1 Network

1.x Overview

[...] is constructed as an overlay network on top of some existing network technology, such as the Internet. Throughout this report the underlying technology will be assumed to be the Internet and TCP/IP (IPv4) if not otherwise specified. Data structures in this report are designed for use with TCP/IP and has special fields for things like IP-address and port number but this can easily be modified for other underlying networks technologies.

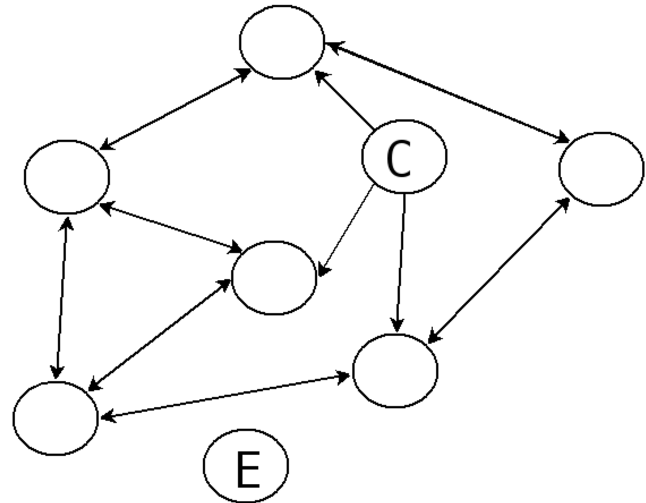

Figure 1: A typical network

The virtual network is constructed as a mesh where any node can connect to any other node in the network. Because of this it might be hard to get a picture about the actual structure of the network and how the data flows from the *content node* (C) to other nodes (N). The content node is the node that creates/has the actual data and inserts it into the network. The network structure is not deterministic because every time a new node joins it randomly selects nodes in the network to connect to.

However, the authors of DONet made an analysis of their network which resulted in that the average distance between the content node and all destinations are O(log N) where N is the number of nodes in the network. The average distance between the content node and other nodes reflects the end-to-end delivery latency and a logarithmic relation between the number of nodes and the radius is rather good. [...]'s network is constructed in a similar way to DONet and should have the same properties.

Apart from the ordinary client nodes that connects to each other and exchanges data there are some nodes with special properties that are required for the network to operate correctly.

The content node as mentioned earlier inserts data in into the network and do not need to download data from any other node, the only purpose of the content node is to provide data to others. From a client node's point of view the content node do not differ from any other node in the network but is just another node like any else.

The *entrance node* (E) is the entry point for all new clients to the network. The client gets a list of existing nodes in the network from the entrance node and will then connect to a random subset of them.

1.x Node management

The entrance node always has a complete list of all the nodes in the network and other nodes has a subset of this list, this list is called the *node cache*.

An entry in the node cache consists of:

- Identifier - 64 bits (assigned by entrance node)
- Internet Address - 32 bits
- Port - 16 bits
- NAT/firewall flag - 1 bit
- Expected amount of descriptors - 8 bits (described later, MDC)

To identify a node in the network each new node that connects to the network is assigned a unique identifier by the entrance node. The address in the underlying network, internet address and port, is also required to be able to establish a connection to it. The NAT/firewall flag indicates whether it is possible to connect to the node or not. It is not always possible to create a new connection because of firewall or NAT. The expected amount of descriptors field will be described later.

A subset of the node cache is the *active set* and the rest is the *backup set*. The active set is a set of nodes that a node has active connections to and exchanged data with. If not enough data is available from the nodes in the active set in a given moment it is possible to expand the active set with nodes from the backup set.

1.x A new node connects

The entrance node is universally known and the location of this must be known before connecting to the [...] network. This can be given to the user as an URL or similar and must uniquely identify the node in the underlying network technology.

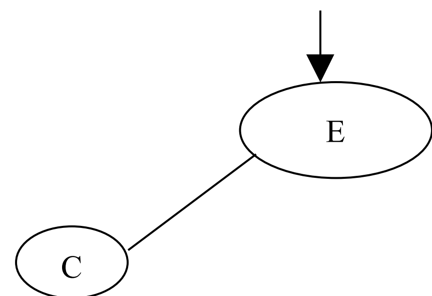

Figure 2: Enter through the entry node (E)

The new node makes a connection to the entrance node (E) and provides some information about itself such as the listening port and the expected amount of descriptors to download. Figure X illustrates this scenario for an empty network. It also requests a list of nodes in the network. The entrance node will return a list of nodes including itself. If the node cache contains to many entries it possible to send just a random subset of the list but that is usually not required since each entry is small.

The entrance node will then make a reverse connection to the new node to check whether it is firewalled or behind NAT. A new entry with information about the new node is added to the node cache of the entrance node and the NAT/firewall flag is set correctly.

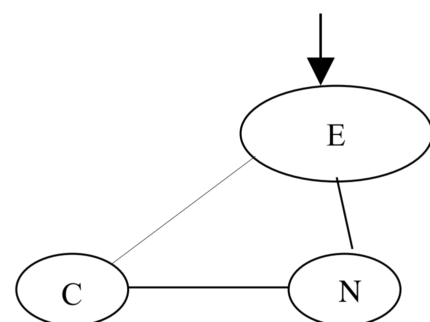

Figure 3: Network after successful connection

Once the new node is registered and has retrieved a node list it will then create an active set from the node cache, this algorithm will be described in section [MDC]. It will create a TCP connection to each node in the active list and exchange data availability information with them. Figure [X] illustrates the network from Figure [X-1] after a successful connection. The remaining nodes from the node cache that are not in the active set will be put in the

backup set.

1.x Network Address Translation
Today many users are using Network Address Translation (NAT) to connect to the Internet. NAT has some nice features such as shared connection to the Internet and improved security but unfortunately also a number of drawbacks. The biggest drawback for a computer using NAT is that it has to configured to allow incoming connections which can be rather complicated for a normal user. A similar problems can also happen if the computer is equipped with a firewall that blocks incoming connections by default.

Nodes that are not connectible has to be treated in a special way and never given to any other node in the network since they are not connectible anyway and useless for a node to try to connect to. A client behind a firewall or NAT always has to be the active part and connect to others that are connectible. But once a TCP connection is made the connection is bidirectional and it is possible to send data in both directions.

1.x Multiple description coding
Multiple description coding (MDC) is a method to split a video into many layers, *descriptors*, and to later use these descriptors to get a working video to display. The good thing about MDC is that it is possible to combine an arbitrary unordered amount of the original descriptors to get a resulting video of varying quality, the more descriptors the better resulting quality. With other words, it is possible to have just a small number or even a single of these descriptors and still get something that works.

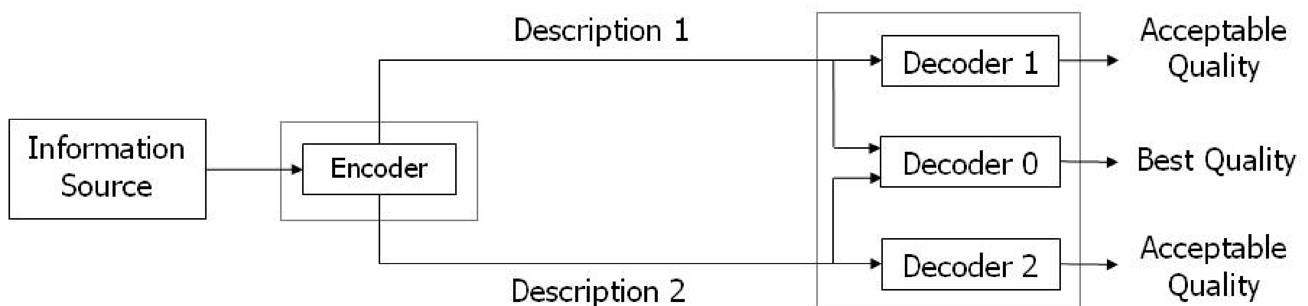

Figure 4: How MDC can be used. Referens !

With help of MDC it is possible to satisfy clients with different bandwidth requirements. A client with high bandwidth simply downloads all descriptors for each segment of the movie and gets the full quality as originally sent by the content node while a client of low quality can select a subset of the descriptors and get a video of lower quality but that requires lower bandwidth. Without this technique the client with low bandwidth would be left out, a client would either have to download everything or not download anything at all.

MDC provides good features for allowing clients with different bandwidth but gives a high impact on how the network should be constructed and how nodes selects their active set. It would be bad for a client with high bandwidth to only be connected to clients with low bandwidth since that makes it harder to download all required descriptors. Nodes that uses all descriptors should therefor prefer to be connected to other nodes that also wants all descriptors. (Also if two nodes with low bandwidth only uses one out of five available descriptors and if these two descriptors are different these two nodes would not be able to share any data at all.)

Vet inte hur MDC fungerar i detalj. Om det är lätt att byta descriptors etc. Men det är lätt att modda. Att ha base layers gör att det e enkelt att använda layered coding om man skulle vilja ha det.

The most logical solution would be to make each node prefer to connect to other nodes that wants to have the same descriptors. An easy way to do this would be to have a logical ordering of the descriptors even tho MDC allows an arbitrary set of descriptors in the end to recreate the video. By having a base descriptor and then a order in which additional descriptors should be downloaded all nodes that have the same amount of wanted descriptors also wants the same actual descriptors. This makes it easy to make nodes prefer to connect to nodes with the same actual descriptors as itself by having it prefer nodes with the same amount of descriptors as itself. The amount of wanted descriptors for all known nodes can be found in the node cache.

It is however not good to have a node only connect to other nodes with the same amount of descriptors, it is preferred not required. Data must be able to flow from high bandwidth nodes to low bandwidth nodes and vice versa. The behavior of preferring nodes with the same amount of descriptors should only be used to a certain level and not for all nodes in the active set. (Behöver utveckla mer, hur väljer vi exakt ut de vi ska föredra? 50% av noderna?)

1.x Node failure

By design any peer-to-peer network will suffer from unreliable clients that connects and disconnects randomly. This is normal and expected behavior and nothing that can be ignored, any sensible peer-to-peer network needs to handle this to be useful. Nodes must be expected to be able to fail at any time, even during active transmission. In [...] all nodes in the network should have some active connection to someone else, otherwise it is not part of the network. This means that as soon as someone fails some other node will detect it after a timeout. That of course assumes that many nodes not fails at the same time and that not all nodes in the failed nodes active set also fails, which unfortunately is not that unlikely. This problem will be addressed later.

To handle failing nodes a *gossip based protocol* is used that forwards news about failing nodes to other nodes. In a typical gossip algorithm a node selects a random subset of it neighbors and forwards the message to them, these node do the same thing in their round etc. To avoid flooding the network the messages has a time to live field that decreases when forwarded. A node that decreases the time to live to zero simply drops the message.

When a node detects that one of the nodes in its active set fails it creates a new gossip message and forwards it to its other nodes in the active set and removes the node from its node cache. The news about the failing node then spreads from node to node until the time to live limit has been reached and all nodes hopefully has got the news and removed the entry about for the failing node from its node cache. This gossip based protocol also has the property of detecting old messages, that is messages that has already been seen. Detecting old messages and dropping them instead of forwarding them helps to reduce the overhead and avoid flooding even better. To do this failed nodes are added to a special *removal set* and will be stored there for a certain amount of time, about 1 hour. Entries in this set contains the identifier of the failed node and a time to live value that specified the time when it should be removed. It is then easy to check if incoming news about a failed node already has been detected by checking for a entry in the removal set with the identifier of the node.

If many nodes fails at once it might happen that the gossip message will not reach all nodes in the network and especially the entrance node can have traces of nodes that have left the network a long

time ago. But when nodes joins the network or requests an updated node cache these leftover nodes that are unavailable can be retrieved. When a node will try to connect to the unavailable node it will detect that the node do not respond and will assume that the node has failed. It will then run the ordinary failed node algorithm for that identifier and will notify other nodes in the network about the failed node.

1.x Graceful disconnect

A graceful disconnects happens when a client shuts down the application correctly and not just cuts the connection without notification or forces the application to terminate. This is similar to how a failed node is handled but instead of other nodes detecting that the node is unavailable the disconnecting node can generate that message itself and send to the nodes in its active set. This means that it is not required to wait for the timeout to trigger before detecting that the node has disappeared and the network should adopt to the new state faster.

2 Data transfer

2.x Send and receive data

Once the network is set up the actual data transfer can begin. [...] has the possibility of downloading from many different hosts simultaneously. To do this the data is divided into *segments*. A segment is slices of data from the stream and is identified by a unique time stamp, a big integer. The segment has one *slice* for each descriptor where each slice has some common predefined size. All slices in the same segment have the same deadline. To identify a slice one first identifies a segment and then uses the identifier of the descriptor to find the slice. Each segment or actually slice can be downloaded individually from an arbitrary node in the active set. Being able to distribute the load and take pieces from different places opens the possibility of achieving good download speed even if a single node is not able to offer full upload speed. The downloaded segments will continuously be put together into a continuous steam that is possible to watch with a media player.

To know which segments can be downloaded from where nodes will continuously exchange data availability information with nodes in their active set. Nodes sends information about which slices are already downloaded and thus available for others to download and which are not yet downloaded. For this purpose a special data structure called a *buffer map* was created. The buffer map is a matrix where each cell (slice) is just a single bit, "0" indicates that the segment is not yet available and "1" indicates that it is available. The real identifier of each segment is of course not just a singe bit, that would just allow a total of two segments in the entire system. To be able to represent a segment with just a single bit it is required to have a "start value" which identifies which segment the first cell in the matrix represents and then each consecutive cell (bit) represents the next following segment in time. The reason this buffer map is represented as a matrix and not an array is to support multiple descriptors (MDC), each column in the matrix represents the same segment and each row represents the same descriptor. The main idea about how the buffer map is constructed is illustrated in *Table X*.

		Segments						
		Segment 1	Segment 2	Segment 3	Segment 4	Segment 5	Segment 6	...
Multiple Description Coding	Descriptor 1	0/1	0/1	0/1	0/1	0/1	0/1	0/1
	Descriptor 2	0/1	0/1	0/1	0/1	0/1	0/1	0/1
	Descriptor 3	0/1	0/1	0/1	0/1	0/1	0/1	0/1
	...	0/1	0/1	0/1	0/1	0/1	0/1	0/1

Table X: Main idea of the buffer map layout, echo cell with 0/1 is a slice

Each node creates it own buffer map specifying with slices it has available and will periodically distribute it to the nodes in its active set. Receiving nodes will then reply with a new buffer map where the segments it wants to download is marked with a "1". Exactly how the decision about where to download individually segments is made is described in the *scheduler*.

2.x Analysis of delay

All tests were made from a cable connection in the Comhem network in Göteborg, Sweden.

Test on delay to a number of hosts:

	rtt-min	rtt-avg	rtt-max
chalmers.se (Sweden)	27.919	30.385	38.823
kth.se (Sweden)	19.583	21.076	23.632
www.tv4.se (Sweden)	14.879	19.392	23.648
www.mit.edu (USA)	118.108	120.599	124.815
www.colorado.edu (USA)	148.619	155.006	158.569
www.u-tokyo.ac.jp (Japan)	276.061	281.144	286.693
www.yahoo.co.jp (Japan)	367.199	370.821	380.414
www.cityofsydney.nsw.gov.au (Australia)	366.889	368.822	370.993
solo.ucc.usyd.edu.au (Australia)	329.792	335.560	344.945
www.adamson.edu.ph (Philippines)	370.597	382.991	394.606

Summary of average RTT:

- Inside Sweden: About 20 30 ms
- To USA: About 150 ms
- To Japan: About 350 ms
- To Australia: About 350 ms
- To Philippines: About 380 ms

The delay in the tests is the round trip time (RTT) which is the time it takes for a package to reach the other end and the response to be retrieved, both ways.

Note that the websites used in these tests are either well known and established universities, companies or organizations and should have good connection to the Internet. End users might have slower connections and higher bandwidth especially if the network is congested when many users use the network at once. But they give a good general idea about what delay one can expect to different countries and should in normal cases not differ that much from the test values.

The goal of these tests was to analyze the average delay to different parts of the world to see what can be an expected average delay. In the worst case the delay was 380 ms to the Philippines which is almost as far away as one can come from Sweden so the delay should not be much higher than that on a non congested network. In general to make it easy the delay can be said to be not higher than 500 ms but in most cases about 100 200 ms.

2.x Scheduler

The scheduler is one of the most complex parts of the system and is responsible for deciding where and when each segment and slices should be downloaded. As input the scheduler requires a set of buffer maps and the respective node identifier for each buffer map. The buffer map specifies which slices each node can offer, combined these buffer maps tells for the receiver where each segment can be downloaded. To make a schedule the scheduler considers the amount of available clients for each slice, the end-to-end properties of clients, such as bandwidth and network delay, and the deadline for when each segment must be available. Fulfilling all these constraints at the same time and create a optimal schedule is a hard task, especially considering the heavily changing network topology in a peer-to-peer network. In fact this problem is a variation of *Parallel machine scheduling* [REF TILL DONET SCHEDULING] which is known to be NP-hard [25, DONet paper]. Instead of trying to create an optimal solution the proposed scheduler uses heuristics to try to create a "good" schedule.

One of the most important constraints that need to be fulfilled is to prioritize segments with few potential suppliers but at the same time segments with deadline close in time should also be given high priority. Segments with few potential suppliers are important to prioritize so they can allocate bandwidth before others segments with more suppliers take it, if not segments with few suppliers can end up with no one to offer the data at the end. Segments with a deadline close in time is important to get before segments with deadline further ahead in time so the movie can continue to play without interruption.

The proposed scheduler first priorities segments with few suppliers and then segments with deadline close in time. This is based on the belief that it is more important to allocate slots to segments with few suppliers so they not are starved than to try to download the segment with closest deadline, in the end all segments are needed anyway. To do this the scheduler makes a sorted list that is first sorted by the number of suppliers and then by the closest deadline in time. It will then take segment after segment in order and for each assign the best supplier. The selection of the best supplier is based on the bandwidth and the network delay. Of all clients that has enough left over bandwidth to handle the segment the one with the lowest delay will be used.

The network delay is easier to achieve than the bandwidth which usually is more varying but both of them do vary with time. The scheduler needs to continuously analyze the performance with each node in the active set and try to figure out how much bandwidth it has available for use and see if the delay varies. The network delay can be found by sending a package back and forth and compute the difference between the time the package was transmitted and the response retrieved but the bandwidth parameter required more sophisticated methods to achieve.

To estimate the available bandwidth a TCP like algorithm is used. The bandwidth in TCP is the same thing as the number of packages that can be sent at once and the same idea can be used for segments instead of packages. TCP uses a so called "slow start" algorithm that first increases the bandwidth exponentially but halves the bandwidth and increases the bandwidth linear after package loss is seen (congestion avoidance). During the exponential increase phase the number of packages that can be sent simultaneously are increasing by 1 each time a package is transferred completely. But during congestion avoidance phase the number packages are increased by 1 for each round-trip time, that means for example that if 10 packages were requested the number of packages are increased by 1 after all 10 are transferred completely. This method can fairly easy be adapted to handle segments instead of packages. A segment can simply be seen as a package and then the "slow start" algorithm works more or less unmodified. The scheduler just need to keep the state of each node that it transfers data with and

update the status when segments are transmitted and on other similar events that affects the scheduling.

Once download nodes are assigned to all slices the actual data transfer can begin and the target nodes are notified about which segments the client wants do download from them. The buffer map can be reused for this purpose. All slices that will be downloaded from the same node will be marked in a buffer map and this buffer map will then be sent to the target node. The target node checks in the buffer map which segments are about to be sent to the requesting client and the data transfer can begin.

Pseudo code for the scheduler:

```
Input: Map of Node identifier -> Buffer Map
Input: Active set of nodes

Map slices = map of (segment identifier, descriptor identifier) ->
set of node identifiers that can offer the slice
Sort the map segments by the number of suppliers then by closest
deadline

Set clients = set of nodes in active set
Sort the set clients by the shortest delay

for each slice from slices do
     client = best from clients set that has free bandwidth
     client.assignSegment(slice)
     client.reduceBandwidthWithOneSegment()
     segments.remove(slice)
end for

for each client from clients
     Set slices = client.getAssignedSlices()
     BufferMap bm = new BufferMap(slices)
     client.setBufferMap(bm)
end for
```

Senare nämna hur vi moddat schedulern, den är väldigt lik. Men var dåligt med detaljer exakt hur de gjorde I artikeln så vi fick komma på massa saker själva osv.

2.x Lack of segments

If a client detects that it is impossible to retrieve all segment from the nodes in the active set it can expand the set with new nodes from the backup set. If that is not enough a node might have to go back to the entrance node and request a new list of peers to expand the backup set with. When at least one additional node is found in the backup the standard node selection algorithm is used to select some additional nodes to add to the active set, the same algorithm that was used when first connecting to the network. It then need to exchange the buffer map with the new node(s) as usual and try to schedule again.

Except for adding new clients it is also required to be able to remove bad clients in order to keep a

reasonable amount of active nodes. Nodes are scored by how data they have transmitted, in both directions. How much data is transmitted between the nodes in the active set can be seen as the number of successful simultaneously segments that have been sent/received at once. The number of received segments at once from each node in the active set is easy to get since that is used in the scheduler and it is possible to just fetch that value but the number of sent is not used anywhere else. But it is possible to store the number of marked segments in the buffer map that is received from nodes in the active set and store that value somewhere. The maximum of the sent and the received number of simultaneously segments is used as a score the one with the lower score can be discarded. This way the one discarded is not used much for either transmission or retrieval of data and can rather safely be removed.

2.x Multiple description coding

Hmm, behöver vi skriva nåt om detta här egentligen? Vi har ju inga direkta detaljer om hur MDC och crap fungerar... men kanske kan vara skoj att diskutera lite vad man kan göra med datan här och kanske använda LC istället och sånt.

2D Buffer Map