UMB Networks Project CS425

Y5488 Utkarsh Upadhyay.
Y5458 Subhonmesh Bose.
Y5264 Makarand Mijar

Problem Statement

The problem is to design a peer-to-peer file transfer protocol where a user can simultaneously download parts of a file from different sources and assemble it. We can assume that each host has a complete list of the chunk information of different files on each machine on this network. The transfer must use UDP and the program thus must tackle packet reliability on its own.

Our assumptions

1. We have bypassed the fact that every machine has a static chunk information at other hosts. We rather use a Multicast group where each user joins in and telecasts its information regarding its chunks to other hosts in this group. This information is sent periodically and thus takes care of the fact whether a host has gone down or has added/removed any shared file or its chunks.
2. We assume that the peers would be on the same LAN as IITK routers do seem to have a problem routing multicast packets. Thus we require the hosts to be able to join a multicast group that restricts us in IITK to use the same LAN.
3. We assume the availability of some ports on the machine. The ports we require are from 5000 to 5010, out of which 5000 and 5001 are essential ones. If the program cannot bind to a socket enable to communicate over these two ports, it automatically shuts down giving an error message. If even one among the rest is opened, the program starts running.
4. The user can only ask for a complete file and cannot ask to get just a chunk. It is the duty of the program to find the chunks distributed across the network and report to the user if it could fetch a complete file or not.
5. However, for requesting a file to another host, the program internally decides to ask for chunks rather than a complete file from other hosts in the network.

Our approach

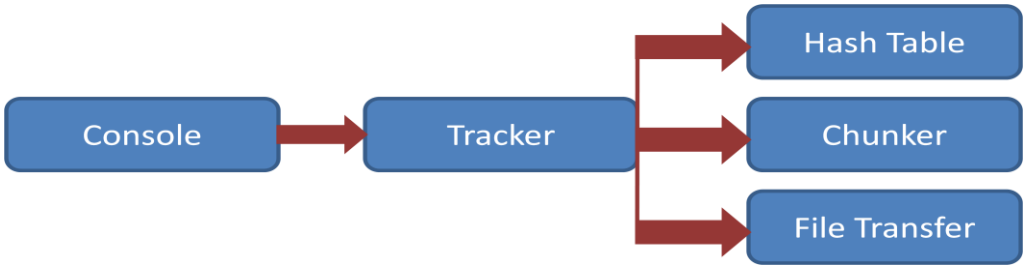
We used a highly modular approach to the problem. We figured out that there are basically 5 parts to the program:

1. **Database Manager:** This takes care of the list of chunks of different files available on the network.

2. **Chunk Maker/Assembler:** This creates chunks of a file and maintains a mechanism for testing the integrity of each chunk. It also assembles the chunks into a complete file when all the chunks of a file have been downloaded.
3. **Chunk Sender/Receiver:** This communicates on a single port with another host on a defined port and transfers file reliably. This throws back problems if encountered in the process or flags a success message if it is successful.
4. **User Interface:** This is where the user interacts with the program. We have 2 such interfaces, one is a GUI and another is a console one. Here the user can ask to share a file on the network and fetch a file from the network.
5. **The Head:** This interacts with every other part and decides what to do when. It basically deploys the work to other modules and also performs a 3-way handshake before a communication begins on a defined port using the Chunk Sender/Receiver.

Now we describe each module in detail in the next section.

The Design



ManagedHashedTable: The Database Manager

It is initialized by THE HEAD with a socket where it can communicate with other ManagedHashTables to keep knowledge of what chunks are available on the network and exactly which are the sources that has these chunks. Thus this part interacts both with THE HEAD and other ManagedHashTables on other hosts on the network. Its function can be categorized as:

a. Given a port address by THE HEAD, it quietly adds itself to a Multicast group and keeps listening for incoming messages multicast from similar ManagedHashTables present on the same network. It keeps track of chunks, their possible sources and their hashes, as well as how many chunks is this chunk a part of. It manages synchronized hash tables to keep renewing this information on a regular basis as Timeout messages are received at regular intervals from all other hash tables joined in the multicast group. It also generates time out messages for those chunks for which it is a source.
b. Also, it has an API open to the tracker program, using which files can be added to this hash table. Once added to this table, the message will be multicast to all the hash tables online at that moment and this information will be renewed at regular timeouts. Also, for files it is the source of, it keeps some information, like the Time Out interval and the

true full path on this computer which is private to only this ManagedHashTable, since it will never be needed by any remote machine.

All peers have information about each chunk on its own machine and on the network. So the overhead of one RTT for asking information about a file is removed. This information is conveyed via a multicast and is refreshed periodically thus making the design scalable. Any request from the client side is avoided because that would lead to unnecessary congestion over the network. This has a trade-off. Any new host that comes up has to wait one *timeout* period before it has all the information regarding the files shared on the network.

Chunker: The Chunk Maker/Assembler

This module mainly performs the following tasks:

a. Given a file, it makes chunks for the file and its SHA hashes and stores them at the same place as the file. Thus we physically maintain different files for chunks.
b. Given a target file, SHA hashes for chunks and the path for the chunks (which it obtains from the fullpath of the target file), it performs the reassembly and clean up.

FileTransfer: The Chunk Sender/Receiver

This is the part that manages the transfer of ONE chunk from a HOST:PORT to another HOST:PORT. There are essentially 2 parts to this, the Downloader and the Uploader. The communication of a chunk happens in packets which can be of maximum 32KB in size. It implements Sliding Window Protocol with fixed window size to download a chunk. Here a congestion control with negotiable window size would not be of much use as the OS usually provides us with 1MB of memory size anyways. If we are communicating with at max 10 users (those are the number of ports we are opening for file transfer and each chunk transfer occupies a single port till it is done with it.) Modern computers easily provide us with that much memory space.

Sender: Requires the following:

1. Own port to use.
2. Peer host's IP address and its port to use.
3. Filename corresponding to a chunk to send.

Downloader: Requires the following:

1. Own port to use.
2. Peer host's IP address and its port to use
3. Filename with complete path where to write a chunk.
4. Number of packets to be received.

We use cumulative acking system with proper timeouts. If the Downloader or Uploader fails to transfer a chunk fully, it reports back to THE HEAD with its failure.

The User Interface

This is where the user interacts with the program. We have provided two interfaces. The classes for these are:

- edu.networksproject.GUITracker
- edu.networksproject.ConsoleTracker

The GUI requires javax.swing libraries to work. Here we basically lets the user to do the following:

1. Ask to share a file on the network.
2. Display the available files on the network
3. Fetch a file from the network.

Tracker: The Head

This is the controller and manager of each of the modules. Upon the creation of a new objet of this type, it captures a port meant only for Tracker communication. Then it also has an active and a passive part. On the listener, it listens for communication from other trackers which are online on the network. All the interactions are independent of any other interaction that happens as a part of the ManagedHashTable or FileTransfer. Then it opens for the user an API with the following functions:

a. ***To upload the file whose full path is give:*** void upload(String fFullName)
b. ***To get a list of files currently known to the ManagedHashTable:*** String[] getFileList()
c. ***To request an exit while shutting down the program:*** void requestExit()
d. ***To fetch a file from the network:*** boolean getFile(String fName)

The 3-way handshake

The tracker when requested to fetch a file from the network, it does the following:

a. Asks the ManagedHashTable for the information of the locations of the chunks.
b. Now for each chunk, it contacts the Tracker of another host sending a Type1 packet requesting a chunk.
c. The peer host's tracker sends back a packet which can be:
    - Type2 packet: This says that the peer host has accepted the request and it is designating a port for sending the chunk.
    - Type3a packet: This says that the peer host does not have the chunk requested and thus is negating the connection.
    - Type3b packet: This says that the peer host has the chunks but currently does not have any free ports to take the request.
d. If Type3a is received then the tracker tries to request the file from another source, if available.
e. If Type3b is received then the tracker would look for other sources and if it runs out of other sources it ask the same host after some time.
f. If Type2 packet is received, the Tracker sends a Type4 packet that carries the information about which port of this user would be listening for the packets and starts the Downloader.
g. On reception of Type4, the Uploader is called.
h. If on any of these communications, a timeout is faced, it is gracefully handled.

Thus we achieve a 3-way handshake similar to TCP for starting up the chunk transfer.

This kind of a handshake happens for every chunk transfer. We then ask the Downloader/Uploader to simply transfer the file over a designated port. For every get file request, the Tracker starts transferring chunks in a batch of 5. This ensures that no get file request hogs all the ports available for communication.

Conventions Used

All packets transferred through our program start off with the byte encoding of "UMB". The control packets sent to communicate between trackers on different hosts are suitably designed with byte encoding. The details are available in the file edu.networksproject.TrackerThread.java. We maintain that if the packets received do not follow the conventions, we drop the packet.

Implementation

All the four logical modules explained above were actually different projects in NetBeansIDE6.5. The implementation for most parts was done by different people and each project was accompanied by a few Test programs implemented as JUnit tests. The use of Junit 4.5 caused a little consternation on Windows on which a similar library was not immediately found. Hence, most functions had a sanity check inbuilt in them. Also, since we adopted a bottom-up approach to designing the Project, we initially had interfaces for three of the four modules, i.e. for all except Tracker, on which work began relatively late. Each module was developed to meet the specifications of the corresponding interface. These can be found at:

1. edu.networksproject.ManagedHashTable (The Download Manager implemented by edu.networksproject.SimpleHashTableManager)
2. edu.networksproject.filetransfer.<Uploader/Downloader>(Chunk Sender/Receiver defined by the interface in edu.netwoksproject.filetransfer.TransferListner
3. edu.networksproject.chunker.Chunker (implemented by edu.networksproject.chunker.ChunkSHA)

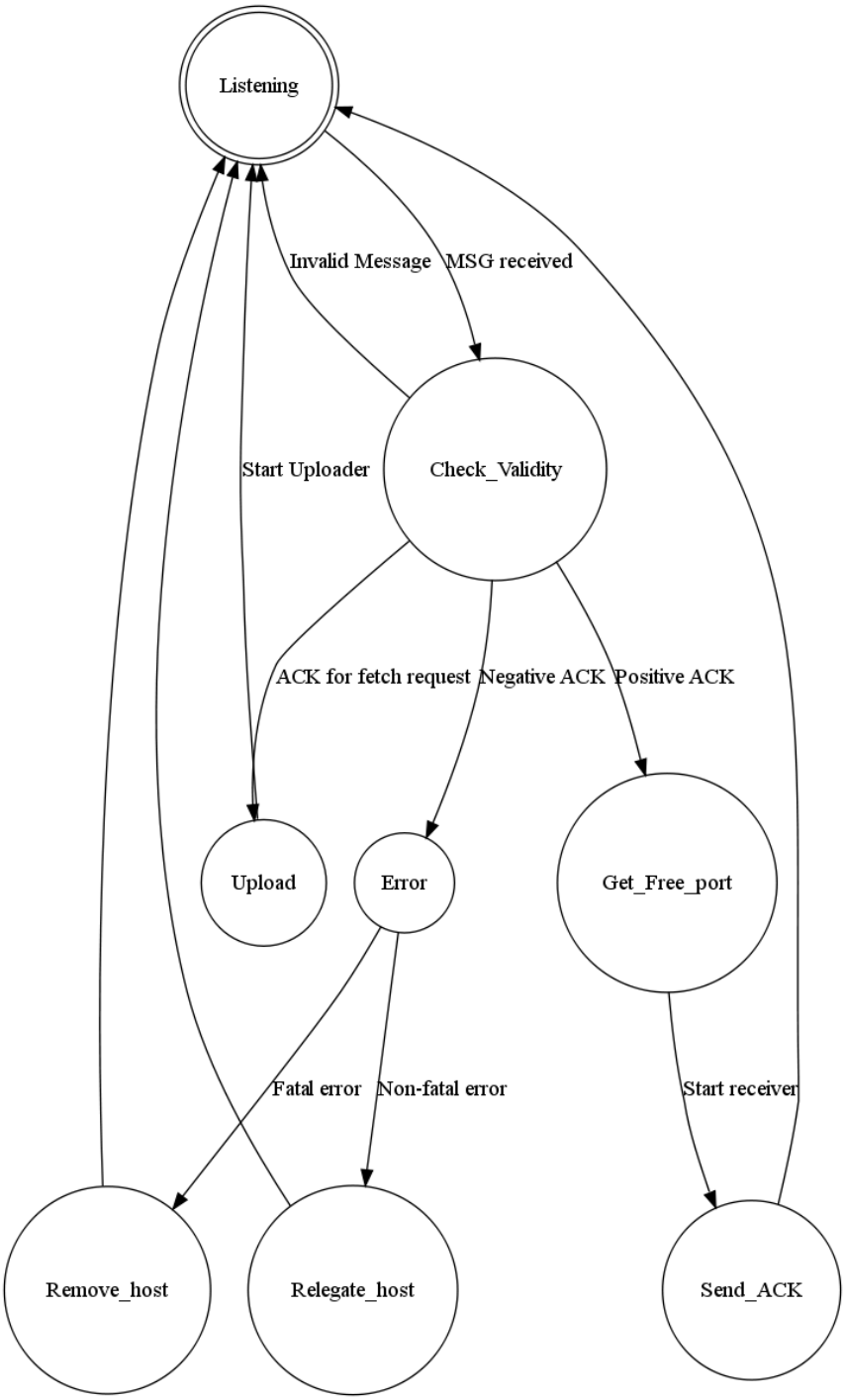
Testing units

We used standard JUnit tests which require JUnit 4.1 or higher. Apart from these, we had dedicated classes for the same. Some of them are:

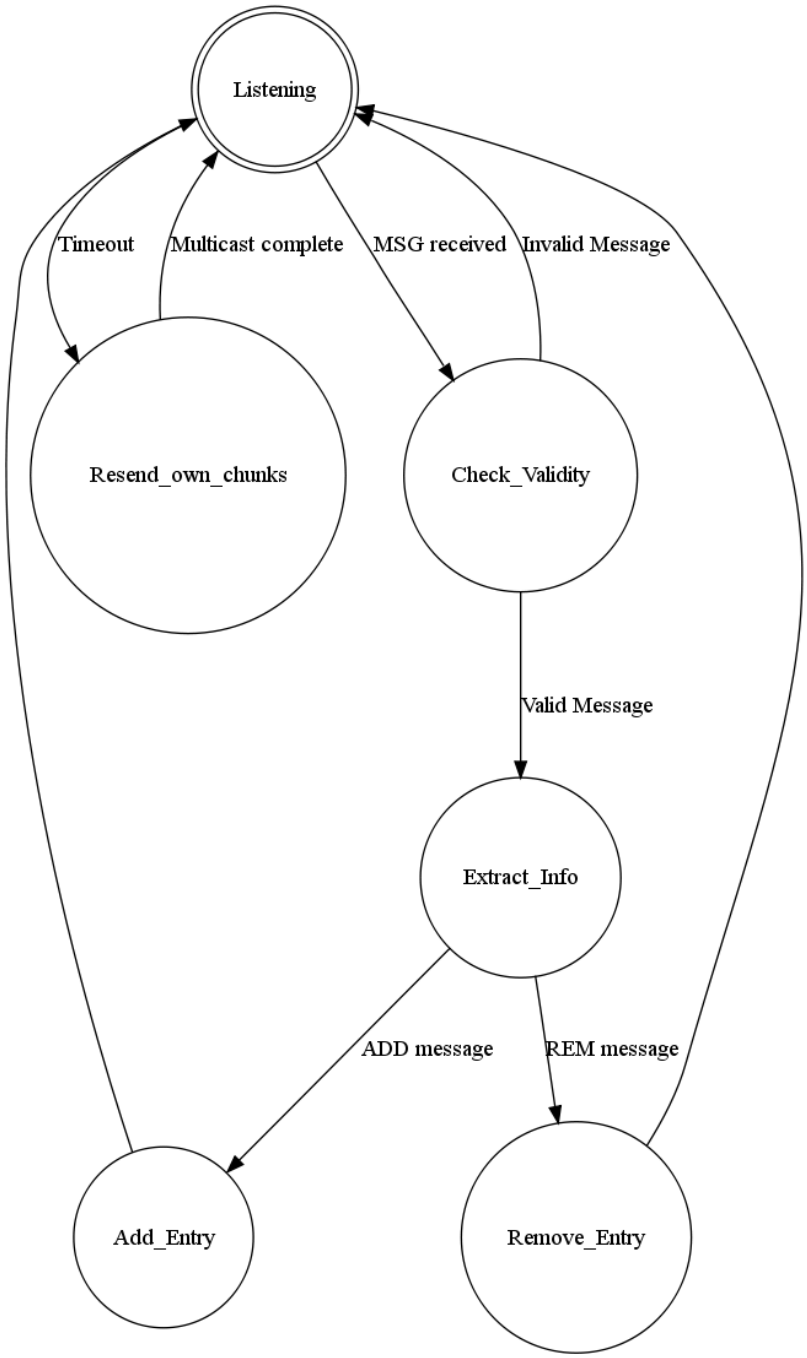
- edu.networksproject.MultiCastTester package
- edu.networksproject.ReceivingMachine
- edu.networksproject.SendingMachine

These files were run from the console and helped in diagnosing problems with specific parts of the project. These tests were run from peer-to-peer (FileTransfer tests) or as Multicasters (ManagedHashTable).

Finally while putting all the pieces together, we assembled them into the Tracker, which apart from having a few its small share of JUnit tests, was put through heavy testing using the edu.networksproject.ConsoleTracker across two machines in adjacent rooms. This testing phase continued for a fairly long time.



Tracker Thread State Diagram



Managed Hash Table State Diagram

Challenges Faced

a. Trying to work across hostels, we understood that the routers in IITK do not forward Multicast packets. Thus the Database Managers (MANAGEDHASHTABLEs) would not be able to communicate across hostels.

b. The String encoding of different systems does vary widely and thus we needed a Byte Stream reader/writer for successful working of the file.

Comments and Remarks

SHA1 hashes were used for verifying each chunk transferred. Details of this hashing scheme can be found at http://java.sun.com/j2se/1.4.2/docs/api/java/security/MessageDigest.html

1. ***GUI*** as well as a ***Console*** based front-end was made.
2. The program handles orphan threads and errors gracefully.
3. The various Timeout constants were fine-tuned while testing the program over the LAN. These are defined in a file edu.networks.project.fileTransfer.Constants. The timeouts for ManagedHashTable are put separately. These can be readily changed and can also be made dynamic for a larger system.
4. The design of the project was such so that it would be easy to divide work and work independently. We used the ***Bazaar Version Control System*** and the ***NetBeans IDE6.5*** for programming. The code base is deployed even now at:
   sftp://networksproject@172.24.0.237/~/networksproject/{MakeChunk,FileTransfer,ManagedHashTable,Tracker}
   It is password protected and is accessible using bzr at most hours while LAN is working.
5. The project is ***extension-friendly*** and owing to its modular nature is extremely maintainable and deployable.