

Project Phase 2

Again in this phase, you are to work in groups of size at most 2. You do not have to stick with the partner, if any, from assignment 1. Please start very early since it is quite a bit more complex (and interesting) than assignment 1.

1. Project Overview

You are to implement a program based on a simplified version of the Gnutella Protocol Version 0.6, which we shall refer to as **Simpella (version 0.6)** from now on. The simpella protocol specification is in a later section. The Gnutella protocol version 0.6 is available at

<http://rfc-gnutella.sourceforge.net/index.html>

Simpella is a distributive search/file sharing protocol. Unlike Napster, where you search, download and share your files (mp3, mpeg, dvi, asf, ...) via one or several central servers, Simpella is entirely distributive in nature. There are a lot of potential killer-apps which could come out of this kind of peer-to-peer (P2P) computing, beside the fact that P2P networks are highly fault tolerant.

Each program implementing Simpella is called a *Simpella client*. Do not confuse this client with a typical TCP client or UDP client. What we mean by Simpella client is that this is a client to the simpella protocol, not a client to a central server somewhere.

Yet another terminology is *servent*. A servent is a program which could function as both a **SERVer** and a cliENT. **All Simpella clients are servents**. Basically, you act as a server to people who want to download files or request information from you, and as a client to download files and request information from people. In this context, the words "Simpella client", "servent", "host", "peer", "node" have the same meaning. They refer to a participant in the Simpella network.

Simpella clients establish TCP connections with each other (not a complete graph) to form the Simpella network, exchange information about their shared files, and eventually establish a connection to a specific Simpella client to download a file.

Your task is to write a command line version of a Simpella servent using Java. No other programming language is allowed. The specific commands' definitions and functionalities are described later.

2. The Simpella Protocol Overview

2.1 Connecting to the network

To connect to the Simpella network, a servent only needs to know one (or more) [IP, port] of another servent on the network. The new servent establishes a TCP connection to the existing servent(s). The new servent then sends a special announcement to the servents that it is just connected to. The announcement gets flooded through the whole network. The flooding process will be described later. Other servents who get the announcement reply with information about themselves (IP, port, their files' information, etc.) along the reverse routes of the paths that the announcements came from.

2.2 Searching for shared files

Searching for files works in a similar fashion as join announcing. Search requests get flooded over the network, and each servent getting the request searches its shared directory. If some matches are found

at a server, the server replies with a *result set* which contains information about the matched files (type, size, exact name, etc.). For example, if the search query is "psy gangnam ", then all files whose names contain either "psy" or "gangnam" have their entries in the result set. The result set also contains the IP of the server having the file and the TCP port to which one can establish a connection to, in order to download the file.

2.3 Downloading

After searching and getting matched results, if a server wants to download a particular file as instructed by the user, then it establishes a TCP connection to a specific port of the server(s) who has the file. Then, a subset of the HTTP protocol is implemented to download the file. In essence, each server is also a little web server with limited functionalities.

2.4 Traffic Monitoring

Since all search queries go through everyone, every server knows what people are looking for in the network. Each server provides the user with a monitor which tells the users information like what files people are searching for, how many bytes are shared, how many users are there in the network, etc. The monitoring is done in real time.

3. The Simpella Protocol Specification v0.6

3.1 Connecting to the network

Each server handles at most 3 incoming connections and 3 outgoing connections. In total, a server has up to 6 concurrent connections at the same time. Again, "incoming" means that someone connects to you, and "outgoing" means that you initiated the connection. Other than that, all connections are equivalent.

To connect to another server given its IP and port number, a server tries to create a TCP connection. The other server accepts the connection and they start to handshake.

The connection initiator sends a string (case-sensitive)

```
SIMPELLA CONNECT/0.6\r\n
```

Here, 'r' is the carriage return character (ASCII code 13), and 'n' is the line feed character (ASCII code 10).

The server accepting this connection sends back

```
SIMPELLA/0.6 200 <string>\r\n
```

if it wants to accept this connection. The **status code** 200 is meant to indicate that the server accepts the connection. The <string> that comes after should be "OK", but it could be a welcome message. The initiator prints out the welcome message, and also confirms the connection with

```
SIMPELLA/0.6 200 <string>\r\n
```

The other end prints out the <string> too. This is like a "thank you for accepting me" message.

If the receiving server does not want to accept the connection, it replies with

```
SIMPELLA/0.6 503 <error-message>\r\n
```

indicating the error condition or the reason for not accepting this connection. For example

```
SIMPELLA/0.6 503 Maximum number of connections reached. Sorry!\r\n
```

For this version of our protocol, the only reason that a servent does not want to accept a new connection is that it has got 3 incoming connections already. The connection initiator prints out the error message it received so that the user knows what's going on.

Note that this handshaking protocol applies each time a new connection is to be established. This is not limited to just the case when a servent tries to get on to the network for the first time. After an initial connection has been established, a new servent will get to know more identities of other servents via various ways to be described later.

3.2 Message formats

After connecting to the network, the new servent and other servents exchange messages. Each message has a 23 byte header followed by the payload of arbitrary length. **All fields are in big-endian format.** Also, all IP addresses are in IPv4 format (32-bit integer).

Precisely, each message is of the following format:

Bytes	Field Name	Description	
0-15	Message ID	A Unique Message ID. This should be a GUID (globally unique ID). Servents SHOULD store all 1's (0xff) in byte 8 of the GUID. (Bytes are numbered 0-15, inclusively.) This serves to tag the GUID as being from a modern servent. Servents SHOULD initially store all 0's in byte 15 of the GUID. This is reserved for future use. The other bytes SHOULD have highly random values.	Header
16	Message Type	One of four possible types: 0x00 = PING 0x01 = PONG 0x80 = QUERY 0x81 = QUERY HIT	
17	TTL	Message's Time To Live, which is an unsigned integer, set initially by the original sender and decreased by 1 at each servent that receives it. A message must not be forwarded if TTL is 0. Originally, the TTL should be set to 7.	
18	Hops	Number of hops the message has passed through. Initially 0. Increased by 1 at each receiver. At all time, TTL+Hops should be the original TTL.	
19-22	Payload length (PL)	The length of the message's payload immediately following this header. The next message header is located exactly this number of bytes from the end of this header i.e. there are no gaps or pad bytes in the Simpella data stream. Messages SHOULD NOT be larger than 4 kB. In other words, you only need a buffer of 4096 bytes for each payload.	

23- (22+PL)	Payload		Payload
----------------	---------	--	---------

Abuse of the TTL field in broadcasted messages (Query) will lead to an unnecessary amount of network traffic and poor network performance. Therefore, servents **SHOULD** carefully check the TTL fields of received query messages and lower them as necessary. Assuming the servent's maximum admissible Query message life is 7 hops, then if $TTL + Hops > 7$, TTL **SHOULD** be decreased so that $TTL + Hops = 7$. Broadcasted messages with very high TTL values (>15) **SHOULD** be dropped.

Each servent is responsible for generating the message ID so that it remains unique during its life time.

3.3 PING messages (0x00)

Right after connected to the network, or after instructed to by the user, a servent sends a PING message to all its neighbors, that is, all servents connected to it, whether or not the connections are incoming or outgoing. Again, the terms incoming and outgoing are only meaningful technically (who initiated the connection). Other than that, all connections are full-duplex and treated the same way.

A PING message does not have any payload. Thus, $PL=0$ for a PING message. Upon receiving a PING message, a servent check to see if it has seen this PING before. If not, the servent forwards it to all neighbors except the neighbor where the PING came from. This is done by maintaining a routing table of up to 160 entries of the most recently received PING messages. Each entry in the table also indicates where (which neighbor) the PING came from. The particular data structure used to maintain this routing table is irrelevant in this discussion. You can use a linked list (then searching is pretty slow, but space efficient) or an array of size 160 (then searching is relatively fast, but taking up space).

3.4 PONG messages (0x01)

A PONG is a response to a PING, and has the same message ID as the PING it responds to. A PONG message is sent back to the neighbor where the PING came from. Upon receiving a PONG, a servent looks up its routing table and forwards the PONG back to the neighbor who sent the corresponding PING. This way, PONG messages propagate back to the original servent who sent out the original PING. Obviously, all PONG messages are not forwarded anymore by the original PING sender. PONG's payload is 14 bytes long and has the following format:

Bytes	Field Name	Description
0-1	Port	The port and IP address where this servent can accept incoming connections. These are connections for the Simpella network, not the file downloading connections.
2-5	IP address	
6-9	Number of Files Shared	The number of files that the servent with the given IP address and port is sharing on the network.
10-13	Number of Kilobytes Shared	The number of kilobytes of data that the servent with the given IP address and port is sharing on the network.

The original PING sender collects all these information and store them somewhere for displaying and for later usage. After getting PONGs back from a new PING, the saved data has got to be refreshed. Some servents might be down, data shared are different from the last time we got the information.

3.5 QUERY messages (0x80)

A QUERY's payload has the following format:

Bytes	Field Name	Description
0-1	Minimum speed	The minimum speed (in kb/second) of servents that should respond to this message. A servent receiving a Query message with a Minimum Speed field of n kb/s SHOULD only respond with a Query Hit if it is able to communicate at a speed $\geq$ n kb/s. In fact, the semantics of these 16 bits in Gnutella 0.6 are fairly complex. As far as Simpella is concerned, we just set it to 0.
2-	Search string	This is a null-terminated string containing search text as typed in by the user.

Since Query messages are broadcast to many nodes, the total size of the message SHOULD not be larger than 256 bytes. Servents should drop Query messages larger than 256 bytes, and MUST drop Query messages with payload larger than 4 kB.

QUERYs are forwarded in a similar fashion as PING messages. The only difference is that when a servent gets a new QUERY, it searches its shared files to see if there is any file whose name contains one of the given words. If there is, then the servent replies back (to where the QUERY comes from) with a HIT message. It is entirely up to you if you want to maintain a separate routing table for QUERYs.

3.6 QUERY HIT messages (0x81)

A QUERY HIT is a response to a QUERY if one or more file matches were found. Similar to the PONG case, a QUERY HIT has the same ID as the corresponding QUERY and gets forwarded back to the querier in the same fashion.

Query messages with TTL=1, hops=0 and Search Criteria=" " (four spaces) are used to index all files a host is sharing. Servents SHOULD reply to such queries with all its shared files. Multiple Query Hit messages SHOULD be used if sharing many files. The TTL SHOULD be set to at least the hops value of the corresponding query plus 2, to allow the Query Hit to take a longer route back, if necessary. The TTL value MUST be at least the hops value of the corresponding query, and the initial hops value of the Query Hit message MUST (as usual) be set to 0.

QUERY HIT's payload has the following format:

Bytes	Field Name	Description		
0	Number of hits	The number of matched files		
1-2	Port	The port and IP address where this server can accept incoming connections for file downloading. This port is different than the port used for Simpella network connections. File downloading is to be done based on the HTTP protocol as described later.		
3-6	IP address			
7-10	Speed	The speed, in Kbps of the responding host. For now, let's fix this field at 10Mbps = 10,000 Kbps. Note that this is a 32-bit integer to be stored in big-endian format.		
11-	Result set	A set of responses to the QUERY. This set contains "number of hits" elements contiguously, each with the following structure:		
		Bytes	Field Name	Description
		0-3	File Index	A number, assigned by the responding host, which is used to uniquely identify the file matching the corresponding query.
		4-7	File Size	The size (in bytes) of the file whose index is File Index.
		8-...	File Name	A null-terminated name of the file
Last 16	Server ID	A 16-byte string uniquely identifying the responding server on the network. This SHOULD be constant for all Query Hit messages emitted by a server and is typically some function of the server's network address. The server Identifier is mainly used for routing the PUSH message. Simpella does not implement the PUSH messages. However, to be interoperable with Gnutella we shall conform to the protocol.		

3.7 File transfer

After receiving a QUERY HIT, a server may elect to initiate the direct download of one of the files in the result set. Files are downloaded out-of-network, i.e. a direct connection between the source and target server is established in order to perform the data transfer. File data is never transfer over Simpella network.

File downloading is recommended to be done via HTTP 1.1. However, HTTP 1.0 can be used instead. For our purposes, the following super-small subset of HTTP1.1 is sufficient.

The server initiating the download sends a request string of the following form to the target server:

The server initiating the download sends a request string on the following form to the target server:

```
GET /get/<File Index>/<File Name> HTTP/1.1\r\n
User-Agent: Simpella\r\n
Host: 123.123.123.123:6346\r\n
Connection: Keep-Alive\r\n
Range: bytes=0-\r\n
\r\n
```

where <File Index> and <File Name> are one of the File Index/File Name pairs from a QueryHit message's Result Set. The Host header is required by HTTP 1.1 and specifies what address you have connected to. It is usually not used by the receiving servent, but its presence is required by the protocol. For example, if the Result Set from a QueryHit message contained the entry

File Index: 2468
File Size: 3456789
File Name: Foobar.mp3

then a download request for the file described by this entry would be initiated as follows:

```
GET /get/2468/Foobar.mp3 HTTP/1.1\r\n
User-Agent: Simpella\r\n
Host: 123.123.123.123:6346\r\n
Connection: Keep-Alive\r\n
Range: bytes=0-\r\n
\r\n
```

The servent receiving this download request responds using a HTTP1.1 compliant header such as

```
HTTP/1.1 200 OK\r\n
Server: Simpella0.6\r\n
Content-type: application/binary\r\n
Content-length: 3456789\r\n
\r\n
```

The data file then follows and should be read up to and including the number of bytes specified in the content-length provided in the server's HTTP response.

If the file is not found, then the reply should be of the form

```
HTTP/1.1 503 File not found.\r\n
\r\n
```

The requesting servent should detect this error code and print out the error message. For the purpose of this project, "file not found" is the only reason for rejecting a download request.

Note: in HTTP headers the numbers are represented as a decimal string. For Simpella, you can ignore all requests whose byte range are not "0-", i.e. we do not support partial file downloads. Lastly, you can close the connection once all bytes for the requested file has been sent. Simpella does not support persistent connection either.

You have several choices to implement file downloading: let a single process handle everything (Simpella connections, downloading, ...) using non-blocking I/O, spawning a new thread to handle a file downloading request, etc. For this version of Simpella, the number of simultaneous downloads could be arbitrary to simplify implementation. In reality, a busy servent should respond with a code 503 header to deny a request.

4. The Program Command Line Interface

4.1 Invocation

Your program named **simpella**, say, starts with 2 command line parameters

```
java simpella <port1> <port2>
```

If no parameter was given, <port1> is defaulted to be 6346 and <port2> 5635. <port1> is the port where other servers can establish connection to you to connect to the Simpella network. <port2> is the port for file downloading using HTTP.

Sample output

```
smathew2@pollux (~/Simpella) % java simpella 6346 6745
Local IP: 192.168.0.16
Simpella Net Port: 6346
Downloading Port: 6745
simpella version 0.6 (c) 2002-2003 XYZ
```

```
simpella> bye
no such command: bye
simpella> quit
smathew2@pollux (~/Simpella) %
```

4.2 The commands

Note: brackets are used to indicate alternatives. For example, [abc] means either a or b or c.

4.2.1 info

info [cdhnqs] - Display list of current connections. The letters are:

- **c** - Simpella network connections
- **d** - file transfer in progress (downloads only)
- **h** - number of hosts, number of files they are sharing, and total size of those shared files
- **n** - Simpella statistics: packets received and sent, number of unique packet IDs in memory (routing tables), total Simpella bytes received and sent so far.
- **q** - queries received and replies sent
- **s** - number and total size of shared files on this host

Sample outputs: (Packs: x/y means there have been x packets sent and y packets received on this connection. Bytes: x/y is similar).

```
simpella> info h
HOST STATS:
-----
Hosts: 6046  Files: 1.937M  Size: 13.378T
simpella> info c
CONNECTION STATS:
-----
1) 65.83.181.222:6346      Packs: 571:735      Bytes: 22.66k:32.37k
2) 64.2.56.31:6346       Packs: 305:1871     Bytes: 16.30k:70.78k
3) 65.32.58.209:6346     Packs: 0:0          Bytes: 0:0
```

```

simpella> info n
NET STATS:
-----
    Msg Received: 7.60k  Msg Sent: 3.01k
    Unique GUIDs in memory: 5136
    Bytes Rcvd: 321.52k  Bytes Sent: 200.47k
simpella> info d
DOWNLOAD STATS:
-----
    1)24.26.127.119:6346          4.2%    159.74k/3.825M
    Name: Paul Simon - The Boxer.mp3
simpella> info q
QUERY STATS:
-----
    Queries: 6606          Responses Sent: 1437
simpella> info s
SHARE STATS:
-----
    Num Shared: 2164  Size Shared: 39.666M

```

4.2.2 share and scan

share [dir | -i]- specify the directory all whose files are shared. **To simplify matters, you can assume that only one directory is shared, when share is invoked the second time, the new directory overrides the old one.** If the directory name does not begin with a "/", then it's a relative directory, otherwise it is an absolute path name. If "-i" is specified, prints the current shared directory.

scan - scan the shared directory for files' information

sample outputs:

```

simpella> share this
simpella> share -i
    sharing /home/smathew2/Simpella/this
simpella> scan
    scanning /home/smathew2/Simpella/this for files ...
    Scanned 2164 files and 3.96663e+07 bytes.
simpella>

```

4.2.3 open and update

open <host:port> - open a connection to host at port. The field <host> could either be a host name or a host IP.

sample outputs:

```
open hadar.cse.buffalo.edu:6346
```

update - send PINGs to all neighbors

4.2.4 find, list and clear

find <string> - start looking for files containing words in the string. All matchings are case insensitive.

```

simpella> find paul simon
    searching Simpella network for `paul simon'
    press enter to continue
    20 responses received

```

(the 20 keeps changing, then <enter> pressed)

```

-----
the query was 'paul simon'
1) 192.168.0.1:6346      Size: 5.3M
   Name: Simon and Gafunkel - The Boxer.mp3
2) 192.168.0.2:6346      Size: 6.4M
   Name: Simon and Gafunkel - Scaborough Fair.mp3
simpella>

```

list - list files returned by find, notice that there could be several *finds* done. You just append them on the list. You can impose a maximum number of entries on the list and overwrite old ones if there are too many *finds*.

clear [file-no] - clear file whose number is file-no from the list. If no argument was given, clear all files.

4.2.5 download

download <file-num> - start downloading the file specified. **Note that the download is done in the background. User inputs should not be blocked on a download command.**

```

simpella> download 2
   downloading 'Simon and Garfunkel - Scaborough Fair.mp3'
simpella>

```

4.2.6 monitor

monitor - display the queries people are searching for

```

simpella> monitor
MONITORING SIMPELLA NETWORK:
Press enter to continue
-----
Search: 'Elvis Presley'
Search: 'Chopin'
Search: 'Bread If'
Search: 'Moonlight sonata'
Search: 'the boxer'
   (enter pressed)
simpella>

```

4.3 Some other requirements, suggestions:

1. **Each servent has to try its best to maintain at least 2 connections at all time (incoming or outgoing).** After the first open, it's got only one connections. Based on the PONGs received, it gets more information and automatically try to connect to other hosts. So, if one of the neighbors quits, the servent searches in its recent PONG database and try to find another person to connect to.
2. Your program should handle invalid input graciously, reports "unknown command" for totally off inputs, reports the command's usage if the input parameters were not expected.
3. For file downloading: the sender should try to write blocks of data of size 1K each. The reader use 'read' directly to keep reading from the socket descriptor until '0' is returned. In that case, it means that the sender has finished writing and the downloading is done. **You do not want to allocate a buffer of 10M for MP3 files.** While reading, you write whatever you've read to a

temporary file, change the name to the right name after downloading is done.

4. You should also implement the commands "quit", "bye" and other features as you think appropriate
5. Your program should respond in a nice way (tell user what's happened) when the total number of outgoing connections or incoming connections have reached its limit.
6. Download file to the shared directory too. Hence, after downloading you could do a 'scan' to share more files.

4.4 Minor points:

Officially, the Gnutella protocol is almost identical to the one we describe here, except that it can handle hosts behind firewall by another message type called PUSH. Also, officially Gnutella requires all connections to be TCP. There are good things and bad things about establishing too many TCP connections.

There are many other issues if you were to make a real-world working servent. We have simplified the commands' options quite a bit. However, the essential ideas are there.

Lastly, this is a really really good networking exercise for you . It illustrates some of the most important networking concepts: routing tables, table lookup, flooding, protocol, packet formatting, distributive collaboration.