Integration of Mechanisms for Video Distribution between Clusters in a Hierarchical P2P Architecture

Abstract. Video distribution and video streaming have grown dramatically during the last years in the digital era. Every day people streaming video from different sites or sharing multimedia content with friends or coworkers. This paper presents an integration of different mechanisms for video distribution between clusters which are organized on a hierarchical P2P architecture. Inside of this architecture, peers are organized into small clusters which need interchange video blocks. After the contents have been distributed to all peers in a cluster, all these peers can work as new source nodes for lower clusters in the hierarchical structure. In this paper are described the flow control, caching and redistribution mechanisms, and how they work together for the video distribution.

Keywords: Peer-to-peer networks, video streaming, content distribution.

1 Introduction

Video streaming has gained a high popularity during the last years. Many video streaming solutions have been deployed using Application-layer multicast as an alternative to IP Multicast. In an application-layer multicast all multicast tasks are implemented at the end-hosts exclusively while the network infrastructure is maintained. Clustering has received considerable attention in the scientific community [1], [2], [4], [5], [6], [12]. Clustering plays an important role in many large scale distributed systems [12]. Complex networks such a social networks or the World Wide Web exhibit a high degree of clustering and scale-free [1], [3]. Perhaps, P2P networks are a clear example of complex networks. The hierarchical collaborative multicast scheme [7] assumes that clusters with high interconnected nodes combine into each other in a hierarchical network. Hierarchical collaborative multicast first forms a hierarchical structure and then evolves into meshed clusters. In hierarchical mode, data must be distributed from one cluster to another, from top to bottom. A peer may belong to two clusters located on different layers of a hierarchical tree. Clusters are an elementary unit of hierarchical architecture, which involves one source peer and several requesting peers. The peers inside a cluster are fully connected. Each peer inside a cluster is a receiving and forwarding peer at the same time. Due to the fact that the upload capacity of all peers is also used, the bandwidth consumption from the source can be reduced. The requesting peers form a small cluster with their neighbors based on proximity. Each cluster is represented by a cluster head, while the other nodes close to the cluster head are integrated into the cluster. The flow control, caching and redistribution mechanisms are integrated together for the video transmission between clusters in the system [13]. This paper describes the operation of these mechanisms.

The rest of this paper has the following organization. Section 2 explain the flow control mechanism. Operation of the caching mechanism is presented in Section 3. The redistribution mechanism and its operation is introduced in Section 4. Integration of these mechanisms for the clusters in the hierarchical architecture is explained in Section 5. This article concludes in Section 6.

2 Flow Control Mechanism

In a multicast system, the content are originated from the source, and therefore the upload capacity of the source has become the most important resource. This work assumes that the upload capacity of each peer is the only constraint. This assumption is motived by the fact that peers usually have larger download capacity than upload capacity (e.g. DSL lines) on the Internet. Thus, the flow control mechanism is based on the upload capacity of the source and the requesting peers. Initially, the source does not know the upload bandwidth of all requesting peers. Under this scenario, the source cannot send content to all peers as quickly as possible. The reasons are the following. First, if the download capacity of a peer is much greater than its upload capacity, the packets may be overstocked, and this peer must take more and more space to store packets in the memory. Second, a peer may have a much broader upload capacity and forwards the packets immediately, and mostly it waits for the packets from the other peers. For example, the source sends packet 301 to peer R1 and packet 302 to peer R2. Peer R2 has abundant upload capacity, while R1 has a scarce upload capacity. Peer R2 forwards packet 302 to the other peers immediately, while it is still waiting for the packet 101, which was previously received by peer R1 (before packet 301). Therefore, peer R2 must also store packets in the range from 102 to 302 in memory and cannot play the content from packet 101 even it has received packet 302 already.

To deal with this problem, the flow control mechanism assumes that a peer does not read packets from the source link, if one of the forwarding buffers is full. This scenario is shown in Figure 1. Here, the forwarding buffers of node R1 to R2 and R3 are full. The peer R1 stops reading packets from source S. Then the sending buffer of the source to the peer R1 will also be full soon, and the source will not load packets to this buffer any more. On the other hand, the peers R2 and R3 have large upload capacity and are reading packets from their sockets very quickly. Thus, the source can rapidly load packets to its other two sending buffers and send them to the peers R2 and R3. Following the sequence number of packets in the buffers, the source sends packets much faster to peers R2 and R3 than to peer R1. In figure 1, the packets 316, 317 and 318 are sent to peers R2, R3 and R1, respectively. However, in the following round, the source send packets 319 and 320 to peers R3 and R2, respectively. Also, the packets 321 and 322 are sent to peers R3 and R2. In this way, the source distributes more packets between the peers with large upload capacity. If each forwarding buffer is still available for more packets, the corresponding bit in the socket related to source will be set in descriptor, which will be checked by the source. If the bit is still set, the peer will read data from the socket of the connection from the source until the whole packet has been received. Then, this packet will be put into each buffer of the forward link, to be forwarded to other peers in the same cluster. The payload of this packet is stored in the memory block.

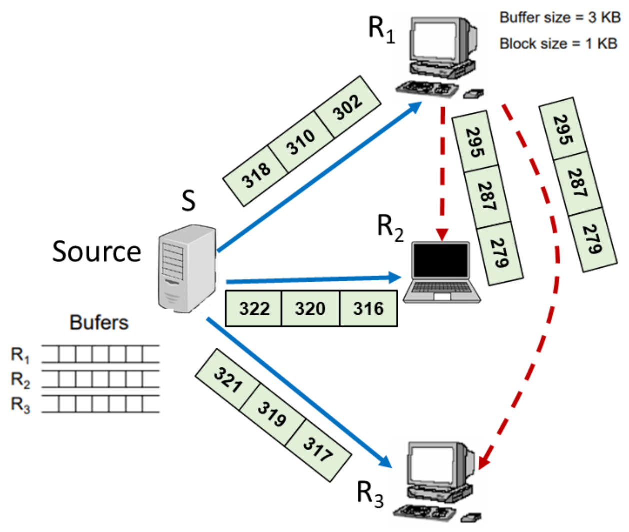

Fig. 1. Buffers for redistribution in the flow control mechanism

Another scenario is the loss of the uplink. If a peer stops on the uplink, the other peers can continue transferring. And these peers will receive all packets with continuous sequence numbers after packets they have not yet received, which are still in the blocked peers' buffers. The stream on them can also continue playing if a small jump of several packets is carried out. This is a dynamic flow control idea. It can efficiently utilize uplink performance and minimize storage cost on peer nodes.

3 Storage mechanism

In a hierarchical collaborative multicast, the requesting peers need to efficiently store the payloads of the packets received from the source and other peers in the same cluster. This task can be done by a caching mechanism which must store contents in separate storage according to the ID of the source and the requesting peers. This ID is labeled in the header of the packet. The design assumes that there is enough space to store the content. In each peer, the payload of the packets should be sorted by sequence number and stored continuously. However, two problems are present: media streaming size is unknown and the packets are arriving out of order from different peers. Arrays can solve these problems. They are a standard solution used for storing data. An array is a structure consisting of a group of elements that are accessed by indexing [9], [10]. This allows for very fast access because the code can do a little jump quickly to any location in the array, and the elements are all grouped together so they tend to be in memory at the same time [8]. However, the size of the array must be fixed and predefined before distribution, and is impossible to extend its size later. On the other hand, in a list, there is no fast way to access the N-th element, but the size is easy to append, which means the size does not need to be predefined before the distribution. In hierarchical collaborative multicast, the size of the content is unknown and an array can be used to store the packets according to the sequence number that is specified in the packet header. Thus it is not suitable for storing large size content. Multimedia streaming is played frame by frame, as soon as each frame is completely received, it can be read out of the buffer for playback, and the space of this frame in the memory can be released. Furthermore, the order of the incoming packets depends on the upload bandwidth of the redistributing peers, and is most probably out of order. The size of the buffer should be dynamic in order to optimize the efficiency of the memory usage.

Figure 2 shows how the arrays and double-linked lists are employed to store the received packets. When using double-linked lists, a block can be inserted or removed in the middle of the list, or added at the beginning or the end. After removing or inserting that block, the linked-list should be reconnected, and the size or the index updated if necessary. In addition, the size of the list can be specified or unlimited. This mechanism minimizes the usage of memory space, and it is suitable for application of playing the stream during receiving.

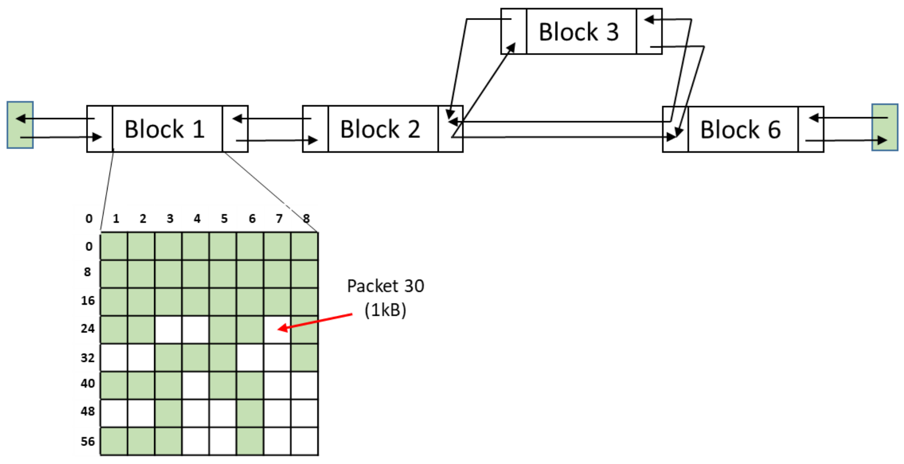

Fig. 2. Storage blocks

In a cluster, each requesting peer receives packets from the source (via a direct link) and other members in the same cluster (via forwarding links), which have different upload capacity. The packet distance shows the difference of performance between these route terms of memory usage. A large packet distance indicates that a packet must wait a long time in the memory before it can be forwarded. So, the larger the packet distance, the larger the memory usage in each peer. The flow diagram in Figure 3 shows the steps for storing packets in the storage blocks.

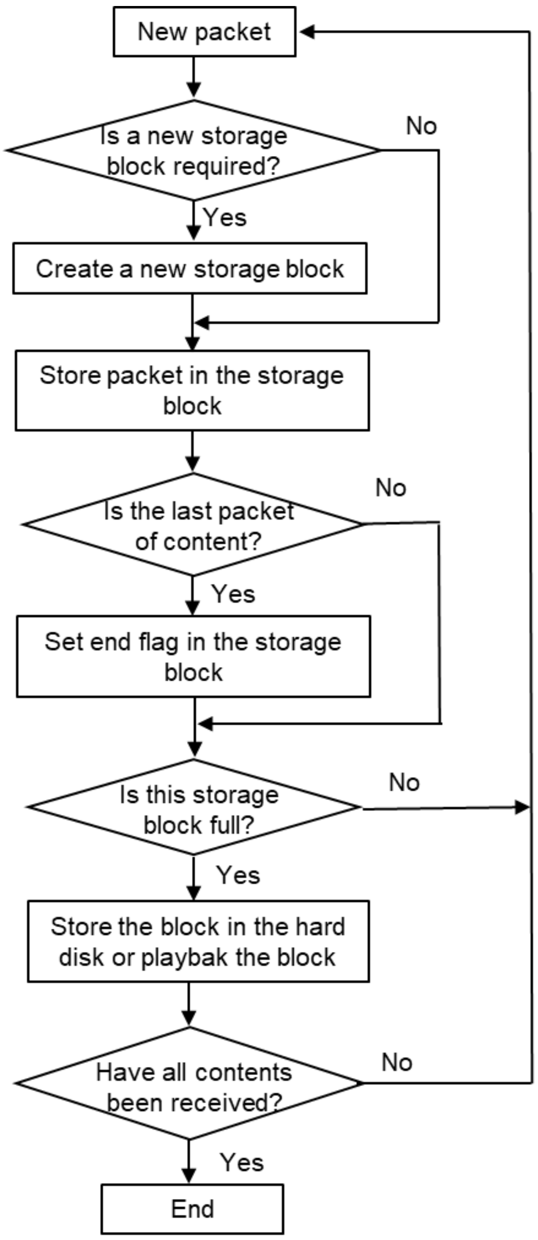
New packet
Is a new storage block required?
No
Yes
Create a new storage block
Store packet in the storage block
Is the last packet of content?
No
Yes
Set end flag in the storage block
Is this storage block full?
No
Yes
Store the block in the hard disk or playbak the block
Have all contents been received?
No
Yes
End

Fig. 3. Flow diagram for the storage process

Once a new packet is received, the peer checks first the sequence number that is specified in the packet header, then determines the position of the packet inside the block. Although at the source the packets are distributed in order for the requesting peers, the packets are received via different links and due to the different network capacity, they typically arrive out of order. If the specific block does not exist, the peer creates a new block, and adds it at the correct position within the double-linked list. After storing the packet the peer checks whether this block is full. If the counter is equal to the number of packets defined for this block, then the block is full and its index must be checked. If the index of the current full block is the smallest, then, this block is the first block of the list (or the previous blocks have been removed), and the data in this block can be read out. After this, this block can be removed from the list, and the peer sets the index of the next block as the smallest.

4 Redistribution Mechanism

In the hierarchical architecture, each requesting peer receives packets from the source and from the other peers in the same cluster. Each peer forwards the packets received from the source to the other requesting peers in the same cluster. If the requesting peers are cluster heads then they redistribute the received blocks to their child-clusters.

A redistribution to the children after the whole content has been received is not feasible, because the playing time difference (redistribution delay) between two levels become too large and the nodes need too much space to store the content. Furthermore, the last level peers experience excessive delay before they can start playing the content (potentially in the order of minutes or even hours). The required time T_{rc} to download all content to the top cluster from the source is given by

$$T_{rc} = \frac{\mathit{Content\ size}}{\Theta} \qquad \textbf{(1)}$$

where Θ represents the overall throughput of the system.

In contrast, the redistribution mechanism approach based on streaming avoid large delivery delays between clusters, and allows that the cluster heads stream the content to their child-clusters as soon as a block is ready. In this case, the delivery delay T_{rb} is given by

$$T_{rb} = \frac{\mathit{Block\ size}}{\Theta} \qquad \textbf{(2)}$$

again, Θ represents the overall throughput of the system.

From equation 2, it becomes obvious that playout delay becomes much shorter using block distribution, because the size of a storage block will typically be much smaller than the whole content.

The hierarchical collaborative multicast uses redistribution queues to handle anomalies such as packet loss and network congestion during content distribution. The flow chart for the forward link is shown in Figure 4. In this scheme, a data block is removed by a peer from its incoming link in each iterative loop. This block is copied onto the outgoing links connecting all the other requesting peers. The next block from the incoming link is not removed until the last data block has been successfully copied onto all delivery links. This forwarding scheme is used by a peer to redistribute the block received from the source to all the other peers in its cluster. If the peer is a cluster head, then the peer collects the blocks from all peers in the top cluster and sends these blocks to its child cluster as a source peer. Thus, in order to reduce the content receiving delay between a cluster head and its child-peers, the cluster head peer should send the content via the forwarding link after receiving some blocks instead of receiving all content from its parent peer or the root source.

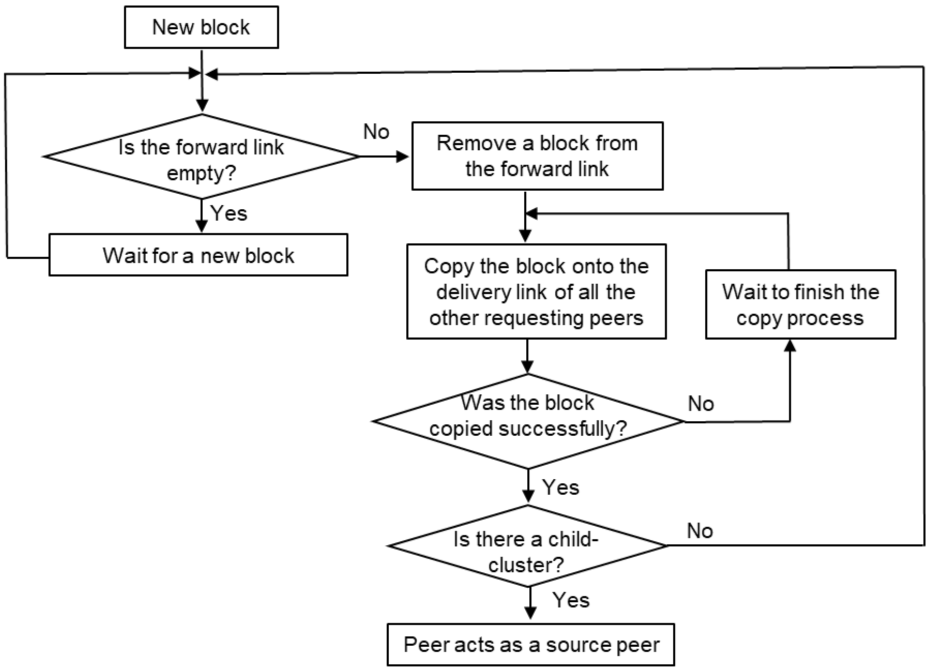
New block
Is the forward link empty?
No
Remove a block from the forward link
Yes
Wait for a new block
Copy the block onto the delivery link of all the other requesting peers
Wait to finish the copy process
Was the block copied successfully?
No
Yes
Is there a child-cluster?
No
Yes
Peer acts as a source peer

Fig. 4. The operation flow of the forward link on requesting peer.

In the delivery links of the requesting peers a block can arrive from other peers into the same cluster or from the source. Blocks to be redelivered to the other requesting peers have a higher priority than the blocks to be consumed in the local peer.

5 Integration

For the integration of these mechanism, we consider that all the peers (except the root source) send and receive packets at the same time, and the distribution of blocks among the requesting peers is implemented using threads. In computer term, a thread is a lightweight process, which is smaller, faster and more maneuverable than a traditional process [11]. In this work, the threads are used in order to ensure distribution and storage without delay. Each node runs 3 threads in parallel: sending, receiving and measurement. According to the task of each peer in the system, they are classified as source, requesting peer or cluster head. A peer can be a source and request peer at same time as is shown in Figure 5. In this case, the peer first works as requesting peer (R), and as source peer (S). In order to reduce the content receiving delay between this peer and its child-peer, the content should be sent to child-peers after receiving some blocks instead of receiving all content from its parent peer.

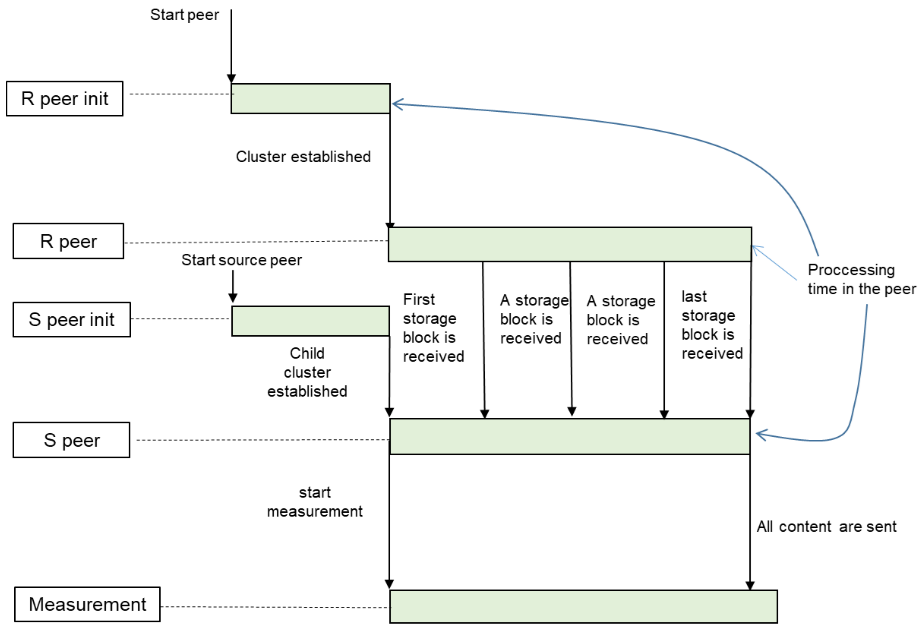
Start peer
R peer init
Cluster established
R peer
Start source peer
S peer init
Child cluster established
First storage block is received
A storage block is received
A storage block is received
last storage block is received
Proccessing time in the peer
S peer
start measurement
All content are sent
Measurement

Fig. 5. Threads in a peer working as source and requesting peer at same time

A sender module runs at the root source, which is located at the top of the hierarchical architecture. This module does not receive any content, and sends content only. The sender module includes two threads. Figure 6 shows this scenario. The first thread called source thread delivers content, while the second thread, called measurement thread measures the upload throughput. The sender thread performs the initialization of the source and distributes the content to the requesting peers after the cluster has been established. Meanwhile, measurement thread starts measuring the upload throughput until it receives an asynchronous ending message from the sender thread.

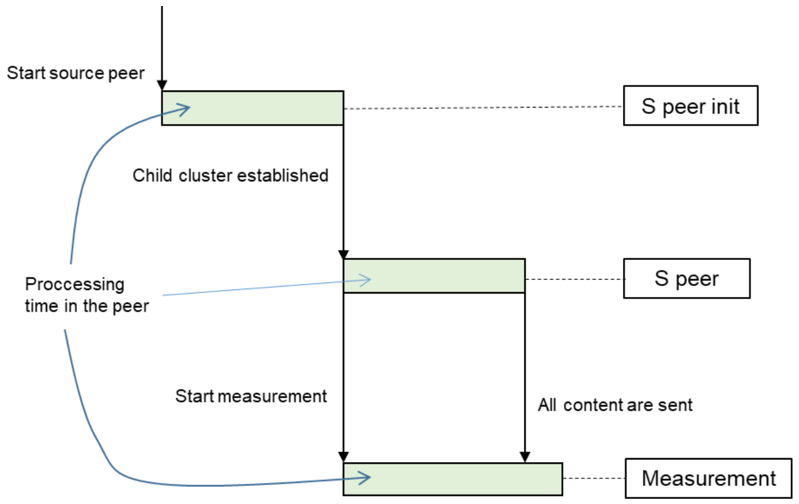

Fig. 6. Threads on a sender module.

A receiver module runs on every requesting peer, which has no child-cluster. This request module has only one thread, which receives and forwards the content received from the source to the rest of the peers in its cluster. A cluster head peer works as a requesting peer and server peer (as it shown in Figure 5). To this end, a cluster head module is run by each cluster head peer. The cluster head module combines "sender" and "receiver" modules. Thus, three threads called source thread, requesting thread and measurement thread are enabled in each cluster head. The communication between threads in provided by the redistribution queue. The requesting thread puts the storage blocks into the queue and the source thread redistributes them to the child-cluster one by one.

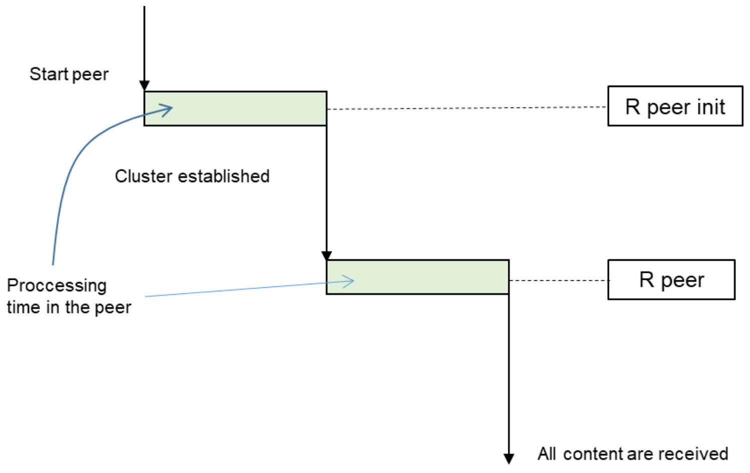

Fig. 7. Threads on a requesting peer.

To run the protocol, initially, the source starts listening on a pre-defined port and waits for the socket connection request. Then, each requesting peer establishes a connection with the source. The source copies the IP address and listening port of every requesting peer and sends this information to all requesting peers. The cluster initialization takes place in the very beginning, in order to define the hierarchical collaborative structure to be used. To this end, each requesting peer uses the IP address and Round Trip Time (RTT) to calculate its proximity to the rest of the requesting peers. The local clusters can be organized by combining proximity information with information about the upload capacity of the peers. Each peer starts a new thread, which listens and waits to establish the forward link for transferring content between two peers. All peers maintain a list of all peers in the cluster. So the forwarding connection could be established very quickly. After the initialization of the forward link, an initialization finished signal is received from the source indicating that the cluster is initialized and all links are ready for content distribution. Then, the source and the requesting peers begin the data transfer.

The source has a connection to each requesting peer, and a sending buffer is associated to each connection by the source. In these sending buffers, the source loads packets from its hard disk and sends these packets through TCP connections. After the blocks are received at each requesting peer, it forwards the block to the rest of the peers in the cluster. If the requesting peer is a cluster head, it organizes the received blocks and forwards them to its child-cluster. For this task, the requesting peers check each forwarding buffer. If forwarding buffer is still available for more packets, the source or cluster head will be informed via the socket descriptor. The source checks this information and the next packet is put into each buffer of the forward links. The block size is set to 1KB, so each block can be sent using a single TCP/IP packet. The buffer size is set to 3 KB, which corresponds to 3 packets. A loading pointer indicates how many packets remain in the buffer and how much space is available.

6 Conclusions

Video has been present as an important media of entertainment and communication in the society from many years. However, the recent advances of the Internet and computing technologies have opened up new opportunities to multimedia applications, where video delivery and streaming over the Internet has gained significant popularity. In this paper are presented different mechanisms for video transmission between clusters deployed on a hierarchical architecture. These mechanisms are used for flow control, caching and redirection of video between the clusters. These mechanisms are integrated into a hierarchical collaborative multicast scheme for video delivery from one source to multiple receivers using peer-to-peer (P2P) networks. Tree and mesh structures are combined for video delivery. The hierarchical structure organizes all clusters as a scalable distribution tree. Requesting peers are organized into small meshed clusters, which are hierarchically located through a unique distribution tree. Content is distributed from the root source to every peer through the tree. If a requesting peer has a child-cluster, then this peer is called a cluster-head peer. A cluster head peer and its child-peers comprise a distribution cluster. The hierarchical clustering is based on the upload capacity of the requesting peers and their mutual proximity. In each cluster, all the participating peers are fully interconnected, and they are in fact receivers and senders at the same time. Integration and interoperability of the flow control mechanism with the caching and redistribution mechanisms are very important in the video distribution between the clusters.

References

1. Lee, C., Kang, S-G. and Nayyar, A.: Location-proximity-based clustering method for peer-to-peer multimedia streaming services with multiple sources. In: Multimedia Tools and Applications, Vol. 81 pp. 23051-23090 (2022)
2. Xu, R. and Wunsch, D.: Survey of Clustering Algorithm. In: IEEE Transactions on Neural Networks, Vol. 6, Num. 3, pp. 645-678 (2005).
3. Barabási, A. L. and Bonabeau, E.: Scale-free Networks. In: Scientific American, pp. 50-59, (2003).
4. Azimi, R., Sajedi, H., Ghayekhloo, M.: A distributed data clustering algorithm in P2P networks. In: Appl. Soft Comput 51(Supplement C):147–167 (2017)
5. Hammouda, K. and Kamel, M.: HP2PC: Scalable Hierarchically-Distributed Peer-to-Peer Clustering. In: Proc. of the Seventh SIAM International Conference on Data Mining (SDM07), Minneapolis, MN, (2007).
6. Liang, C., Guo, Y. and Liu, Y.: Hierarchically Clustered P2P Streaming System. In: Proc. of the IEEE GLOBECOM 2007, pp. 236-241, Washington, DC, USA, (2007).
7. Lopez-Fuentes, F. A. and Steinbach, E.: Collaborative Hierarchical Multicast. In: Proc. of the 15th ACM Multimedia Conference (ACM MM'07), pp. 763-766, Ausburg, Germany, (2007).
8. Stevens, W. R., Fenner, B. and Rudoff, A. M.: UNIX Network Programming. Vol. 1, (2003).
9. Wirth, N.: Algorithms and Data Structure. Prentice Hall, (1985).
10. Aho, A. V., Ullman, J. and Hopcroft, J. E.: Data structures and Algorithms, Addison Wesley, (1983).
11. Butenhof. D. R.: Programming with POSIX threads. Addison Wesley Longman, (1997).
12. Demirci, S., Yardimci, A., Sayit, M., Tunali, E. T. and Bulut, H.: A Hierarchical P2P Clustering Framework for Video Streaming Systems. In: Comput. Stand. Interfaces, vol. 49, pp. 44–58, (2017).
13. Lopez-Fuentes, F. A.: Video Multicast in Peer-to-Peer Networks, PhD thesis, Technische Universität München, (2009).