Contents

List of Figures

List of Tables

1 Introduction

1.1 Motivation

In the case of the actual network, the traffic in the networks has a strong temporal variability. During peak hours congestion occurs and during off-peak hours the resource can not be fully utilized. Therefore we want to shift some work in the peak hours to the off-peak hours. The solution to this is caching. A cache is a hardware or software component that stores data so future requests for that data can be served faster. In our project, we use caching technique to reduce peak traffic rates by placing content/data into cache memories at the clients. In my project we use decentralized caching, because it is more realistic compare to the centralized caching[3], which has a central coordination in placement phase. The central coordination results in a limitatiom of its applicability.

1.2 Objective of Our Works

The first objective is to realize the idea "Decentralized Coded Caching" from paper "Decentralized Coded Caching Attains Order-Optimal Memory-Rate[2] Tradeoff". The second objective is to design the search algorithm, which is not mentioned in the paper and is used to find the XORed files. The third objective is to consider the unreliability of UDP and solve this problem with KODO. Because the paper do not consider the unreliability of UDP.

1.3 Structural of This Document

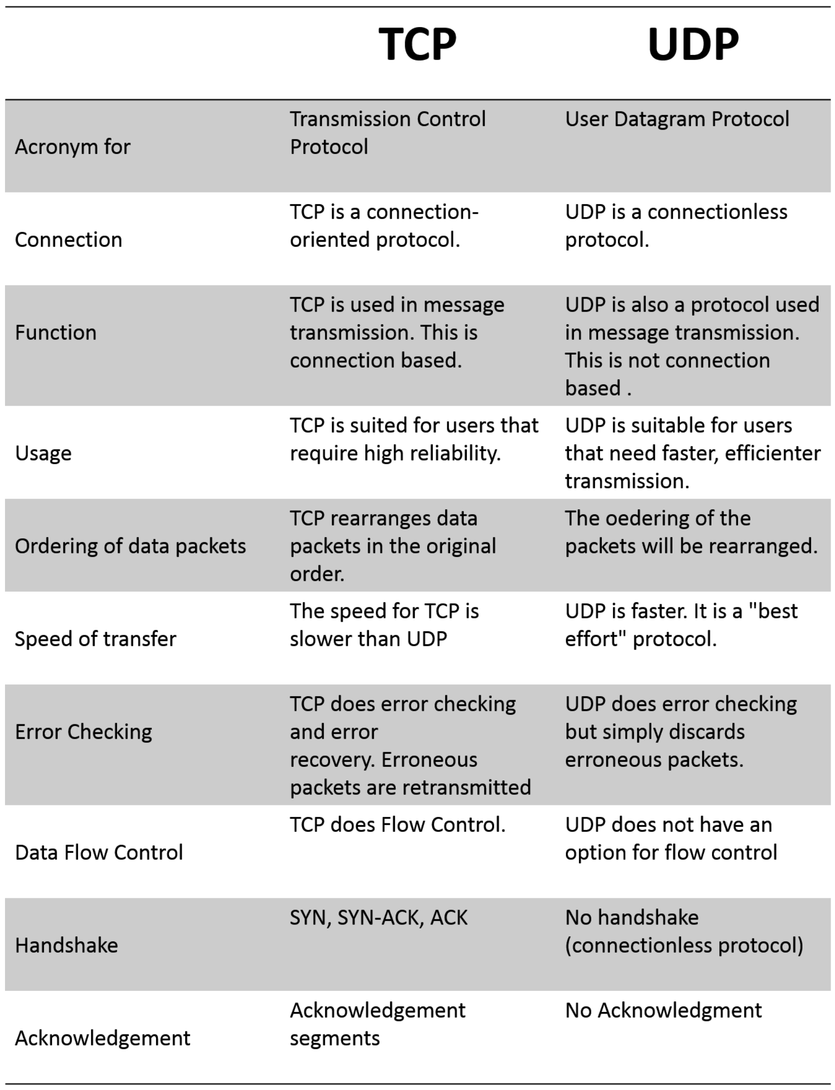
In the 2. Chapter, we will briefly introduce TCP and UDP, also the differences and similarities between them. In the 3. Chapter, we will illustrate "Decentralized Coded Caching" by a example. In the 4. Chapter, we will explain how to create the multicast data(In other word, multicast data is the coded data with XOR operation), which is important to decentralized coded caching In the 5. Chapter we will briefly illustrate KODO and RLNC (Random Linear Network Coding). Then we talk about the Realization with

KODO. In the 6. Chapter we will explain the implementation of our project through program flow chart of server and client. After this we will show the simulation's results of our program. The 7. Chapter is about conclusion and outlook.

2 TCP and UDP

2.1 Comparison between TCP und UDP

Internet protocol suite has two important core members. They are TCP (Transmission Control Protocol) and UDP (User Datagram Protocol). TCP provides reliable, ordered, and error-checked delivery of a stream between users. TCP is connection oriented – once a connection is established, data can be sent bidirectional. If users do not require reliable data stream, then they can use User Datagram Protocol (UDP), which provides a connectionless service that emphasizes reduced latency over reliability. UDP is a simpler, connectionless Internet protocol. Multiple messages are sent as packets in chunks using UDP. The comparison chart in the below will illustrate the differences and similarities between TCP and UDP. We can see Figure 2.1.



	TCP	UDP
Acronym for	Transmission Control Protocol	User Datagram Protocol
Connection	TCP is a connection-oriented protocol.	UDP is a connectionless protocol.
Function	TCP is used in message transmission. This is connection based.	UDP is also a protocol used in message transmission. This is not connection based .
Usage	TCP is suited for users that require high reliability.	UDP is suitable for users that need faster, efficienter transmission.
Ordering of data packets	TCP rearranges data packets in the original order.	The oedering of the packets will be rearranged.
Speed of transfer	The speed for TCP is slower than UDP	UDP is faster. It is a "best effort" protocol.
Error Checking	TCP does error checking and error recovery. Erroneous packets are retransmitted	UDP does error checking but simply discards erroneous packets.
Data Flow Control	TCP does Flow Control.	UDP does not have an option for flow control
Handshake	SYN, SYN-ACK, ACK	No handshake (connectionless protocol)
Acknowledgement	Acknowledgement segments	No Acknowledgment

Figure 2.1: The Comparing between TCP and UDP

3 Decentralized Coded Caching

3.1 The Basic Algorithm of Decentralized Coded Caching

3.1.1 The Basic Algorithm of Decentralized Coded Caching

We now present an algorithm (referred to as decentralized coded caching in the following), which was proposed from the paper "Decentralized Coded Caching Attains Order-Optimal Memory-Rate Tradeoff". The proposed algorithm consists of a placement procedure and two delivery procedures. In the placement phase, the clients cache data from server. After the placement, the clients will request files from the server. Then the server enters into the first delivery phase. In this phase the server will send the XORed data (coded data), which can minimize the peak rate over the shared link. After the first delivery phase, the server will carry out the second delivery phase, in which the server will send the not XORed data (uncoded data). F denotes the number of files. The size of every file is S. U denotes the number of clients. M denotes the caching memory size of every client. We use the notation $[F] = \{1,2,3\ldots,F\}, [U] = \{1,2,3\ldots,U\}$

```
procedure PLACEMENT for $u \subseteq [U], f \subseteq [F]$ do
        Client u independently caches a part of file n, which has a size of $\frac{M}{F}$
        randomly and uniformly
end for
end procedure

procedure DELIVERY 1
for $f \subseteq [F]$ do
        Server sends the XORed data(coded data)
end for
end procedure
```

```
procedure DELIVERY 2
for $f \subseteq [F]$ do
        Server sends the not XORed data(uncoded data)
end for
end procedure
```

3.1.2 Example for Decentralized Coded Caching

Firstly we consider a server connected through a shared link to two clients, the size of each client's cache memory is M bits. This server has two files, respectively A and B. The size of each file is F bits. Each user caches a subset of bits of each file independently at random, satisfying the memory constraint. The all subsets of A and B are included in two set S_A and S_B. Now the file A and file B are divided into 4 subsets, respectively, on the other word:

$$S_A = 0{,}1{,}2{,}12, S_B = 0{,}1{,}2{,}12$$

NOTE: The subset{12} represents client1 and client 2 cached this subset already. The subset{0} represents client 1 and client 2 did not cached this subset. The other subsets have the simialr representation.

Subsets of $A\{1{,}12\}$ are cached in client 1 and the subsets of $B\{2{,}12\}$ are cached in client 2, the placement phase are accomplished.

Following the search algorithm, which is used to search the XORed files, server finds two subsets $A\{1\}$ and $B\{2\}$, which can encode with XOR operation. After this server also find that subsets{0} of file A and file B are not cached in the clients memory, subsets{0} of both files are clarified into no XORed data, i.e subsets{0} of A, B are transmitted in the form of original data by the server in the delivery phase. As shown in the figure 3.1 below:

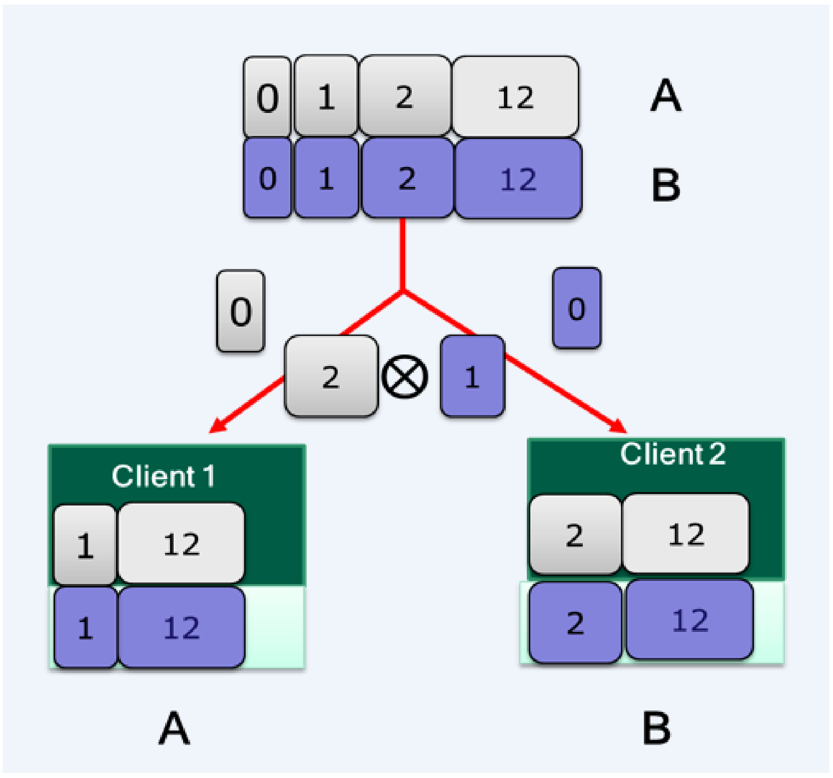


Figure 3.1: The Example for Decentralized Coded Caching

4 Multicast Data

4.1 Multicast Data

In the 3. Chapter, you may be noticed that the server will send the XORed data and original data(without XOR) in the delivery phase. We call these Multicast data. But how can the server create these data, especially for the XORed data. In this section, we will explain the searching algorithm. The server use this algorithm to find the files, which can be encoded woth XOR operation, and then the server get the XORed data (encoded data). In the case the server knows the XORed data, it is easy to find the not XORed data.

According to the requested files from the clients and the cached data, server can calculate the XORed data which need XOR operation, in order to minimize the resulting peak rate over the shared link, and the not XORed data, which do not need XOR operation. The algorithm is presented in the below.

We use the variable XOR-files, noXOR-files to respectively denote the files, which need XOR operation and the files, which do not need XOR opertion. Notation $V^n_{k,p}$ denotes that the p. cached part of file k in client n. $V^n_{k,p}$ denotes client n do not have the p.part of file k. [U] denotes clients with the notation $[U] = \{1,2,3\ldots,U\}$.

The first circulation of the algorithm is used to search for the clients, who request the same file from the server. The second circulation of the algorithm is used to search for the clients, who request different files from the server.

```
procedure XORFILESEARCH

for $n \subseteq [U]$ do

    for $m \subseteq [U]$ do
```

Client n requests file R and has the α part of file R is $V_{R,\alpha}$, if client m requests the same R and has the different part β of file R, $V^n_{R\beta}$. This two file

```
$V^n_{R,\alpha}$ and $V^n_{R,\beta}$ can be XORed and thus will be added to the variable XORFiles
        end for
    end for
    for $n \subseteq [U]$ do for $m \subseteq [U]$ do
        Client n requests file R and has a part of file G, $V^n_{G,\alpha}$. Client m requests file
        G and has a part of R, $V^n_{R,\beta}$. If client m do not have the file $\overline{V}^n_{G,\alpha}$ and client
        n do not have the file $\overline{V}^n_{R,\beta}$. This two files $V^n_{G,\alpha}$ and $V^n_{R,\beta}$ can be XORed and
        thus will be added to the variable XORFiles
        end for
    end for
    for $n \subseteq [U]$ do
        for $m \subseteq [U]$ do
            Client n requests file R and has a part of file G, $V^n_{G,\alpha}$. Client m requests file
            G and has a part of R, $V^n_{R,\beta}$. If client m do not have the file $\overline{V}^n_{G,\alpha}$ and cilent
            n do not have the file $\overline{V}^n_{R,\beta}$. This two files $V^n_{G,\alpha}$ and $V^n_{R,\beta}$ can be XORed and
            thus will be added to the variable XORFiles
        end for
    end for
    for $n \subseteq [U]$ do
        Client n requests file R. After the delivery of XORFile, cilent n still lacks file
        $\overline{V}^n_{R,p}$. The file $\overline{V}^n_{R,p}$ will be added to the variable noXORFiles
    end for
end procedure
```

5 KODO

When we realize the “Decentralized Coded Caching”, we find that some packets will lost and the ordering of the packets will change. Thus the loss packet need to be retransmitted. But it is difficult to design protocol. So we solve this problem with KODO[1]. In this chapter, we will firstly explain KODO, Then briefly explain how to use KODO or the realization with KODO.

5.1 KODO

KODO implements a number of different erasure correcting codes. In our project, The Full RLNC[4] (Random Linear Network Coding) is used. The Full RLNC is one of the most common RLNC variants, and possess several advantages that RLNCs have over traditional erasure correcting codes. However, we just used it as a standard erasure correcting code, namely to encode and decode the video file data.

In the following, we want to introduce some key parameters, which are important in our project when configuring an erasure correcting code:

$$\text{The number of symbol: } \textit{symbols}$$
$$\text{The symbol size: } \textit{symbolsize}$$

In general when a block of data is encoded, the block of data will be split into a number of *symbols*, each symbol’s size is *symbolsize*. In network applications, a single symbol typically corresponds to a single packet.

symbolsize denotes the size of each symbol in bytes. The choice of symbol size typically depends on the application. For network applications we may choose the symbol size according to the network MTU (Maximum Transfer Unit) so that datagrams do not get fragmented as they traverse the network. In those cases symbols sizes are typically around 1300-1400 bytes.

symbols denotes the number of symbols in a block/generation. Increasing this number will have the following effects:

1. The computational complexity will increase, and can therefore slow down the encoding/decoding.
2. The per-symbol decoding delay will become larger. The reason for this is that when we increase the number of symbols that are encoded the decoder has to receive more symbols before decoding.
3. The protocol complexity can be decreased. If the number of symbols is increased so that all the data which is to be encoded can fit in a single generation, the protocol will only have to handle a single generation. If multiple generations are needed, the receivers will have to tell from which generations the server should send data, and hence increasing the complexity of the protocol.

The reason why we use the KODO is that UDP protocol is not reliable. For example in transmission some packets will be lose, so we need to retransmit the loss packet. But in multicast the retransmitted packets maybe are not useful for the other clients and it will become more difficult to design the protocol. In addition to this UDP has no congestion control, reordering and etc..

If we use KODO, the clients and the server do not pay attention to the lost packet. The clients just receive the packet until the decode is successful, then send a acknowledgement to the server. The other side, the server send the packet all the time until all the acknowledgements from the client are received.

5.2 Realization with KODO

In this section, I will explain how to achieve encoding using KODO.

In my program, every file will be separated into many blocks/Generation. Every block's size equal s to *symbols** *symbolsize* (*symbols*=1024, *symbolsize*=512). Then every block is separate into M (equals to *symbols*) symbols as shown in Figure 5.1. Every symbol corresponds to a single packet. With KODO a lot of linear equations will be calculated as shown in Figure 5.2.

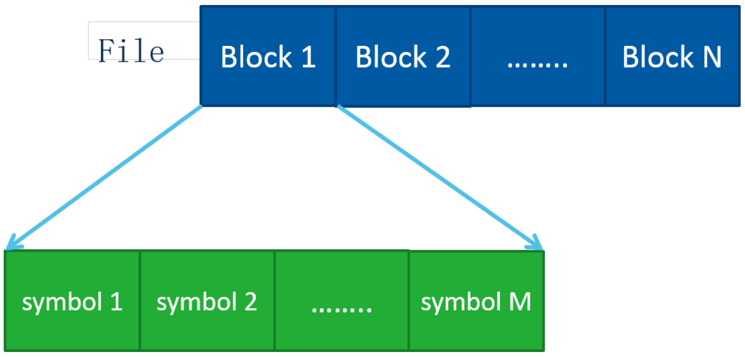
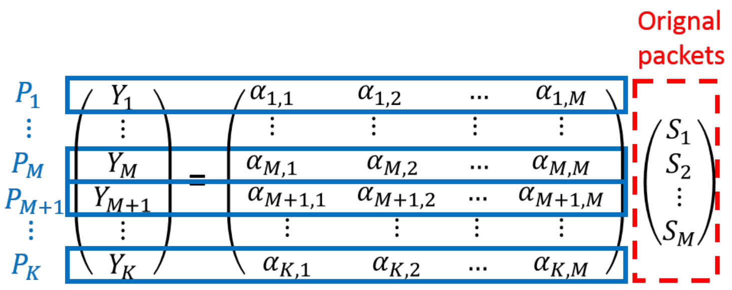


Figure 5.1: Separation

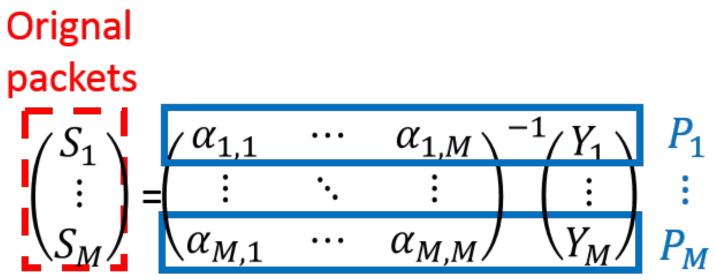
And in fact $(S_1, S_2 \ldots S_M)$will not be transmitted, but $(P_1, P_2 \ldots P_M)$ will be transmitted until ACKs from clients are received. Figure 5.2 In the client, it



$$\begin{array}{c|c} P_1 & \\ \vdots & \\ P_M & \\ P_{M+1} & \\ \vdots & \\ P_K & \end{array} \begin{pmatrix} Y_1 \\ \vdots \\ Y_M \\ Y_{M+1} \\ \vdots \\ Y_K \end{pmatrix} = \begin{pmatrix} \alpha_{1,1} & \alpha_{1,2} & \cdots & \alpha_{1,M} \\ \vdots & \vdots & \vdots & \vdots \\ \alpha_{M,1} & \alpha_{M,2} & \cdots & \alpha_{M,M} \\ \alpha_{M+1,1} & \alpha_{M+1,2} & \cdots & \alpha_{M+1,M} \\ \vdots & \vdots & \vdots & \vdots \\ \alpha_{K,1} & \alpha_{K,2} & \cdots & \alpha_{K,M} \end{pmatrix} \begin{pmatrix} S_1 \\ S_2 \\ \vdots \\ S_M \end{pmatrix}$$

Figure 5.2: Encoding

just need to receive M equation and then can calculate the original packets $(S_1, S_2 \ldots S_M)$ as shown in Figure 5.3



$$\begin{pmatrix} S_1 \\ \vdots \\ S_M \end{pmatrix} = \begin{pmatrix} \alpha_{1,1} & \cdots & \alpha_{1,M} \\ \vdots & \ddots & \vdots \\ \alpha_{M,1} & \cdots & \alpha_{M,M} \end{pmatrix}^{-1} \begin{pmatrix} Y_1 \\ \vdots \\ Y_M \end{pmatrix} \begin{array}{c} P_1 \\ \vdots \\ P_M \end{array}$$

Figure 5.3: Decoding

5.3 KODO in Our Program

In our project, KODO is our second encoded algorithm. Server firstly encode the data with XOR, then use KODO to encode again. After receiving the data, client decode them with KODO at first, then do the XOR operation, which is the second decoding process, to get the original data. The following figure 5.4 shows this process with KODO and XOR operation.

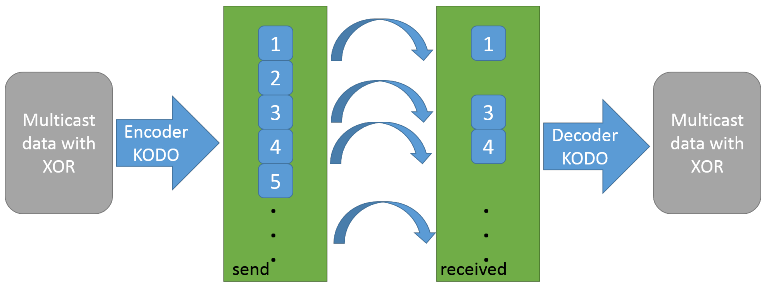


Figure 5.4: Graphical Representation

6 Implementation of The Decentralized Coded Caching With KODO

In this chapter, we will illustrate how do we implement the decentralized coded caching with KODO. Then we show some simulation results of our program.

6.1 Implementation

The tools we use to realize the decentralized coded caching with KODO are listed below:

1. Programming language: Python 2.7
2. Library: python-KODO, matplotlib, numpy
3. Operating system: Ubuntu 16.04 LTS

6.1.1 Program Flow Chart of Server

The figure 6.1 is the program flow chart of server. In initial, server will separate every files into many subfiles according to the number of the files in the server. Then server go into the placement phase and wait for the connection from clients. Once a connection is established, client will cache some subfiles from the server. (The accessed probability of each subfile is the same). After caching, the client will disconnect the server. Once the placement phase finishes and clients request files from server, then server will enter into delivery phase and begin to create the XORed data. Then all the data will be encoded with KODO and transmitted to the clients using multicast. During transmission the clients will send server ACKs (acknowledgement) when the clients decode successful with KODO. After the ACKs are received by the server, then the server will begin to send the next block/generation data with KODO. Server doing this until all the data are sended.

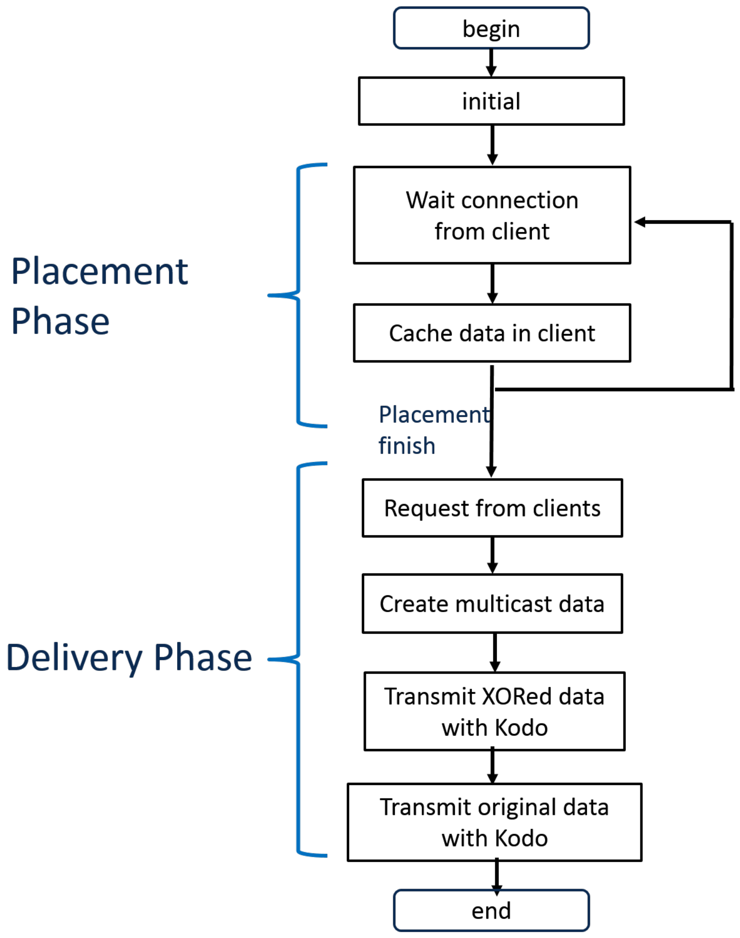


Figure 6.1: Program Flow Chart of Server

6.1.2 Program Flow Chart of Client

The figure 6.2 is the program flow chart of client. After connecting the server, client will cache some files from server according its cache memory size. Then client will request a file from server and enter into delivery phase. In the delivery phase, when server begin to send data, client will check if it needs this file. If client does not need this file, it will wait until the next file and check again. If client needs this file, it will receive this file's data and judge if it can decode with KODO. If no, client will continue receive data from server. If client can decode, it will send a ACK message to the server and then decode with KODO at first and decode again with XOR if needed.

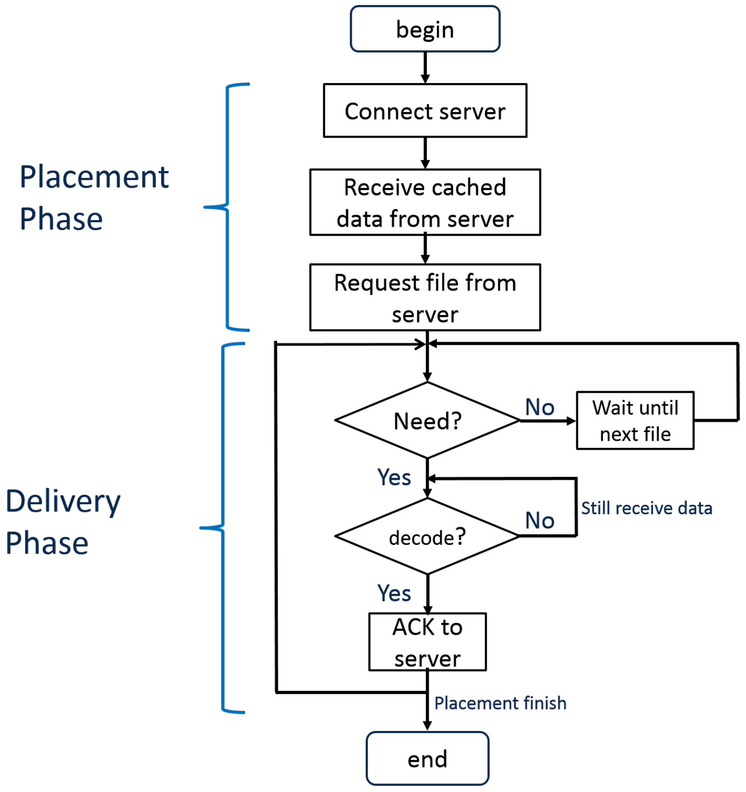


Figure 6.2: Program Flow Chart of Client

6.1.3 Communication between Server and Client

The figure 6.3 shows the communication between server and client. In the placement phase, server and client use TCP to communicate. In addition to cached data, server will send some important information to client, for example, the key parameter of KODO. These information is important, so we use TCP in this phase. In the delivery phase, server and client communicate with each other using UDP. With KODO we can say we "reduce" the unreliability of UDP and better to solve the problems of loss packet and reordering.

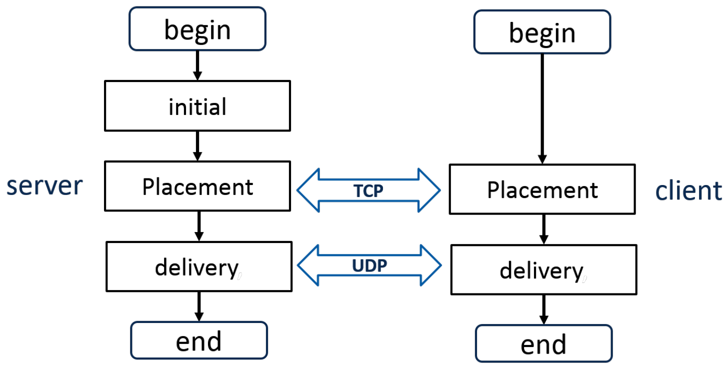


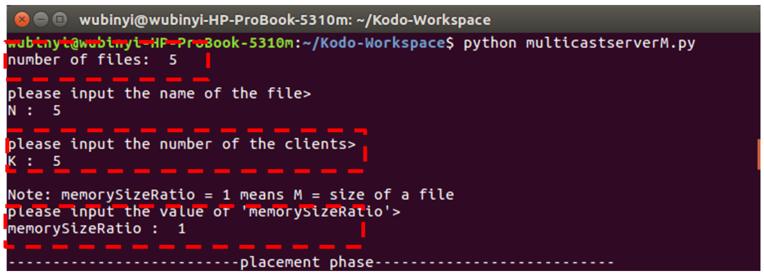
Figure 6.3: Communication between Server and Client

6.2 Results from Running Our Program

In this section, we will show our simulation results of our program.

6.2.1 Information of Server in Initial Phase

The figure 6.4 shows the "Initial Phase". Server has 5 files and 5 clients connect to the server. The cache memory size in each client is 1 (normalized by the size of file).

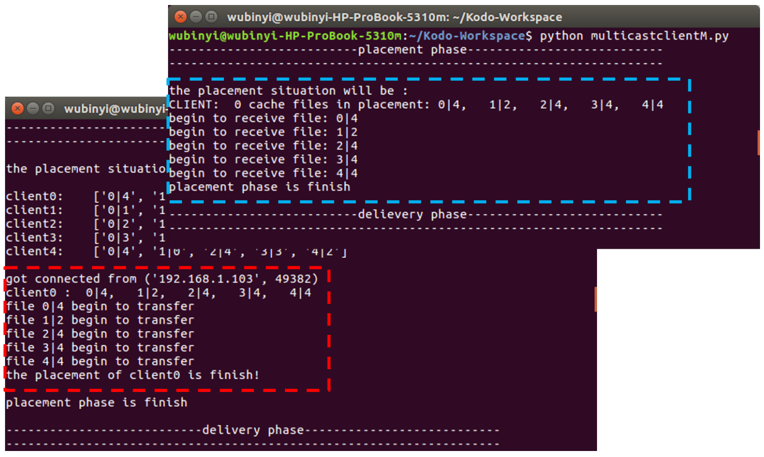


```
wubinyi@wubinyi-HP-ProBook-5310m: ~/Kodo-Workspace
wubinyi@wubinyi-HP-ProBook-5310m:~/Kodo-Workspace$ python multicastserverM.py
number of files:  5

please input the name of the file>
N :  5

please input the number of the clients>
K :  5

Note: memorySizeRatio = 1 means M = size of a file
please input the value of 'memorySizeRatio'>
memorySizeRatio :  1

-------------------------placement phase--------------------------
```

Figure 6.4: Initial Phase

6.2.2 Information of Placement Phase

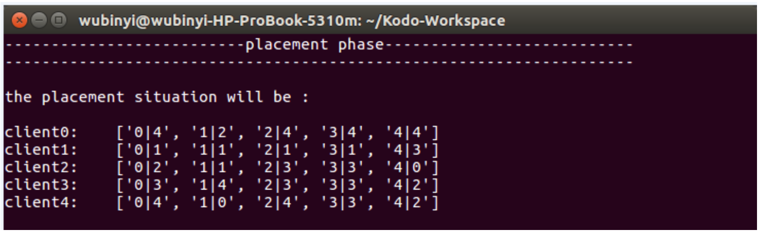
The figure 6.5 shows the "Information of Placement Phase". Client cache file from server. The caching situation like the following figure 6.6 shows.



```
wubinyi@wubinyi-HP-ProBook-5310m: ~/Kodo-Workspace
wubinyi@wubinyi-HP-ProBook-5310m:~/Kodo-Workspace$ python multicastclientM.py
-------------------------placement phase---------------------------
--------------------------------------------------------------------

the placement situation will be :
CLIENT:  0 cache files in placement: 0|4,   1|2,   2|4,   3|4,   4|4
begin to receive file: 0|4
begin to receive file: 1|2
begin to receive file: 2|4
begin to receive file: 3|4
begin to receive file: 4|4
placement phase is finish
-------------------------delievery phase---------------------------
--------------------------------------------------------------------
```

```
wubinyi@wubinyi
----------------------
----------------------

the placement situatio

client0:    ['0|4', '1
client1:    ['0|1', '1
client2:    ['0|2', '1
client3:    ['0|3', '1
client4:    ['0|4', '1|0', '2|4', '3|3', '4|2']

got connected from ('192.168.1.103', 49382)
client0 :  0|4,   1|2,   2|4,   3|4,   4|4
file 0|4 begin to transfer
file 1|2 begin to transfer
file 2|4 begin to transfer
file 3|4 begin to transfer
file 4|4 begin to transfer
the placement of client0 is finish!

placement phase is finish

-------------------------delivery phase-------------------------
----------------------------------------------------------------
```

Figure 6.5: Placement Phase



```
wubinyi@wubinyi-HP-ProBook-5310m: ~/Kodo-Workspace
-------------------------placement phase--------------------------
------------------------------------------------------------------

the placement situation will be :

client0:    ['0|4', '1|2', '2|4', '3|4', '4|4']
client1:    ['0|1', '1|1', '2|1', '3|1', '4|3']
client2:    ['0|2', '1|1', '2|3', '3|3', '4|0']
client3:    ['0|3', '1|4', '2|3', '3|3', '4|2']
client4:    ['0|4', '1|0', '2|4', '3|3', '4|2']
```

Figure 6.6: Caching Situation

6.2.3 Information of Multicast data

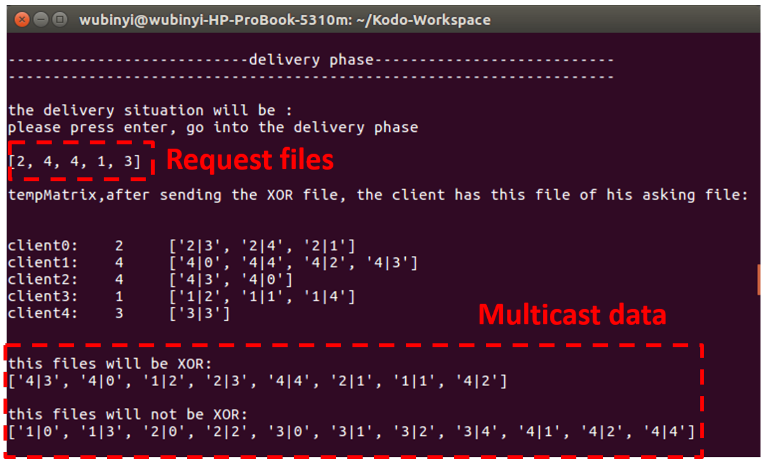
The following figure 6.7 show the requested files from clients and the multicast data.

6.2.4 Information of the First Delivery Phase

The following figure 6.8 show the information in the first delivery phase, for example, the key parameters of KODO and the number of loss packets(the loss rate of packet is 10%).

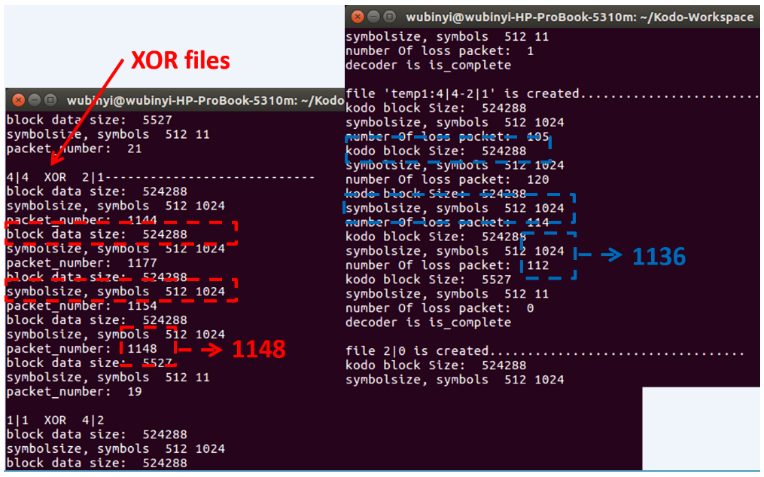
6.2.5 Information of the Second Delivery Phase

In the second delivery phase, server send the original data (without XOR) with KODO. The following figure 6.9 shows the printed information of server and client.



```
--------------------------delivery phase--------------------------
------------------------------------------------------------------

the delivery situation will be :
please press enter, go into the delivery phase
[2, 4, 4, 1, 3]
tempMatrix,after sending the XOR file, the client has this file of his asking file:


client0:    2     ['2|3', '2|4', '2|1']
client1:    4     ['4|0', '4|4', '4|2', '4|3']
client2:    4     ['4|3', '4|0']
client3:    1     ['1|2', '1|1', '1|4']
client4:    3     ['3|3']


this files will be XOR:
['4|3', '4|0', '1|2', '2|3', '4|4', '2|1', '1|1', '4|2']

this files will not be XOR:
['1|0', '1|3', '2|0', '2|2', '3|0', '3|1', '3|2', '3|4', '4|1', '4|2', '4|4']
```

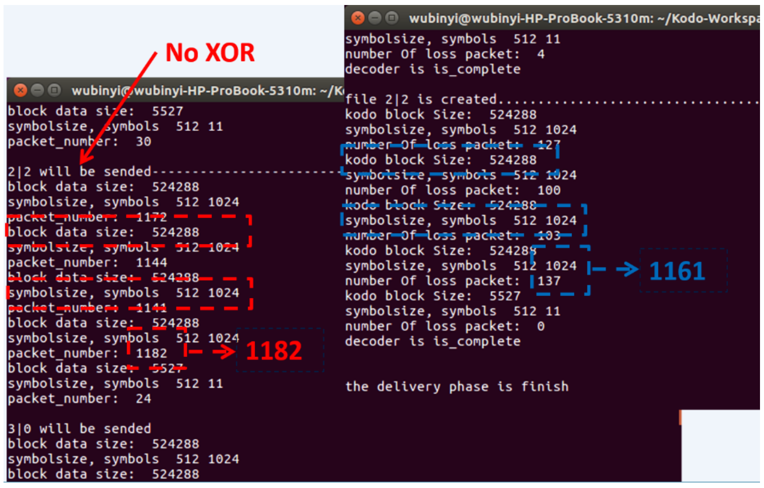
Figure 6.7: Multicast Data


XOR files
wubinyi@wubinyi-HP-ProBook-5310m: ~/Kodo
1148
wubinyi@wubinyi-HP-ProBook-5310m: ~/Kodo-Workspace
1136

```
block data size:  5527
symbolsize, symbols  512 11
packet_number:  21

4|4  XOR  2|1----------------------------
block data size:  524288
symbolsize, symbols  512 1024
packet_number:  1144
block data size:  524288
symbolsize, symbols  512 1024
packet_number:  1177
block data size:  524288
symbolsize, symbols  512 1024
packet_number:  1154
block data size:  524288
symbolsize, symbols  512 1024
packet_number:  1148
block data size:  5527
symbolsize, symbols  512 11
packet_number:  19

1|1  XOR  4|2
block data size:  524288
symbolsize, symbols  512 1024
block data size:  524288
```

```
symbolsize, symbols  512 11
number Of loss packet:  1
decoder is is_complete

file 'temp1:4|4-2|1' is created........................
kodo block Size:  524288
symbolsize, symbols  512 1024
number Of loss packet:  105
kodo block Size:  524288
symbolsize, symbols  512 1024
number Of loss packet:  120
kodo block Size:  524288
symbolsize, symbols  512 1024
number Of loss packet:  114
kodo block Size:  524288
symbolsize, symbols  512 1024
number Of loss packet:  112
kodo block Size:  5527
symbolsize, symbols  512 11
number Of loss packet:  0
decoder is is_complete

file 2|0 is created................................
kodo block Size:  524288
symbolsize, symbols  512 1024
```

Figure 6.8: First Delivery Phase



```
wubinyi@wubinyi-HP-ProBook-5310m: ~/K
block data size:  5527
symbolsize, symbols  512 11
packet_number:  30

2|2 will be sended---------------------------
block data size:  524288
symbolsize, symbols  512 1024
packet_number:  1172
block data size:  524288
symbolsize, symbols  512 1024
packet_number:  1144
block data size:  524288
symbolsize, symbols  512 1024
packet_number:  1141
block data size:  524288
symbolsize, symbols  512 1024
packet_number:  1182
block data size:  5527
symbolsize, symbols  512 11
packet_number:  24

3|0 will be sended
block data size:  524288
symbolsize, symbols  512 1024
block data size:  524288
```

```
wubinyi@wubinyi-HP-ProBook-5310m: ~/Kodo-Workspa
symbolsize, symbols  512 11
number Of loss packet:  4
decoder is is_complete

file 2|2 is created...................................
kodo block Size:  524288
symbolsize, symbols  512 1024
number Of loss packet:  127
kodo block Size:  524288
symbolsize, symbols  512 1024
number Of loss packet:  100
kodo block Size:  524288
symbolsize, symbols  512 1024
number Of loss packet:  103
kodo block Size:  524288
symbolsize, symbols  512 1024
number Of loss packet:  137
kodo block Size:  5527
symbolsize, symbols  512 11
number Of loss packet:  0
decoder is is_complete


the delivery phase is finish
```

Figure 6.9: Second Delivery Phase

6.3 Simulation

The figure 6.10 is the simulation results of the performance of caching for 100 files and 20 clients. The red dash line represents the situation of uncoded caching and without error. The blue line represents the situation of decentralized coded caching without error. The green line represents the ideal situation of decentralized coded caching with error(error rate is 10%). We conclude, the decentralized coded caching is better than the uncoded caching even though decentralized coded caching has a 10% error rate. But we can also see that, there is difference between the red line and the green line. There are two reasons for this difference. The first is that the green line represents the decentralized coded caching with 10% error. The second reason is that the searching algorithm is not good enough.

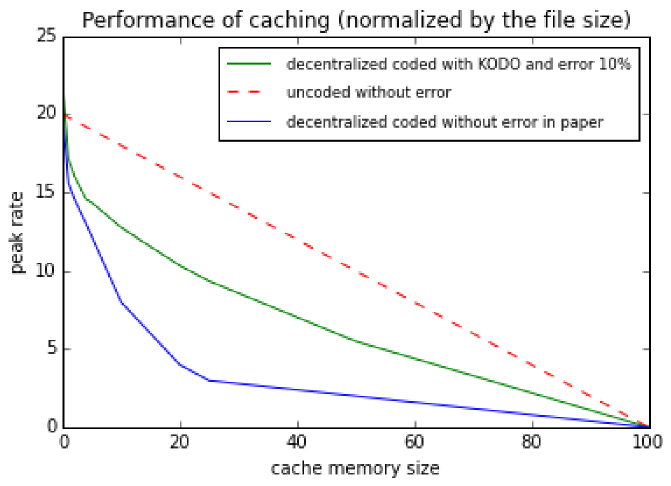


Figure 6.10: Simulation for 100 Files and 20 Clients

7 Conclusion and Outlook

7.1 Conclusion

Decentralized coded caching was proposed in the paper but without considering the errors. In our program we realize this idea and use KODO with errors. As we can see in the simulation, decentralized coded caching is really better than the uncoded caching. In addition to this, we also design the algorithm to search the files, which can do the XOR operation in order to minimize the peak rate.

7.2 Outlook

1. We can still optimize the search-XOR-files-algorithm, because there is a gap between our simulation results and the theoretical value.

2. We should play the videos during transmission.

References

[1] Steinwurf ApS. *Kodo document in steinwurf.* 2016. URL: `http://docs.steinwurf.com/overview.html` (cit. on p. 11).

[2] Mohammad Ali Maddah-Ali and Urs Niesen. "Decentralized coded caching attains order-optimal memory-rate tradeoff". In: *IEEE/ACM Transactions on Networking (TON)* 23.4 (2015), pp. 1029–1040 (cit. on p. 1).

[3] Mohammad Ali Maddah-Ali and Urs Niesen. "Fundamental limits of caching". In: *IEEE Transactions on Information Theory* 60.5 (2014), pp. 2856–2867 (cit. on p. 1).

[4] wiki. *Linear network coding.* 2016. URL: `https://en.wikipedia.org/wiki/Linear_network_coding` (cit. on p. 11).