CS4105 Practical 2 Report: Control-Plane

170004680

University of St. Andrews

November 2020

Contents

1 Introduction:

In this practical, a peer-to-peer file-sharing application has been developed for local area networks (LANs) using IPv4 UDP multicast and TCP. Particular attention has been given to the design and implementation of a text-based control-plane protocol, which allows user to discover and query file-spaces connected on the network.

The basic requirements for this practical involved the implementation of file-browser discovery, remote browsing, distributed search, and remote file download. These features have been successfully developed and are documented in sections 3, 4, 5, and 6 respectively.

The additional requirements for this practical involved the implementation of remote file upload and deletion, which have been fully developed and are documented in sections 7 and 8. An analysis of the scalability, performance, and privacy/security concerns with the developed file-browsing application is given in section 9.

2 Compilation And Execution Instructions:

From within the `code/` directory, the `FileTreeBrowser` program can be compiled using (see figure 1):

javac FileTreeBrowser.java

Alternatively, a `makefile` has been provided, allowing for the program to be compiled using 'make'. The resulting compiled class files can be removed using 'make clean'.

After compiling the program in the `code/` directory, `FileTreeBrowser` can be executed using (see figure 2):

java FileTreeBrowser

Figure 1: The following screenshot shows that the program can be compiled within the St. Andrews School of Computer Science lab environment without error. All of the `.java` files have been compiled into `.class` files that can be used for program execution.

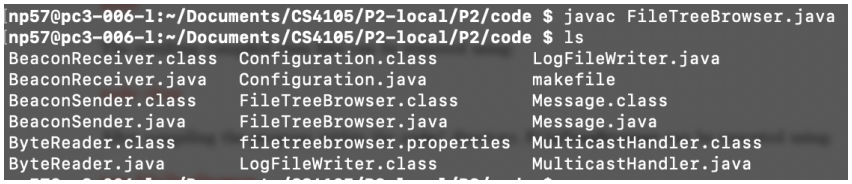
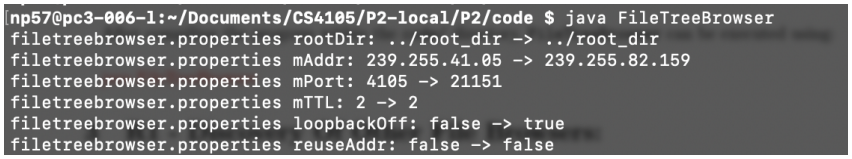


Figure 2: The following screenshot shows that the program can be executed within the St. Andrews School of Computer Science lab environment without error. As seen, the program begins execution by reading in a properties file that affects the program configuration.



3 R1 - Discovery Of Other File Browsers:

3.1 Description:

An IPv4 multicast mechanism has been implemented which allows for the `FileTreeBrowser` to discover other instances of itself on the network of Linux hosts in the School lab. This has been implemented using the beacon messages defined in the protocol specification. To see other instances of the `FileTreeBrowser` on the network, the '`:showBeacons`' option can be invoked within the program.

3.2 Testing:

This section demonstrates some of the testing carried out to ensure the correctness of the solution for the R1 requirements. To this end, log files and screenshots have been used to verify the program behaves as expected.

Figure 3: In this test, the `pc3-006-l` and `pc3-010-l` lab clients were executing `FileTreeBrowser` on the School lab (LAN) network. In the top screenshot, the program output shows that the beacons of another `FileTreeBrowser` are read and displayed. Additionally, the `pc3-006-l` log file (bottom screenshot) indicates that beacons for discovery are being sent and read correctly. Also, the log file shows that each `FileTreeBrowser` ignores its own transmissions when the `loopbackOff` option is set to true.

Figure 4: This test verifies that many instances of `FileTreeBrowser` are discovered and their beacons are displayed to the user. In this test, four clients executed the program. The screenshot shows that for a given lab client (e.g. `pc3-044-l`), the other three client beacons could be received and shown. This verifies that the discovery protocol scales as more lab clients connect the the multicast group. The log file shown in the bottom screenshot (for `pc3-044-l`) also verifies that the correct behaviour is shown at scale.

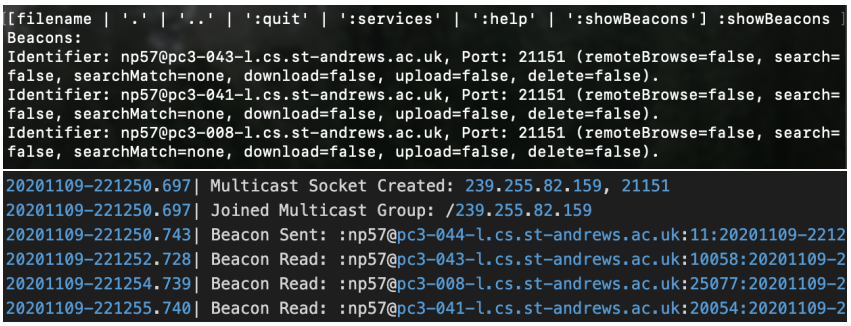
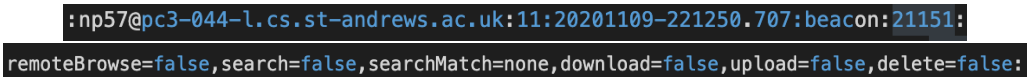


Figure 5: This test verifies that expired beacons are removed. A beacon expiry occurs if a new beacon is not received within a given amount of time since a previous beacon was received. In this case, four lab clients were connected in the scenario from figure 4. Then, two of these clients (`pc3-008-l` and `pc3-041-l`) disconnected. The top screenshot shows that these beacons are removed as available remote file browsers. The log file in the bottom screenshot shows that, after two of the clients disconnected, the beacon for the only remaining other peer is received.

Figure 6: This test verifies that, when no other clients are connected to the multicast group, no beacons are received and shown to the user (`loopbackOff` is true). In the top screenshot, the program output shows that an empty list is displayed to the user when there are no beacons to show. The bottom screenshot shows that this is because, after all other clients have disconnected, there are no beacons received from the multicast group.

Figure 7: This test verifies that the beacon format is as expected by the protocol specification. It can be easily verified from the screenshots, which represent a single beacon, that the protocol specification regarding the sending and receiving of beacons has been adhered to.



4 R2 - Remote Browsing:

4.1 Description:

Remote browsing capabilities have been added to `FileTreeBrowser` via TCP. Using beacon information from the discovery protocol, the user can connect to a remote instance of `FileTreeBrowser` and browse its file-space with the same text commands as used for local browsing. To connect to a remote file-space, the user can invoke the '`:remoteBrowse`' command. Whilst remotely browsing, the user may invoke the '`:localBrowse`' command to return to local file-space browsing.

4.2 Testing:

This section demonstrates some of the testing carried out to ensure the correctness of the solution for the R2 requirements. To this end, screenshots have been used to verify the program behaviour is as expected.

Figure 8: This test verifies that the validation checks work as expected when the user has to specify a beacon to connect to for remote file-space browsing. As seen in the screenshots, a host-name and port number for an active beacon must be specified to indicate which remote file browser in the multicast group to connect to.

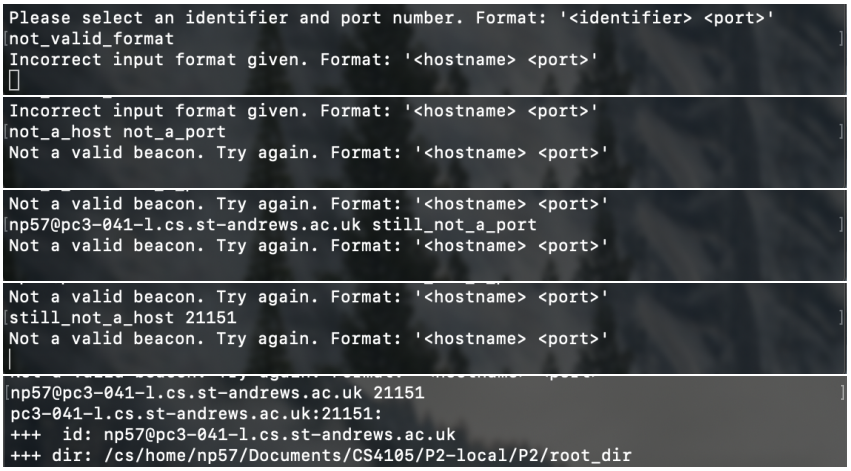


Figure 9: This test verifies that the user is returned to local browsing when there is a complication in specifying a beacon to connect to. The top screenshot shows that if the user attempts to carry out remote browsing when there are no active beacons, then they are informed of this and local browsing resumes. Additionally, the bottom screenshot shows that if all active beacons expire whilst selecting a beacon to connect to, then the user is informed and local browsing resumes.

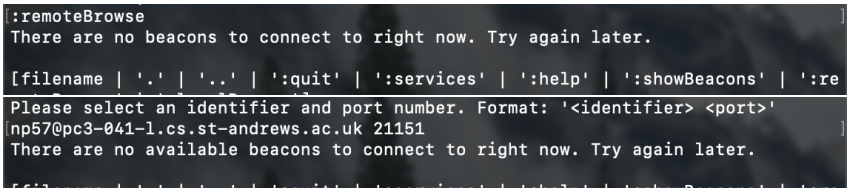


Figure 10: This test verifies that all of the available text commands (i.e. '.', '..', '`:help`', '`:services`', and '`<path_name>`') work as expected when navigating a remote file browser. In each of the screenshots below, a given command has been tested on a remote file-space. When a command is executed on a remote file-space, the host-name and port number of that file-space is provided as feedback so the user understands where the file-space they are navigating is located. All of the screenshots show the working order of the program and the outputs are as expected. Some screenshots do not show the full output. This is for conciseness and the screenshot still verifies that the correct behaviour is exhibited by the program.

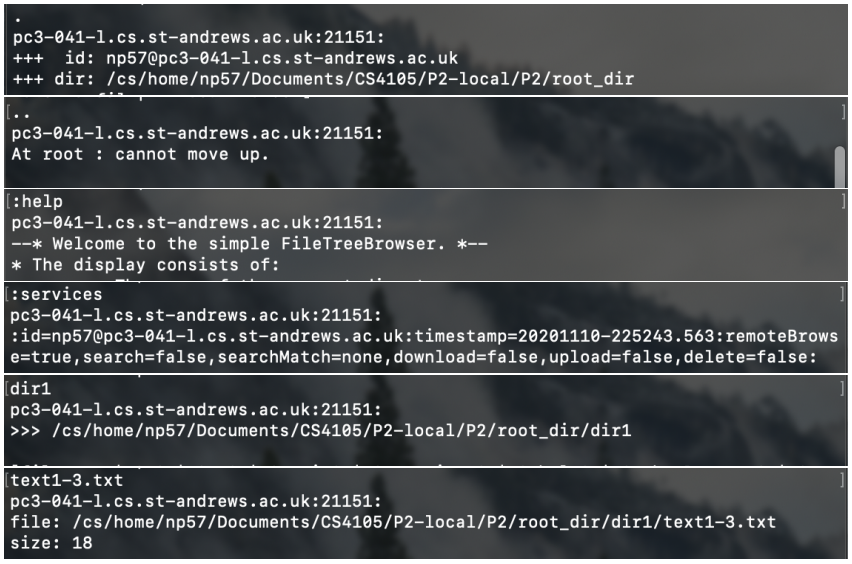


Figure 11: This test verifies that the TCP server at a given `FileTreeBrowser` instance can manage multiple incoming connections and allow for many different clients to browse the file-space of the server. In this example, each of the screenshots represent an individual client, all connected to the same peer (`pc3-041-l`). At each client, a different part of the `pc3-041-l` file-space is being browsed. Each client has requested the information of a leaf file in the file-space hierarchy, which is correctly provided by the server. Multiple commands from each client were required to reach the desired file location, all of which returning the correct result. Thus, the screenshots verify that multiple different clients can access a single server file-space.

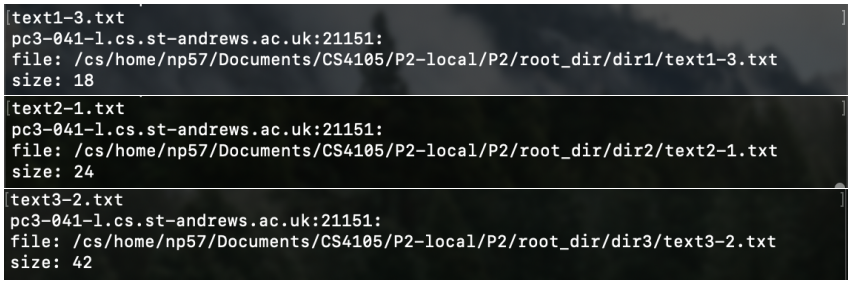
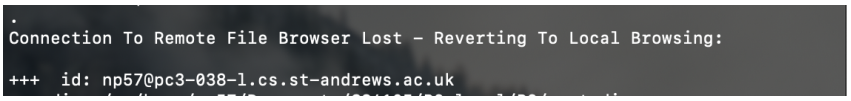


Figure 12: This test verifies that the '`:localBrowse`' command correctly disconnects the user from a remote file-space, and local file-space browsing is initiated. The screenshot shows that upon entering the command, the root directory information for the local file-space is shown, which indicates that the client has correctly disconnected from the remote file-space.

Figure 13: This test verifies that the TCP client of a `FileTreeBrowser` appropriately manages a disconnection from a TCP server, which is likely to occur in the peer-to-peer scenario. The screenshot shows that, if one `FileTreeBrowser` is disconnected from another when browsing its remote file-space, the program informs the user that the peer has disconnected and local file-browsing is resumed.



5 R3 - Search Capability:

5.1 Description:

Distributed searching using UDP multicast has been implemented using the provided protocol specification. This allows for users to search for remote files at peers in the multicast group. The user can perform three different search types: 'path', 'filename', and 'substring', which are explained in the protocol specification. To perform searches, the user can invoke the '`:search`' command, which then prompts for a search type and search string.

5.2 Testing:

This section demonstrates some of the testing carried out to ensure the correctness of the solution for the R3 requirements. Screenshots have been used to verify the requirements have been met and that the program behaviour is as expected.

Figure 14: This test verifies that when performing a distributed search using the `:search` command, the user must specify one of the supported search types.

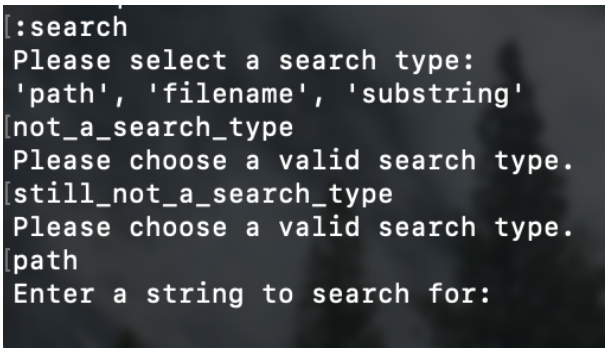


Figure 15: This test verifies that when performing a distributed search using the `:search` command, the user must specify a non-empty query string.

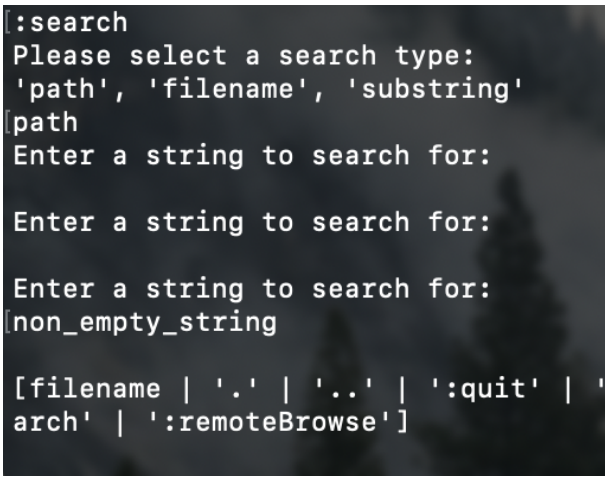


Figure 16: This test verifies that when a user performs a search of the 'path' type, all search responses for active peers are shown to the user. In the example shown, a total of 4 peers were connected to the multicast group. Therefore, when one peer sent out a search request (top screenshot), the other three peers sent responses to the request (bottom screenshot). As seen, all three peers sent a `search-result` response, which is expected as the path given in the search leads to a valid file at all of the peers.

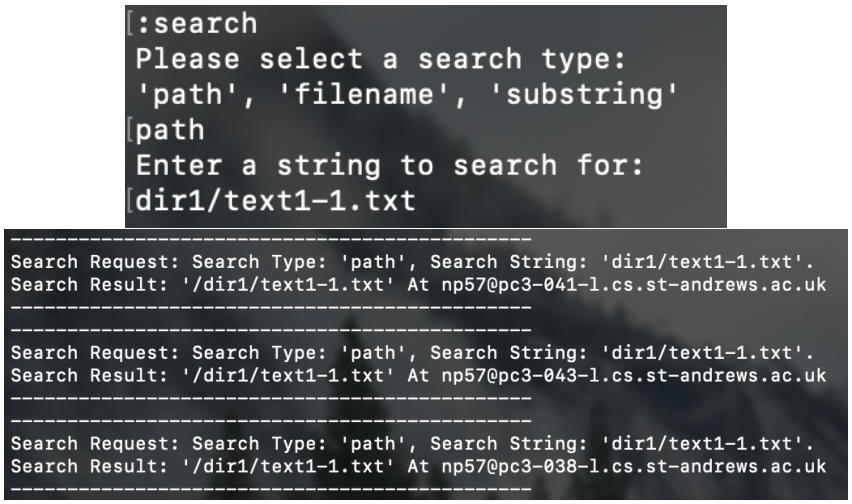


Figure 17: This test verifies that, when a user performs a search of the 'filename' type, all search responses for active peers are shown to the user. In the example shown, 4 peers were connected to the multicast group. Therefore, when one peer sent out a search request (top screenshot), the other three peers sent responses to the request (bottom screenshot). As seen, all three peers sent a `search-result` response, which is expected as the file-name given in the search leads to a valid file at all of the peers.

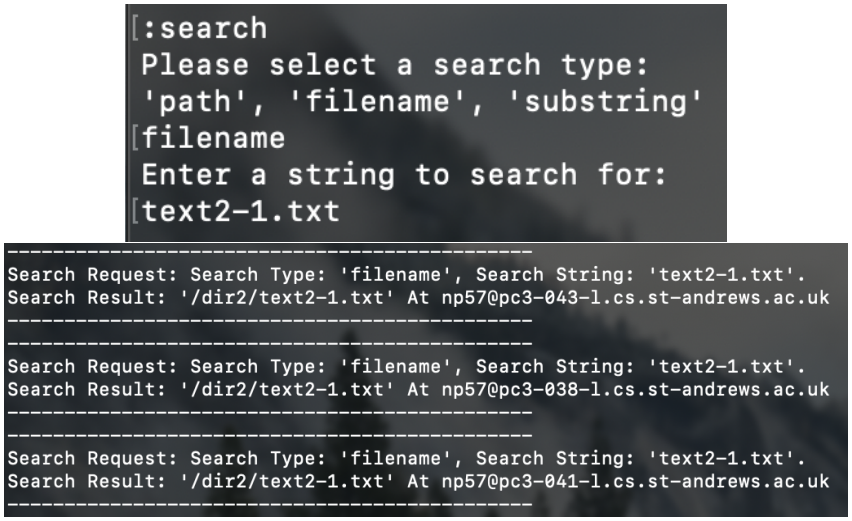


Figure 18: This test verifies that, when a user performs a search of the 'substring' type, all search responses for active peers are shown to the user. In the example shown, a total of 4 peers were connected to the multicast group. Therefore, when one peer sent out a search request (top screenshot), the other three peers sent responses to the request (bottom screenshot). The search request essentially requested all text files at all active peers. Thus, most of the responses to this request have been left out for conciseness, however, all responses were checked for correctness.

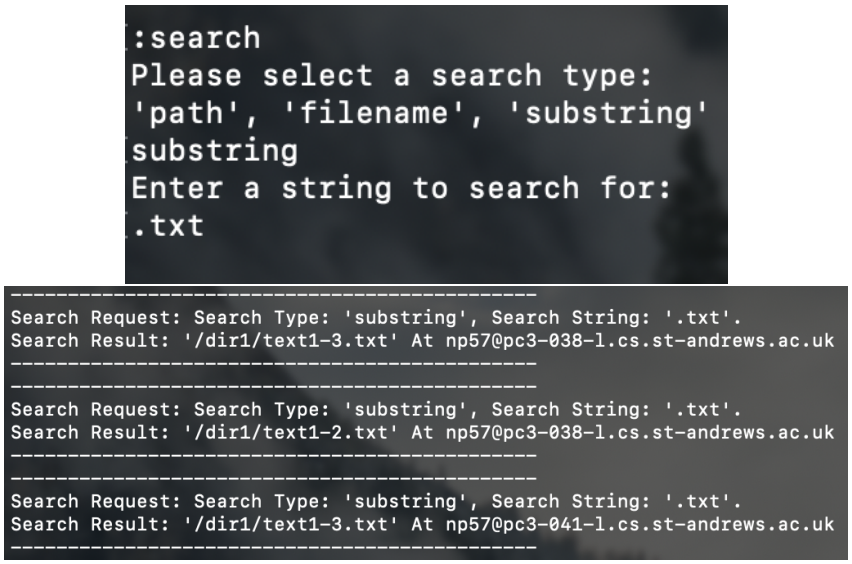


Figure 19: This test verifies that, when a peer does not have a result for a given request, it is reflected in the output to the user. As seen in the top screenshot, a 'path' search was conducted with a path that isn't valid at any peers. As a result, all peers sent back a `search-error` response. In practice, this type of response wouldn't need to be shown to the user. However, it has been included to demonstrate the correct and full implementation of the R3 requirements and distributed search protocol design.

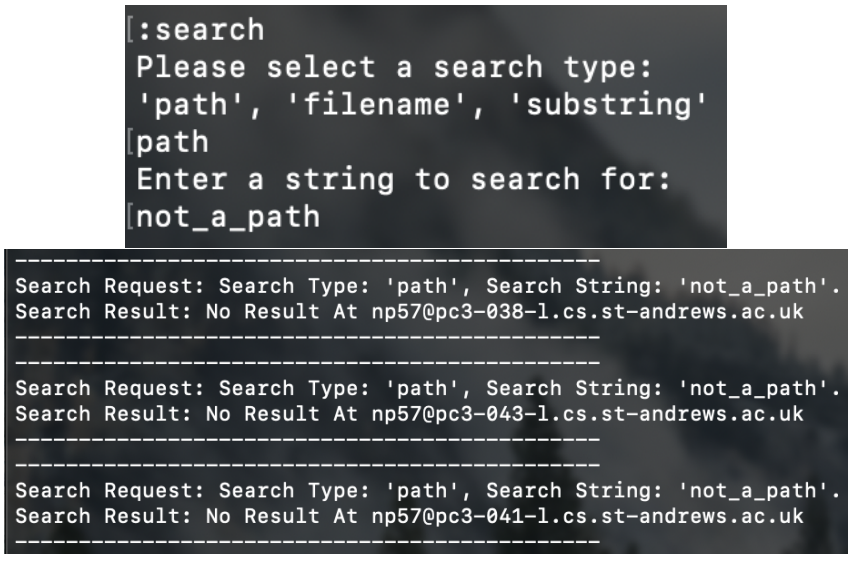
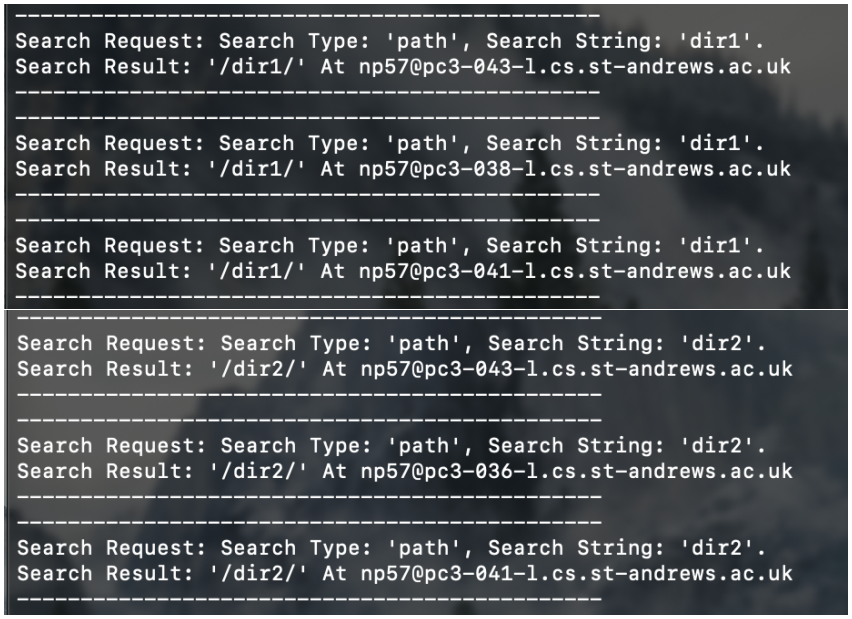


Figure 20: This test verifies that, when multiple concurrent searches are occurring in the multicast group, a peer will ignore all responses that do not correspond to one of their previously sent requests. In the examples shown, the top screenshot shows the results for a request sent by one peer (`pc3-036-l`), whilst the bottom screenshot shows the results for a request sent by another peer (`pc3-038-l`) at the same time. As seen, both peers successfully get their responses from the other peers and both ignore responses that don't correspond to their own search requests.



6 R4 - Download Design And Implementation:

6.1 Description:

Remote file downloading has been implemented by extending the provided text-based control-plane protocol. This protocol acts over IPv4 UDP multicast for download message signalling, whilst TCP has been used for reliable file transfer of the files to be downloaded. The user is able to download a desired file from a remote file-browser by invoking the ':download' command.

6.2 Design:

This section describes and justifies design decisions regarding the remote downloading extension of the text-based control-plane protocol. The protocol definition for remote downloading can be found in the provided `cs4105_p2_protocol_specification.txt` file.

The remote downloading extension to the protocol is designed to be simple. The added protocol structures closely replicate the existing protocol structures for distributed search. Namely, there are three forms of message when remote downloading: `download-request`, `download-result`, and `download-error`. This is similarly the case when searching. The message types share the same meaning, except that the context changes depending on whether search or download is occurring. By replicating existing protocol structures, additional functionality can be added to the protocol whilst minimally affecting the protocol complexity. As a result, the overall protocol is simpler to understand and easier to implement. The protocol simplicity aids performance; yet, the protocol has not been optimised to maximise performance.

Fields that have been added to the protocol are added out of necessity. For example, the fields <`target_identifier`> and <`target_file_path`> indicate which host and file is desired to be downloaded in a request. The <`file_transfer_port`> is added to responses, which indicates how to connect via TCP and download the file. All other required fields are re-used from the original protocol definition. Only necessary fields have been added to the protocol. While optional fields could provide useful information for downloading, they are unnecessary and are not always likely to be implemented. Therefore, they been excluded from the protocol extension. Again, this keeps the overall protocol simple, which aids with ease of implementation and protocol scalability.

The entire protocol has been designed using text-based, variable-length fields, which assists scalability by not constraining field sizes. For example, the protocol definition when downloading permits arbitrary-sized file paths to be used, which allows for the size of remote file-space hierarchies to scale. Despite this, the text-based control-plane protocol uses UDP datagrams for packet delivery. Therefore, the protocol is limited by the maximum payload size of a UDP packet. Signalling messages in the control-plane may be larger than the maximum UDP packet size via packet splitting; however, this would make the protocol more complex and would negatively impact protocol performance. These issues are further emphasised by the fact that packets are delivered unreliably by UDP.

The download extension to the protocol preserves modularity. Therefore, changes to any one of the functionalities supported by the protocol can occur independently. This allows for different functionalities supported by the protocol to be developed independently and with different evolution timescales. Also, parts of the protocol that have been re-used, such as <`response_id`>, have the same semantic meaning, which keeps the protocol simple and preserves modularity.

6.3 Testing:

This section demonstrates some of the testing carried out to ensure the correctness of the solution for the R4 requirements. Screenshots of program output and log files have been used to verify the requirements have been met and that the program behaviour is as expected. Throughout the testing of the remote download functionality, multiple file-browsers were connected to one another performing tasks like remote browsing to ensure that the presence of the uploading functionality would not impair the other functionalities implemented.

Figure 21: This test verifies that, when downloading a file, the user must specify a non-empty file path to send as a query to the remote peer, which is shown in the screenshot. As the state of the root directory is not known to the client, no further validation can be performed on the file path provided by the user. Instead, any issues with the file path are signalled by the remote file-browser once the download request is processed.

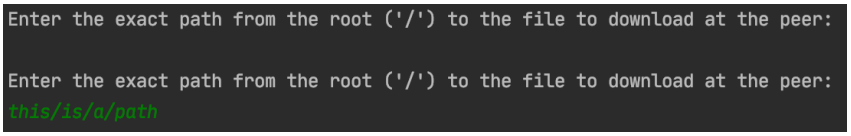


Figure 22: This test verifies that, when a download request is processed, the file path leading to the file to download must exist, must not be a directory, and must be within the root directory. This is seen in the screenshots. The responses from a remote-file browser to download requests where these conditions are not met is given, which shows that the download requests were rejected.

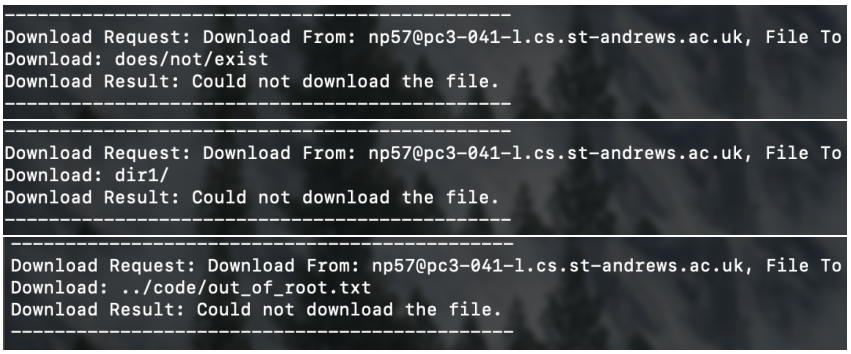


Figure 23: This test verifies that the user must specify a valid file path to save the downloaded file to when a download request is approved by a remote file-browser. As seen in the screenshot, the user must provide a valid path within the root directory that leads to a filename (which or may not exist), and the extension of the file must be provided.

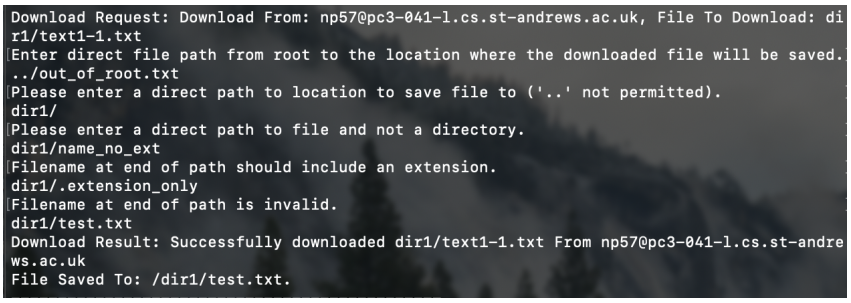


Figure 24: This test verifies that text files at a remote file-browser are successfully downloaded for valid download requests. The top screenshot shows the program output of a successful download, whilst the bottom screenshot shows the log file output, which indicates the bytes of the file were successfully written to the local file-space. For the `text1-1.txt` file, 6 bytes is the correct file size. Whilst this is only one example to illustrate the working order of the program, all text files were downloaded from a remote file-browser with various sizes to ensure correctness. The contents of these files were inspected to ensure the bytes written to the file were those expected.

Figure 25: This test verifies that image files at a remote file-browser are successfully downloaded for valid download requests. The top screenshot shows the program output of a successful download, whilst the bottom screenshot shows the log file, which indicates the bytes of the file were successfully written to the local file-space. For the `crawdad.jpg` file, the amount of bytes written is correct. This is one example to illustrate the working order of the program; however, all image files were downloaded from a remote file-browser with various sizes to ensure correctness. The contents of these files were viewed to ensure the bytes written to the file were those expected.

Figure 26: This test verifies that video files at a remote file-browser are successfully downloaded for valid download requests. The top screenshot shows the program output of a successful download, whilst the bottom screenshot shows the log file, which indicates the bytes of the file were successfully written to the local file-space. For the `all_the_twitters-web.mp4` file, the amount of bytes written is correct. This is one example to illustrate the working order of the program; however, all video files were downloaded from a remote file-browser with various sizes to ensure correctness. The contents of these files were watched to ensure the bytes written to the file were those expected.

7 R5 - Upload Design And Implementation:

7.1 Description:

Remote file uploading has been implemented by extending the provided text-based control-plane protocol. This protocol acts over IPv4 UDP multicast for upload message signalling, whilst TCP has been used for reliable file transfer of the files to be uploaded. The user is able to upload a desired file to a remote file-browser by invoking the ':upload' command.

7.2 Design:

This section describes and justifies design decisions regarding the remote uploading extension of the text-based control-plane protocol. The protocol definition for remote uploading can be found in the provided `cs4105_p2_protocol_specification.txt` file.

The remote uploading extension to the protocol specification is near identical to the protocol definitions for remote downloading. The context simply changes from downloading to uploading, which mostly affects the TCP file transfer in the user-plane and not the signalling of the control-plane. For instance, <`target_file_path`> indicates the location to upload to instead of download from. Also, <`file_transfer_port`> indicates the port the client should connect to in order to send a file as opposed to read a file using TCP. As the protocol definitions for remote upload are nearly identical to remote download, the protocol is kept simple and minimally adds fields, which assists scalability, performance, and ease of implementation.

The near identical definitions for remote upload and download mean that these two functionalities could have been implemented under a single 'file-transfer' message type, with a single field indicating which of the download or upload context the message relates to. However, the two functionalities have been defined separately to preserve modularity. As a result, the remote download and upload features can be developed independently, with changing requirements to one of these features having no effect on the other in the protocol.

7.3 Testing:

This section demonstrates some of the testing carried out to ensure the correctness of the solution for the R5 requirements. Screenshots of program output and log files have been used to verify the requirements have been met and that the program behaviour is as expected. Throughout the testing of the remote upload functionality, multiple file-browsers were connected to one another performing tasks like remote browsing to ensure that the presence of the uploading functionality would not impair the other functionalities implemented.

Figure 27: This test verifies that, when the user is trying to upload a file to a remote file-browser, they must provide a non-empty file path that specifies the location to upload to. As the state of the remote root directory is not known to the local file-browser, no further validation is performed on the provided file path locally. Instead, the remote file-browser signals if there is an issue with the provided location to upload to with an upload error message.

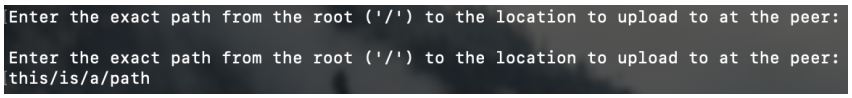


Figure 28: This test verifies that, in order for an upload request to be accepted by a remote file-browser, the user must specify a location to upload to that is within the root directory, is not a directory, and must be valid. This is seen in the screenshots below, where invalid upload locations resulted in the upload requests being denied.

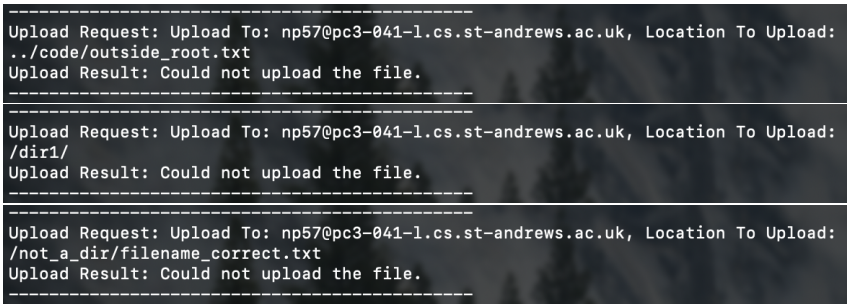


Figure 29: This test verifies that, when an upload request is accepted, the user must provide the file path for the file to upload. The file path must be within the root directory hierarchy, not be a directory, and must exist. This is observed in the screenshot.

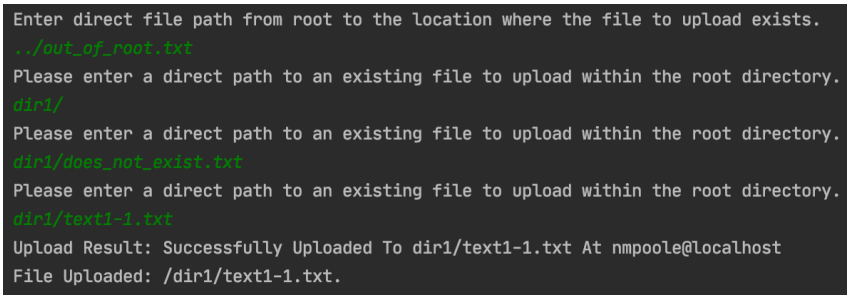


Figure 30: This test verifies that text files are successfully uploaded to a remote file-browser for valid upload requests. The top screenshot shows the program output of a successful upload, whilst the bottom screenshot shows the log file output, which indicates the bytes of the file were successfully written to the remote file-space. For the `text1-1.txt` file, 6 bytes is the correct file size. Whilst this is only one example to illustrate the working order of the program, all text files were uploaded to a remote file-browser with various sizes to ensure correctness. The contents of these files were inspected to ensure the bytes written to the remote file were those expected.

Figure 31: This test verifies that image files are successfully uploaded to a remote file-browser for valid upload requests. The top screenshot shows the program output of a successful upload, whilst the bottom screenshot shows the log file output, which indicates the bytes of the file were successfully written to the remote file-space. For the `crawdad.jpg` file, the number of bytes written is correct. Whilst this is only one example to illustrate the working order of the program, all image files were uploaded to a remote file-browser with various sizes to ensure correctness. The contents of these files were viewed to ensure the bytes written to the remote file were those expected.

Figure 32: This test verifies that video files are successfully uploaded to a remote file-browser for valid upload requests. The top screenshot shows the program output of a successful upload, whilst the bottom screenshot shows the log file output, which indicates the bytes of the file were successfully written to the remote file-space. For the `all_the_twitters-web.mp4` file, the number of bytes written is correct. Whilst this is only one example to illustrate the working order of the program, all video files were uploaded to a remote file-browser with various sizes to ensure correctness. The contents of these files were watched to ensure the bytes written to the remote file were those expected.

8 R6 - Delete Design And Implementation:

8.1 Description:

Remote file deletion has been implemented by extending the provided text-based control-plane protocol. This protocol acts over IPv4 UDP multicast for deletion message signalling. The user is able to delete a desired file at a remote file-browser by invoking the ':delete' command.

8.2 Design:

This section describes and justifies design decisions regarding the remote deletion extension of the text-based control-plane protocol. The protocol definition for remote deletion can be found in the provided `cs4105_p2_protocol_specification.txt` file.

The remote deletion extension to the protocol specification replicates the same structure as distributed search, remote download, and remote upload. Namely, there are three deletion message types: `delete-request`, `delete-result`, and `delete-error`. As mentioned, re-using protocol structures when possible extends to supported functionality of the protocol whilst minimally increasing the protocol complexity, which aids protocol scalability, performance, and ease of implementation.

As with the protocol extensions for remote download and upload, the protocol definitions for remote deletion signalling are minimal in the number of required fields. This is most prevalent for `delete-result` and `delete-error` messages where the payload has no fields other than the payload type and <`response_id`> included. The response ID indicates which request the response message is in reply to, whilst no other fields are required as the type of message itself fully conveys the required meaning. A `delete-result` indicates the specified file was successfully deleted, whilst a `delete-error` indicates the specified file could not be deleted. Additional fields could be useful for these message types but, as they are not always necessary, they are not likely to always be implemented. As a result, protocol simplicity was prioritised and any additional fields for usefulness were excluded. For example, the `delete-error` message could have a specific error message field which further explains why the file specified could not be deleted. In such cases, the file may not exist or the file may be protected and cannot be deleted without permission. However, whilst useful, this field would not always be used as the main result (that the deletion failed) is the dominant information required.

Finally, all of the added features for remote upload, download, and delete use a <`target_identifier`> field to indicate which peer connected to the multicast group is to perform the download, upload, or delete requested. This establishes one-to-one communication between two peers using the one-to-many communication provided by IPv4 UDP multicast in the control plane. As a result, any peer on the network can view the communications between other peers. This raises security and privacy concerns that were not considered for the protocol specification design but are addressed in section 9. This also presents possible performance issues at scale as many requests can be sent into the multicast group, which are read by all peers even though only a small proportion of these requests will apply to each peer. However, the use of the <`target_identifier`> field for one-to-one communication over UDP multicast is justified for two main reasons.

Firstly, one-to-one communication is required for remote download, upload, and deletion, whilst one-to-many communication is required for file-browser discovery and distributed search. Using a single communication technology is desired to reduce the complexity of the protocol design and difficulty in implementation. A one-to-one communication technology, like TCP, for the control-plane would yield no simple method for file-browser discovery and would not scale well; an individual connection would be required at a peer for every single other file-browser peer connected. Therefore, UDP multicast is most appropriate for providing the communication required by this peer-to-peer, LAN-based file-browser.

Secondly, one-to-one communication over UDP multicast for requesting remote downloads, uploads, and deletions is required because, although a specified file path in the request may be valid at multiple peers, the file contents may differ at the different peers, which makes the files different. As a result, a peer must specify a single file to download/upload/delete, which must involve specifying the peer that the file is located at. An alternative approach would have a peer send out a download/upload/delete

request to all other peers and then the user can simply select the intended file from the list of responses. However, this wastefully creates traffic in the multicast group. For remote uploading and downloading, this would result in the creation of many TCP servers at all peers to send/receive files when ultimately a single server is connected to. For remote deletions, this would result in many files being deleted at all peers that have the same path, even though the files themselves may all be different and not all intended for deletion.

8.3 Testing:

This section demonstrates some of the testing carried out to ensure the correctness of the solution for the R6 requirements. Screenshots of program outputs have been used to verify the requirements have been met and that the program behaviour is as expected. Throughout the testing of the remote deletion functionality, multiple file-browsers were connected to one another performing tasks like remote browsing to ensure that the presence of the deletion functionality would not impair the other functionalities implemented.

Figure 33: This test verifies that, when the user is trying to delete a file from a remote file-space, they must provide a non-empty file path that specifies the location of the of th file to delete. As the state of the remote root directory is not known by the local file-browser, no further validation is performed on the provided file path locally. Instead, the remote file-browser signals if there is an issue with the provided file path via a download error message.

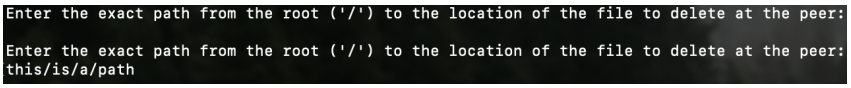


Figure 34: This test verifies that, when the user is trying to delete a file from a remote file-space, the remote file browser sends an error when the specified file path is outside the root directory hierarchy, ends with a directory, or simply doesn't exist. This is seen in the screenshots below, where invalid file paths resulted in the delete requests being denied.

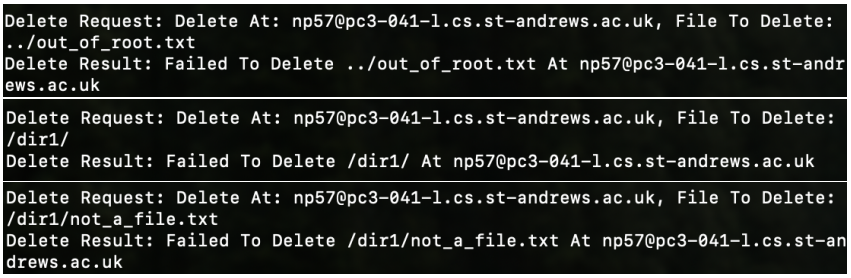
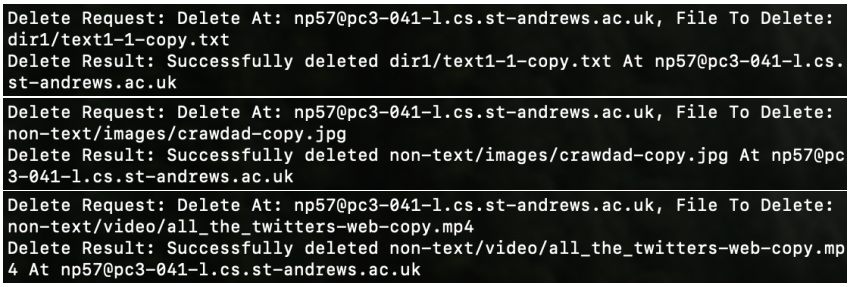


Figure 35: This test verifies that text, image, and video files can be deleted from a remote file-space when delete requests are valid. In the screenshots shown, three example files have been deleted, which is reported to the user as a success. The file-space at the remote file-browser where the deletions occurred was inspected to ensure the files were deleted.



9 R7 - Analysis:

9.1 Scalability:

This section analyses the scalability of the control-plane protocol design and implementation with respect to an increasing number of peers (file-browsers) that may be connected on the LAN.

As mentioned in the design sections for R4, R5, and R6, the control-plane protocol has been designed with a simple structure and with unconstrained fields. This aids with the scalability of the program which implements the protocol specification.

When implementing the protocol specification in a LAN-based, peer-to-peer file-browsing application, several design decisions were made for scalability. For instance, each of the functionalities supported by the protocol are managed by their own thread. This allows for the resources available at a given peer to be evenly distributed for processing different control-plane messages, whilst simultaneously allowing the user to act as a peer in the network by creating requests, browsing remote file-spaces, etc. As a result, the developed program should effectively manage the increasing network load given by increasing the number of connected peers in the LAN.

However, the use of threads to implement the protocol specification is not entirely beneficial to scalability. Currently, the thread which manages remote browsing consists of a TCP server which waits for incoming connections. When a connection is made by a peer in the network, the listening TCP server will create a socket connection for the peer, which is then delegated to a new thread for processing the remote browsing commands given by the peer. As a result, the number of threads being executed at a given peer is linear with respect to the number of remote peers wishing to browse the file-space at the given peer. Whilst a great deal of threads can be created and managed by the program, there is a much smaller limit on the number of threads that can be created before the user experiences issues with the responsiveness of the program.

Therefore, the scalability of the protocol implementation may be improved by using UDP as opposed to TCP. Since UDP is connectionless, a single thread can be used to handle all incoming remote browser requests from many different peers. This would make the total number of threads required by the program fixed, improving scalability. Even though reliable delivery is traded-off when using UDP instead of TCP, this should not be a concern as the program is designed to be used in a LAN-based context, where packet loss is ordinarily minimal.

Furthermore, the protocol has been implemented in a program that creates a peer-to-peer network structure when instances connect to each other on a LAN. The peer-to-peer network structure is very scalable as files demanded by peers in the LAN become evenly distributed amongst the peers, which distributes workload more evenly across the peers.

9.2 Performance:

This section analyses the performance of the control-plane protocol design and implementation.

The protocol specification was designed to be simple, with the key factor effecting simplicity being the repeated use of message structures for different functionalities that were added (i.e. distributed search, remote download, remote upload, and remote deletion all share message structure definitions). As a result, the protocol implementation assigns threads for each of the functionalities supported by the protocol. The delegation of different tasks to separate threads evenly distributes resources between the different tasks the peer must perform, which results in the program being responsive to both the local user and to interacting with other peers communicating across the multicast group.

A performance improvement that can be made to the current implementation of the protocol specification comes from the parsing of messages. Currently, all incoming messages from the multicast group are parsed and distributed to the thread that manages the parsed message type. In the case of download requests, upload requests, delete requests, and all response messages, the majority of the incoming messages are disregarded. Therefore, time and resources are wasted on fully parsing and

distributing the messages to the correct thread. Alternatively, when messages are first received, they should be partially parsed to determine if they are relevant to the peer or not. Only once the message has been determined to be relevant should the message be fully parsed and distributed to the relevant thread for processing. This is a simple improvement that will result reduce the resource and memory overhead at each peer, which improves performance.

Another performance improvement will result from implementing search caching. Repeated requests for remote download, upload, and deletion of a file are unlikely, but it is reasonably likely that a user will repeat search queries when using the file browser application. When a search request is sent out to the multicast group, n responses are received, where n is the total number of currently connected peers to the LAN. Even though only the peer that requested the search will use the responses, all peers in the network will receive all responses to the initial search request. Then, all peers will have to partially process all of the responses so they know they can ignore them (except for the individual peer that made the request). The distributed search generates a lot of network traffic that each peer will have to process. The amount of traffic is emphasised by the fact that search requests can come from all connected peers and each request may have multiple results, which are sent out individually. Therefore, caching frequently made search requests can reduce network traffic and improve performance at each of the peers in the network. However, the effectiveness of search caching is dependant on the frequency by which new peers connect and disconnect from the LAN, and on the frequency by which their file-spaces alter significantly as to produce new results to search requests. The lifetime of the cache for frequently made search requests should be tailored to account for the frequency in changes to the available peers and their file-spaces.

9.3 Privacy And Security:

This section analyses the privacy and security concerns related to the control-plane protocol design and implementation.

The control-plane protocol has been designed and implemented using IPv4 UDP multicast for plain-text-based communication. As a result, several security and privacy concerns arise due to the transparency of information communicated across the control-plane between connected peers.

All peers in the LAN are able to view the search, download, upload, and delete actions carried out by all other peers connected to the multicast group, which presents a privacy issue. Also, processes for file download and upload can be intervened on by users with modified clients. For instance, a modified client can connect to TCP servers for downloading/uploading a file before the intended recipient, which would deny them from downloading/uploading a file. Furthermore, a modified client can issue incorrect messages designed to disrupt other peers. For instance, the control-plane protocol specification does not specify a standard character set. The implementation uses the 'ASCII' standard for interpreting UDP datagrams; though, a modified client could use characters not supported to potentially cause disruption. Additionally, the developed protocol does not support differential file permissions between peers, which means that all peers are equally capable of performing actions that can be potentially destructive, such as when performing remote deletions.

Despite these privacy and security issues, the likelihood and severity of the concerns should be considered in the context of the application when discussing how to address privacy and security with the developed program.

For instance, most of the privacy and security issues discussed arise from the user having a modified version of the program or creating a program to snoop on traffic in the multicast group. However, both of these scenarios are unlikely in the context of a LAN, as it is far easier for a malicious user to simply go watch the target user interacting with the program (i.e. looking over their shoulder). The concept of a malicious user in the LAN is also unlikely given that a LAN should be comprised of trusted users. Therefore, attention should be paid towards securing the LAN, as opposed to alternative approaches like implementing encryption for all messages.

Encryption is a poor solution for the privacy and security concerns expressed given the context of the program and design of the protocol. The resource overhead resulting from the implementation of

encryption in the program is undesirable. Additionally, the entirety of the vocabulary that can be encoded in the encrypted messages is known to all users and is very limited. A truly malicious user could use this information to more easily decode messages (or simply encode and compare all possible messages if weak encryption is used).

Additionally, the severity of the privacy and security issues being enacted is very low. Unless a malicious user is able to modify remote peer programs, all peers are confined to the domain of the root directory used by the program, which is not equal to the root directory assigned to the users. Therefore, confidential or damaging information cannot be retrieved by the files included in the remote root directory used by the program; the files in this directory are intended to be freely available.

10 Conclusion:

In conclusion, a peer-to-peer file-sharing application has been developed for local area networks (LANs) using IPv4 UDP multicast and TCP. Particular attention has been given to the design and implementation of a text-based control-plane protocol, which allows user to discover and query file-spaces connected on the network.

The basic requirements for this practical were achieved through the implementation of file-browser discovery, remote browsing, distributed search, and remote file download. These features have been appropriately developed, tested, and reported.

The additional requirements for this practical were achieved through the implementation of remote file upload and deletion, which have been fully developed, tested, and documented. An analysis of the scalability, performance, privacy, and security concerns with the developed file-browsing application was also given.