NetworksP2P

A peer-to-peer file-sharing service made for our Computer Networking final project.

Compiles with included Makefile - just use `make`. Creates an executable named `p2p` that takes no command line arguments that listens on port 8080.

Peer-to-peer structure

This peer-to-peer file-sharing service is designed for use amongst a relatively small set of distinct hosts that are all known to each other. Hypothetically, the number of hosts in participating in the service is limited only by the number of sockets allowed to a process, since each client is connected to every other server running the program. A client can only request files from a set of hosts whose IP addresses are known before starting the program. Since the service does not restrict which files a client can request, this is a necessary security feature to prevent a malicious client from taking sensitive information from arbitrary servers.

Concurrency

Each instance of our peer-to-peer file-sharing service running on a single host has five threads: two client threads, two server threads, and a main thread that spawns the other four threads and cleans up when they exit. The division of these threads allow hosts to send and receive files concurrently, as well as to connect to newly-active hosts while performing other tasks.

Client threads

There are two client threads: a connector thread and a requester thread. The division of the client into a connector thread and a requester thread allows the client to connect to all available servers and simultaneously make requests to all of them.

The connector thread repeated loops over a list of known hosts that may be running the program, attempting to establish connections with their servers. When a connection is made, the connector thread saves the socket file descriptor associated with the connection in `client_fd_list`, a shared vector with the requester thread. The connector thread continues to loop even after all active hosts have established connections because 1) more hosts may join the network at any time and 2) a host may drop out of the network and come back at any time, and we want the connector thread to be able to re-establish a connection.

The requester thread waits for the human user to input a file to request from the other servers. It then determines which of the other servers have the file and sends requests for chunks of the file to the other servers until it has received the entire file. The user can then request more files.

The separation of these threads allows for a client to connect to servers concurrently with waiting for user input or requesting files from other servers. Both threads use `select()` to add additional concurrency. The connector thread uses `select()` to establish a timeout period for its `connect()` calls, which ensures that the thread can attempt to connect to other hosts if it encounters one that is unresponsive. The requester thread uses `select()` so that it can wait for input from multiple servers concurrently.

Server threads

There are two server threads: a listener thread and a reader thread. The division of the server into a listener and a reader thread allows the server to service requests from multiple different clients.

The listener thread sets up a listening port (port 8080) and just waits for clients to make connections. When a new connection comes in, the listener thread accepts it and puts the socket file descriptor for it in an `fd_set` `server_read_fds` and a vector `server_fd_list`. It then returns to listening for a new connection. This thread uses `select()` to establish a timeout period for its `accept()` calls. This is necessary to ensure that the listener thread is responsive to messages from the other threads indicating that the program has quit and the thread should end.

The reader thread uses the `server_read_fds` `fd_set` to repeatedly call `select()` on all of the server's open connections. When it receives a request from a client, it determines what part of what file is being requested and sends it back to the requesting client. This use of `select()` allows the server to fulfill the requests of multiple different clients concurrently.

Protocol and Messages

Our file-sharing service uses two main message structures and, within each, several types of messages. Clients send messages with the `clientMessage` structure and servers send messages with the `serverMessage` structure. Since clients never communicate with other clients and servers never communicate with other servers, clients can always expect to receive information in a `serverMessage` format and servers can always expect to receive information in a `clientMessage` format. We describe the structure of the client and server messages, as well as the types of information they each carry, below.

Both `clientMessage` and `serverMessage` have fixed sizes. `clientMessage` has a size of 137

bytes, most of which is allocated for the filename. `serverMessage` has a size of 1046 bytes. Most of these bytes (1KB) are allocated for data being sent in response to client data requests. A small number of `serverMessage` messages - specifically, file existence queries and data responses containing data from the very end of a file - may leave a large amount of this space unused. However, the majority of `serverMessage` packets will use all of the data, since most files are significantly larger than 1KB. Thus, the overhead of having unused space due to the fixed size of packets is amortized over the large number of packets that do use all of the space.

Client messages

A `clientMessage` struct has the following fields.

- `char fileName[128]` : specifies the name of the file that the client is requesting from the server. Always used, because a client message is always a file request or a query to determine if a specific file exists on a server.
- `long portionToReturn` : indicates which part of the file we would like from the server in a file request. Should be an integer N corresponding to the Nth kilobyte of the file. Must be a long to work correctly with some system calls.
- `char haveFile` : a boolean value indicating whether the message is a file request or a file existence query. When the message is a file request and the client expects a kilobyte of the file's data back, set to 0; when the message is a file existence query and the client just wants to know if the file exists on the server, set to 1.

Server messages

A `serverMessage` struct has the following fields.

- `long positionInFile` : an integer indicating the position that the data in the file came from. Should be an integer N corresponding to the Nth kilobyte of the file.
- `int bytesToUse` : the number of bytes in the file that are "real" data; that is, bytes that are not header information or padding. Typically, this will be 1024, unless we have reached the end of the file and there are not 1024 more bytes to send.
- `long fileSize` : the total size of the requested file in bytes.
- `char data[1024]` : the actual data being returned by the server. The server never sends more than 1KB in a message at a time.
- `char hasFile` : a boolean value used to answer file existence queries. Indicates whether the server has the file specified in the query. Set to 1 if the server has the file and 0 if it does not.
- `char outOfRange` : a boolean value that is only set if a client requests data that is beyond the size of the requested file. Since the client stops requesting data when it has received the number of bytes specified in `fileSize` , this flag should be set, but in some cases

where servers drop out during the client's data requests, the client may not receive the full file before starting to request out-of-bounds data.

Design and Functionality

Here, we describe other important aspects of the design of the file-sharing service that have not yet been addressed.

Fault Tolerance

One of the main motivations of peer-to-peer file-sharing services is their ability to tolerate the failure of file servers. In a traditional client-server model, if the single server that is fulfilling the client's request fails during the connection, the client will not receive the entire file. Peer-to-peer file-sharing services use multiple servers to fulfill client requests, so if one server fails, the others can take on its workload without a human user on the client side even noticing that anything went wrong. For N connected servers, our file-sharing service is designed to be tolerant of up to N-2 failures during the data request process and still be functional. There are cases in which the service is not tolerant of this many failures, but we believe these situations (described below) are relatively rare. The two remaining hosts must be a single server that has the desired file and the client that is requesting the file. If there are no servers to fulfill the request or if the client fails, it is clear that the client will not be able to receive the entire file.

To provide fault tolerance, we make use of `select()` to notice when other hosts have disconnected. When a host disconnects from the network of active hosts, those active hosts' client requester threads' `select()` calls will indicate that the disconnected host's socket has data to read, but a subsequent `read()` or `recv()` call on that socket will return no data. When this sequence of events happens in a client requester thread, the thread removes the file descriptor and its associated IP address from vectors shared with the client connector thread and the two server threads. This ensures that 1) the resources associated with this connection are freed and 2) if the server comes back up, we will be able to reconnect to it. We then break out of the current round of data requests and start the next one, iterating through an updated list of servers that no longer includes the disconnected one.

To ensure that the client receives the entire file even in the case of server failures, we keep a `map` of integers and booleans that keeps track of which data chunks have been requested and whether they have been received yet. For example, if there are three servers, at the end of the first round of requests, this map will contain keys 0, 1, and 2. The values associated with these keys are set to `false` when we initially send the request for the chunk and `true` when we receive the request. At the end of each round, we check this map to see if there are any chunks that were requested a while ago (before the previous round) that have still not been received. If this is the case, we re-request the missing chunk from one of the active servers. It

is important to note that our service may NOT be tolerant of the failure of the chosen server that we re-request a missing chunk from; behavior upon the failure of this server is undefined.

Quitting the program

Since there are five threads running concurrently, it takes a bit of extra effort to quit the program. When the user types in "quit" (rather than a filename to request), the client requester thread sets a variable `stopped` to 1. This variable is shared with the other client thread and the two server threads, and each thread checks this variable at the beginning of their respective continuously-running `while` loops. If they see that the variable has been set to 1, they free any necessary resources and return. The main thread calls `pthread_join()` on all threads and waits until they exit, then frees the last of the resources and exits itself.

Sometimes, we may need to quit the program due to an error, rather than a "quit" command from the user. If a thread encounters a fatal error - generally an error return by a necessary system call - the thread sets the `stopped` variable to 1, frees its resources, and returns. Since the other threads (including the client requester thread) already regularly check this variable, they will free resources and exit as well.

Our approach is a bit more CPU-intensive than is likely necessary, since we repeatedly acquire the lock that protects the `stopped` variable and check the variable itself, even when the variable has not been set. However, this approach is simple to implement and understand, and does not have an noticeable impact on performance.

Assumptions

We have made a few simplifying assumptions in order to make this project doable within the time frame. We assume that all servers have the same version of a file, and we do not check for differences in file size or content between files with the same name on different servers. It is certainly possible that different hosts running the program, especially if they are end users' personal computers, may have different versions of a file with the same name, but we assume that this will not be the case. We also assume that each server will be able to same the same file at the same file path; the server-side threads do not search for a requested file outside of the specified filepath. Similarly, we assume that the client requesting the file wants the requested file to be written to the a location at the end of the same filepath it specifies to the server, and that if that filepath specifies a location outside of the directory that the program is running from on the client's, the client actually has the directories that form that same filepath. These are perhaps not realistic assumptions to make for a real peer-to-peer file sharing service, but since our goal is just to make a relatively fault-tolerant networked application, we have left out extra file system functionality to ensure that the basic goals of the project are met. We also assume that files cannot be deleted while a server side thread is reading from

them to fulfill client requests.

Known Issues

Due to a lack of extensive testing resources, we were not able to test all possible fault conditions to ensure that our service is completely N-2 fault-tolerant. Our program can handle failures that occur between message `send()` and `recv()` calls, but it is infeasible to test for server failures that happen during these calls. Thus, we cannot promise that the service will be tolerant of failures that occur during message-transmission system calls. As noted above, the system is also not tolerant of failures during missing message re-requests.

We are also aware of several cases in which the program is definitely not tolerant of failures. These cases are feasible to implement, but had to be skipped primarily due to time constraints. Our main goal was to implement fault tolerance in cases where servers dropped out during data request rounds, so we feel that we have still met the goals and requirements of this project even though it is missing a small amount of functionality. Specifically, the cases that have not been implemented are those in which the server which is fulfilling a repeated request drops out, and those in which a server drops out after a client has connected to it but before the client has send any messages to the server. However, if we assume that any hosts on the network spend a majority of their time requesting data from active servers, these cases are rare. Additionally, any adverse effects that the user sees from the second cases can easily be rectified by simply restarting the program.

We are also aware that our program is more CPU-intensive overall than it has to be. In some places, we implemented functionality that was simple rather than efficient due to the time constraints of the project and the desire to make our service as reliable as possible. There are places where things like signals, condition variables, semaphores, or better lock management could improve the CPU efficiency of the program.