INFO 341 Project TWIT - Part 3

Due: Friday December 7, 2012 by 10:00pm

During this project you will be building a peer-to-peer microblogging application. An example of microblogging is Twitter. In this last part you put all the pieces together.

For Part 3 of this project you will modify the code you wrote in the first two parts of this project and extend it so that it can announce participation in the peer system, discover other peers participating, follow an arbitrary number of peers, and communicate status updates to an arbitrary number of peer followers.

You may use the code from Part 2 provided by the instructor and build your own Part 3 based on the instructor's solution to Part 2. However, if you choose to use the instructor's code your solution to Part 3 will be charged 10% of the total possible points for Part 3. If you were not successful with Part 2, this will allow you to continue working on the project to complete Part 3. Note that you can *read* the instructor's solution for free.

Part 3, Task 1 – Service Announcement

Each peer needs a way to announce participation in the peer-to-peer micro-blogging system. In the DNS-SD system this is called "registering" the service. Only a registered service can be discovered by other peers. However, the basic registration does not allow us to determine the specific user who has announced/registered that service. One way to connect specific users to specific peer instances is through a DNS TXT record. A TXT record (text record) consists of key-value pairs. You should minimally allocate four keys "`DisplayName`" and "`UserID`" that will hold the values of the name and unique ID from the local user profile, a "`Version`" that contains the protocol version (you should set this to 1.1), and a "`EncryptSupport`" that is a text boolean that should be set to "`false`" for now. An example of creating a text record is:

```
TXTRecord txt_rec = new TXTRecord();
txt_rec.set("Version",”1.1”);
txt_rec.set("EncryptSupport",”false”);
txt_rec.set("DisplayName",name);
txt_rec.set("UserID",uid);
```

This `TXTRecord` can then be used in the `DNSSD.Register()` call with other parameters. An example of this call is below. Other important parameters of the `DNSSD.Register()` call are the `name` which will be the user name from the local profile, `sd_service` which is a text string of the service type. You get the service type text string from the `DNSSDServiceName` field in the profile.txt file that you created in Part 2, Task 1. The `DNSSD.Register()` method also requires the `port` which is the protocol port number, and `myRegisterListener` which is an object that implements the `RegisterListener` interface.

```
DNSSDRegistration reg = null;
reg = DNSSD.register(0, DNSSD.ALL_INTERFACES, name,
      sd_service, null, null, port, txt_rec, myRegisterListener);
```

The `DNSSDRegistration` object returned by the `DNSSD.register()` method is a thread, which keeps running until you call the `stop()` method. That thread will call the `serviceRegistered()` method of the `RegisterListener` interface once when your service has been appropriately registered.

You might consider implementing a services object that you can use to initiate the service announcement and that also implements the `RegisterListener` interface. If you choose to do that, the `myRegisterListener` object would be a reference to the service object that you create.

In this task, you should modify your `Console` to implement an `announce` command that will announce the availability of your peer when the command is typed at the command line. An example might be something like:

```
--> announce
--> Registered: Berk Beachbum
      Service Type: _twit._tcp.
      Domain: local.
```

In the example above, the `announce` command results in the initialization of a `TXTRecord` and a call to `DNSSD.register().` The object that implements the `RegisterListener` interface then prints some output to the `Console` confirming that the local user has registered the indicated service type.

Part 3, Task 2 – Browsing Services

The second modification you need to make to your client is to make sure it can discover other peers on the network. The DNS-SD API calls this "browsing" for services. Much like the `DNSSD.register()` routine, browsing for services requires both making a method call and implementing a required interface. The call is `DNSSD.browse()` and it returns an object of type `DNSSDService` which is also a thread. The `browse()` method only takes two parameters, the `sd_service` and an object that implements the `BrowseListener` interface.

```
DNSSDService browser = null;
browser = DNSSD.browse(sd_service,myBrowseListener);
```

Unlike Part 3, Task 1, where the `serviceRegistered()` method of `RegisterListener` is called only once on success, the `serviceFound()` and `serviceLost()` methods of the `BrowseListener` are called frequently as services come and go – as other peers announce their services and as they stop their services.

```
public void serviceFound(DNSSDService br, int flags, int ifIndex,
      String serviceName, String regType, String domain)
```

```
public void serviceLost(DNSSDService br, int flags, int ifIndex,
      String serviceName, String regType, String domain)
```

You should use the required methods `serviceFound()` and `serviceLost()` to maintain a list of peers who are available on the network. Make sure that the routines post notices to the `Console` when peers come and go off the network. Modify your `Console` so that when the command `browse` is typed at the command line your system starts browsing for services. An example of that might be:

```
--> browse
--> Online: Jonny B. Goode
Online: Sandy Surfer
Online: Berk Beachbum
```

In this case, the `browse` immediately found three possible peers. Notice that the `serviceFound()` method writes to the `Console` each time a new peer is found.

Modify your `Console` so that the command `showOnline` shows a list of peers who are online. Providing an ordinal value for each peer in the list will make it easier to follow someone or get a profile of someone who is on the network. An example of this would be:

```
--> showOnline
      1: Jonny B. Goode, (ff4da7bd) @ PuffinMini.local.:21255
      2: Sandy Surfer, (0b3cffa7) @ lisa-mbpro.local.:21222
      3: Berk Beachbum, (ac340145) @ dwmc-mbpro.local.:21232
```

You might notice that the online list also contains the hostname and the protocol port that the peer is using. This is because the code for `serviceFound()` makes a service query (called a "service resolution") each time a new peer is found. Service query is described in the next task (Part 3, Task 3 – Service Query).

Part 3, Task 3 – Service Query

While the browse operation tells you who comes online and who goes offline, the notification does not have enough information to contact the peer and communicate through the protocol you wrote in Part 2, Task 3. That requires a service query, also called a "service resolution" or a "resolve." Like registration and service browsing, a resolve creates a thread. The thread is started by a call to `DNSSD.resolve()` which returns a `DNSSDService` object.

```
public DNSSDService resolver = null;
resolver = DNSSD.resolve(0, DNSSD.ALL_INTERFACES, name,
      sd_service, "local", myServiceResolver);
```

Like the other calls, `DNSSD.resolve()` assumes that you have an object that has implemented the `ResolveListener` interface.

```
public void serviceResolved(DNSSDService serv, int flags, int ifIndex,
      String fullName, String hostName, int port, TXTRecord rec) {
```

The `serviceResolved()` interface provides three pieces of information that you need to resolve who is connected to a specific peer. It provides the `hostName` and `port` which provide the two pieces of information your code needs to contact a specific peer, but it also provides the `TXTRecord` that allows you to get the `UserName` and `UserID` to validate which user is at a specific peer.

You should start a `DNSSD.resolve()` anytime your code for `serviceFound()` finds a new peer. That way you will always have the hostname and protocol port when it is needed, like for a `status` delivery or for a `profile` request.

A service query does not need to tell the `Console` when it resolves a specific peer, but the local peer should keep track of how to contact followers from the `followers.txt` file and any others who are online. You will need that information to be able to deliver status messages to existing followers, to request profiles from followers or other peers who are online, or to follow/unfollow new peers.

Part 3, Task 4 – Status Delivery

Now that you have implemented service announcement, service browsing, and service resolution, you are ready to re-implement `status` messages. Change the `status` command in your `Console` so that it is sent to every follower of the local peer who is currently online. In Part 2 you only needed to send status messages to the one peer to which the local peer was connected. However, now, your local peer does not need to maintain any connections. Your local peer should simply look through the list of followers who are online and send those followers the `status` update. You can connect to one online follower peer at a time, send the status message, and then close the connection. Your client will only need to maintain one (short lived) outgoing connection.

Part 3, Task 5 – Requesting a Profile

You should now re-implement the `getProfile` command so that you can request the profile from any user who is in the list of online peers. Assume the `showOnline` command lists the following:

```
--> showOnline
      1: Jonny B. Goode, (ff4da7bd) @ PuffinMini.local.:21255
      2: Sandy Surfer, (0b3cffa7) @ lisa-mbpro.local.:21222
      3: Berk Beachbum, (ac340145) @ dwmc-mbpro.local.:21232
```

Then the following `getProfile` command

```
--> getProfile 2
```

should request and display the profile of the peer "Sandy Surfer," who is the second item in the list of online peers. Note that depending on how you implement the list of online peers, the local peer might be discovered by the `browse` command. You should trap that possibility so that the user does not request the profile of the local peer. That is, a local peer should be prevented from requesting a profile from itself.

Part 3, Task 6 – Following

You should now re-implement the `follow` command so that the local peer can follow or stop following any user who is in the list of online peers. Given the list of online peers in the section above, the command:

```
--> follow 2
```

would result in the local peer following "Sandy Surfer" which means that when the local peer is online, it would receive any status messages from Sandy Surfer. You can only `follow` or `follow remove` a peer who is currently in the online list. Much like the re-implementation of `getProfile`, you should detect if the local peer is in the online list and prevent an attempted `follow` or `follow remove` of the local peer by the local peer.

Part 3, Task 7 – Modifying Console

As a quick review, you need to make a number of modifications to the `Console` command line to complete Part 3. In this part of the project, your `Console` will have implemented or re-implemented the following commands:

`announce` – This command announces your peer using a DNSSD.register() to make the service available to other peers.

`browse` – This command starts a DNSSD.browse() to discover available peers.

`showOnline` – This command shows the list of peers that have been found through browsing.

`status` – This command should now deliver status messages to all followers who are currently online.

`getProfile <#>` - This reimplemented command now accepts a parameter (a number) which is the number of the peer in the online list from which a profile should be requested.

`follow <#>` - This reimplemented command now accepts a parameter (a number) which indicates the peer in the online list that should now be followed.

`unfollow <#>` - This reimplemented command now accepts a parameter (a number) which indicates the peer in the online list that should no longer be followed.

`halt` – It might not be obvious, but since almost all of the DNSSD.* calls result in an object that creates a thread, your halt should make sure that all threads are appropriately stopped before quitting the Console.

Turning In
Your Java code should compile and run correctly from the command line. If you use a development environment you should make certain that your code will compile and run from the command line.

All of your Java classes should be in a `package` that is named by your lastname and the last three digits of your student ID. So if your last name is "McStudent" and the last three digits of your student ID were "820" your code would include a statement like:

```
package mcstudent820;
```

If you have your Java CLASSPATH variable set correctly you would be able to run your Console from the command line by typing:

```
java mcstudent820.Console
```

You will turn in your Java source files, your profile.txt, and followers.txt files as one Zip file. Since packages are assumed to be in directories that are named the same as the package (e.g., in the example above, your code would be in a directory named "mcstudent820") you can just Zip that directory and submit the single file through the course Catalyst Dropbox. Absolutely no assignments will be accepted through email.