Event Driven Simulator for a network cache system

Rahul Vemula[1], Lahari Barad[2]
CIS6930 - Probability for Computer Science and Machine Learning
Computer Science, University of Florida
Gainesville, Florida
rahulvemula@ufl.edu
lahari.barad@ufl.edu

Abstract: An event driven simulator is created to compare the effectiveness of different cache implementations. LRU (Least Recently Used), FIFO (First In First Out) and LF (Largest First) cache eviction policy implementations are compared. In every simulation, new file request events are generated from poisson sample. Popularities and sizes of files are generated using Pareto distribution from the provided inputs.

***Index Terms*—LRU, FIFO, Largest First, Probability, Event Simulator, Cache implmentation**

I. Introduction

CACHING is a technique that stores a copy of a given resource and serves it back when requested. Web caches reduce latency and network traffic and thus reduce the time needed to display resource representations. Different cache-replacement policies are implemented as it is possible for a user to request any random file. A few are developed and tested to predict what files will be requested next by the client. We developed a simulator in JavaScript(Node) to compare LRU, FIFO and LargestFirst cache replacement policies.

The simulator created represents a remote server from which files are requested. This simulator is implemented using a combination of functional and object-oriented programming. Our simulator uses Pareto distribution to generate file sizes and file popularities and Poisson sample to generate new file request event times.

II. Simulator

A. Event Classes

The event class is an abstract parent class which is extended by four other classes: NewRequestEvent class, FileReceivedEvent class, ArrivedAtQueueEvent class, DepartAtQueueEvent class

1) NewRequestEvent class

This event corresponds to a new user request for a file. The process function of this class checks if the cache has this file, and generates a new fileReceived event if the file is present in cache, else creates a arrivedAtQueue event with the requested file id.

2) FileReceivedEvent class

This event is to see if a file has been received by the user.The process function of this event calculates and records response time for a particular file.

3) ArrivedAtQueueEvent class

This event corresponds to file in FIFO queue. The process function of this class checks if the queue is empty and generates a new depart-queue-event. If the queue is not empty, adds the file to the end of the queue.

4) DepartQueueEvent class

This event represents that access link has finished the transmission of file. Once the file is processed, it will trigger FileReceived event.

B. Cache Classes

To implement different cache replacement policies, FIFO, LRU and LF, we created three different classes. Each class has three functions

1) size

This function returns the sum of sizes of all the files in the cache.

2) get

This function fetches and returns the requested file from cache, it returns null if the requested file doesn't exist in the cache.

3) add

This function adds the given file to the cache if it doesn't exist already. It also checks if the cache files are exceeding the capacity and removes the files from cache according to the eviction policy and inserts the new file entry.

C. Queue class

The queue class is an implementation of a queue data structure, which stores the files using a linked list. This implements three functions, enqueue, dequeue and isEmpty. enqueue adds the given value to the end of the queue, dequeue removes the element from the front of the queue and isEmpty returns if the queue is empty.

D. Priority Queue class

The priority queue class is an implementation of priority queue datastructure based on heaps where each element has a priority assigned to it and all the elements in the queue are arranged in the order of the calculated priority. This class also implements an enqueue function, which adds the element to the priority queue and dequeue function, which removes and returns an element with highest priority in the queue and isEmpty function which returns if the priority queue is empty.

E. Files module

This has a fileStore object which stores the array of all the files with fileId, size and probability and cumulative weights array of all the files. The function setupFiles populates this fileStore object according to the generated pareto distributions based on the parameters provided in the config. The function getSampleFile picks a random file based on cumulative weights, which is passed on to newRequestEvent. The function mean calculates the mean of sizes of all the files. The function sum calculates the sum of sizes of all the files.

F. Other modules

index.js module in the entry point or main function of the simulation. utils module has the utility functions used in various modules. stats module stores all the statistics collected as part of the simulation and provides a function to export the statistics to a csv file.

G. Input Parameters

The following is the list of input parameters which can be changed by user via config.json file and is provided to other modules using config.js: totalRequests, numFiles, timeLimit, requestRate, networkBandwidth, accessLinkBandwidth, roundTrip, paretoAlpha, cacheSize, cacheType, paretoSeed

III. Cache Replacement Policies

As discussed above, the simulator implements three cache replacement policies - LRU, FIFO and LF

A. LRU

LRU cache policy stands for Least Recently Used, i.e., the file which has been accessed least recently will be removed. The files entering the cache are stored in an ordered map, with file id as key and file itself as value. Every time a file requested is found in cache, it is removed from its index and added to the end of the map, so, eventually least recently used files are put in the front of the map. When the cache reaches its capacity, files at the start of the map are removed.

B. FIFO

FIFO cache policy stands for First In First Out and files are stored in queue. When a new file is added to the cache, it is added to the end of the queue. When the cache reaches its capacity, the oldest file added, which is at the start of the queue is removed and new one is added to the back of the queue.

C. Largest First

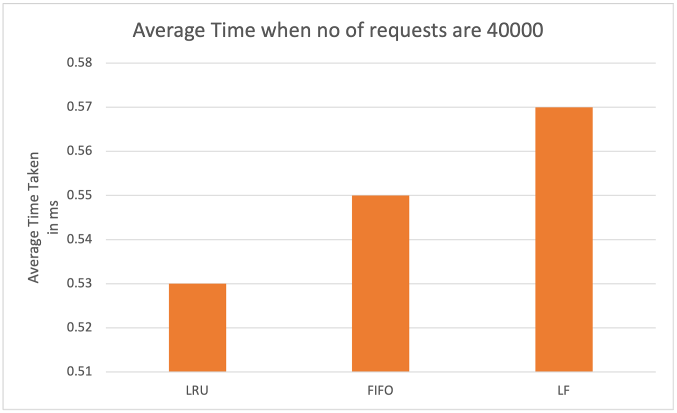
In largest first cache policy, the largest file would be removed. Similar to LRU, The files entering the cache are stored in a map, with file id as key and file itself as value. If cache reaches its capacity, largest file i.e the file with highest size is found and removed.

IV. Results

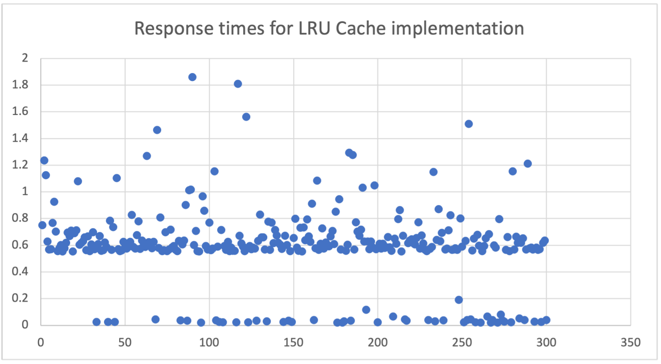
The effectiveness of the LRU, FIFO and Largest First cache replacement policies are compared by changing various input parameters.

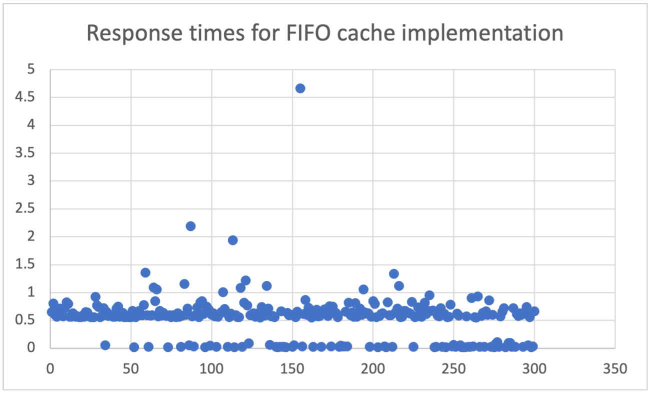
A. LRU vs FIFO vs Largest First

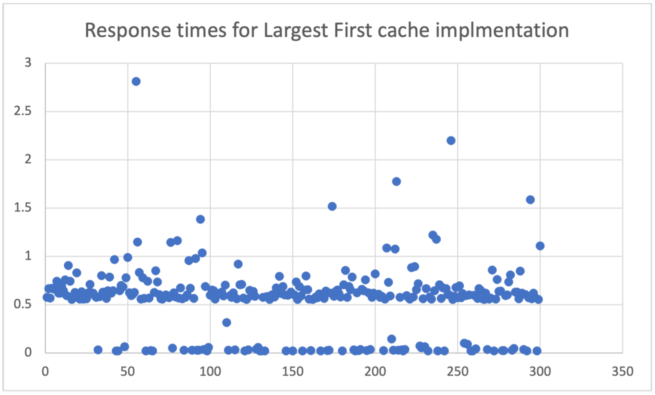
In general, LRU cache performed the best, followed by FIFO cache, followed by Largest First. Largest First and FIFO cache's performance was similar while LRU cache's performance was noticeably better. The simulator is run for 40000 number of requests and for each request, the response time is recorded to a csv file using file system. The average of these times are calculated for each cache, which are: LRU: 530 ms, Largest First: 570ms and FIFO: 550ms.



The recorded response times for each file request are plotted for different cache replacement policies

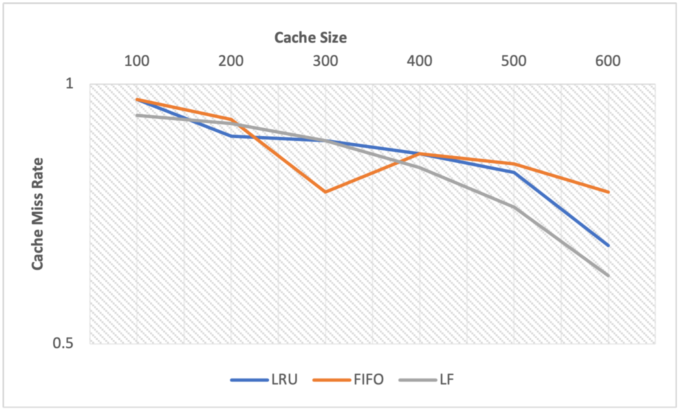






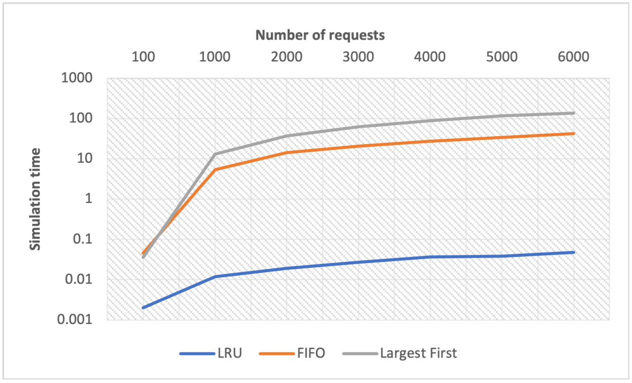
B. Cache size

The cache miss rate for all types of caches is recorded. As expected, It is noticed that as the cache size increases, the miss rate decreases.



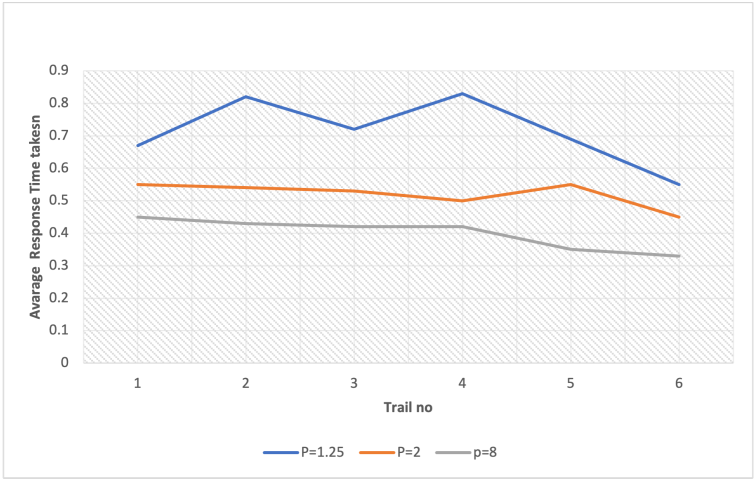
C. Number of Requests

The simulator is run with different number of requests and time taken for simulation is recorded. It is noticed that as the number of requests increases, simulation time increases and as mentioned LRU has the least simulation time,followed by FIFO , which is followed by Largest First



D. Pareto Shape

It is noticed that Pareto shape has a huge impact on average response time. When Pareto shape value is less, the Pareto distribution will have a large tail. In that case, there would be a few really big and a few really popular files. When Pareto shape value is large, all files would have similar size and popularity, thus making it difficult for cache to guess which file would be requested next. Due to this, the impact of Pareto distribution of time taken is more than the cache replacement policies



V. Conclusion

From the above results, it can be inferred that the system specifications like cache size, number of requests and file size have more effect on average response time than the cache replacement policy used in the simulator. A few input parameters, like Pareto shape have huge impact on average response time. With all input parameters kept constant, we found that LRU was the best performing algorithm, Largest First being the second best and FIFO being the worst.This might change depending on the input parameter like pareto shape. Depending on the above mentioned results, it can be concluded that the best way to decrease average response time would be to increase network system infrastructure. In the event of lack of resources to do that, improving cache-replacement policies would be the next best option.