ForestCast: A Central Solution to Heuristically Constructing Trees

Ali Ghodsi
Swedish Institute of Computer Science
Box 1263, 16429 Kista, Sweden
ali(at)sics.se

Seif Haridi
KTH – Royal Institute of Technology
Electrum 229, 16440 Kista, Sweden
haridi(at)kth.se

June 1, 2007

SICS technical report T2007:12
ISSN 1100-3154 ISRN:SICS-T–2007/12-SE

1 Introduction

The goal of this paper is to outline a tentative and simple solution to building multicast trees for live video streaming. This problem is challenging as one needs to:

- Maximize the total utilization of upload bandwidth
- Minimize latency
- Dynamically reconfigure the trees during network dynamism

The first requirement stresses that the solution needs to ensure that the *actual* available upload bandwidth at each node should be utilized as much as possible. Any solution must, therefore, adapt to the given upload bandwidth of the individual nodes. This implies that even nodes with petty upload bandwidth should be utilized (see Section 2.5 on multiple streams).

The second requirement puts focus on latencies between the actual nodes. It also implies that the depth of the multicast trees should be shallow to minimize latencies (this is at least true given identical node latencies).

Finally, the solution should continuously reconfigure the system, as there will always exist some network dynamism. By network dynamism we refer to i) nodes joining/leaving/failing, or ii) network capacity changing due to network congestion etc.

1.1 Outline

Section 2 outlines the basic centralized solution. It will also discuss how multiple streams, using MDC or layering, can be used to build multiple trees. Finally,

Section 3 hints how the centralized elements can be distributed to make the solution scale.

2 ForestCast

Nodes can have three roles:

- *source* – the node which has the video to be streamed
- *server* – central server that constructs the trees
- *peer* – a node (customer) which downloads and/or uploads the stream

The server will have complete information about every peer. Each time a peer joins the system, it will contact the server and provide it with its own bandwidth (up/download) capacity. The server will also approximate the latency of the peer, e.g. it can use synthetic coordinates [1] to approximate the joining peer's latency. The server will inductively construct trees, which it maintains as peers join, leave, and fail.

Multicast Trees The existing multicast trees will only consist of peers. Hence, we assume that the root of a tree receives its video stream from the source, which is not part of any multicast tree. We assume that the video stream has a minimum bandwidth requirement, M. If a peer i has total bandwidth B_i, we say that peer i has $\left\lfloor \frac{B_i}{M} \right\rfloor$ *slots* available. Nodes that have zero slots available are referred to as *closed*, while nodes with more than zero slots are referred to as *open*. Certain peers might always be closed, as their number of available slots is always zero. We refer to such nodes are *clients*, since they can only download, and not upload. Hence, the server knows at all times the shape of all the trees, as well as number of slots available at each peer, and the latency to each peer. We assume that all peers have at least M download bandwidth, such that they can receive the live stream. Nodes that have less than M download bandwidth are simply excluded from ForestCast, as they do not have sufficient download capacity to view the live stream. Furthermore, if there are no open peers in any tree, the server will start a new separate tree, which is feeded by the source directly.

Latencies. Peers can measure their latency to other peers. We denote the latency between i and j by $l(i, j)$. The *total latency* of a peer is $TL(i) = l(i, j) + TL(j)$, where j is i's parent in the tree, in which it participates. The peer that is the root, r, of a tree has $TL(r) = l(r, source)$. The server knows the total latency at each peer. This information is updated each time a tree changes.

2.1 Join Procedure

It is the job of the server to decide from which node a joining peer should receive its live stream. This decision will be based on the existing trees and the information (e.g. total latency and open slots) about the joining peer. How this decision is taken, will determine the efficiency of the system. We provide some basic approaches, which we believe can later be improved.

Breadth First Positioning. In breadth first positioning, the server makes a list of all open peers in all the trees. From that list, the server picks those peers that are positioned lowest in the multicast trees (a root is at depth 0). From the remaining list of peer, the server can pick peers to either minimize latency, or to increase reliability. To minimize latency, the server picks the peer with the lowest total latency, and lets the joining peer become that node's child. The increase reliability, the server picks the peer with fewest children, and lets the joining peer become that peer's child. Hence, reliability is increased, as "super-peers" are avoided so that the failure of a single node will result in few children becoming orphans. The breadth first positioning attempts to keep the depth of the trees low, thus minimizing the total latency of the peers.

Fair Positioning. An alternative approach would be to first determine whether the joining peer is a client or not. If the joining node is not a client, it is placed according to breadth first positioning. If the joining node is a client, it should be placed in the highest depth, i.e. become the child of the open leaf with lowest latency. This approach is more "fair", as nodes that cannot contribute are placed deep down in the tree, while non-clients are placed such as to minimize the depth of the tree.

A similar approach is to let the joining peer's available slots determine its position in the tree. Hence, peer's with many open slots would tend to end up near the root of a tree.

Greedy Positioning. In greedy positioning, the server tries to give the joining node the best possible latency. This is achieved by letting the new node become the child of the node with lowest total latency. Note that this could be hard to achieve, as the latency between the joining peer an the other peers varies. Synthetic coordinates could be used as an approximation. Alternatively, the server could pick a small subset of candidate peers, which the joining node can measure its latency to.

2.2 Leave Procedure

Graceful leaves are always signaled by the leaving peer to the server. A leaf can leave the system as soon as it has signaled its departure to the server. As it comes to peers, the most simple approach to deal with leaves is for the server to let the leaving node's children rejoin the tree. Since the server has full information about the tree, it can ensure that the children do not rejoin at a

node that happens to be a descendant of the leaving node. After the children have rejoined the tree and a handover has been done, the leaving peer will be a leaf in the tree, and can thus leave the network.

The above described method to deal with leaves can be fine-tuned. For example, whenever a node requests to leave, the server attempts to determine whether the leaving peer has a child which has enough slots to feed the rest of the leaving peer's children. If so, the leaving node is essentially "replaced" with one of its children, which then feeds the other children. If no such powerful child exists, the more generic method of rejoins can be used.

Note that peers usually have ample download bandwidth, which is under-utilized. Whenever a peer is rejoining, it could make use of that by receiving the same stream in parallel from its current parent, and its new parent. Only when the new parent is providing the stream, the peer can disconnect from its previous parent.

2.3 Failure Handling

The system needs to ensure that a fail-over is transparently done whenever a node fails. We describe to basic approaches. In the naive case, the children of a failing node rejoin the tree as described in the case for graceful leaves. Note that we assume that the server signals to its children whenever it is lagging and has no more data to send to its children. Thus, we assume that a child can distinguish between the failure of its parent, and the case where its parent is alive, but has no data to pass on.

To facilitate transparent fail-overs, the server could upon the join of a peer always assign to it a *backup parent*. This could be the joining peer's parent's parent, or some other peer using any of the approaches described in the join procedure. Hence, as soon a failure occurs, the children of the failed peer can quickly reconnect to their backup parents. For this approach to work, the server needs to ensure that there are enough slots in the backup parents of any set of siblings. Thus, if a peer fails, its children can be sure that they can fail-over to their backup parent without risking that the backup parent runs out of open slots.

2.4 Dynamically Reconfiguring the Trees

The idea of rejoins, which was used for leaves and failures, can be used at any time to optimize the current multicast tree. For example, the server could occasionally pick the node with the lowest total latency and let it rejoin a tree. Similarly, the server could re-arrange peers, e.g. moving a peer up in the tree, which would automatically also move that peer's subtree. The server could also destroy a tree by letting its nodes join other existing trees.

Yet another approach is to create a new tree from scratch for optimization purposes. The server then picks high bandwidth nodes and lets them join a new tree. This could be desirable if some trees have have a high depth due to the absence of peers with high upload capacity. If later high capacity nodes join,

these nodes could be moved to a new tree to recreate a tree that has a lower depth.

The server could decide to always create a new tree whenever the depth of the existing trees reaches a certain threshold, or if the total latency of all open peers reaches a certain threshold.

Finally, the way this problem has been described in the beginning of this paper makes it clear that it is an optimization problem. Hence, the goal is to maximize bandwidth utilization, and minimize latency. This can be solved using local search or constraint programming. If the server is periodically re-optimizing the tree, it is important that the new tree, which is the solution to the optimization problem, is not too different from the previous tree. It is therefore desirable to add the previous state of the trees as a parameter to the optimization problem, and add minimization of the cost of reconfiguration as a requirement of the solution.

2.5 Dealing with Multiple Streams

So far we have assumed a single-stream, requiring M bandwidth. It is however possible to split a single-stream into multiple, using some form of data partitioning. In this section, we first discuss two different ways to do data partitioning, and thereafter discuss how this can be used to build separate trees for every stream.

2.5.1 Data Partitioning

We focus our attention to two forms of data partitioning: Multiple-Description Coding (MDC) [2] and Layered Coding [3]. Single-streams provide two levels of quality of service (QoS): everything or nothing. Multiple-streams, such as MDC or Layered Coding, on the other hand, provide several levels of QoS, which can be chosen depending on the current state of the system (available bandwidth, congestion etc.).

The advantage of MDC is that every received stream (referred to as *descriptor*) guarantees some quality increase. Layered coding, however, does not give such guarantees. Instead, a specific descriptor, which we refer to as descriptor 0, provides the lowest level of quality. For the next level of quality, descriptor 0 and another specific descriptor, which we refer to as descriptor 1, is necessary, etc. Hence, if a node does not have descriptor 0, it cannot view the stream. Thus, a node will receive QoS level i ($i \geq 1$) if and only if it has received all the descriptors in the range $0, \cdots, i-1$, but not descriptor i. Otherwise, it will receive level 0, which implies that it cannot decode the stream. This disadvantage might be bearable, given that MDC codecs and implementations are not widely available.

2.5.2 Descriptor Trees

This section motivates the use of multiple trees, one for each descriptor of the partitioned stream. We advocate the basic idea promoted by SplitStream [4]. The technical solution proposed by SplitStream is intricate and has some disadvantages. We therefore believe that its main contribution is sometimes belittled. We first describe their contribution, and thereafter show how it can be incorporated into ForestCast.

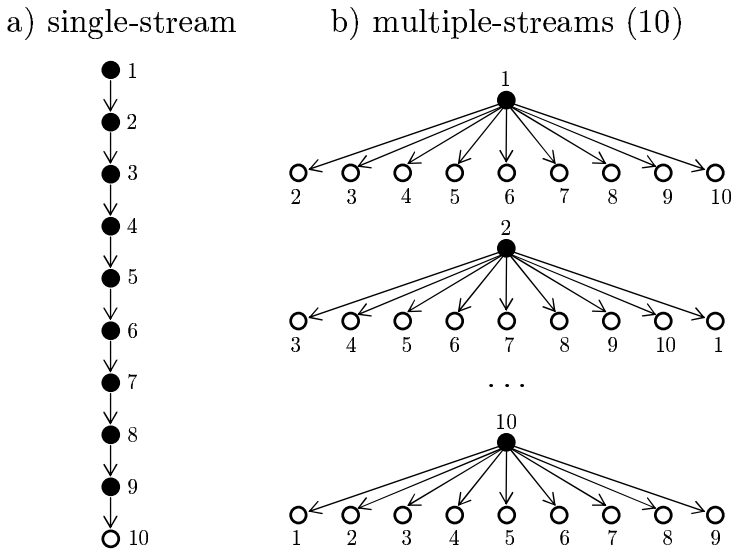

Figure 1: A single stream vs. multiple streams. Black circles indicate closed peers, while white circles represent open peers.

Aside from the intrinsic advantages of using multiple streams, there is a great advantage of using one multicast tree for every descriptor (from now on *descriptor trees*). Using descriptor trees allows for better bandwidth utilization. The following simple example will exemplify this. Assume 10 peers desire to watch a live stream. Each peer has a 900 Kbit/sec upload capacity, and 1000 Kbit/sec download capacity. Furthermore, assume that a node needs 1000 Kbit/sec to see the live stream, i.e. $M = 1000$. For this set of peers, there exists *no multicast tree*, since no single peer has enough slots to upload to any other peer. In other words, a peer–to–peer solution does not exist. Using multiple streams, the stream could be split into 10 descriptors. Hence, downloading a descriptor would require 100 Kbit/sec ($M = 100$). It is now possible to construct 10 descriptor trees, each one having a different root peer, which has all 9 other peers as its children, to which it uploads its descriptor. Each peer thus downloads $10 \cdot M = 1000$ Kbit/sec, and uploads $9 \cdot M = 900$ Kbit/sec. This is illustrated in Figure 1b. If instead each peer would have 1000 Kbit/sec upload capacity, and

1000 Kbit/sec download capacity, it would be possible to construct a peer–to–peer solution for the single-stream case. Nevertheless, the resulting tree would have to be a chain, with each peer having a single child, making it a tree with a high depth (see Figure 1a), which makes it undesirable.

The technical solution proposed in SplitStream attempts to built perfect trees, which ignore the actual number of slots available at each peer. Hence, they advocate building multiple trees, which have a high fanout, resulting in low depth. This, however, has the disadvantage that a large majority of the nodes in a tree will be leafs. This is solved by having multiple descriptor trees and by following the principle of *interior node disjoint trees*, which ensures that each node is an interior node in one tree, and a leaf in the rest of the trees. This ignores the current upload bandwidth available at the peers, and is difficult to maintain as peers join, leave, and fail.

Descriptor Trees in ForestCast. The server in ForestCast maintains k different sets of trees, one for each descriptor. Our previous usage of the number of available slots M should be changed changed to the bandwidth required to receive a single descriptor, i.e. the number of open slots increases with a factor of k. In our single stream solution, a peer joined a single tree. In the multiple stream solution, each peer joins k descriptor trees. A peer which has f slots free for upload will have f slots free in each of the trees in which it participates. The server, however, chooses how to assign joining peers to existing peers. For example, it may choose to prioritize descriptor 0, and reject requests to higher descriptors if the total amount of upload slots available is meager. This would be particularly useful in the case of Layered Coding, as lower descriptors have higher value.

The ForestCast solution to multiple streams has the advantage of being flexible. For example, if a single peer has many upload slots available, it can be an interior node in several descriptor trees. Furthermore, a single peer can have a very high fanout if it has much available upload bandwidth. This is not possible in the original SplitStream solution. Note that all the ForestCast mechanisms described for a single stream can be applied to the multiple stream scenario.

3 Decentralized Solution

ForestCast has up to now relied on a central server. Since only control messages are passed to and from the central server, such a solution should suffice for a commercial solution. We see no intrinsic value in decentralizing the central server. Note also that almost *all* operations can be parallelized. It is only required that changes to the trees, which will just be a data structure, are done atomically. Nevertheless, we hint at how the central server can be decentralized to increase scalability.

The central server can be replaced by a cluster of servers. The trees in the system are partitioned such that every server takes responsibility for one set of

trees. The partitioning can be done along the descriptors, if multiple streams are used, or just randomly. Regardless of how the partitioning is done, the mapping of trees to servers can be done using *consistent hashing* [5]. According to this scheme, each server picks an identifier in a circular identifier space, and the trees are mapped to the same space. Each server is then responsible for the trees that have an identifier between the predecessor of the server and itself. The servers have routing tables, maintaining routing pointers to *all* other servers on the ring. The advantage of this approach is that servers can be removed and added without the responsibility of trees changing too much. Each server can also maintain a snapshot of the state of the other servers, or some statistical information about the other servers (number of free slots, size of tree, depth etc.). This information can be updated periodically. When a node joins, it can contact any of the servers, which then forwards the request to whichever server which has trees with maximum number of available slots.

4 Conclusion

We have presented a flexible and centralized solution to building a forest of trees. We have described a number of heuristic strategies to handle joins. We have described how leaves can be handled through rejoins. To enable fail-overs, we have described a scheme where the server gives each peer a backup parent such that it is guaranteed that the failure of any peer can be handled. We have also described how this can be efficiently coupled with multiple streaming to allow for better bandwidth utilization. Finally, we showed how the centralized solution can be decentralized.

Advantages. The provided scheme has three main advantages.

- *Security.* It is difficult for peers to hack ForestCast as all decisions are taken by the central server. A peer just follows the server's orders about where it should download its stream. It is also possible to use a PKI scheme, where a peer can verify whether it should give its stream to another peer.
- *Flexibility.* As the server has complete information about the state of all trees, it can optimize the number and shape of trees based on any metric, e.g. total latency, bandwidth utilization, robustness against failures etc.
- *Dumb clients.* The client software, running on the peers, contains little intelligence. Hence, it will be simple and can therefore be adapted for various OS and environments. Furthermore, most updates will be to the server infrastructure. A decentralized solution would need software updates to be applied to all peers around the world.

References

[1] Cox, R., Dabek, F., Kaashoek, M.F., Li, J., Morris, R.: Practical, Distributed Network Coordinates. In: Proc. of the Second Workshop on Hot Topics in Networks (HotNets-II), Cambridge, Massachusetts, ACM Press (November 2003)

[2] Goyal, V.K.: Multiple description coding: Compression meets the network. IEEE Signal Processing Magazine **18**(5) (September 2001) 74–93

[3] Khansari, M.R.K., Zakauddin, A., Chan, W.Y., Dubois, E., Mermelstein, P.: Approaches to layered coding for dual-rate wireless video transmission. In: International Conference on Image Processing (1). (1994) 258–262

[4] Castro, M., Druschel, P., Kermarrec, A.M., Nandi, A., Rowstron, A., Singh, A.: Splitstream: High-bandwidth content distribution in a cooperative environment. In: Proceedings of the 2nd International Workshop on Peer-to-Peer Systems (IPTPS03), Berkeley, CA (2003)

[5] Karger, D.R., Lehman, E., Leighton, T., Levine, M., Lewin, D., Panigrahy, R.: Consistent Hashing and Random Trees: Distributed Caching Protocols for Relieving Hot Spots on the World Wide Web. In: Proc. of the 29th ACM Symp. on Theory of Computing (STOC'97), New York, NY, USA, ACM Press (May 1997) 654–663