COMP90015 Distributed System

Assignment 1 - Written report

By Group 77:

Taylor Tang 1323782

Qingyun Wu 1370606

Table of Contents

Brief design of this implementation: 3

Question 1 - Strategy for Peer Selection and Block Selection 4

Question 2 – Improve Protocols 6

Question 3 – Socket Timeout Analysis 8

Question 4 – Testing and Limitations.....9

Reference.....12

Appendix A - JConsole Screenshots 13

Appendix B - Htop Screenshots 20

Brief design of this implementation:

This program on the peer side utilizes one thread to communicate with the index server. For each download request, a new thread is initiated to handle multiple new download threads.For each upload file, a new thread is initiated to handle a sharing port. Peer can open multiple ports to share one file.

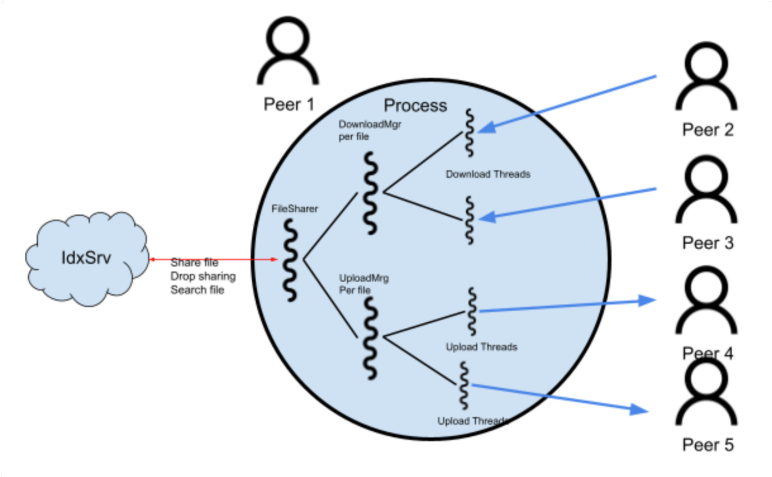


Diagram 1: Overall file-sharer processes

Question 1 - Strategy for Peer Selection and Block Selection

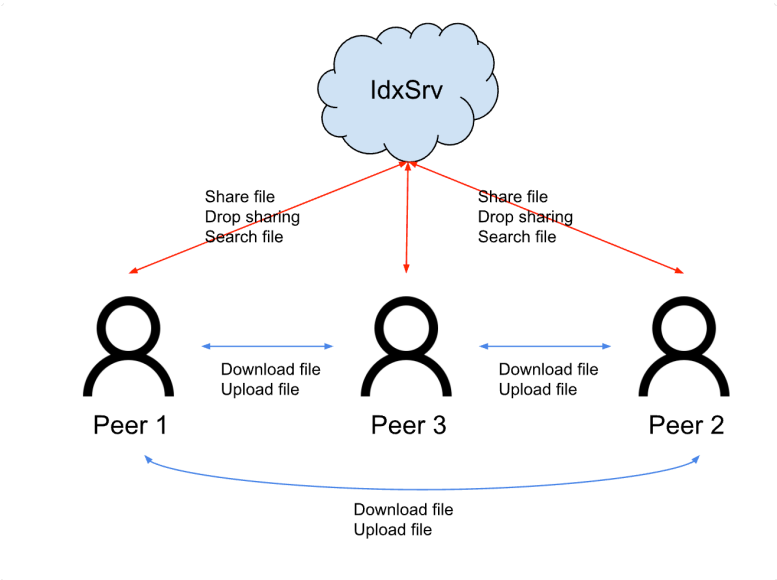


Diagram 2: IdxSrv - Peer relationship & Peer-peer relationship

Strategy for peer selection:

In this implementation, the peer(D) (downloading peer) receives a list of information from the index server, including fileName, fileSize, md5 value, sharing IP and ports etc. It is difficult for Peer(D) to decide which peers to go for as peer(D) doesn't know other peers' network conditions, current availability or propagation delay.

Therefore, to distinguish and separate target peers from the available list, before the actual downloading process, peer(D) first ping the IP address of each peer, sort the returned delay values, choose X number of peers (X = Min(1/2 number of peers, number of file blocks)) as its targets to download from. The strategy behind this is to choose peers that have a faster connection with peers (D) and exclude slow peers.

Strategy for file block selection:

In this implementation (*Diagram 3*), the fileMgr uses Random Access File type to write down a file, in which a file is divided into several blocks. During the writing down process, fileMgr takes each block's number and checks its block hash value against its fileDesc stored hash values. This way, it can perform random access to every file block. Hence, there is no need to write down a file in a certain order. To cope with this feature, the download process of this program is designed to perform multi-thread mix-order downloading, each thread takes one block from the total number of block queues whenever this thread finishes its previous task. All file blocks are temporarily saved in a data pool, a fileMgr is responsible for writing down the file to disk when the pool is full.

Downloading in the natural order could be slow if one file block got seriously delayed. For example, if one block of the file is slow in transferring (either network congestion, or package loss), the other left files have to wait until this one is clear (arrived late or resend). While in mix-order downloading, the process can skip the block that failed to download and retry later.

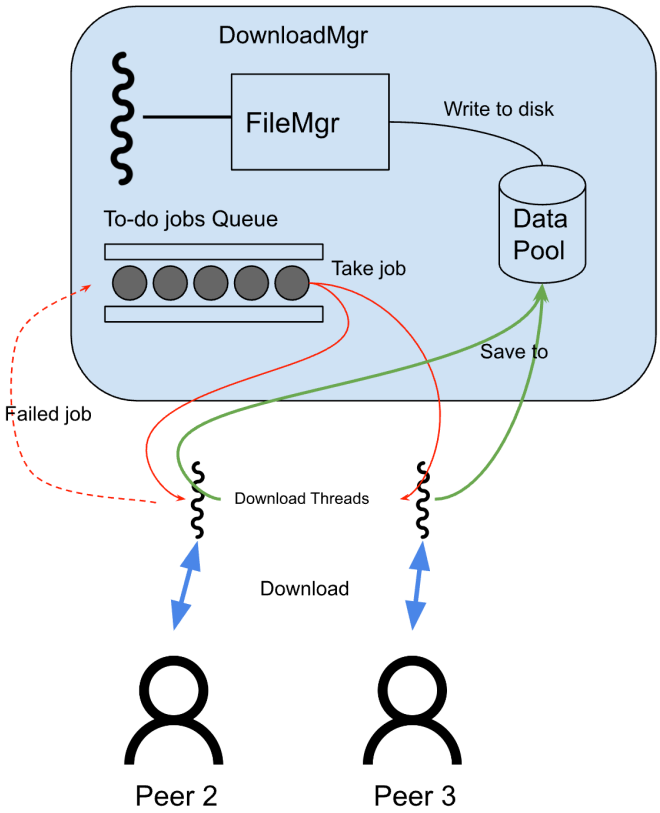


Diagram 3: multi-thread downloading process

Question 2 – Improve Protocols

Under current protocol, to achieve communication between peer and server (refer to diagram 3 above), it requires:

Peer start communication channel

Server sends welcome msg to peer

Peer acknowledge welcome msg, sends authenticateRequest msg to server

Server acknowledge authenticateRequest msg, sends authenticateReply msg to peer

Peer acknowledge authenticateReply msg, sends actual requests to server

Server receive request, process it, sends result to peers

Server stop communication channel

Diagram 4: Protocol between index server & peer

Under current protocol, to achieve communication between peer and peer (refer to diagram 3), it requires:

Peer download side = Peer(D), Peer upload side = Peer(U)

Peer(download) start communication channel

For each file-block:

Peer(D) sends blockRequest to Peer(U)

Peer(U) acknowledge blockRequest msg, sends blockReply msg to Peer(D)

Peer(D) acknowledge blockReply msg, save bytes to data pool

Peer(D) sends goodbye msg to peer(U)

Peer(U) stop communication channel

Diagram 5: Protocol between peer & peer

Reduce the overhead improvements:

1. Decrease acknowledge msg size

   For every action (share, drop & search) between index server and client file-sharer, it requires a three-step acknowledgement before actually querying, steps are namely: send welcome message, authentication request message, and authentication reply message. Therefore, if we shorten the bytes required for each message, the overhead of each query could be reduced. However, this is not backwards compatible, both parties need to update the protocol.

2. Combine welcome and authenticate msg, goodbye not in use

   In this program, the connection between two parties is based on java-socket, which uses TCP reliable connections [1]. Therefore, the step of authentication is redundant in this program. We could reduce the overhead by writing the sharer secret, into queries. If the sharer secret does not match, the other party simply denies the query. Also, the goodbye message can be removed with no side effects; one party doesn't rely on this message to check if a file has finished downloading.
   Not backwards compatible, both parties need to update the protocol.

3. Increase each blockReply msg size

   Currently the block length is set at 16 MB. A file with 160 MB needs at least 10 blocks to be transferred. We could increase the block size and reduce the number of blocks needed. In this way, we are sending fewer blockRequest messages (other than the actual data bytes). Hence, overhead is reduced.
   Not backwards compatible, both parties need to update the protocol.

4. Send all blockRequest messages without waiting for the blockReply messages

   In this program download protocol, one download thread sends a blockRequest, waits for its corresponding blockReply to arrive, and then starts to send the next blockRequest. This protocol can be improved: one thread sends all predefined number of blockRequests and processes all incoming blockReplies. In this way, the propagation delay between each sending blockRequest is removed. Hence, overhead is reduced.
   Backwards compatible, one party can change protocol while the other remains the same.

Question 3 – Socket Timeout Analysis

TCP requires a three-way handshake to establish and a four-way wave to terminate connections reliably. Without a timeout mechanism, dead connection would remain for a long time, which delays other connection requests.

This java program utilizes the timeout mechanism in the same way. ServerSockets use a TCP connection during communication with a SO_Timeout. If one side of the program does not respond accordingly, it will cause the other side of the program to wait continually and not process further codes. Therefore, we need a timeout mechanism to prevent this. A timeout mechanism can help utilize resources more efficiently and prevent wasting resources on dead links.

If the timeout value is too small, the connection will never be established or the connection becomes fragile such that minor fluctuations in delay will cause the connection to be timed out.

If the timeout value is too big, dead connections will last for quite a long time, which delays the connection request from others and causes data congestion. Also, code issues are hard to diagnose with a large timeout. It would be challenging to know the exact problem causing the long waiting time when it could be a network traffic situation or low-efficiency code lines.
According to the queueing theory, which can describe the socket model, the average time each socket spends from processing to closing is $\frac{L}{\lambda(1 - P_{\text{drop}})}$, where lambda is the mean requests per second, L is the average number of socket requests in the system, P-drop is the portion of socket requests dropped.

Theoretically, because ServerSocket is a stream-based transfer, it will generate IOException errors. When the blocking method receives an interrupt request, it may lead to an InterruptedException, and a timeout may lead to a SocketTimeoutException. these exceptions can be caught and printed out by try-catch for easy debugging. There is also the possibility of a busy port, which can be resolved using another port. A chunk hash mismatch is solved by requesting it to be sent again.

Question 4 – Testing and Limitations

The testing codes are to run a single index server on the virtual machine, and a Filesharer on the virtual machine and local machine. While taking action, the peer will send requests concurrently. Each request presents as one peer (different port number). This illustrates multiple peers querying at the same time.

Laboratory environment:

In order to estimate how many peers a single server can support, we have to consider the performance of the virtual machine. The parameters for the cloud server CPU and memory refer to *Diagram 6* below. We installed xfce4, TigerVNCServe and JConsole to monitor the performance of the cloud server and the java program.

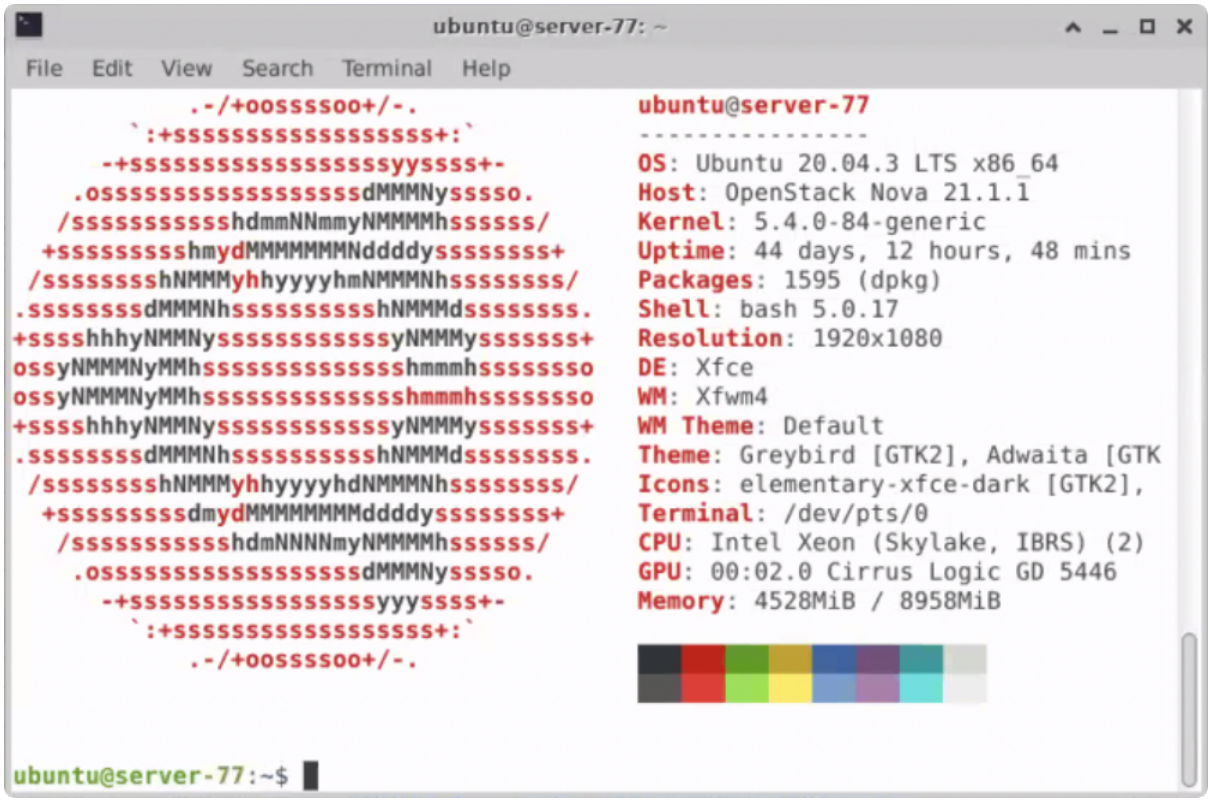


Diagram 6: System Information

Server limits in the optimal environment:

To test the server's capability in an optimal environment, we have accomplished these tests on the condition of running a single index server on the virtual machine, and running a file-sharer peer on the virtual machine as well. Test results are listed below:

File size: 55 KB (typical torrent file size)

	Sharing		Searching		Dropping		Downloading	
Repeat	2,000	20,000	2,000	20,000	2,000	20,000	100	1000
Times	9.8 s	101.2 s	6.2 s	63.5 s	1.4 s	15.3 s	121.2	1232.6

	Sharing		Searching		Dropping		Downloading	
	Index Server	**File sharer**	**Index Server**	**File sharer**	**Index Server**	**File sharer**	**Index Server**	**File sharer**
CPU	21%	35%	42%	5%	22%	18%	17%	35%
Memory	50 Mb	250 Mb	60 Mb	150 Mb	80 Mb	300 Mb	450 Mb	1 Gb

Details refer to Appendix A & B

Based on the above testing, we estimate the QPS (query per second) for this cloud server is approximately 200 for sharing, 300 for searching, 1400 for dropping, 0.8 for downloading.

This system is scalable in this environment, we could have the server responds to more requests if we increase the CPU capability, or add a load balancer to share the loads to more threads.

Server limits in the normal environment

To test the server's capability in an average-user-like environment, we have accomplished these tests on the condition of running a single index server on the virtual machine and running a file-sharer peer on a local machine. Test results are listed below:

	Sharing		Searching		Dropping	
Repeat	200	2,000	200	2,000	200	2,000
Times	40.3 s	392.5 s	37.2 s	384.5 s	35.3 s	364.5 s

	Sharing	Searching	Dropping
	Index Server	**Index Server**	**Index Server**
CPU	1%	0.5%	0.5%
Memory	40 Mb	55 Mb	50 Mb

Details refer to Appendix A & B

Based on above testing, we estimate the QPS (query per second) for in this situation is approximately 5 for sharing, searching, and dropping.

Comparing to the optimal situation, we notice that the CPU load is lower, and the QPS is lower too. This leads to the conclusion that the network condition is the main threshold of this system.

This system is not scalable in this situation. Even if we add more computing power to the server, the system won't process any faster. The system can only perform better when user the has a lower delay to the server.

Limits of a single server:

Based on our testing, the server processes peer's requests in order of the requests' arrival. Therefore, if the waiting time (from the standing to get a reply) for one user to send one quest when the server is idle is $\boldsymbol{T}$, then the average waiting time for $\boldsymbol{n}$ number of users send one request at the same time is:

$$n * T$$

We assume that users expect waiting time to be 30 seconds. Based on the above formula and established QPS value of 5, the server could then handle 150 queries at a given time.

The system could grow until either users number at one time is far exceeding system capacity, that causes significant delay and become unusable, or storage reaches the limit. In this 4 GB available memory environment, we estimate it could store:

$$4\,GB * 1024\,MB * 8\,bits\,/\,40\,Mb * 2000\,records = \mathbf{1.6\,million\,records\,for\,sharing}$$

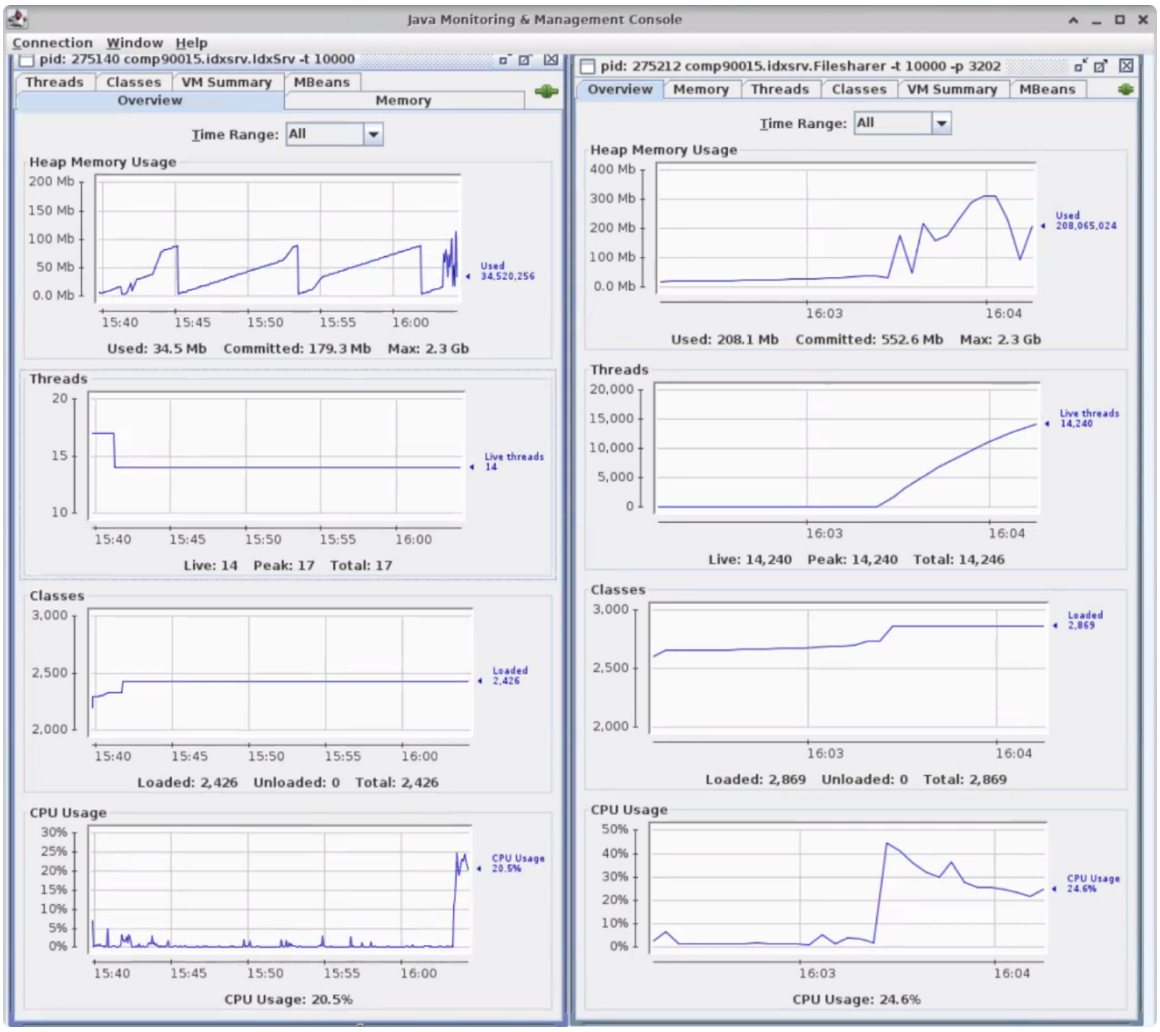
or

$$4\,GB * 1024\,MB * 8\,bits\,/\,55\,Mb * 2000\,records = \mathbf{1.2\,million\,records\,for\,searching}$$

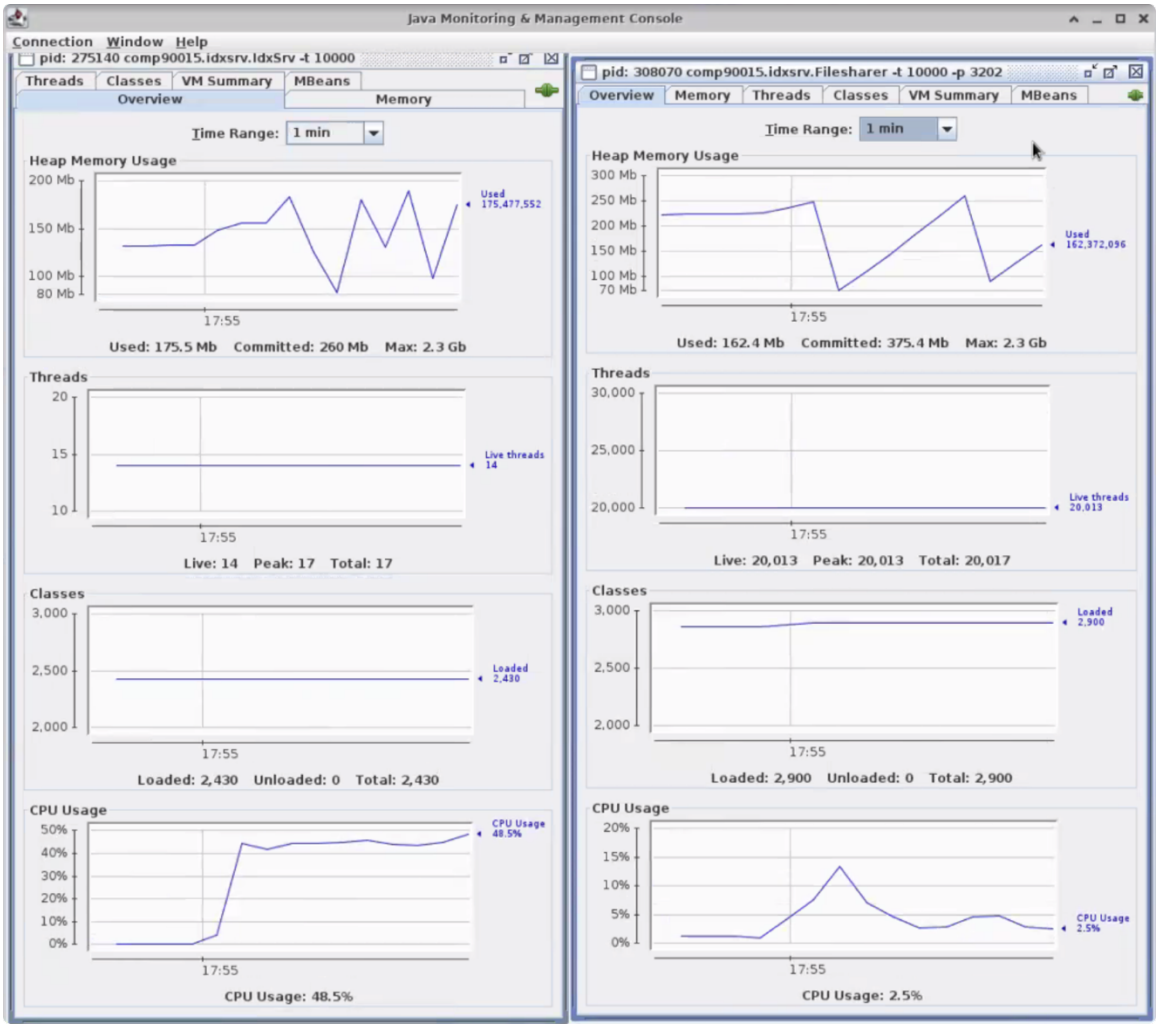
Reference

[1]"Socket (Java Platform SE 7)", *Docs.oracle.com*, 2022. [Online]. Available: https://docs.oracle.com/javase/7/docs/api/java/net/Socket.html. [Accessed: 16- Sep- 2022]

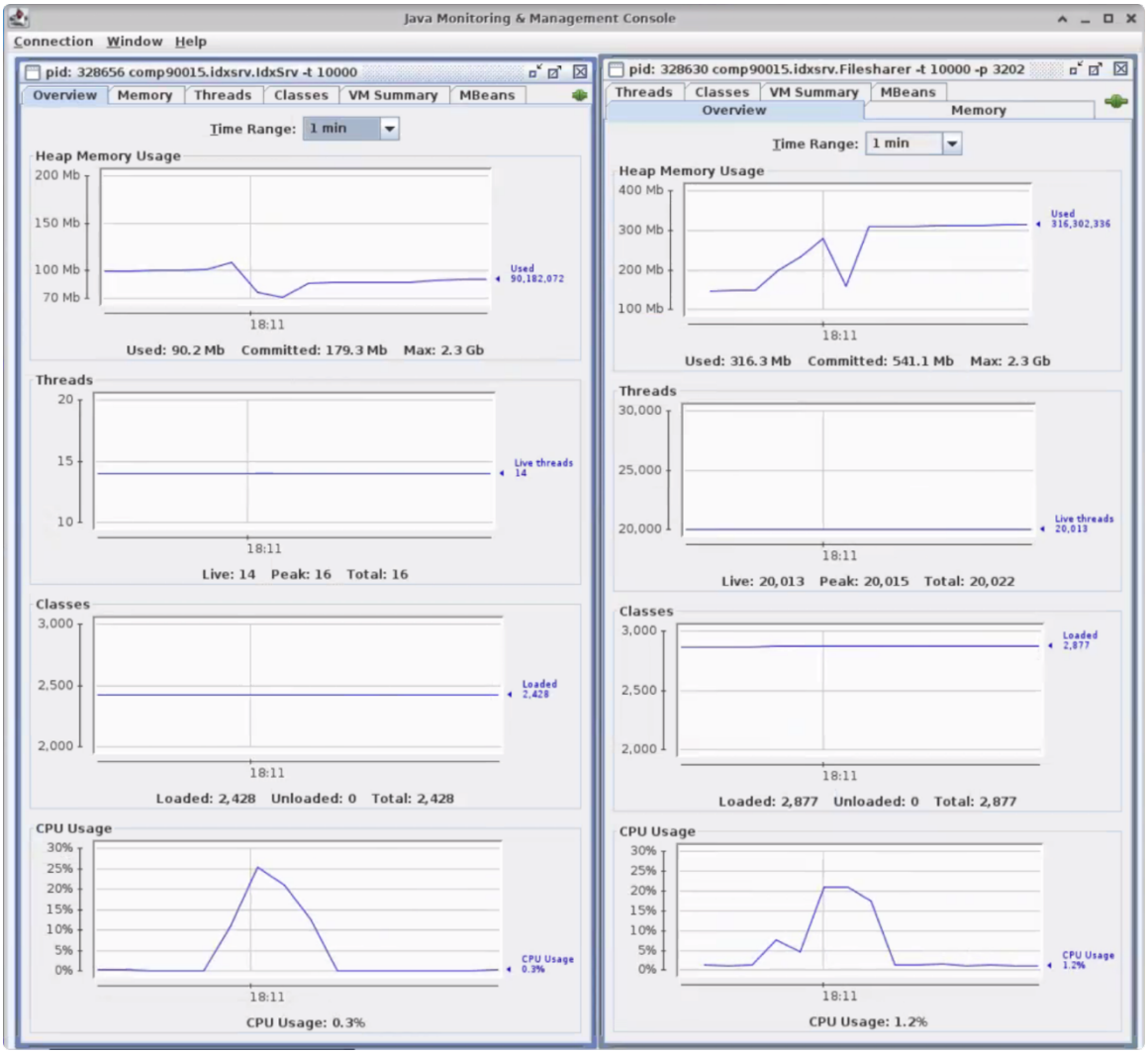
Appendix A - JConsole Screenshots



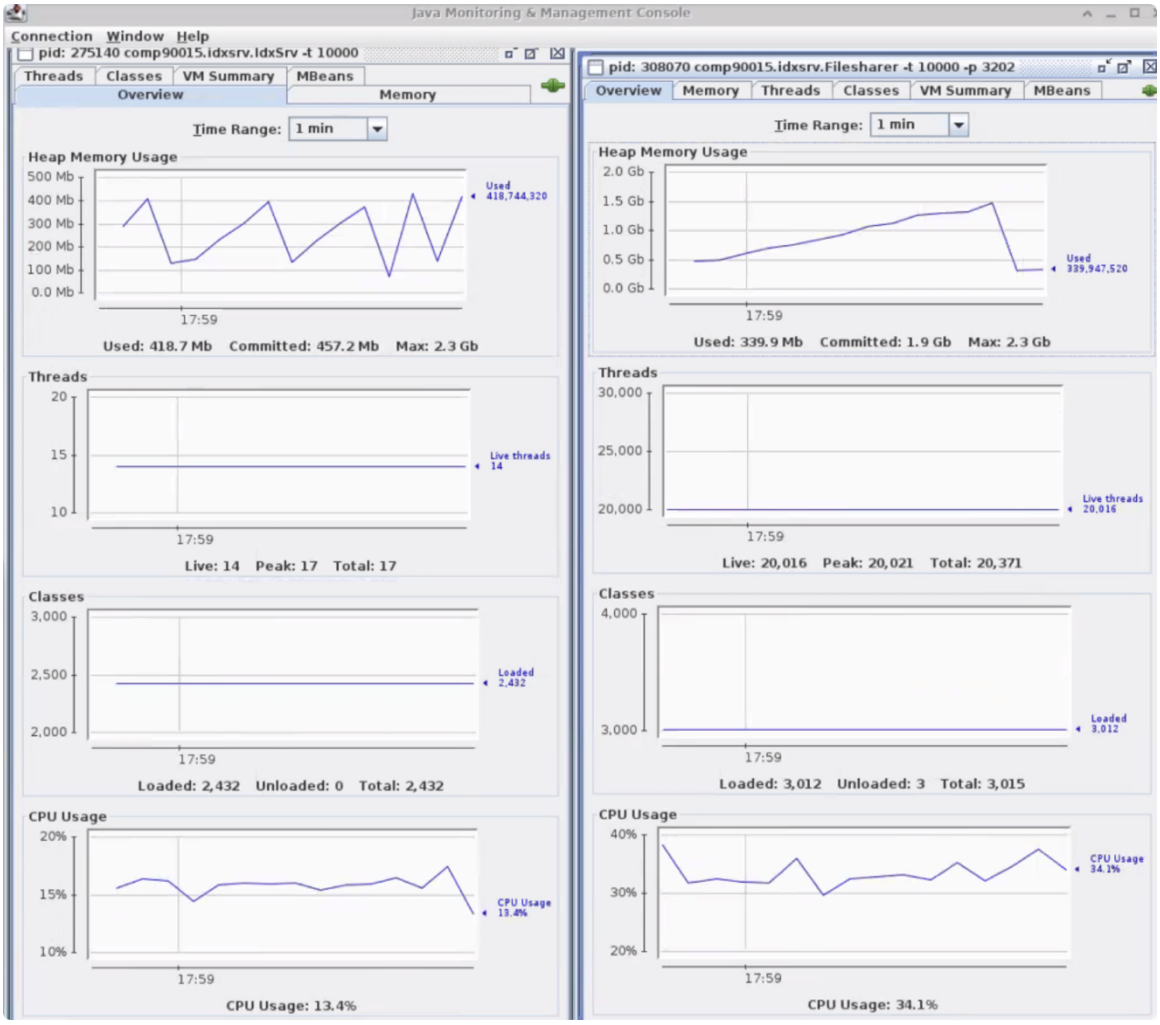
JConsole for Sharing Requests



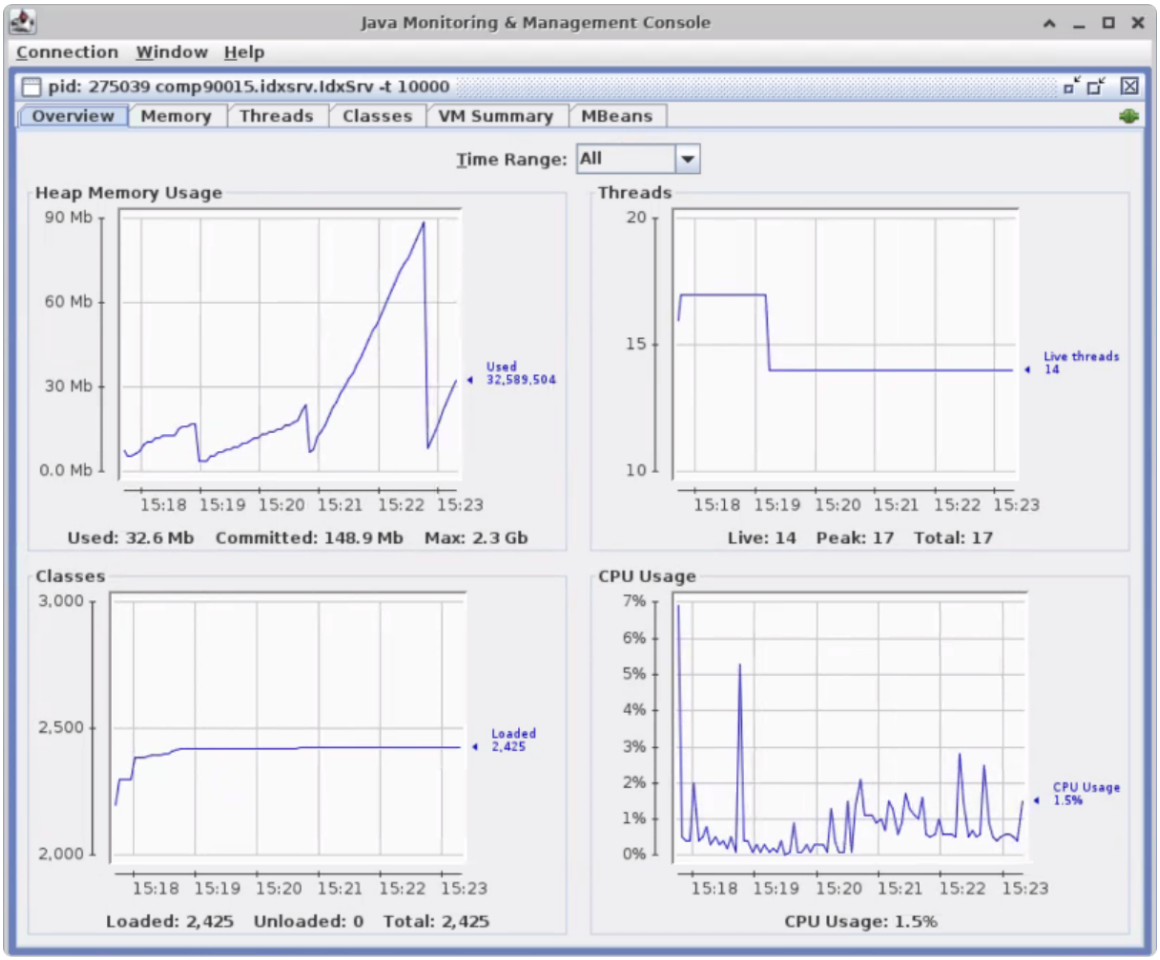
JConsole for Searching Requests



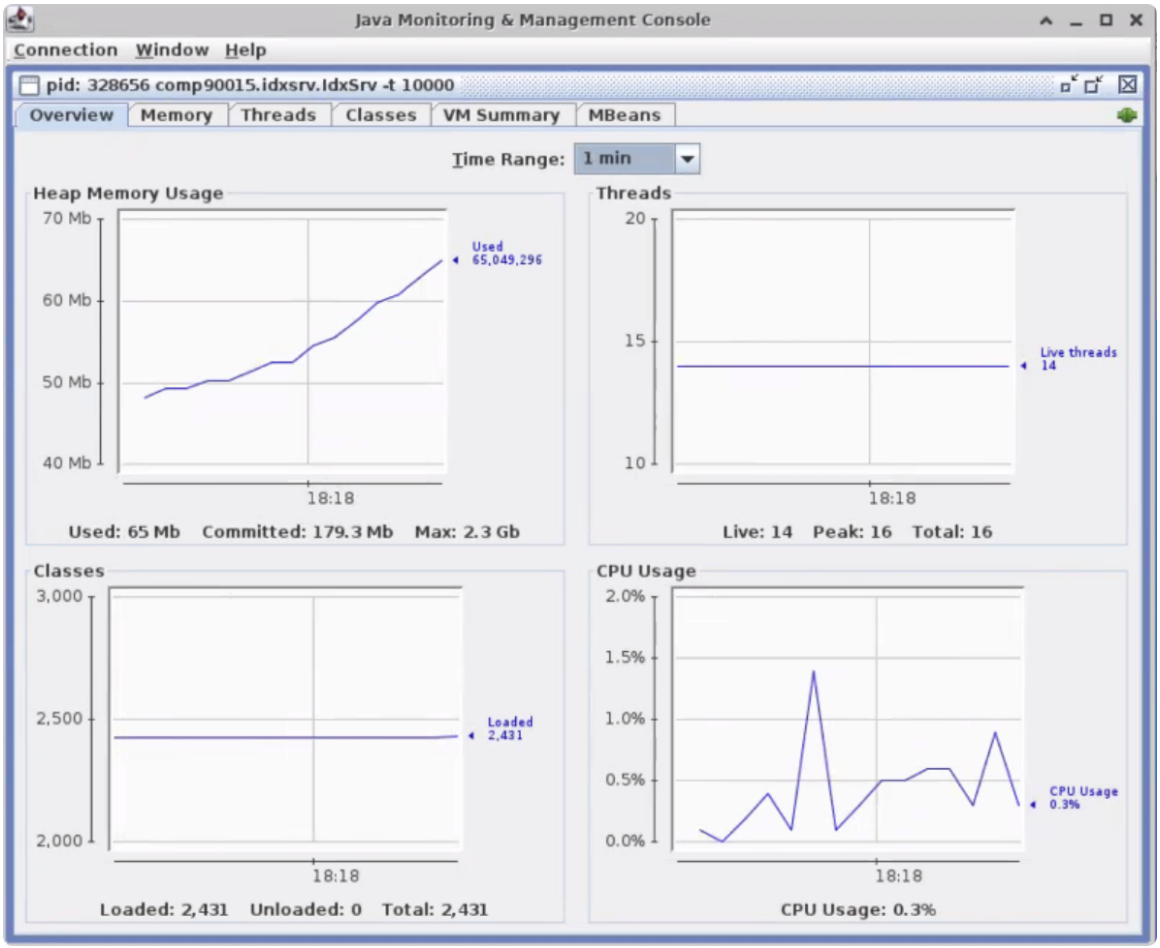
JConsole for Dropping Requests



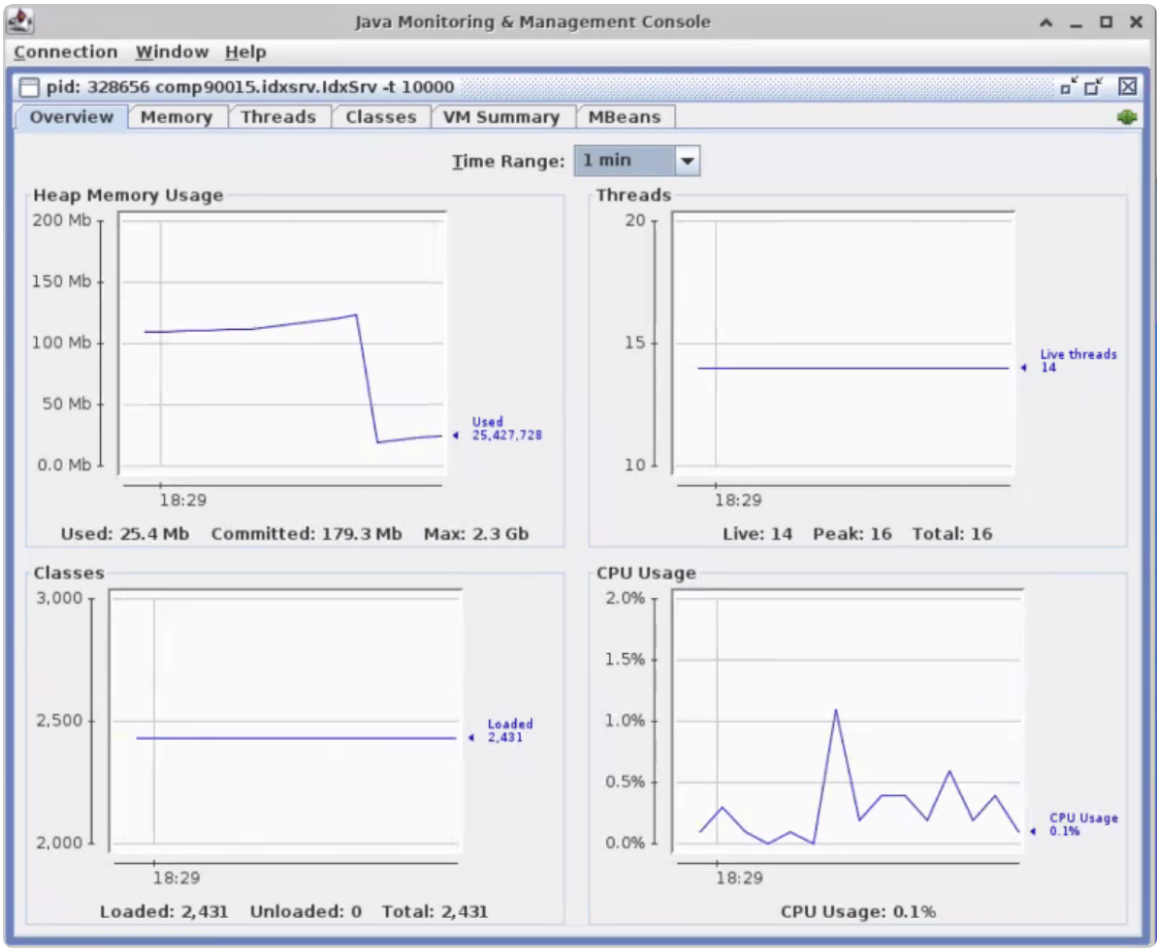
JConsole for Downloading Requests



JConsole for Sharing Requests from Local Machine

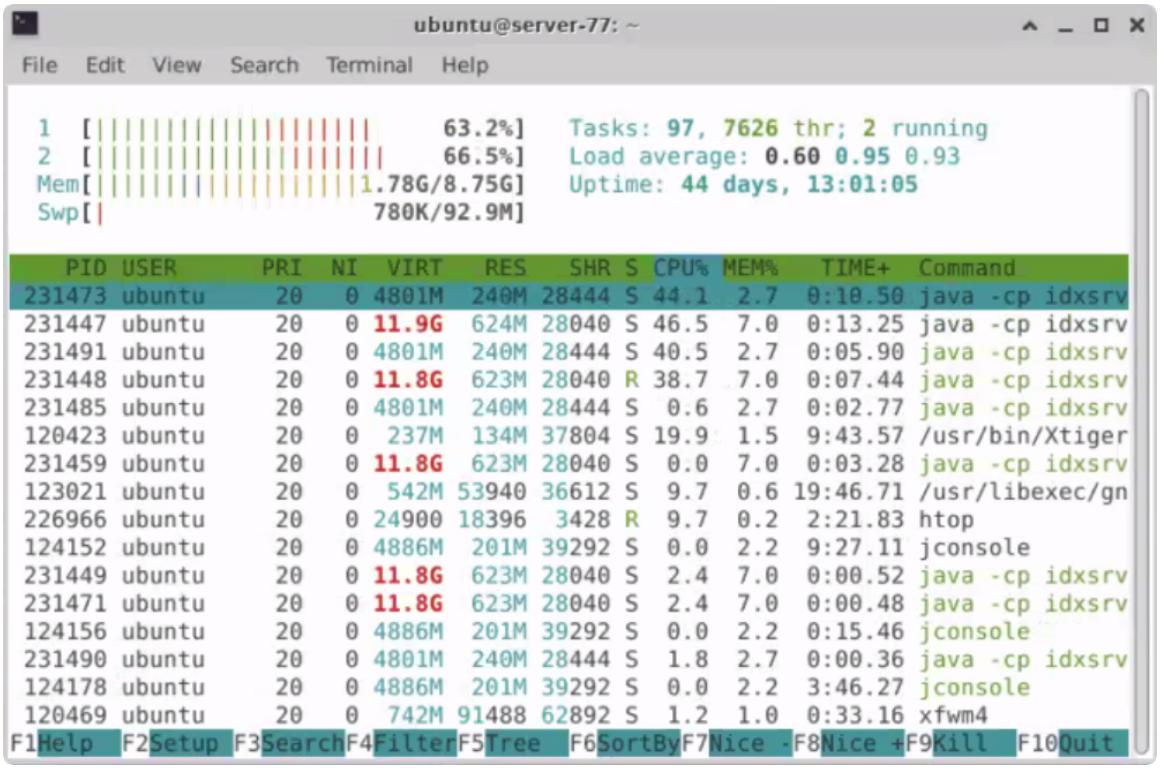


JConsole for Searching Requests from Local Machine

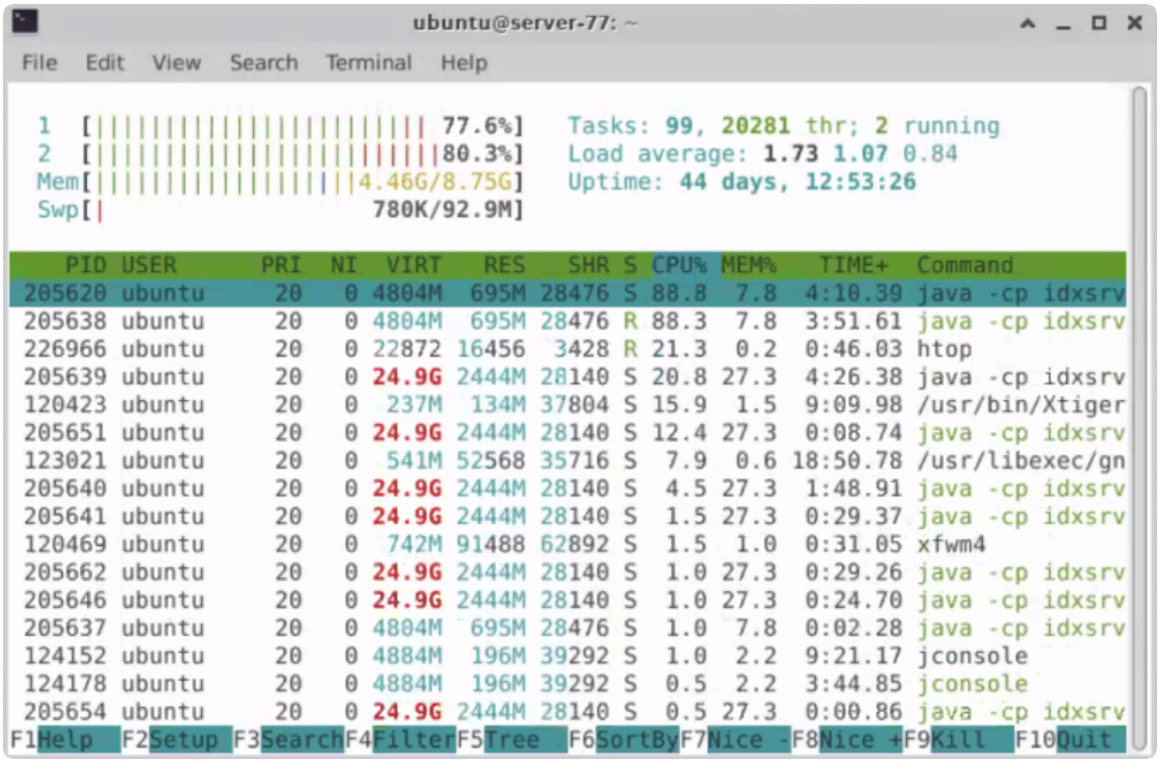


JConsole for Dropping Requests from Local Machine

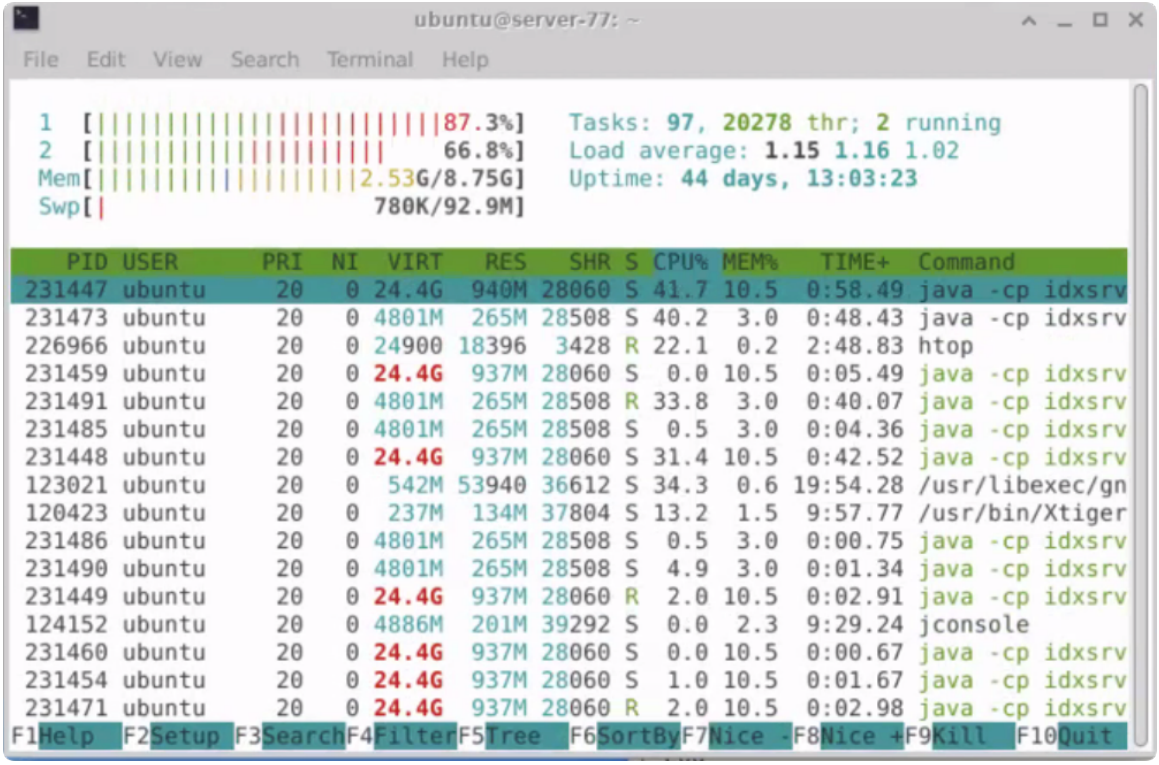
Appendix B - Htop Screenshots



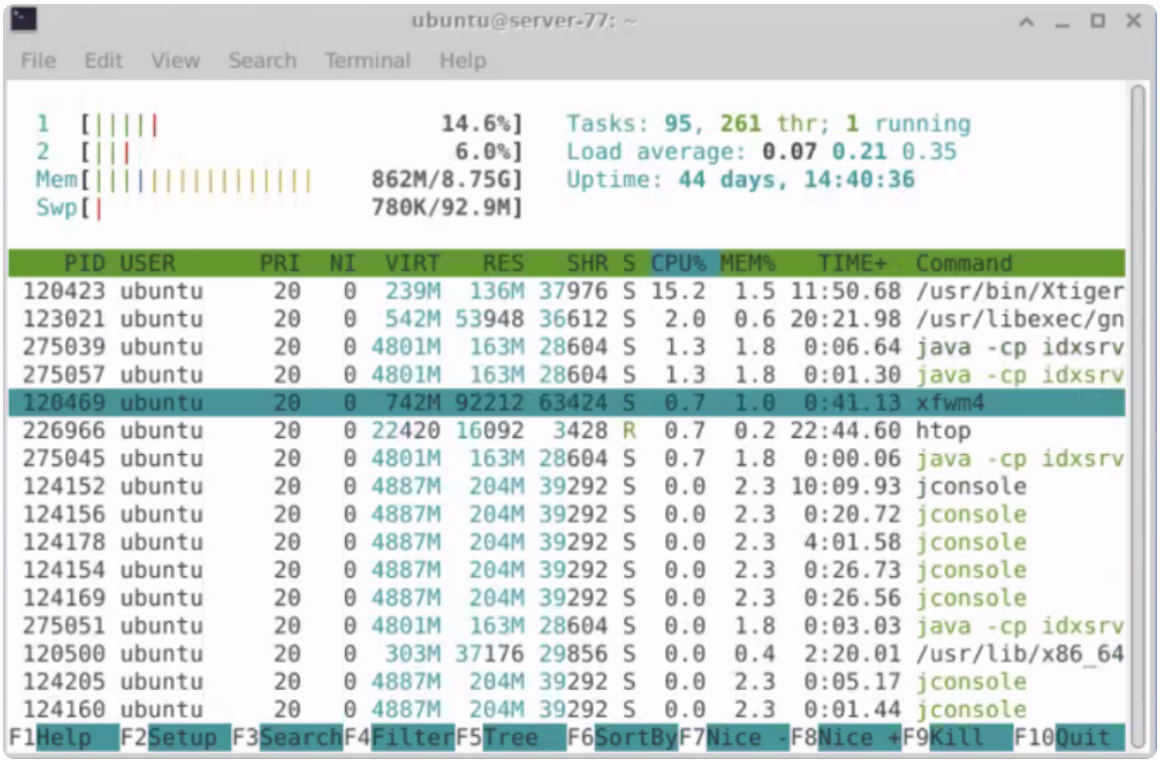
CPU Information for Share Requests



CPU Information for Search Request



CPU Information for Dropping Requests



CPU Information for Sharing Requests from Local Machine