startService(startProxy);
```

You can find more information on starting services on Android here.

Sample application

For a demonstration of how TOR connectivity can be implemented into an application, a sample application has been developed and is hosted as open source code on Github here.

HTTP library

Developers should note that existing HTTP implementations may not be appropriate for applications they submit to the store. This is because some HTTP implementations may unintentionally leak data related to network requests even if they are configured to use the TOR connectivity described above. For example, the HTTP implementation may send and receive unencrypted HTTP data for a destination outside of the TOR network. This data can be intercepted or manipulated by an attacker as detailed by this interactive webpage.

To ensure secure HTTP communication, the developers of the store application has also released their HTTP library as an open source project on Github here.

Although this implementation may not be as fully featured as developers are used to with previous implementation, you are encouraged to make use of this library when working with the TOR connectivity the store provides. Since the library is open source, you may extend it and contribute to it according to your requirements.

Importing the library

The library is provided as an Android library (AAR format) that can be imported into any standard Java based Android project. You will need to clone the HTTP library and compile the Android library file on your computer.

How you import the library will depend on the development tools and integrated development environment you are using, information on importing Android libraries is available here.

It is important to note that the library has a dependency on Realm which you will also need to import separately into your application. This is caused by a limitation with Realm which requires not only our library to import it as a dependency, but your application to import it as a dependency too. Instructions on how to import Realm into your project are provided here.

Get requests

Once you have the HTTP library imported you can begin to make HTTP requests and receive HTTP responses. Requests are created using the Java builder pattern. To create a request, your application must provide details of the TOR proxy instance, the url to send the request to and the method to use when sending the request.

The library is built to work with the TOR connectivity provided by the store and requires the host and port that the TOR proxy instance is currently running on. These details should be encapsulated in a Proxy object your application can create before making a request. The following code adapted from the sample application demonstrates how this can be done:

```
String host = …;
int port = ...;
Proxy proxy = new Proxy(host, port);
```

Assuming you have created a Proxy object, your application can create a GET request to, for example “https://example.com”, using the following code:

```
Request request = new Request.Builder()
        .proxy(proxy)
        .url("https://example.com")
        .get()
        .build();
```

Once your application has defined a request, it can use the library to attempt a HTTP connection and receive a response. Being careful of Android’s restrictions on network connections on the user interface thread, simply call “request.execute()” for the library to synchronously attempt a HTTP connection. This will either return a Response object if the connection was successful or cause an IOException to be raised if it was unsuccessful. The following code adapted from the sample application demonstrates how to handle executing a request:

```
try {
        Request request = …;
        Response response = request.execute();
        ...
} catch (IOException ioe) {
        //ignored
}
```

With a Response object you can retrieve the status, the headers and the body of the HTTP response. The status code of the response can be checked with the “response.getStatusCode()” method. All of the headers for a response can be retrieved with the “getHeaders()” method or a particular header can be retrieved with the “getHeader(name)” method. Finally, the raw byte version of the response body can be retrieved with the “getMessageBody()” method or the String version, based on the character set given by the content type, can be retrieved with the “getMessageBodyAsString()” method.

Post, put and delete requests

The library also provides support for post, put and delete requests. Creating these requests is similar to creating get requests but with using post, put or delete instead of get when building the request. For example the following code creates a post request to “https://example.com”:

```
Request request = new Request.Builder()
        .proxy(proxy)
        .url("https://example.com")
        .post()
        .build();
```

These requests can be executed in the same way as get requests in order to retrieve their responses. It is likely that you will want to include a form body with these requests which provides your backend services with the data they need to operate. To do this, simply add an argument to the post, put or delete method that is of the FormBody type.

The library currently only has builtin support for form bodies which are of the “application/x-www-form-urlencoded” format using the UrlEncodedForm class. Other formats can be used with the library but will require you to create their necessary implementation based on the FormBody abstract class. To do this, simply create a new class that defines your implementation and extends the FormBody class. You will need to define behaviours for the “getContentType()” and “getBytes()” methods. The “getContentType()” method provides your implementation the opportunity to set a custom content type header. The “getBytes()” method allows your implementation to create the main body of the request to be sent.

Caching responses

The library also provides some caching capabilities which can reduce the number of requests that actually get sent across the network. This reduces latency and bandwidth usage of network connections your application makes.

In order for responses to be cached, you need to inform the HTTP library that it should cache responses. A Cache object can be created using the Java builder pattern. This object can then be passed to the library so it has the necessary information it needs to cache responses correctly.

The following code taken from the sample application demonstrates how you can inform the library to cache up to 10 megabytes of data from responses for the application in its cache directory:

```
Cache cache = new Cache.Builder()
        .context(getContext())
        .maxSize(1024 * 1024 * 10)
        .cacheDirectory(getCacheDir())
        .build();
Request.setCache(cache);
```

The cache only has to be set once during the application’s lifecycle and will automatically be applied to future requests.

Developers are advised to use the private storage their application is assigned to store cached files to ensure that only their application and the operating system can access these files.

Sample application

For a demonstration of how TOR connectivity can be implemented and combined with the HTTP library to make requests, a sample application has been developed and is hosted as open source code on Github here.