**File Nexus**

Optimized Multi-file Uploads with Task Balancer

Owners: Dung Ha, Katie Le

**Introduction**

Task balancers are essential tools in various fields, from software development to manufacturing. They distribute workloads efficiently across resources, ensuring optimal performance and resource utilization. Whether it's balancing computational tasks in distributed systems, distributing requests in cloud computing, or allocating tasks in manufacturing, task balancers play a critical role in enhancing productivity and system performance.

File Nexus is a client-to-server file storage system that employs a task balancer using the TCP protocol. This architecture allows for efficient handling of multiple file uploads simultaneously from multiple clients.

In this system, a multi-threaded client uploads multiple files to a storage server. Before the files are transferred, a task balancer (non-multi-threaded) analyzes the server's load and directs the file to the appropriate worker server, ensuring load balancing and minimizing queuing time. The worker server stores a copy of each uploaded file in the local system. The storage server, which is multi-threaded, sends requests to the worker server to upload files in a round-robin manner. This project facilitates simultaneous file uploads from multiple clients, with support for at least 2 clients concurrently.

**System Architecture**

A diagram of a work server

Description automatically generated

The idea of this system is to route the files uploaded by clients to a centralized storage in a scalable way using TCP networking protocol. The Task Balancer and the Worker Servers act as intermediaries which help spread the workload in the uptime of the network by receiving and storing files from the clients in a distributed manner and continuously uploading those files to the Storage Server. Hence, in the uptime, the clients do not have to wait in a long queue to upload files, and in the downtime, the Worker Servers can focus their resources on transferring the files to the final storage destination. The system can be divided into four main components: the Clients, the Task Balancer, the Worker Server, and the Storage Server. The flow of a file is as follows. When a client wants to send a file to the storage system, it first asks the Task Balancer for the address of the appropriate Worker Server. It then sends the file to the Worker Server. The Worker Server then waits to send the file to the final destination, the Storage Server. All communications among components are deployed using TCP.

Components and messages:

1. Main classes:
   * **Clients (MultiThreadedClient.java):** These multi-threaded clients utilize FileSend (implements Runnable) to either send a “file sending request” to the Task Balancer or a “client transfer request” to the Worker Servers. The former requests a Worker Server’s address, while the latter sends a file to the Worker Server.
   * **Task Balancer (TaskBalancer.java):** This server informs the client of which Worker Server to send a file to. Currently, it assigns Worker Servers’ addresses in a round-robin manner.
   * **Worker Server (WorkerServer1.java/WorkerServer2.java):** These multi-threaded servers handle the requests of either storing a file sent by the client (“client transfer request”) or uploading a file to the Storage Server (“storage transfer request”).
   * **Storage Server (StorageServer.java):** Multi-threaded server responsible for sending “storage transfer request” to the Worker Servers and received and store files uploaded from the Worker Servers.
2. Helper classes
   * **FileSend.java:** Client’s helper class which sends a “file sending request” to the Task Balancer and receives a Worker Server’s address. It then transfers a file to the Worker Server with the address just obtained using the “file transfer request”.
   * **FileTransferRequest.java:** Worker Servers’ helper class which either handles “file transfer request” (store the file sent by the client) or “storage transfer request” (upload files to the Storage Server)
   * **FileReceive.java:** Storage Server’s helper class which sends a “storage transfer request” to a Worker Server so that files can be uploaded to the final storage destination.
   * All (helper) classes implement Runnable. All messages and files between hosts are sent and received with TCP.

**System Implementation**

1. Programming language: Java
2. APIs:

* BufferReader:
  + Read Worker Server’s address obtained from Task Balancer (FileSend) and file transfer status (FileReceive)
  + Read client’s request (TaskBalancer)
  + Distinguish requests from clients and Storage Server (FileTransferRequest)
* DataInputStream:
  + Read client’s file content (FileTransferRequest, FileReceive)
* DataOutputStream:
  + Send Worker Server’s address to clients (TaskBalancer)
  + Send client’s files to Worker Server (FileSend)
  + Send responses to clients if file is received (FileTransferRequest)
  + Sends “storage transfer request” to Worker Server (FileReceive)
* FileInputStream
  + Read clients’ files locally (FileSend, FileTransferRequest)
* FileOutPutStream
  + Save files locally through TCP (FileTransferRequest, FileReceive)
* Socket
  + Establish TCP connections between multiple components (all classes)
* StringTokenizer
  + Extract values like Worker Server’s address, file name, file path, and client’s ID from servers’ responses (FileSend, FileTransferRequest, FileReceive)
* File
  + Create file objects for storing local copies (FileSend, FileTransferRequest, FileReceive)

1. Platform

* Currently, hosts (servers and clients) use localhost but with different sockets.

1. How to execute this application:

* Open the source code folder **/fileNexus** in a code editor (we use VS Code editor for this project). Download the Java Extension for the VS Code. Then the “Run” button will appear to run the classes. Otherwise compile each class by entering the bash command “javac <SourceFileName>.java”, and run each class by using the bash command “java <SourceFileName>.java”.
* Run each class in the following order:
  + - StorageServer.java
    - WorkerServer1.java and WorkerServer2.java
    - TaskBalancer.java
    - MultiThreadedClient.java and MultiThreadedClient2.java
* Monitor the terminal for client and server activity.
* You will see the following newly created folders:

1. /storage: stores copies of files successfully uploaded by multiple clients, organizing them into folders based on each client’s ID
2. /worker1: stores copies of files successfully transferred from clients assigned to Worker Server 1, which will then be used for uploading to the storage server.
3. /worker2: stores copies of files successfully transferred from clients assigned to Worker Server 2, which will then be used for uploading to the storage server.
4. Testing

Suppose you want to test the server's support for more than 2 clients, simply follow these steps:

1. Copy the template of **MultiThreadedClient.java.**
2. Modify the **username** field in the new client to assign it a unique ID.
3. Add files you want to upload to the **/clientFiles** folder.
4. Specify these file names inside the **String files[]** field within MultiThreadedClient.java.

Here is an example of adding a third client to the system:

* Add files file1.txt, file2.mp3, and file3.png to the /clientFiles folder.
* Modify the following fields:

public class MultiThreadedClient3 {

public static void main(String argv[]) throws Exception {

...

// Client’s username

String username = "Client\_3";

// Files to be uploaded

String[] files = { "file1.txt", "file2.mp3", "file3.png"};

...

**Extra Feature – Local State Storing of File Transfer**

1. Description

We've introduced a new feature to enhance the resilience of Worker Servers in managing file transmissions. This feature enables the Worker Servers to maintain the status of file transfers even in the face of network disruptions or server disconnects.

For context, a hash map is currently used by the Worker Servers to contain task IDs and file names of files to be transfered to the Storage Server. Also, an array is used by the Worker Servers to record the state of what files are available to be sent to the Storage Server. This new feature extracts information from this existing hash map and array to allow the servers to preserve the state of file transmission status. This state information is stored in two text files:

1. worker<no.>stateMap.txt: The state map file containing the task ID and file name



2. worker<no.>state.txt: a transfer status file for the transfer status (1 for available, 0 for not available). Notice that index 0 is not used to record files availability status (task ID starts at 1), so we use it to record task with the highest task ID that made the change to this file.



Each time the Worker Server successfully receives a new file from a client, it updates the <ID,name> map and toggles the value (currently 0) of the appropriate index the status array. Consequently, the corresponding entry is removed from the state map and the value (currently 1) of the appropriate index is toggled in the state array once it has been transferred to the Storage Server. Each request processed is followed by an update to the state array file and the map file (except for when the Worker Servers has no file to send to the Storage server).

Should a Worker Server experience downtime followed by recovery, it consults these text files (loads the information to the file status array and file <ID, name> map to resume transferring any remaining files with a transfer status of 1. This process dramatically increases the reliability of the whole storage system.

1. How to test the local state storing feature

Note: Please refer first to the steps outlined in “How to execute this application” under the “System Implementation” section above for instructions on how to run each class below, either from an editor or a terminal.

1. Initiate the TaskBalancer.
2. Start WorkerServer1 and WorkerServer2.
3. Run MultiThreadedClient instances.
4. Monitor state files in the /worker1\_state and /worker2\_state directories once both Worker Servers finish processing all tasks from clients.
5. Launch the Storage Server.
6. Wait 6 seconds, then halt both Worker Servers to simulate disconnection.
7. Review the state files and the files received by the Storage Server to identify pending tasks that weren't transferred due to disconnection.
8. Restart the Worker Servers. Wait for some time (15 seconds), and check the state files, where all pending tasks should have been cleared. Also, check if the Storage Server has received all the files.

**Conclusion**

The benefits of our system are mostly in theory. This is because we have not been able to test the system using different machines. However, we assume that if each host resides in different machines, the system will minimize sending time from many clients with large amounts of data since there are many servers handling their requests instead of just one. Also, the system does not have any known errors as files can be sent and received successfully throughout the system. In the future, we hope to add an authentication feature, so that clients need to be authenticated with a pair of passwords and usernames to upload files to the system.

In addition to the proposed future authentication feature, ensuring consistency across both the worker server and storage server is crucial for maintaining the integrity of data transactions within the system. With the added extra feature of locally storing file transfers on the Worker Servers, this system can guarantee reliable and accurate data storage and retrieval operation, even in instances of disconnection from either the Storage Server or Worker Servers.