Κινητος και Διαχυτος Υπολογισμος

ON-DEMAND CODING-BASED BROADCASTING NETWORK

February 20,2022

Poulianou Chaido 2613
Polykarpou Christoforos 2668
Varnava Anastasis 2669

Contents

Chapter 1

General Description

1.1 General Description

In this project, we implemented the straightforward approach for on-demand code-based data broadcast from G.G. Md. Nawaz Ali, Kai Liu, Victor C.S. Lee, Peter H.J. Chong, Yong Liang Guan, Jun Chen. On their article "Towards efficient and scalable implementation for coding-based on-demand data broadcast" they covered three approaches, the straightforward, efficient and approximate algorithms. We will discuss these algorithms and their performance in detail below.

Chapter 2

Straightforward Approach

2.1 Algorithm Overview

This algorithm is a coding based(encoding-decoding) on-demand broadcast algorithm. It takes advantage of cached and requested data items to broadcast an encoded packet of the most requested data items according to chosen QoS.
The straightforward approach is using the following scheduling algorithms:

- **FCFS**: First come first Serve or also known as FIFO. Clients that send a request first to the server, have priority over later arriving requests.
- **MRF**: Most Requests First, it takes the data item with most pending requests as the one of top priority. The coding based implementation later finds the maximum clique with the highest priority.
- **EDF**: Earliest deadline first. Every request comes with a deadline time. The server calculates remaining time left to broadcast item to client by subtracting deadline to current time, thus prioritizing clients by the smallest deadline.

The program then calculates maximum clique according to highest priority items based on one of the above algorithms. A maximum clique is a fully connected graph of clients. The server then encodes the required items and broadcasts them in the network. Each client stores items that the server broadcasts in their cache.

2.2 Implementation

The program is implemented in C using the following structures and variables:

- **Client(struct):** contains client-id and Cache.
- **Request(struct):** contains client-id(the one requesting), requestedItem, lru(stores position of last item used in cache), cache_size, deadline(used for EDF), start_t(clock_t variable used to compare performance)
- **Graph(struct):** Adjacency list of lists. Contains an array(AdjList *array) pointing to each head of client's list(AdjListNode *head). Graph is used to calculate max_clique.
- **MRI(String):** Most Requested Item.
- **MRIclients(Array):** stores clients id's requesting MRI.
- **DataToBroadcast(Array):** list(based on scheduling) of items for broadcasting.

On execution, two threads are created. One for simulating clients and one for simulating the server. The server waits until all clients on the network made their request(request item is generated randomly using rand() function).

HOW CLIENTS RECEIVE REQUESTS FROM SERVER

client_list is initialized where it stores information for each client.
newClient_request() is used by new clients in order to append their request to the server.
Then **connect()** is called to add edges on the graph between client nodes.
Edges between two clients are added if either one of the following rules applies:

- First client requests the same data item as second client.
- The requested items of both clients are stored in each other's caches.

Thus **addEdge(firstClient, secondClient)** is called.

The server wakes up after all requests from clients are made and calculates the most popular data item to broadcast. Popularity depends on the chosen **QoS**. By going through all requests that clients demand, it stores the most popular item on MRI variable. By calling **maxClique()** the server finds all clients requesting MRI, and stores them in MRIclients array. The algorithm then runs through the graph list and checks for each one of MRIclients neighbors whether they also are common neighbor of all other MRIclients. This is a requirement to build a fully connected graph that is the final max clique If a common neighbor exists, add its id and requested data item to the maxClique[] array. **DataToBroadcast[]** is used to store all the encoded requested items of the maxClique clients. Finally the server broadcasts the encoded items, removes requests from queue and also removes all edges of MRIclients from the Graph by calling **removeMaxCliqueEdge()**. The server then calls **fillCache()** to add the encoded items to all network clients caches. If their cache is already full, LRU replacement policy is used. A client's cache does not contain items that have been requested by the client.

Broadcast is a simulation and does not work as a real broadcasting network. The program in reality outputs the results in a file named "output.txt". Outputting is considered as broadcasting.

```
// Calling this function adds all necessary edges to the clients graph.
// s is the total number of clients
void connect(struct Graph* graph, struct client_request** client, int
    client_num, char *req, int s){
  int exit = 0;
  //check with every other already inputeed client if we can connect the
    newnode
  for(int i=0; i<s-1; i++){     // s-1 because we don't want to connect
    client vertex to same client
    // check all client's caches and add edge between them if req is equal
    if(strcmp(client[i]->req_item, req) == 0){  // first case
      addEdge(graph, client, client_num, req, i);
    }

    // Second case
```

```
  else{
   //cache array is a 2d array, or a pointer to an array of 4 elements(for client0)
   //each element constists of 2 chars, so cache array points to a total area of $*2=8 bytes
   //so to find the number of elements, divide by 2
   if((client[i]->cached_items != NULL) || (client[client_num]->cached_items != NULL)){
    int size = client[i]->cache_size; // sizeof(client[i]->cached_items)/2;
    for(int j=0; j<size; j++){

     if(strcmp(client[i]->cached_items[j],req) == 0){
      int size1 = client[client_num]->cache_size;//sizeof(client[client_num]->cached_items)/2;
      for(int z=0; z<size1; z++){
       if(strcmp(client[client_num]->cached_items[z], client[i]->req_item) == 0){
        addEdge(graph, client, client_num, req, i);
        exit = 1;   // Flag
        break;
       }

      }
       if(exit==1){
         exit = 0;
         break;
       }
     }
    }
   }

  }
 }
}
```

Listing 2.1: Code for connect()

```c
void addEdge(struct Graph* graph, struct client_request** clients, int client_num, char *req, int i){
  struct AdjListNode *newNode, *tempNode = NULL;
  int flag = 0;
  if(graph->array[i].head != NULL){ //if not first entry in the list

    tempNode = graph->array[i].head;
    while(tempNode->next != NULL){
        tempNode = tempNode->next;
        if(tempNode->client_num == client_num){//check if already a neighbor
          flag = 1;
        }
    }
    if(flag == 0){
      if(graph->array[i].head->neigbors > 0 ){
        graph->array[i].head->neigbors++;
      }
      else{
        graph->array[i].head->neigbors = 1;
      }
      newNode = (struct AdjListNode*)malloc(sizeof(struct AdjListNode));
      newNode->client_num = client_num;
      newNode->next = NULL;
      tempNode->next = newNode;
    }
  }
  else{
    graph->array[i].head = (struct AdjListNode*)malloc(sizeof(struct AdjListNode));
    graph->array[i].head->client_num = i;
    graph->array[i].head->next = (struct AdjListNode*)malloc(sizeof(struct AdjListNode));
    graph->array[i].head->next->client_num = client_num;
    graph->array[i].head->next->next = NULL;
    graph->array[i].head->neigbors = 1;
```

```
}

flag = 0;
// Since graph is undirected, add an edge from
// dest to src also
if(graph->array[client_num].head != NULL){ //if not first entry in the list

  tempNode = graph->array[client_num].head;
  while(tempNode->next != NULL){
      tempNode = tempNode->next;
      if(tempNode->client_num == i){//check if already a neighbor
        flag = 1;
        break;
      }
  }
  if(flag == 0){
    if(graph->array[client_num].head->neigbors > 0 ){
      graph->array[client_num].head->neigbors++;
    }
    else{
      graph->array[client_num].head->neigbors = 1;
    }
    newNode = (struct AdjListNode*)malloc(sizeof(struct AdjListNode));
    newNode->client_num = i;
    newNode->next = NULL;
    tempNode->next = newNode;
  }
}
else{
  graph->array[client_num].head = (struct AdjListNode*)malloc(sizeof(struct AdjListNode));
  graph->array[client_num].head->client_num = client_num;
  graph->array[client_num].head->next = (struct AdjListNode*)malloc(sizeof(struct AdjListNode));
  graph->array[client_num].head->next->client_num = i;
  graph->array[client_num].head->next->next = NULL;
  graph->array[client_num].head->neigbors = 1;
}
```

```
}
```

Listing 2.2: Code for addEdge()

```
void maxClique(struct Graph *clients, struct client_request **max_clique, struct client_request **clients_requests){
  int mri = 0;

  // find clients of MRI
  int counter = 0;
  for(int j=0; j<max_clients; j++){
    if(strcmp(MRI, clients_requests[j]->req_item) == 0){
      MRIclients[counter++] = clients_requests[j]->client_num;
    }
  }
  mri = MRIclients[0];
  struct AdjListNode *vertex = clients->array[mri].head;

  struct AdjListNode *neighbor = NULL;
  struct AdjListNode *temp = NULL;//the adjlist item of the neighbors list
  int neighbor_neighbor_num = 0;
  int neighbors = 0;
  counter = 0;//if counter == #of MRIClients neighbors, we add that client to max clique
  int flag = 0;//if 1, then the neighbors neighbor also an MRIlient neigbor, there exist the edges: MRIClient->neighbor and neighbor[i]->neighbor

  if(vertex == NULL){//MRIClient has no neighbors, so max_clique only consists of MRIcllients[0](there only exists 1 mriclient, otherwise it would have a neighbor)
    //initialize max_clique array
    for(int y = 0; y <max_clients; y++){
      max_clique[y] = (struct client_request*) malloc(sizeof(struct client_request));
      max_clique[y]->client_num = -1;
    }

    max_clique[0]->client_num = mri;
    max_clique[0]->req_item = clients_requests[mri]->req_item;
```

```
    int l = mri;
    struct client *temp = client_list;
    while(l > 0){
      l--;
      temp = temp->next;
    }
    //last step, remove the max_clique requests from the graph
    clients_requests[mri]->req_item = NULL;
    clock_t end_t = clock();
    temp->latency = end_t - clients_requests[mri]->start_t;

    return;
  }
  else{

    neighbors = vertex->neigbors;//num of neighbors of MRIClient, compare it with counter to see if node to be included in max clique
  }

  int reject_flag = 0;//if 1, then reject the neigbor from comparing it

  //initialize max_clique array
  for(int y = 0; y <max_clients; y++){
    max_clique[y] = (struct client_request*) malloc(sizeof(struct client_request));
    max_clique[y]->client_num = -1;
  }

  for(int i=0; vertex != NULL; i++){
    counter = 0;
    neighbor = clients->array[vertex->client_num].head;//take the first neighbor of MRIClients adjlist
    //and loop for each of its own adjlist items
    mri = MRIclients[i];
    for(int j=0; neighbor != NULL; j++){
      neighbor_neighbor_num = neighbor->client_num;
      temp = clients->array[mri].head;
      //check for each of the MRIClients neighbors if they are also neighbors of the current neigbor
```

```
    for(int k=0; temp != NULL; k++){
      if((neighbor_neighbor_num == temp->client_num) || (neighbor_neighbor_num == mri)){//we have a matching neighbor
        flag = 1;
        break;
      }

      temp = temp->next;
    }
    if(flag == 1){//match
      counter++;
      flag = 0;
    }
    else{//at least one not common neighbor, the current neighbor is not in max clique, break
    //put current neighbor in reject list (((?????????)))
      reject_flag = 1;
      flag = 0;
      break;
    }

    clock_t end_t = clock();
    //check if current neighbors all connections with MRIClient
    if(counter == neighbors){
      max_clique[i]->client_num = vertex->client_num;
      max_clique[i]->req_item = clients_requests[vertex->client_num]->req_item;
      int l = vertex->client_num;
      struct client *temp = client_list;
      while(l > 0){
        l--;
        temp = temp->next;
      }
      //last step, remove the max_clique requests from the graph
      clients_requests[vertex->client_num]->req_item = NULL;
      temp->latency = end_t - clients_requests[vertex->client_num]->start_t;
    }
    neighbor = neighbor->next;
```

```
    }
    vertex = vertex->next;
  }
}
```

Listing 2.3: Code for maxClique()

2.3 Runnning the programm

The source file name is "project.c". By running command "make", an executable with name "project" is created. To run it, use command

```
./project [arg1]
```

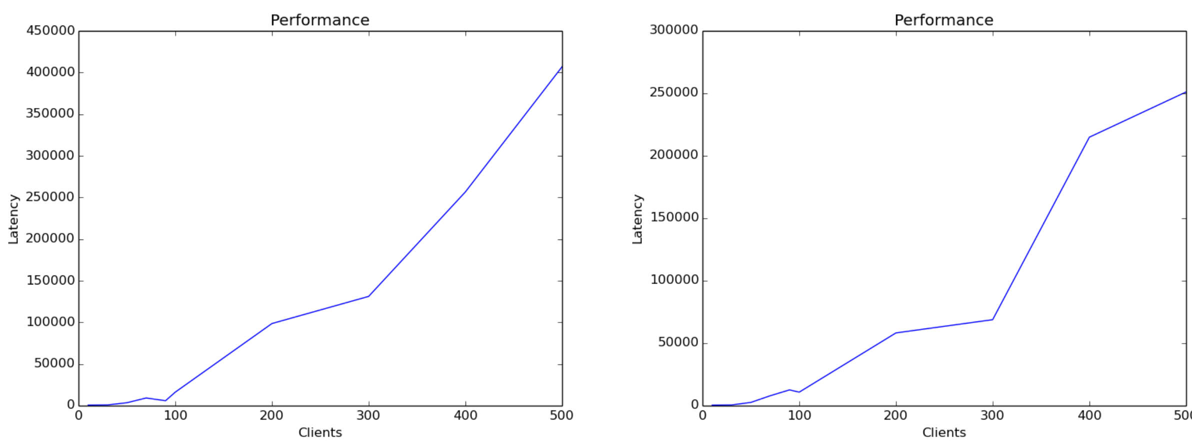
Where arg1 implies the applied scheduling algorithm to calculate the data items priority. If arg1 = 0, MRF is applied, otherwise if arg1=1, EDF and if arg1=2 FCFS. If no argument is given, it defaults to MRF.

The number of max data items (MAX_DATA_ITEMS) , clients (max_clients), rounds of requests (REQ_ROUNDS) and rounds of client number changing (ROUNDS) can be specified by changing the defined variables in the source code for different tests. The results are outputed in a text file called "output.txt", where each row represents number of ROUNDS, and first column the average latency for the number of clients given in the second column.

Chapter 3

Performance

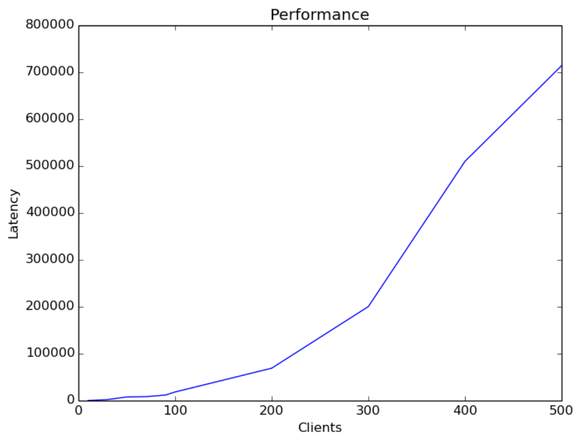
In this chapter we are analyzing the performance of straightforward, efficient and approximated approaches for on-demand data broadcast. We implemented a program in python where it reads from the file we were outputting in C and creates the following graphs:



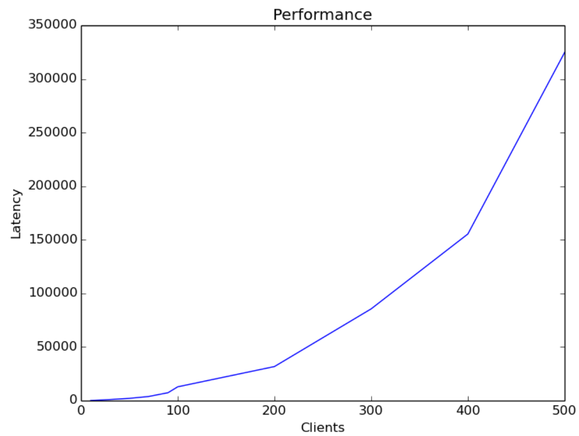
(a) 5 data items

(b) 10 data items

Figure 3.1: Straightforward approach using MRF scheduling

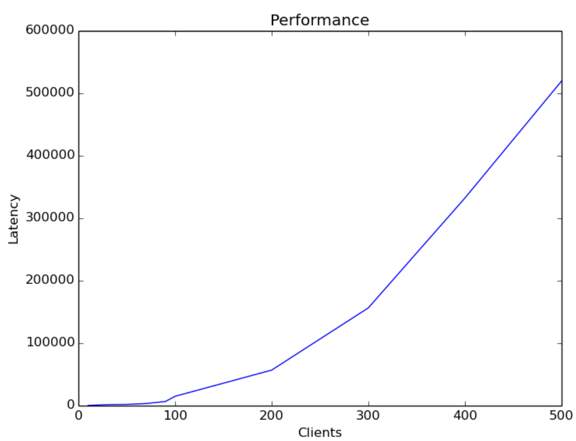


(a) 5 data items

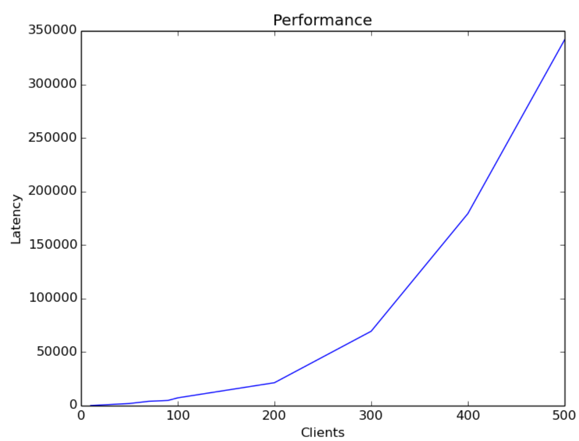


(b) 10 data items

Figure 3.2: Straightforward approach using EDF scheduling

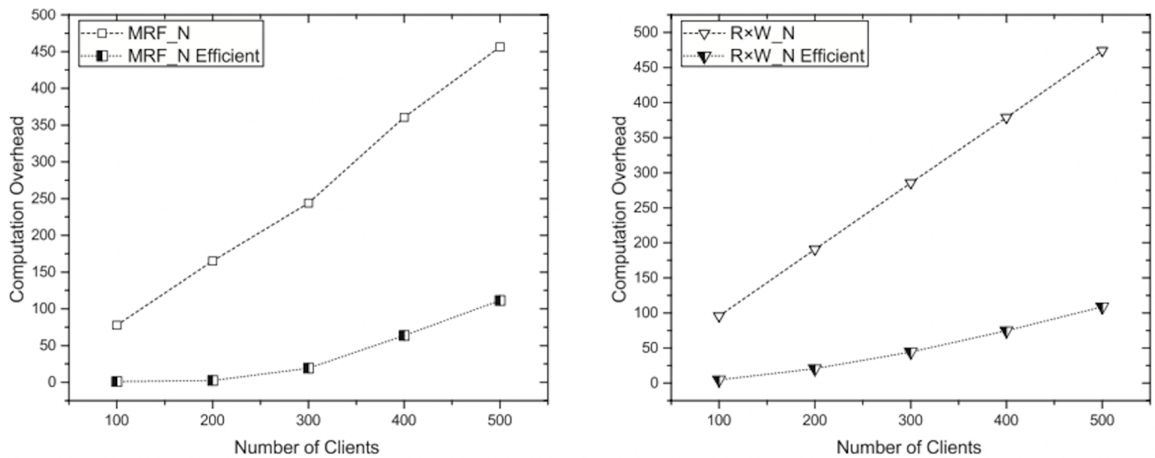


(a) 5 data items



(b) 10 data items

Figure 3.3: Straightforward approach using FCFS scheduling



(a) Straightforward MRF_N vs. Efficient MRF_N

(b) Straightforward R×W_N vs. Efficient R×W_N

Figure 3.4: Straightforward vs Efficient

Based on the graphs above, it is clear that as clients increase in the network, so does the average latency. Also, by incrementing the maximum available data items we see that the average latency decreases. That could be, because the probability of clients requesting the same item decreases, so searching cost for their a maximal clique decreases as well. The latency also depends on the selection of QoS. As we can see, MRF scheduling algorithm is outperforming EDF and FCFS.

Last graph represents straightforward's and efficient's Computation overhead for MRF and RxW as scheduling algorithms. As we can see from the graph, the straightforward approach is much more computational heavy, thus it has more overhead. Efficient implementations are a lot faster.

Chapter 4

Final Verdict

Based on the research from G. G. Md. Nawaz Alia, Kai Liuc, Victor C.S. Leed, Peter H.J. Chonge, Yong Liang Guanb, Jun Chenf on "Efficient and scalable implementation for coding-based on-demand data broadcast", we conclude that their efficient implementation has lower average latency compared to our straightforward approach. This result was expected from the paper too, since the main drawback of the straightforward approach is the search cost for maximal clique. Because of the nested loops that this approach has, it results in high overhead compared to the efficient one. The efficient approach overcomes this problem by using a vertex degree to skip some computations for the maximal clique. This is done by searching a sub-graph of the vertices instead of the main graph, thus reducing the searching cost for the maximum clique.

This makes the efficient approach a much better choice for on-demand coding-based broadcasting networks.