Appendix A
P2P Network Command Structure and Code Listing

Commands sent in the P2P network that I implemented for this project can be divided into three categories as follows:

1. Commands sent from peers to the tracker server
   a. Create session command (COMMAND_CREATE_SESSION)
   b. Join session command (COMMAND_JOIN_SESSION)
   c. Get session details (COMMAND_GET_SESSION)
   d. Get list of all sessions (COMMAND_ALL_SESSIONS)
   e. Leave session (COMMAND_LEAVE_SESSION)

2. Commands sent from the tracker server to peers
   a. Peer updates when new peers join (COMMAND_PEER_UPDATE)
   b. Exit update when peers leave (COMMAND_PEER_EXIT)

3. Commands sent from peer to peer.
   a. Send a simple hello message (COMMAND_PEER_HELLO)
   b. Initiate a ping session (COMMAND_OPEN_PING)
   c. Initiate a spruce test (COMMAND_SPRUCE_START)

The #define signatures for the commands are shown in parentheses. The actual commands are strings, their values can be seen in the common.h header file that is identical in both the TrackerServer and P2P_test applications.

All commands are sent in a uniform JSON format. The advantage of JSON is two-fold. First, JSON data is extremely simple to create and to parse in Objective-C thanks to the NSJSONSerialization class. Second, JSON data allows for expandability. If a developer were to decide to add new keys later in the life of this command protocol, they could do so without affecting old versions of the program. This is because keys in a JSON dictionary that are not actively requested have no effect on anything programmatically (except the size of the payload).

Commands generally conform to the following structure:

```json
{
  COMMAND_KEY: "command name",
  PARAMETER1: "value1",
  PARAMETER2: "value2",
  ...
  ...
}
```

Where COMMAND_KEY (and all other keys that will be capitalized from here on out) is a constant defined in common.h. PARAMETER1 is an unspecified key constant specific to the command.

Responses generally follow the following structure:

If an error occurs:

```json
{
  STATUS_KEY: STATUS_ERROR,
  ERROR_CODE_KEY: 123,
  ERROR_MESSAGE_KEY: "Error code 123: something bad"
}
```

If no error occurs, the value of STATUS_KEY is STATUS_OK and command-specific parameter keys and values follow.
COMMAND_CREATE_SESSION
1. Parameters:
   • COMMAND_KEY: COMMAND_CREATE_SESSION
   • NAME_KEY: name for the new session
   • PEER_PORT_KEY: port number on which the sending peer is listening
2. Response parameters: (Session detail response packet)
   • STATUS_KEY: STATUS_OK
   • PEER_COUNT_KEY: number of peers in the session (1 for create)
   • PEER_ARRAY_KEY: array of peer JSON dictionaries with the keys below:
     o PEER_ADDRESS_KEY: peer IP address
     o PEER_PORT_KEY: peer listening port number
     o PEER_ID_KEY: unique ID of peer
   • NAME_KEY: name of session
   • SESSION_ID_KEY: unique ID of session
   • PEER_ID_KEY: unique ID of the peer that sent the original command

COMMAND_JOIN_SESSION
1. Parameters:
   • COMMAND_KEY: COMMAND_JOIN_SESSION
   • SESSION_ID_KEY: unique ID of the session to join
   • PEER_PORT_KEY: port number on which the sending peer is listening
2. Response is the same as the create session response

COMMAND_GET_SESSION
1. Parameters:
   • COMMAND_KEY: COMMAND_GET_SESSION
   • SESSION_ID_KEY: unique ID of the session to join
2. Response is the same as the create session response

COMMAND_LEAVE_SESSION
1. Parameters:
   • COMMAND_KEY: COMMAND_LEAVE_SESSION
   • SESSION_ID_KEY: unique ID of the session to leave
   • PEER_ID_KEY: unique ID of the peer that's leaving
2. Response: There is no response to this session since the peer sends it when the app wants to quit and the peer doesn't want any more traffic. So it's a send and hope for the best situation for the peer.

COMMAND_PEER_UPDATE
1. Parameters:
   • COMMAND_KEY: COMMAND_PEER_UPDATE
   • PEER_INFO_KEY: dictionary with the following values
     o PEER_ADDRESS_KEY: peer IP address
     o PEER_PORT_KEY: peer listening port number
     o PEER_ID_KEY: unique ID of peer
2. No response for efficiency in large number of peers. Not worth wasting time computing error cases

COMMAND_PEER_EXIT
1. Parameters:
   • COMMAND_KEY: COMMAND_PEER_EXIT
   • PEER_ID_KEY: unique ID of the peer that exited
2. Again, no response because this is potentially sent to a large number of peers simultaneously
Code Listing

All of the code was written using Apple’s Xcode IDE for both OS X and iOS applications. Therefore, it can only run on a Mac with Xcode installed (including the iOS SDK packages). It can be opened in Xcode by opening the *xcodeproj file in each application folder. This will present the user with a file browser and code editor for the entire project. Then you can simply make sure that the build target is correct in the top left (application name and device type, either “My Mac 64-bit,” “iOS device,” or “iPhone/iPad simulator”) and press the run button. The build settings are already set up so that the code will compile and link properly and the application will start. To end a Mac application, either quit as normal or press the stop button. To end an iOS application in the sim, press stop or quit the simulator. To end an iOS application on a device, press the stop button or unplug the device. Alternatively, one might double tap the home button to see backgrounded apps and press and hold the icon until a red button appears to remove it from memory. If you do this while the debugger is attached, then the debugger will think that the app crashed and show errors.

NOTE: You will NOT be able to run the iOS code on a device unless you are a member of the iOS developer program and sign the code with your developer certificate.

The relevant code files for each application are listed below:

The TrackerServer OS X application:

AppDelegate.h and .m
• These files define the application delegate object, which the main function specifies when it begins the application run loop. In the case of the TrackerServer application, the app delegate serves as a controller for the main Window and as the main server program. It is the delegate of a GCDAsyncSocket listening socket and implements callbacks for server events and command parsing and handling.

Common.h
• The header file of common definitions across all source files. Defines the command keys and parameter keys as well as error values, etc.

GCDAsyncSocket.h and .m
• This is an open-source socket object with an API for asynchronous socket programming under the GNU public license. See References for a github link.

MainMenu.xib and SessionDetailWindowController.xib
• XML Interface Builder files that specify the layout of the main GUI and the session detail panel GUI respectively. These files contain links to outlets in code specified by the keywords IBOutlet and IBAction.

SessionDetailWindowController.h and .m
• Controller object that controls the session detail panel that pops up when the user double clicks a session in the list in the main window.

Peer.h and .m, Session.h and .m
• These are data objects that store the details of the peer and session data session maintains a strong pointer reference to a list of peers and peers contain a weak reference to their session. The reason for one being weak is to avoid retain cycles in memory management. Since Objective-C frees memory that has a 0 retain count, they cannot both retain one another. Both data objects have the ability to dump themselves as JSON data.

TrackerClient.h and .m
• This object manages a list of open client sockets. The application can use it to send commands to peers in way that is functionally distinct from the server and also receive responses.

The P2P_test iOS application:

AppDelegate.h and .m
• As above, the application delegate that responds to application loop callbacks (lifecycle
callbacks). And serves as our entry point for beginning the application

Common.h and GCDAsyncSocket.h/.m (same as above)

LocalPToPServer.h and .m
• This is a singleton object that implements the server callbacks for a listening socket that the
iOS application maintains. This is also the primary data manager for the application. Since it
receives the data about sessions from the tracker it also stores it in memory. Since it is a
singleton, it can be asked for this data without a reference to the server being passed around.

LocalClient.h and .m
• Like with the TrackerServer application, here the client object is a singleton that manages a
list of open client sockets for sending commands to peers and to the tracker and for
receiving and parsing responses. It serves as the delegate for many potential
GCDAsyncSocket instances.

RootViewController.h and .m
• The root table view of the applications navigation controller that serves as the root of the
navigation tree for both iPhone and iPad

PeerViewController.h and .m
• The view controller for the main page of the P2P functionality on the application. It acts as a
GCDAsyncSocket delegate and sends set up commands and obtains information from the
tracker before the peer joins a session.

Peer and Session objects
• These objects are data objects very similar to those in the TrackerServer application for
storing information about sessions and peers.

RunningSessionViewContoller.h and .m
• The table view controller for displaying information about a joined session. It listens for
notifications from the server singleton about updates and reloads its UI in real time

Spruce_snd.c and spruce_rcv.c and spruce.h
• Spruce algorithm implementation files. These files are modified from the original open
source downloads (see references). They are modified to allow error message passing,
argument parsing, and proper memory management. They also are modified so that they can
be run multiple times in one application run session.

MainStoryboard_iPhone.storyboard and MainStoryboard_iPad.storyboard
• XML files that organize the application UI layout and segues between UI screens for use in
the Interface Builder.

Other view controllers
• There are a few other view controller subclasses that handle other application functionality
that may be of interest for testing basic client and server functionality. There is also the
SettingsViewController for managing a settings view.

In addition to these two applications, I have submitted an iOS application called Spruce, which is a
simple spruce client designed to run the spruce test between an iOS device and a spruce receiver
running on another machine. This wasn’t really part of the project but was an important proof of
concept application. It also contains modified versions of the spruce source code found online. The
most important modification is that the size of long integers is adjusted for because they are smaller
on iOS devices than on 64-bit Macs, which could cause frame-shift errors in the original spruce
implementation.