A Model for Peer-to-Peer Mobile Interactions and Network Bandwidth Reservation

Michael Brandt
May 1, 2013
CPSC 490: Senior Project for Computer Science
Advisor: Dr. Richard Yang
Abstract

The goal of this project is to propose and implement components of a mobile peer-to-peer (P2P) network and to design a process by which this network could communicate with ISP controllers to reserve bandwidth. This project presents a design for the structure and operation of a mobile P2P network backbone that can be used for more general P2P interactions. Its implementation is also presented. Furthermore, this project proposes a process by which this type of P2P network can communicate with an ISP’s network resource control infrastructure to reserve bandwidth for the duration of the interaction. To do this, the P2P network must have knowledge of its overall topology and capacity. Therefore, the network must have the ability to measure capacity along individual connections accurately and efficiently, even given a large number of concurrent interactions on the network.

The project consists of three components. First, it presents the design and implementation of a mobile P2P network in iOS that maintains network topology information. Second, it implements methods to determine the network’s capacity using the Spruce algorithm. Third, it presents a RESTful API design that a network application could use to notify any network control infrastructure of the P2P network topology and requirements. Then, ISPs could reserve bandwidth “channels” to guarantee the functioning of the network. As a final touch, a proposed Quantum API extension is outlined that the P2P application may use to communicate with a virtual network environment for simulation and testing purposes.

Introduction

Most of what we think about when we discuss online services and interactions today can often be reduced to a series of simple one-to-one interactions between clients and servers. Even massive social networks on the scale of Facebook or Twitter that facilitate huge amounts of many-to-many interactions between individual users follow this model; one-to-one client-server interactions effectively give the appearance of direct interactions between peers.

Problems arise, however, when the interactions between peers are both time- and bandwidth- sensitive as well as high volume (that is, a single exchange between a group of peers requires a large data transfer not necessarily a high volume of exchanges). In addition, when online interactions between individuals are large in scale, an application must devote a large number of servers to any given session of the application. While this is scalable given enough space (both physically in datacenters and digitally), it is not necessarily the best option when given a large number of peers sharing real-time information because there are such high setup costs.

It is easy to envision a few key interactions between groups of individuals that meet the criteria of being time- and bandwidth-sensitive. Sharing large files and streaming video and audio are obvious examples. Online audio and audio-video conferences clearly meet these requirements. The sheer volume of data transferred between peers in a single interaction is large, it is time-sensitive because one peer experiencing a lag in audio can ruin the experience, and it is bandwidth-sensitive,
because if the nature of network traffic changes or some peers have a lower bandwidth connection than others, quality is negatively effected. However, the most salient example of modern day multi-peer online interactions whose scale is already outpacing many networks’ capabilities is online multiplayer gaming.

In a single online gaming session, each end user must be constantly updated in real time on the activities of every other end user as well as other changes to the game environment. They must also continuously upload their own activities so their peers can be updated. In fast-paced action games, this is clearly a very time-sensitive situation. Similarly, a single user without the required bandwidth capabilities can be left with a poor user experience and negatively impact gameplay for other users. Because of this, without extremely well designed, dedicated servers that few entities have both the ability and incentive to maintain, online multi-peer interactions must be limited to relatively small scales. While gaming is an example of a desire for large-scale multi-peer interactions today, it is not out of the question to imagine that many applications that have yet to be realized because infrastructure set-up costs bar the development community at large from creating such interactions. Creation is instead restricted to a few large, dedicated companies, like Blizzard for gaming, or Citrix for business meetings.

A solution to the problem of high infrastructure costs is to look to Peer-to-Peer (P2P) networks to facilitate large-scale multi-peer interactions. The scalability wall that the client-server model runs into is the result of the fact that the number of connections between peers that a server must facilitate grows quadratically as the number of peers increases. P2P networks, on the other hand, have low capacity in the small scale, but the capacity of the network grows as more peers join, because each peer acts as both a client and a server.

An issue that arises in P2P networks, however, is that Internet connectivity is far from uniform across peers. This presents a difficulty in the network setup required for facilitating such interactions, which is that the limits of the interaction are not necessarily known from the outset. Some potential ways to alleviate this problem might be to design the network applications to only update interested parties. For example, the application could require each player to only update nearby players in an online game. However, this solution is only partial and breaks down when all or many parties are interested in updates, such as when audio or video conferencing is involved. Additionally, to alleviate the problem the network application might make higher demands of peers with strong connections, for example by asking them to forward data that weaker peers cannot keep up with. The “Donnybrook” system is an example of an attempt by Microsoft which presents yet another potential solution to this problem. It attempts to estimate what aspects of online gaming interactions are most important to players and convey those with high fidelity and simulate the rest with occasional, lower fidelity updates [1].

While there is great focus on this problem in gaming and other sectors for networked personal computers and gaming consoles, a rapidly growing portion of
the networked community that will require more and more of these large-scale interactions with many peers is that of mobile devices and wireless networks. In the coming years, it will be necessary for mobile networks to confront the problems of large-scale time- and bandwidth-sensitive online interactions. For this reason, this project will focus on network interactions between mobile devices.

**Project Description**

My project consisted of three parts. First was the design and implementation of a P2P network “backbone” for iOS. That is, a working set of network applications that facilitate the creation and maintenance of P2P networks amongst iOS devices. This set of applications sets up the network and enables direct communication between a large number of peers and is designed so that application functionality (e.g. a game) can be added in without worrying about network functionality (hence why it can be thought of as a “backbone”). There seemed to be little precedent for open source P2P network implementations for iOS, so I designed this structure largely from scratch.

For this first part, I needed to create a communication structure that Peers could use to notify coordinating servers (“trackers”) of their intent to join or leave a network. Similarly, the tracker servers needed to be able to update peers on changes in the network. In addition, we needed to create a similar communication structure by which Peers could communicate with one another directly after network setup. It was important for the tracker server to always maintain data structures that represent information about the current state of the network so that it could ultimately perform the necessary interactions with ISPs to reserve bandwidth.

The second part of the project was to research and implement a method of reliably determining a network's bandwidth capacity. The main issue to consider was how to deal with lots of peers in a P2P network sending a lot of network traffic simultaneously. Several papers ([3], [4], [6]) pointed to the Spruce algorithm being a good candidate for the measurement method. So in this stage, I adapted available implementations Spruce algorithm [10] to work in iOS and to interact properly with the P2P network backbone. All code is described and documented in appendix A.

The third and final portion of this project was to design a RESTful API that the client application can use to notify the ISP’s controllers of network topology and bandwidth needs. In doing so, the client P2P application can request the reservation of network resources. The separation of this API from the workings of the P2P network allows for the client application’s peers to avoid conflating their workings with the workings of the network resources of various ISPs. A description of this API design can be found in appendix B.

It was never feasible to actually implement and test such an API, so the original intention was to simulate the allocation of network resources using
OpenStack Quantum. Quantum is an API-driven system for creating and maintaining networks in code and could be used for creating a virtual network. In appendix C, I provide an outline of how the P2P backbone’s tracker server can use the Quantum API [14] and how the API can be extended to provide the ability to simulate real-world bandwidth reservation.

Part 1: P2P Network Design and Implementation

Structure & Design

To design a working P2P network application for iOS, I could find very little to use as a starting point. There are plenty of discussions of implementing a client-server model application in iOS, both in the Apple developer resources [15] and on various online forums. Similarly, there are many open source libraries that can be used to facilitate such designs. However, there are very few P2P implementations in iOS that are accessible enough to suit my needs. I needed to be able to modify and have direct control over the networking code on individual peers and I also needed to be able to modify any coordinating servers to keep track of the network and ultimately, send API calls to reserve bandwidth resources. Apple’s GameKit framework allowed for the first case. GameKit allows for easy “matchmaking” between peers running the same app. The intention is to put peers in contact with one another to play a game against each other. It is then possible (though not as trivial) to implement one’s own P2P code once the peers are in contact with one another. However, the act of coordination is limited in large-scale conditions and is opaque, so it would be difficult for a coordinating server to maintain information of the entire network and make reservation API calls. Ultimately, it made more sense just to design the coordinating server as well as the peer application.

The high level concepts of my network design are based loosely on the concepts of the BitTorrent protocol [11]. My network contains two main types of units, a coordinating (“tracker”) server and peers, which act as both servers and clients. The tracker server is fixed so that peers can contact it at the time of network initialization. The tracker keeps track of potentially many groupings of peers (referred to as “sessions”). Peers announce their intention to join a session or create a new session to the tracker server. Peers then ask the tracker server for information about the session and all the other peers that have joined it. Therefore the P2P network creation can be thought of as a two-phase process.

This two-phase process of network creation is shown in figure 1. In the first phase, peers announce themselves to the tracker and obtain information from the tracker about the current state of the session. In the second phase, peers contact each other without the mediation of the tracker server, but they continue to listen for updates from the tracker about the joining of new peers, which are sent out when those peers are in the first phase.
There are a few independent operational units that had to be implemented for the operation of the network. The actual tracker server program is a unit that is designed to be a stationary server instance that manages the P2P network. It listens for incoming announcement commands, updates the network accordingly, and sends responses. The tracker program has a “client” unit that is independent of the server. When the program updates the representation of a session (in response to a server command), then the client unit is tasked with notifying the whole session of the change. Similarly, the peer application has two independent units, a server that listens for connections, both from other peers and from the tracker, as well as a client unit that can initiate connections with other peers.

**Implementation Details**

The code can be downloaded from the CPSC 490 site for my project. A code listing and description is given in appendix A along with a description of the command format for communication amongst peers and between peers and the tracker. This is occasionally referenced in the description that follows.

There are two applications in the P2P network implementation. Both are written in a combination of C and Objective-C (this is technically redundant, since Objective-C is a proper superset of C). Both leverage an open source Objective-C library called GCDAsyncSocket, which uses both Grand Central Dispatch (GCD) and CFNetwork to allow for thread-safe, asynchronous, BSD socket programming. GCD is a C library provided by Apple that allows a different approach to multi-threaded programming. GCD allows for the creation of queues that run on different threads. Operations can then be added to these queues in the form of functions or blocks. Blocks are a C extension that allow for some measure of functional programming. They are like chunks of code that can be passed around and called. The GCDAsyncSocket API provides the nitty-gritty code for setting up sockets and allows developers to simply provide callback functions (and the GCD queues that they run on) for ease of multi-threaded socket programming. The two applications use a Model-View-Controller design philosophy. Data objects (model), UI objects (view) and the implementation of logic and control flow (controller) are kept separate.
The first application, TrackerServer, is an OS X application written in Objective-C that uses the Cocoa framework for connecting to a GUI. The tracker server maintains two main types of data objects: Sessions and Peers. Sessions contain information about a single P2P session, including its name and unique ID as well as a list of all the peers that have joined that session. A peer object contains information about a peer that has joined a session including a unique ID and the IP address and port on which the peer can be expected to be listening.

When the application starts, the user is presented with a window that contains a text field for inputting a port number to start the tracker server on. When the user presses start, the server begins listening for incoming connections of the proper command format described in appendix A. The main window (class UIWindow) of the application displays a list (class NSTableView) of sessions that the tracker is currently maintaining. This list is empty until a peer announces its intent to create a session. The UI list updates in real time when peers create sessions. Double-clicking a row in the list opens another UIWindow instance of the panel style. This style has a smaller top bar and always stays in the foreground to provide details of the session. In the current implementation it provides a list of peers and their basic information including IP address and listening port. An example of what the TrackerServer application looks like at runtime is shown in Figures 3 & 4.

Figure 3: The main window of the TrackerServer application displays a real-time updating list of current sessions of the network.
The second application, P2P_test, is an iOS app written in Objective-C and C that uses the Cocoa-touch framework for its GUI. For a full description of how it works, see the code listing in appendix A. The application uses a UINavigationController to handle UI transitions. Transitions were implemented using iOS storyboard segues so that the main focus of the application could be the underlying networking rather than the UI. The app is universal, meaning that it contains a native UI for both iPad and iPhone and the code differentiates between the two and branches. However, I tried to design the app in such a way that the view controller objects are reusable for both iPhone and iPad. I was able to do this using NSNotificationCenter to post information. In this way, unique view controllers to one type of UI could respond to notifications and if that view controller doesn’t exist, then the notifications would simply be ignored.

The application itself also uses the GCDAsyncSocket library to send announcement commands to the tracker. The address and port of the tracker need to be set in the app preferences or the user will see a “connection refused” or “connection timed out” error. In a real deployment, the tracker servers would be known and wouldn’t move, so this would be unnecessary. The main screen presents a few options for testing features, but if a user taps “Run P2P” then the app contacts the tracker server and displays the current list of active sessions. An example of this screen is shown in figure 5. The user can tap one of these sessions to send a join command or can tap “Create New Session” to send a create command. In the background, the app tells the singleton object LocalPToPServer to begin running and sends its listening port information to the tracker server. This singleton server instance maintains the session data and peer information locally while the P2P network is running.

After joining a session, the user is presented with a list of IP addresses of peers that are currently in the session (figure 5). In addition, when a new peer joins, the TrackerServer application sends a peer update command to every peer.
currently in the session. The singleton server object that runs on every peer listens for these commands and posts notifications using NSNotificationCenter that the Session information has been updated. The view controller for the session listens for these notifications and updates the data when they are posted, so the user is updated in real time of peers entering and exiting. If the user were to press the back button or close the application, then the app sends a “leave” command to the server and peers are updated in the same way.

Tapping a peer allows the user to send a “hello,” which simply causes a popup on the selected peer’s device or initiate the bandwidth estimation described in the next section.

Figure 5: The main first screen the user is presented with on running the P2P client app (left) and the session info screen that the user sees upon entering a session (right). The session screen is updated in real time as peers enter and leave.

Part 2: Peer Bandwidth Estimation

Once the P2P network backbone was up and running, the second part of the project was to implement an effective method of measuring the bandwidth between peers directly. There are many different methods for estimating bandwidth, but they generally fall into two categories: the Probe Gap Model (PGM) and the Probe Rate Model (PRM) [8]. The first type of estimation algorithm works by observing the change in the gap between a pair of packets that are sent from point A to B with a known initial gap. The increase in the gap between packet pairs can be used to
compute the available bandwidth. In the Spruce algorithm, a PGM method, the initial gaps are randomly distributed relative to the capacity \( C \) of the bottleneck on the target path. The available bandwidth can then be measured using the following equation [8]:

\[
A = C \times (1 - \frac{\Delta_{out} - \Delta_{in}}{\Delta_{in}})
\]

Where \( \Delta_{in} \) and \( \Delta_{out} \) are the input and output gaps of the packets respectively. This method has the drawback of requiring some knowledge of the capacity of the bottleneck in the connection, which we will address momentarily.

Croce, Mellia, and Leonardi tested the performance of several PGM and PRM algorithms and found that Spruce performed the best under the conditions of a large P2P network in which multiple peers were performing bandwidth estimations between one another simultaneously. Spruce calculated that the available bandwidth was approximately the actual bandwidth minus the overhead caused by all the peers using the tool [4]. This is what prompted me to settle on Spruce as the bandwidth estimation algorithm to include in the P2P network backbone.

In order to implement Spruce, I adapted the C code made available by Jacob Strauss online [10]. This code needed to be modified to pass information to the iOS app, including returning error messages, and to clean up memory and sockets in successful and unsuccessful cases so that it could be rerun over the course of the app's lifecycle without causing leaks.

Once it was in working order, the problem of knowing the capacity of the bottleneck ahead of time became an issue. Knowing this information is essential to the Spruce algorithm both for calculating the available bandwidth at its conclusion and for determining the distribution of initial gaps to send. The way I decided to approach this was by implementing a ping command that peers can send and respond to. This command (described in appendix A) simply tells a peer to expect several pings and to respond in kind so that the pinging peer can get a measure of the round trip time (RTT) of pings and use that to get a rough estimate of capacity. Most importantly, this process should take a minimal amount of time (~1 second) so that the bandwidth estimation can proceed. Because of the positive performance of Spruce in large-scale P2P network conditions, Spruce can be run on many peers simultaneously to parallelize the bandwidth estimation so that this calculation runs in time \( O(n) \) in parallel rather than \( O(n^2) \) were it forced to be serial.

Part 3: RESTful API for Bandwidth Reservation and for Quantum Simulations

The third and final portion of the project was to design a RESTful API for communicating to ISPs the topology of the P2P network which the TrackerServer application was careful to keep track of and to reserve bandwidth for the operation of an application that can be added on top of the backbone. The purpose of the P2P
network design described above is to make this final portion of the project possible. The coordinating tracker server was careful to maintain a representation of the networks that it was coordinating so that it could communicate these networks using this API.

The API is described and documented in appendix B. The RESTful nature of the API means that the client (our TrackerServer) communicates with the server (a network controller) by exchanging a series of resource representations and using the default HTTP verbs (GET, POST, PUT, DELETE). There are 4 different resource types in my proposed API. First is a resource that represents the network controller itself that can be requested by the tracker server to determine basic information about the availability of resources. Next is a Session resource that corresponds to the sessions of the tracker server. The tracker server can create sessions using an HTTP POST to /session with a payload as described in appendix B. The Tracker can also create Peer resources by sending an HTTP POST to /peer. The key is that the tracker creates these resources during the first phase of network creation (see Part 1 above). The use of this API by the tracker server eliminates the need for any peers to have any knowledge of the underlying network structure and resources and frees up developers to create apps as they see fit.

The onus is on the tracker server to determine, knowing the complete structure of the network, what resources need to be reserved. Then, after the tracker has updated the network controller of the network structure via our API, it must ask to create Route resources by sending HTTP POSTs to /route. For efficiency, route resources use the “hose” model. This means that bandwidth is reserved in the form of one peer that is guaranteed to have a certain bandwidth when sending to a list of other peers. Generally applications can be organized to follow this model. For example, in a video conferencing application, one peer may be sending streaming video to many other peers. In this case, the bandwidth only needs to be reserved in specific directions.

According to Erlebach and Rüegg [7], the reservation of bandwidth for routes described using the hose model is possible in polynomial time and can be done efficiently and optimally by the network controller. Their work was on VPN bandwidth reservation, but it suggests that the hose model can be efficient if used properly in our P2P case. For this reason the API described in appendix B requires reservations to be described using the hose model.

Why REST?

We chose to design the API using a RESTful design for several reasons. REST encourages a uniform interface between client and server (namely HTTP). In addition, REST encourages scalability, both by making it easy to add functionality after launch without compromising older clients and by making the addition of more and more resources trivial. To be fair, RESTful design takes a hit in efficiency because of the redundant transfer of data, but intelligently implementing and using filtering options in requests can mitigate this. Furthermore, REST has the ability to
easily add layers to the network, like proxies and other intermediate servers, without compromising the workings of the service.

**Other Working examples**

In designing this API, I looked to several working, large-scale RESTful systems as models. Primary among these were the Twitter and Flickr client APIs ([12], [13]), which share many properties with my presented design. In those and in my design, requests are sent to the server using simple HTTP verbs and arguments are added using CGI syntax or JSON payloads. Originally, responses were formatted as XML files, but later as JSON data. JSON is simpler and easier to parse, mainly because it uses only dictionaries and arrays with keys rather than any number of possible tags that XML uses. It also tends to be slightly less verbose. It may be beneficial to include optional arguments in requests to get resources as XML documents instead of JSON simply for the sake of client designer preference. Like the two released designs mentioned above, this system has the potential to be dealing with very large resource sets in any given request. For this reason it is important to include filtering requests and automatic cutoffs with links to further resources (though the option to get more at once should be included).

**OpenStack Quantum**

The Quantum API design by OpenStack was a particularly good design to look at for ideas on how to properly design this API because it was also designed to send requests for information on and set up of networking hardware (albeit virtual). Quantum includes JSON payloads with requests as well as responses and only uses HTTP arguments for filtering purposes. This is a very robust design because the nature of JSON allows easy expansion of the API by the addition of arguments and information without compromising the current working of the API. Another really useful design philosophy of Quantum is the ability to perform bulk creation requests. In a large scale P2P network, a client’s tracking server may wish to make setup requests to the ISP’s server all at once after the network has been established. In this scenario, it would be best for the ISP-side controller to support bulk requests using JSON arrays so that the client doesn’t have to send hundreds or even thousands of requests in rapid succession. It could instead send only one or as many as it sees fit. The usage of JSON in the request payload facilitates this ability as well because of the extensible nature of JSON arguments (using arrays) as opposed to arguments directly in the HTTP request.

Quantum, like Twitter and Flickr, implements the pagination of response data in the response payload. The dictionary that contains the list of response data objects also includes links to the next and previous pages. One interesting note is that Quantum includes the ability for clients to create their own extensions to the API in order to design systems that are specifically geared to their expected usage cases, which I propose taking advantage of for this project in the future.

Quantum uses standard HTTP error codes in order to convey errors in request transmission and processing. However, in our RESTful API design, I feel that
it is more beneficial to create a new format for errors directly in response JSON dictionaries. This allows the system to respond more specifically to errors and to send useful messages to the client for debugging and notifying users of failure.

Quantum Simulation

As a final touch, I propose the use of the Quantum API for the purposes of simulating the network resource side of this project. Since it is infeasible to actually implement and test the bandwidth reservation API described in appendix B without affecting the workings of Internet resources, it may be useful in the future to test the feasibility of the API design with Quantum. The Quantum API already has many of the required features available in its base APIv2.0. In appendix C, I briefly propose how this API might be used and extended for simulating a bandwidth reservation API.

Conclusions

Within this project, I designed and implemented a working, mobile P2P network with a Peer application and Tracker application. I proposed and implemented a method for determining the capacity of this network using the Spruce algorithm along with low-overhead pings. Then I designed a RESTful API for network bandwidth reservation so that the operation of large-scale P2P networks that transfer time- and bandwidth-sensitive information can be guaranteed. Since it is not possible to actually test such an API without access to ISP resources, as a final piece, I proposed an extension of OpenStack Quantum for the purposes of simulating the desired behavior.

References:
Links and Open Source Code
[12] Twitter REST API documentation: https://dev.twitter.com/docs/api