Ahmedabad
University

Parallel and Distributed Systems

CSE524

Section-1

Submitted to: Srikrishnan Divakaran

Enrollment Number	Name of the students
AU1940206	Maulikkumar Bhalani
AU1940183	Mithil Parmar
AU1940028	Moksh Doshi

Problem:

Migration for servicing requests in a Distributed Web Server:
You are given a distributed system consisting of N servers S1, S2, ..., SN executing programs P1,P2, ..., PN respectively and are connected by a network whose topology is defined by a weighted graph G=(V, E, w), where V are the set of servers and E is the set of edges connecting these servers and w is the set of distances corresponding to the edges in E. Each server Si stores a collection of {Pi1, Pi2, ..., PiD} D pages each. Each program Pi makes a sequence of n requests ri1, ri2, ..., rin to one of the pages in the N servers. Now if the requested page is available locally (memory or local storage) then you can serve it immediately elsYou are required to develop a framework that provides the necessary mechanisms for distributed servicing of requests by employing page replication/migration coupled with caching in distributed systems you can serve the request by either moving the requested page from a remote server locally by caching it or permanently migrate the requested page. If you are caching it or migrating the requested page then you will have to replace one of the pages in the local cache.

You are required to develop a framework that provides the necessary mechanisms for distributed servicing of requests by employing page replication/migration coupled with caching in distributed systems. Then using the mechanism provided by this framework design and develop algorithms/policies for optimal page replication/migration and caching.

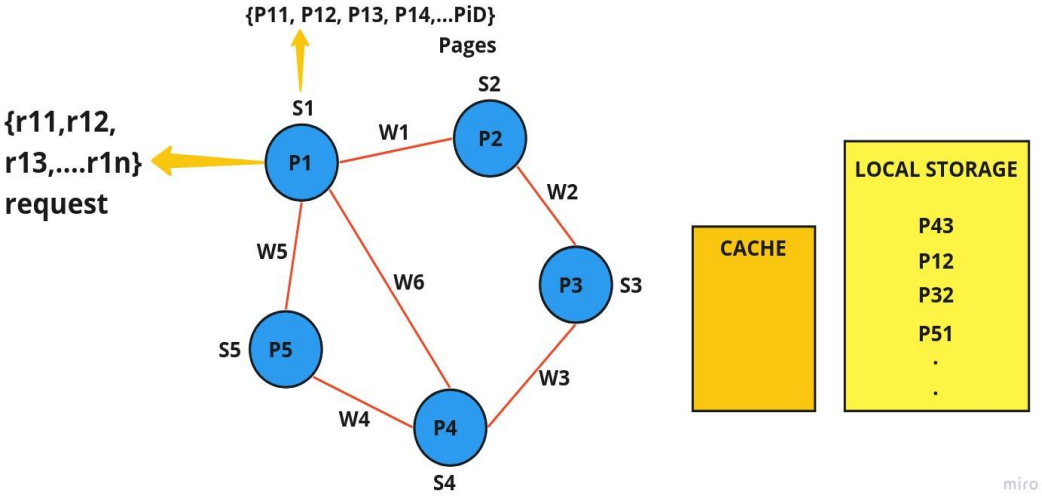
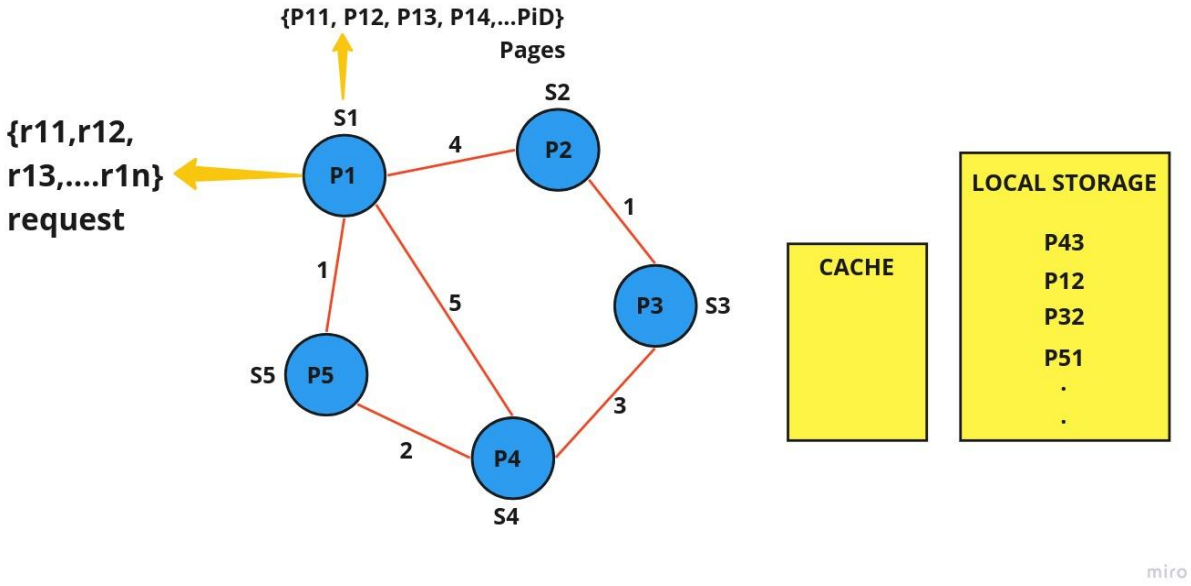


Fig - Distributed web server



Suppose we take weights of edges as w1=4, w2 =1, w3=3, w4=2, w5=1, and w6=5.

Assume that the program P1 is executing from server 1. A distributed system consisting of N servers S1, S2, ..., SN executing programs P1, P2, ..., PN respectively on each of the servers. Suppose that server one is executing program

1. Each server S_i stores a collection of {Pi1, Pi2, ..., PiD} D pages each. Each program Pi makes a sequence of n requests (ri1, ri2, ..., rin) to one of the pages to access the data from any servers. Suppose server one executing program one, and it will request the page P43, which will be found on server four at index 3. At first, the algorithm will check for the page value P43 in the cache memory(cache memory will temporarily store frequently used instructions and data for quicker processing by the CPU); if the requested pages are found in the cache memory, then it will be immediately shared to the process by cache memory.

Case-2:

Suppose that server three is executing process three, and it will request page P54 to proceed further. The algorithm will first check for this page value in the cache memory, and if it finds the same, it will be allocated to the process and proceed further. In case the requested page isn't available in the cache memory. It needs to load it from the main memory by either moving the requested page from a remote server locally by caching it or permanently migrating the requested page.

To provide page P54 (On server 5, index 4) to the process, we need to travel from server 3 to 5. There are three total possible paths:

1) P3→ P2→ P1→ P5
2) P3→P1→P5
3) P3→ P4→P5

The total distance in case-1 is 6, in case-2, it is six, and in case-3, it is equal to 5. For the same, the shortest path in case-3. We can determine the shortest path using Dijkstra's algorithm(This algorithm provides the shortest cost path from the source router to other routers in the network). So, the total time taken by this process is 10. (Client-Server and Server-Client). In the algorithm itself, we will set the threshold value for the total time, and it will depend on the system design, and if the total time is greater than the threshold, then we will put that process in the cache memory to minimize the access time. If for some reason(Like memory might

be full), it is not possible to take that page in the cache memory, we can perform the page migration in order to access the same page quickly in the future.

If the page migration option is performed, the required steps will be described below.

Suppose the process-3 (P3) requested page P54. As discussed earlier, the total distance from server-3 to server-5 is ten and greater than the threshold value. These processes consume more time for processors, so it is advisable to put that page in the cache or perform the migration operation. Suppose P3 requests the page P54 again and again in some interval of time; then we can perform the page migration as mentioned below:

Suppose page P31(Server 3, page1) has not been referenced yet to replace the P31 with the P54, which is called the page migration. Once the migration is done, suppose that the page P31 will be requested to be accessed by many processes in a specific time interval; in that case, the algorithm will load that page and then put it in the cache memory.

Advantages of migration:

- The page which is migrated will still be accessible.
- Nevertheless, the page used frequently in Process is in the server program, so the server's responsiveness will increase.
- Management of requests becomes easy.
- It has reduced network traffic.

Use of the data structure in order to map the request of any specific server

The Use of an Array Data Structure for Mapping a Request to a Particular Website server. If the page won't be found (Because it doesn't exist or there could be some reason) on the local machine, the first priority is to locate the most appropriate server that holds the requested page at this time. An appropriate search algorithm must be implemented in order to achieve mentioned. Each page is assigned an address space based on the size of the server which can be it will be serving. For that purpose, we need to make one table (we will call it a global table) that will keep track of these address spaces. Whenever a specific page address request is made, the request will forward it to that particular location where the table is kept. There are two empty arrays declared which can be used to store those values. The server will use one of them to fill in the requested address, while the remaining array will compare the addresses in the table. A suitable compare procedure followed by the server will then serve the page accordingly.

We'll get an array with a total size of N*D, where N represents the total number of servers and D is the number of pages which was held by each server,.and the array will keep track of the count variable, which tells us which request was made by how much time in terms of address. We'll suppose the cut-off frequency is 8. Therefore, if the value of the count variable for a particular address is more than 8, we'll store it in distributed cache memory as the cache memory is globally declared, and any server can use it with some priority order. The server will search the distributed cache for their request's address, and if it is found in the cache memory, it will be mapped instantly. When the request is accepted, there can be two possibilities. The first possibility is that the data will be called by migrating the entire page from the server. This is the case when the particular size of the page is smaller than the computation to be performed on the page. The migrated page needs to be stored in the cache of the requesting server. The result is calculated at the destination. If the cache is complete, the LRU algorithm is applied, and the page is transferred to the primary memory.The second inference made here is that the Main memory has some amount of additional storage present for managing the replaced pages from the cache. for the second possibility, the computation from the

requested server will be transferred where the page is requested before. The requested page is stored in the receiver's cache(global cache), and the result is sent back to the server.

Replication:

We are keeping a copy of the same data on the multiple nodes. This technique is widely used to manage databases, filesystems, etc. A node which has a copy of the data is called a **replica**. If some of the replicas are faulty, others are still accessible, and we can use the same to get the data and fulfill the request. Primarily, replicas are distributed at different locations, and we might be able to use the local replicas but not that replica that was somewhere remotely. If we got the data that was not changed, it is static, then we can copy the entire data across all of the nodes, and it can be done using the **replicas**.

Suppose that P1 requests page P43, and we created the replica for the same page. So that this process can access the same page in the future with minimum time as the replica stored in the server's local memory. Now, if there is some other process, let us say P3 requests for the P43, and it is not an efficient way to provide the page itself to the P3 because it will increase the total time. Because if we did this, then some other process might come in the future, and that process needs to wait for a long time in order to fulfill its page request, as the main page migrated to some other location. We can remove this limitation by generating another replica of the same page, for the same we need to create the replica of the P43 and migrate with P3 by migration method, and this will minimize the total time for P3 to get the same page in the future as it already exists in its own local memory.

Deadlock:

If a process is holding a resource for a long time and at the same resource is requested by another process, we have to do pre-emption, which is impossible. We cannot forcefully take a resource from a running process that will cause a deadlock.

Mutual exclusion :

Mutual exclusion is a program that will prevent the simultaneous access of the shared resource at the same time. It will prevent two processes from entering the critical section at the same time. In this graphical model, we have searched for an algorithm that will check all processes in the graph from the right and left sides and let only one process access the resource/server. This algorithm is similar to Peterson's algorithm.

```
1 C[side(i)] ← i
2 T ← i
3 P[i] ← 0
4 rival ← C[¬side(i)]
5 if rival != ⊥ and T = i then
6.     if P[rival] = 0 then
7          P[rival] = 1
8      while P[i] = 0 do spin
9       if T = i then
10.         while P[i] ≤ 1 do spin

// critical section goes here
11 C[side(i)] ← ⊥
12 rival ← T
13 if rival 6= i then
14          P[rival] ← 2
```

This is called Yang and Anderson's algorithm.
This will check all the process IDs in 2 ranges and side(I) variable is determined accordingly

1) side(0) for process id in range {1, 2,.....n/2 }
2) side(1) for process id in range {n/2+1……n}

Variable C[0] and C[1] determines which process will be selected out of both sides.

Each process contains a variable P[I] , initially P[I] = 0 for all processes. This variable spins when the process state changes to blocked. P[I] has a value range {0,1,2}. This variable signals if the process is safe to proceed.

Furthermore, variable T if T=i then it is i'th process's turn to wait.

Suppose there are two processes in Graph P1 and P3 Trying to send a request to the same server, S2. Suppose requests are r12 and r32

Suppose P1 requests then set C[side(i)] value so P3 can find P1.
P1 then sets the T value to itself so P3 can also know that P1 is requesting.
P1 blocks itself if C[~side(i)] = 1 and T = i.

This can occur in two ways:

1. P1 really writes the value of T after P3 did.
2. P3 wrote C[~side(i)] but did not write the value of T

In the later case, P3 will signal P1 that the T value may have changed due to the spin of P[i] to 1. P1 has to recheck T because it did write the value of T later. If the value of T for P1 is still I, then it is a surety that P3 is ahead of P1, and its request will surely be accepted. In that case, P1 can wait for P[I] value to spin to 2, indicating that P3 is left.

For an entire tree, this algorithm has the complexity of O(log n).

Dijkstra's algorithm

1. First, find the shortest distance path between the client and server/source. Create the shortest distance path set from it, which will be initially empty.
2. To each server, the node assigns the value of distance/weight. Furthermore, the source node/server (So) assigns distance zero so the algorithm can begin from that point.
3. While the shortest distance path set does not include all the server nodes.

a) pick a server node Si which is not in the set and has the shortest distance from So

b)include Si into the set

c)update the distance value of all adjacent nodes of Si. to update distance value, iterate through all adjacent nodes for every adjacent node Sj

if the sum of a distance value of Si (from So) and weight of edge Si-Sj, is less than the distance value of Sj

then update the distance value of Sj.