P2P Protocol Definition

DISCOVERY
Use JmDNS for discovery. The service type you should use is "_cs6452._tcp.local."

There is still a bit of flakiness in JmDNS. In particular, sometimes requests to resolve a service aren't handled. It's fairly easy to work around this. In my version of the code, did the following:

- When you first detect a service (through serviceAdded), fire off requestServiceInfo to try to resolve the service. Also, add an entry to a dictionary where the name of the service is the key, and the value is a new Date object that provides a timestamp on when the request was sent:

```
self.jmdns.requestServiceInfo(event.type, event.name, 5000)
self.pendingRequests[event.name] = util.Date()
```

- In our serviceResolved method, remove entries from this dictionary once they are resolved:

```
del self.pendingRequests[event.name]
```

- Create a new thread that will wake up periodically and check for services that are still in this list, and have a timestamp that is older than, say, 10 seconds ago. Reissue requests for these to be resolved.

```
self.jmdnsMender = threading.Thread(target=self.mendJmdns)
self.jmdnsMender.start()

def mendJmdns(self):
   while 1:
      time.sleep(10)                                  # wake up every 10 seconds
      for key, value in self.pendingRequests.items()
         now = util.Date()
         if now.time - value.time > 10000:            # milliseconds
            self.jmdns.requestServiceInfo("_cs6452._tcp.local.", key, 5000)
```

PROTOCOL BASICS
Each request should open a new connection to the peer, send a correctly formatted

request, wait the response, and then close the connection.

(NOTE that this is different from the IM program, where every client maintains a persistent connection to the server.)

FINDING OUT ABOUT USERS
Once you've discovered a peer through JmDNS (and thus have its mDNS name), and have the service info for it (the IP address and port on which the peer is listening), you can begin to issue requests to it for additional information, to start chats, etc.

To find out about the user running the peer, issue a GET_ USER_INFO request:

    GET_USER_INFO

The recipient should respond with a pickled dictionary containing key/value pairs that provide additional user information. At a minimum, your dictionary should contain the following data:

Key	Value
"Long Name"	<string containing the name of the user>
"URL"	<string containing a URL to the user's home page>

You can include extra stuff if you want, but receiving peers aren't necessarily expected to do anything with the data.

BROWSING AND GETTING FILES
List the shared files on a remote peer by issuing LIST_FILES:

    LIST_FILES

Peers should return a pickled Jython list containing the names of the files:

Note that these don't have to be "real" full-path filenames. The easiest thing to do is return the short names of files that exist in a specified directory.

Once you know that a given filename is available on a peer, you can request it via GET_FILE:

    GET_FILE <filename>

Note that this should be in a separate connection. Since the filename is just a simple string, there's no need for pickling it. Just look for the first space after the GET_FILE command and treat everything after that as the filename. The peer should respond with the actual bytes contained in the file: just read it into a string and return it. Again, in this case, there's no need to pickle the return result.

To send a file to someone (as a result of a drag'n'drop in the UI for example), you send an OFFER_FILE message:

OFFER_FILE (<mdns_name> <filename>)

The argument is a pickled Jython list containing the sender's mdns name and the filename being offered. There is no reply. If the recipient chooses to take the file, it can use the GET_FILE message as above.

CHATTING

Chat functionality isn't pure peer-to-peer, mostly because I don't want to introduce issues of maintaining distributed consistency of the chat membership list. So, the way to think of a chat is that it is "hosted" by the client that initiated the chat. This client gets to invite people to the chat, and receives the acceptance or rejection notices. This allows the initiating peer to keep the "true" list of current chat members, although it means that other peers (invitees) do not have the ability to invite people into the chat.

Once you decide to initiate a chat with a peer, issue the INVITE message:

INVITE (id, mdnsName)

The arguments are a pickled Jython tuple containing the integer ID and the string mDNS name of the peer issuing the invitation. (Yes, this simple protocol allows you to fake invitations from others.)

The peer should respond with either the string ACK to acknowledge and accept the chat, or NACK to refuse it.

The initiating peer should keep a data structure that contains the names of peers that are currently joined into the chat. Each time an acceptance comes in, the initiator should notify all peers currently in the chat through a CHAT_STATUS message:

CHAT_STATUS (id, mdns_name, [member1, member2, ..., membern])

The argument is a tuple containing the chat ID (as the first argument), the mDNS name of the initiator of the chat (as the second argument), and the updated list of

the chat members (as the third argument). This message indicates to all peers who is currently in the chat, so that they can update their display.

Any peer can generate messages to any other peer in the chat. To send a message to other chat members, use MESSAGE:

MESSAGE (id, sender, text)

The argument is a pickled Jython tuple containing the chat ID, the sender of the message, and the text of the message. You should send this to every other member of the chat.

Any member can leave a chat by sending the GOODBYE message to the original initiator of the chat:

GOODBYE (id, sender)

The arguments are a pickled Jython tuple containing the chat ID and the mDNS name of the party leaving. The original initiator, upon learning of the departure of a member, should notify the remaining peers through a CHAT_STATUS message.

Only the initiator can "close" a chat completely. This is done by sending a CLOSE_CHAT message to any remaining peers in the chat. This can be triggered by the initiating user closing his or her chat window, for example:

CLOSE_CHAT (id, sender)