Push-to-Pull Peer-to-Peer Live Streaming

Thomas Locher, Remo Meier, Stefan Schmid, and Roger Wattenhofer

Computer Engineering and Networks Laboratory (TIK),
ETH Zurich, 8092 Zurich, Switzerland
{lochert,remmeier,schmiste,wattenhofer}@tik.ee.ethz.ch

Abstract. In contrast to peer-to-peer file sharing, live streaming based on peer-to-peer technology is still awaiting its breakthrough. This may be due to the additional challenges live streaming faces, e.g., the need to meet real-time playback deadlines, or the increased demands on robustness under churn. This paper presents and evaluates novel neighbor selection and data distribution schemes for peer-to-peer live streaming. Concretely, in order to distribute data efficiently and with minimal delay, our algorithms combine low-latency push operations along a structured overlay with the flexibility of pull operations. The protocols ensure that all peers are able to obtain the required data blocks of a live stream in time, and that due to the loop-free dissemination paths, the overhead is low.

1 Introduction

Currently, we are witnessing an explosion of online video content provided on websites such as *YouTube*[1]. It is likely that in the near future, the Internet will also revolutionize television. Due to its scalability, peer-to-peer (p2p) technology is an appealing paradigm for providing live TV broadcasts over the Internet. Live p2p streaming is not only an active field of research, but there are already commercial products emerging, e.g., *JumpTV*[2], *PPLive*[3], *SopCast*[4], among others, which provide television to thousands of viewers.

Live streaming faces several challenges that are not encountered in other p2p applications such as file sharing. The streaming content is required to be received with respect to hard *real-time constraints*, and data blocks that are not obtained in time are dropped, resulting in a reduced playback quality. Additionally, a live broadcast ought to be received by all users simultaneously and with *minimal delay*. Moreover, as video streams often already demand a high transmission rate themselves, it is of paramount importance that the overhead caused by redundant transmissions of the protocol itself be minimized. Yet another crucial property of any successful live streaming system is its robustness to peer dynamics: It is likely that peers join and leave the system continuously and concurrently, called *churn*.

[1] See http://www.youtube.com/
[2] See http://www.jumptv.com/
[3] See http://www.pplive.com/
[4] See http://www.sopcast.com/

A. Pelc (Ed.): DISC 2007, LNCS 4731, pp. 388–402, 2007.

While there already exist several solutions both in literature and in practice, many of these systems fail to take all the aforementioned criteria into account. This may partly be explained by the fact that some of the optimization goals are inherently antagonistic. For example, a low delay can be achieved by having each peer immediately forward all incoming data blocks to its neighboring peers (*pushing*). Unfortunately, such a naive solution results in a significant overhead, as a peer may receive the same block repeatedly from different neighbors. Alternatively, peers could request missing blocks *explicitly.* This scheme is referred to as *pulling* since all peers have to initiate the transmission of data blocks towards themselves. While a pull-based approach circumvents the problem of receiving duplicates, it comes at the cost of intolerable latencies, as notifications and requests have to be sent back and forth. Hence, there is a trade-off between overhead and efficiency.

This paper presents and evaluates novel data distribution mechanisms which combine the benefits of pull-based approaches with the advantages of push-based approaches. In our mechanism, a fresh data block is quickly pushed to a well-defined set of peers. Due to the structured, prefix-based neighbor selection policy, this can be achieved without any redundant transmissions. The remaining peers which have not received the blocks in this initial phase use the pull mechanism to distribute the new data block amongst themselves.

We have implemented the algorithms presented in this paper in our own peer-to-peer live streaming system *Pulsar.*[5] Apart from real-world tests such as the broadcast of the IPTPS 2007 conference, we have performed extensive simulations of our protocol. According to our emulations with up to 100,000 peers (using the real code base), the system scales well as the network topology has a low diameter and guarantees small round trip times due to the latency-aware choice of neighbors. The proposed push-to-pull data dissemination policy is efficient: The time required from the moment a fresh data block becomes available at the source until it has reached virtually all peers is around 1,500 ms in a overlay consisting of 10,000 peers, and having 100,000 peers instead of 10,000 incurs a moderate additional delay of less than 250 ms. Finally, the Pulsar system tolerates a large fraction of peers crashing simultaneously without entailing any *underflows* at the remaining peers, i.e., all packets arrive before their playback deadline. This indicates that our protocols also perform well in dynamic environments.

The remainder of this paper is organized as follows. After reviewing related work in Section 2, the design of our protocol is presented in Section 3, followed by its evaluation in Section 4. In Section 5, the paper concludes.

2 Related Work

Although it has been expected that one-to-many broadcast would be offered through *IP multicast* since the early 1990s, it is not used in practice at all due to its limited support by the Internet Service Providers (ISPs). An attractive

[5] See http://www.getpulsar.com/

alternative to native IP multicast is to use a peer-to-peer network overlay built on the application layer to distribute the content.

Existing peer-to-peer approaches are mainly categorized according to the topology maintained among the peers or, equivalently, the neighbor selection algorithms the peers employ. Simple multicast systems are based on overlay *trees* [2,4,13]. Trees have the advantage that the topology is simple, once it is constructed the overhead is small in a static setting, and there are no duplicates as every peer receives its data blocks from its sole parent. However, there are rather serious drawbacks which render such systems inefficient. For example, resources are wasted as all the leaves of such a tree do not contribute anything to the system. Moreover, inner nodes having two or more children need to upload at least at twice the bitrate of the stream. This means that high-quality video streams cannot be transmitted unless one can guarantee that all inner nodes have a lot of spare upload capacity. Finally, the fragile tree structure is not resilient to any kind of node failures or churn. In order to overcome these problems, systems have been proposed to split the content of the stream into several disjoint *stripes* and disseminate this information along multiple disjoint trees. SplitStream [1] is a prominent example which uses *multiple description coding* (MDC) [3] to split the stream into different stripes in order to distribute them on several trees. Multiple description coding allows for the reconstruction of the original stream using any subset of the stripes. As each peer is also required to be an inner node in one of the trees, this approach solves the single tree's problem of having a large fraction of free-riding leaf peers. The CoopNet [7] approach is similar in that it also uses MDC and multiple trees; however, its goal is merely to complement the traditional client-server model as opposed to completely replace it. In this protocol, the server handles all the join requests and centrally manages all trees which limits the system's scalability. MDC is still an active research effort and no implementations for practical use are available. In addition, the overhead of multiple description coding harms the system's efficiency which may raise concerns whether multiple description coding is currently suitable for this kind of application. While maintaining several trees improves the robustness of a system, each tree can break individually, and the overhead potentially increases as more trees have to be repaired continuously (and concurrently).

Various systems using other approaches to cope with the shortcomings of tree-based topologies have also been presented. The *Bullet* [6] system uses a *mesh* on top of an arbitrary tree overlay in order to increase robustness. The additional links introduced by the mesh increase robustness by reducing the dependency of peers on their parents. The stream is split into disjoint blocks and distributed within the tree. Only as many blocks are sent to children as bandwidth is available, and missing blocks are then localized and retrieved using the mesh. However, the encoding of blocks, the duplicates, the requests for missing blocks, and the tree maintenance entail a substantial overhead in Bullet. *ChunkySpread* [12] strives to redeem the shortcomings of tree-based topologies by providing more efficient protocols to build and repair trees. By adding a "weak" tit-for-tat model and locality awareness, additional important aspects of peer-to-peer live streaming are considered.

The overhead of any tree-based protocol is generally large as the trees have to be repaired and the topology maintained. This is particularly true if there is a lot of churn in the network. Another disadvantage of trees is its lack of control over selfish peers: It is difficult to enforce that peers actually forward the data blocks to their designated children. Due to these inherent problems, a lot of research has also focused on tree-less protocols.

Since a rigidly structured overlay requires permanent maintenance, care has to be taken not to burden the individual peers. Therefore, unstructured overlays have been favored over structured overlays, and various protocols based on unstructured overlays have been proposed, e.g., *CoolStreaming/DONet* [15], *Chainsaw* [8] and *GridMedia* [14], all published in 2005. CoolStreaming/DONet makes a strong case for a *data-centric* design of the overlay, which means that the availability of data at certain nodes must steer the content dissemination, in contrast to having the predefined overlay dictate the data flow. Chainsaw and Gridmedia also follow this paradigm and mainly differ in the number of stored links to other peers, block sizes, buffer lengths, etc.—generally, parameters which have an impact on the overlay's robustness and overhead.

Typically, in unstructured overlays, peers have to *notify* neighboring peers about available blocks of data, and peers that are interested in obtaining these blocks must explicitly *request* them before any data is exchanged, because there is no structure in the network that could be used to disseminate data. Note that this scheme has the disadvantage that notifying peers and subsequently requesting data blocks potentially results in long delays before any data is exchanged.

Our approach differs from all these protocols in that it uses a structured overlay, based on a *prefix-routing* neighbor selection policy [9,10]. This policy guarantees a logarithmic diameter, robustness to massive crash failures and churn, and it also ensures that the entire network remains connected. At the same time, the protocol uses the flexibility of this neighbor selection scheme to take latency and bandwidth considerations into account when building up and maintaining the routing tables. Our mechanism further uses novel push algorithms tailored specifically for prefix-routing-based topologies to quickly disseminate the content to a fraction of all peers, thereby significantly reducing the delay experienced in other pull-based protocols. The benefits of push-to-pull strategies are well-known in theory, e.g., in the context of efficient rumor spreading [5]. Hence, this push-to-pull-based technique possesses the advantages of the pull-based schemes and in addition has the efficiency of push-based algorithms.

3 Push-to-Pull Protocol

This section presents the design of our protocol for peer-to-peer live streaming. It is based on two concepts: First, the protocol defines the overlay structure, i.e., it specifies how peers are to select their neighboring peers. The overlay is inspired by the structured topologies of *distributed hash tables* (DHTs) which guarantee connectivity and a logarithmic diameter. The flexibility of the neighbor selection strategy is used to account for additional factors which influence performance, for instance, bandwidth requirements and latency constraints. The topology aims at being resilient to churn and massive correlated failures. Second, the protocol specifies how data is distributed in the overlay network. Concretely, the protocol advocates the data-driven streaming paradigm, and introduces a novel combination of fast pushing operations and robust pull operations.

3.1 Overlay and Neighbor Selection

The proposed overlay consists of an unstructured and a structured part. Initially, a peer is assigned a random set of neighbors by a *network entry point*. Over time, a refinement process takes place as peers learn about other peers from their neighbors and add them to their routing table depending on the following criteria: Since peers strive to maintain several connections to *close-by peers*, new neighbors are continuously accepted based on the latency measured to these peers.

While truly random networks are known to have desirable properties, constraining the choice of neighbors to peers that are close-by may lead to clusters and consequently threaten the efficiency or even the connectivity of the overlay. Therefore, our protocol uses d-bit peer identifiers in order to build a DHT-like topology (of course, without the data storage semantics). These identifiers can be used for prefix-routing, as links to neighbors are stored for different shared prefix lengths. Let β denote the number of bits that can be fixed at a peer to route any message to an arbitrary destination. For $i = \{0, \beta, 2\beta, 3\beta, \ldots\}$, a peer chooses, if possible, $2^\beta - 1$ neighbors whose identifiers are equal in the i most significant bits and differ in the subsequent β bits by one of $2^\beta - 1$ possibilities. For random bit strings, this ensures an expected logarithmic network diameter and peer degree. Similarly to DHTs based on prefix-routing, our solution has the advantage over more rigid DHT structures such as *Chord* [11] that there is a large choice of neighbors for short prefixes, which means that an optimizing secondary criterion can be used to pick neighbors. For example, as the identifiers of roughly half of all peers start with 0, any of those peers can be used as the routing table entry for this prefix, while about one fourth of all peers are suitable for the prefix 00 etc. This freedom is used in our protocol to choose peers according to their latency (locality awareness), but also in order to construct different push mechanisms as described in the following section.

3.2 Pushing and Pulling Data

The prime objective of the pushing component is to quickly distribute a data block to a certain number of peers, in order to fuel the subsequent pull-based exchanges. As we have argued before, such a mechanism is needed due to the long delays of purely pull-based approaches; the pushing phase brings the data block into the vicinity of virtually all peers.

In this section, various aspects of pushing data blocks to neighbors are discussed. In particular, we present two concrete algorithms where each of these algorithms has its own merits. The first algorithm, denoted by $\mathcal{ALG}_1$, is simple, robust, and has a low overhead; it needs fewer neighbors per peer and deals better with heterogenous bandwidths. However, it cannot guarantee that the push mechanism reaches a considerable share of all peers and specific care has to be taken to make sure that no duplicates can occur. The second algorithm, $\mathcal{ALG}_2$, is more sophisticated: All the peers can be reached without the use of the pulling mechanism, and there are provably no retransmissions. Note that a loop-free transmission implies that data is distributed on *induced spanning trees*, which are generally not comparable to structures where the overlay graph consists of one or more trees which must be used to disseminate data. Our graph is still hypercubic, and, in accordance with the data-driven streaming paradigm, each packet can theoretically induce a different tree on which it is broadcast.

Due to the simplicity and robustness of $\mathcal{ALG}_1$, it is better suited for dynamic environments and also in settings where peers may act selfishly. As we will show in Section 4, in order to boost the dissemination process it suffices to push fresh data blocks to a subset of all peers. This implies that the lack of guarantee that many peers can be reached using this push mechanism is not a severe limitation. Nevertheless, the ability of $\mathcal{ALG}_2$ to efficiently push new blocks to practically all peers may be preferable in various scenarios. For example, there may be situations where one wants to precisely control the fraction of peers reached by the pushing operation only. In a more stable network or a network where incentives are of no concern, more peers should be reached by pushing blocks for efficiency reasons, so that only a small number of pulls are necessary to distribute the new block among the remainder of the peers.

In the following, let, for two peers u and v with identifiers $b_0^u \dots b_{d-1}^u$ and $b_0^v \dots b_{d-1}^v$, where b_i^u and b_i^v, denote the i^{th} bit of their respective identifiers, $\ell(u,v) = k$ if $b_j^u = b_j^v$ for all $j \in \{0, \dots, k-1\}$ and $b_k^u \neq b_k^v$. Furthermore, let $\mathcal{N}_v$ be the set of all neighboring peers of v. We first present $\mathcal{ALG}_1$ and discuss its properties. Let β again denote the number of bits that the prefix routing algorithm fixes at each hop. The source selects 2^β peers from its routing table, if possible, such that the identifiers of any two peers differ in at least one bit of the first β bits. A new block is pushed to these peers along with the information that they must only forward the block to peers with which they share the first β bits of their identifiers. Recursively, upon receiving such a push message with the specified prefix length π that they must not modify, a recipient selects 2^β peers that share the prefix of length π with itself and that differ in at least one bit between the $(\pi+1)^{st}$ and the $(\pi+\beta)^{th}$ bit and so on. This straightforward approach to pushing on prefix-based overlays has an obvious shortcoming: Assume that $\beta = 2$ and that the source peer has the identifier consisting of only zeros. It will push the block, among others, to a peer whose identifier starts with 00 which will in turn forward the block to a peer whose identifier starts with 0000. This peer might then forward the block back to the source again, as the identifier of the source also starts with 000000. Such loops can occur on all paths. If a peer v pushes the block solely to all the $2^\beta - 1$ peers that differ in at least one bit from the identifier of v itself, there are no duplicates; however, this reduction would cut off entire branches of peers which could never benefit from the push mechanism.

A viable solution to the duplicates problem is to include a list $\mathcal{L}$ of *critical* predecessors of the induced spanning tree. Only the peer identifiers having a prefix of length $\pi + \beta$ in common are sent along. The push message any peer v receives contains the parent p in the induced spanning tree, the fixed prefix length π, and the list of critical predecessors $\mathcal{L}$.[6] The parent is potentially a critical peer for one of the children, and therefore it is added to the list $\mathcal{L}$. Afterwards, using the local subroutine *getChildren*, the $l \leq 2^\beta$ children are selected from the routing table for which it holds that they all share a prefix of length at least π with peer v itself, the identifiers of any two of those children differ in at least one of the following β bits, and they do not occur in the list $\mathcal{L}$. In the next step, the lists $\mathcal{L}_j$ of critical predecessors are created for all children. Note that any critical predecessor p_i is added to at most one list $\mathcal{L}_j$, and only if it is still critical for this child v_j, i.e., $\ell(v_j, p_i) \geq \pi + \beta$. The source v_0 pushes data blocks containing the parameters $p := v_0$, $\pi := \beta$, and $\mathcal{L} := \emptyset$ to its children. This push strategy $\mathcal{ALG}_1$ is summarized in Algorithm 1.

Algorithm 1. $\mathcal{ALG}_1$: push$(p, \pi, \mathcal{L})$ at peer v.

```
$\mathcal{L} := \mathcal{L} \cup \{p\}$
$\{v_1, \dots, v_l\}$ := getChildren$(v, \pi, \mathcal{L})$
for all $p_i \in \mathcal{L}$ do
    $j := \arg\max_{j \in \{1,\dots,l\}} \ell(v_j, p_i)$
    if $\ell(v_j, p_i) \geq \pi + \beta$ then $\mathcal{L}_j := \mathcal{L}_j \cup \{p_i\}$ fi
od
for $j = 1, \dots l$ do send push$(v, \pi + \beta, \mathcal{L}_j)$ to $v_j$ od
```

It is easy to see that $\mathcal{ALG}_1$ is indeed loop-free, and that the expected length of the list $\mathcal{L}$ is bounded by $\sum_{j=2}^{\infty} \frac{1}{(2^\beta)^j} = \frac{1}{2^\beta(2^\beta - 1)}$ which is less than one entry. However, the worst-case length of the list is $\log(n)/\beta$. Another shortcoming of this algorithm is that it is likely that not all peers can be reached, because once a peer is reached that only has connections to peers that are in the list $\mathcal{L}$ for a certain prefix, this branch of the tree is cut off.

$\mathcal{ALG}_2$ avoids these problems by modifying the topology and using a different routing scheme. For simplicity, we present the neighbor selection strategy and the push algorithm for the case $\beta = 1$. In order to use $\mathcal{ALG}_2$, the peers must store links to a totally different set of neighbors: A peer v with the identifier $b_0^v \dots b_{d-1}^v$ stores links to peers whose identifiers start with $b_0^v b_1^v \dots b_{i-1}^v \overline{b_i^v} b_{i+1}^v$ and $b_0^v b_1^v \dots b_{i-1}^v \overline{b_i^v b_{i+1}^v}$ for all $i \in \{0, \dots, d-2\}$. For example, the peer with the identifier 0000 has to maintain connections to peers whose identifiers start with the prefixes 10, 11, 010, 011, 0010, and 0011. Pseudo-code for the algorithm is given in Algorithm 2.

[6] For simplicity, as the data contained in the push messages does not have any influence on the push procedures, it is omitted in our notation.

Algorithm 2. $\mathcal{ALG}_2$: push(π, v_c) at peer v.

```
$\mathcal{S} := \{v' \in \mathcal{N}_v \mid \ell(v', v) \geq \pi + 1\}$
choose $v_1 \in \mathcal{S}$: $\ell(v_1, v) \leq \ell(\tilde{v}, v)\ \forall \tilde{v} \in \mathcal{S}$
**if** $v_1 \neq \emptyset$ **then** send push$(\ell(v_1, v), v)$ to $v_1$ **fi**
**if** $v_c \neq \emptyset$ **then**
    choose $v_2 \in \mathcal{N}_v$: $\ell(v_2, v_c) = \pi + 1$
    **if** $v_2 = \emptyset$ **then** $v_2 :=$ getNext$(v)$ from $v_c$ **fi**
    **if** $v_2 \neq \emptyset$ **then** send push$(\ell(v_2, v_c), v_c)$ to $v_2$ **fi**
**else**
    choose $v_2 \in \mathcal{N}_v$: $\ell(v_2, v) = \pi$
    **if** $v_2 \neq \emptyset$ **then** send push$(\pi + 1, v_c)$ to $v_2$ **fi**
**fi**
```

The parameters are again the length π of the prefix that is not to be modified, and at most one critical predecessor v_c. If $\beta = 1$, any node v tries to forward the push message to two peers v_1 and v_2. The procedure is called at the source v_0 with arguments $\pi := 0$ and $v_c := \emptyset$, resulting in the two push messages *push(*$1, v_0$*)* to v_1 and *push(*$1, \emptyset$*)* to v_2. The peer v_1 is chosen locally such that the prefix its identifier shares with the identifier of v is the shortest among all those whose shared prefix length is at least $\pi + 1$. This value $\ell(v_1, v)$ and v itself are the parameters included in the push message to peer v_1, if such a peer exists. The second peer is chosen similarly, but with respect to v_c and not v itself. If no suitable peer is found in the routing table, the peer v_c is inquired for a candidate using the subroutine *getNext* which is described in Algorithm 3.

Algorithm 3. getNext(v_s) at peer v

```
$\mathcal{S} := \{v' \in \mathcal{N}_v \mid \ell(v', v) > \ell(v_s, v)\}$
choose $v_r \in \mathcal{S}$: $\ell(v_r, v) \leq \ell(\tilde{v}, v)\ \forall \tilde{v} \in \mathcal{S}$
send $v_r$ to $v_s$
```

This step is required because node v cannot deduce from its routing table whether a peer v_2 with the property $\ell(v_2, v_c) \geq \pi + 1$ exists. In the special case when $v_c = \emptyset$, v_2 is chosen locally, if possible, such that $\ell(v_2, v) = \pi$. In Figure 1, an example spanning tree resulting from the execution of $\mathcal{ALG}_2$ is depicted.

As mentioned earlier, $\mathcal{ALG}_2$ has the property that, at least in a static setting where peers neither join nor leave the network, all peers can be reached. Due to churn, any real overlay can never be considered static. However, this static property implies that this pushing procedure is expected to reach a large number of peers even if some peers appear and disappear during the push phase.

Theorem 3.1 *In a static overlay, the push algorithm* $\mathcal{ALG}_2$ *has the following properties:*

(a) It does not induce any duplicate messages (loop-free), and
(b) all peers are reached (complete).

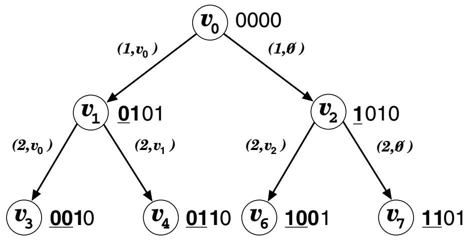

Fig. 1. The spanning tree induced by a push message initiated at peer v_0 is shown. The fixed prefix is underlined at each peer, whereas prefixes in bold print indicate that the parent peer has been constrained to push the packet to peers with these prefixes.

Proof. Throughout the proof, we will use the fact that $\forall u, v, w : \ell(u, v) = \sigma$ and $\ell(v, w) = \tau$ implies that $\ell(u, w) = min(\sigma, \tau)$ which we will refer to as Fact (1).

(a) Loop-free: If a peer v receives a *push message* μ and forwards it to other peers which in turn forward the message and so on, let $\mathcal{C}_v(\mu)$ denote the set of peers that are reached recursively. We first show that if any peer v forwarding push messages μ' and μ'' to two peers v' and v'', these peers will subsequently forward the message to disjoint sets of peers, i.e., $\mathcal{C}_{v'}(\mu') \cap \mathcal{C}_{v''}(\mu'') = \emptyset$. Then, we will show that peers never send messages back to predecessors.

Let v be the peer receiving the push message and let π_v denote the prefix length that peer v can no longer modify. As in the description of the algorithm, the two peers the message is forwarded to are v_1 and v_2. Let further v_p denote the peer that sent the message to v. In order to prove that disjoint sets are constructed, it suffices to show that $\ell(v, v_1) \geq \pi_v + 1$ and $\ell(v, v_2) = \pi_v$ at any peer v.

The first inequality follows immediately from the algorithm. If $v_c = \emptyset$ then $\ell(v, v_2) = \pi_v$ also follows by definition. Therefore we can assume in the following that $v_c \neq \emptyset$. If $v_p = v_c$ we have that $\ell(v_p, v_2) \geq \pi_v + 1$, as v was chosen from $\mathcal{S}$ according to this criterion. It further holds that $\ell(v, v_p) = \pi_v$ because the parameter π_v that p sends to v in the message is precisely $\ell(v, v_p)$. According to Fact (1), we get that $\ell(v_2, v) = \pi_v$. Similarly, if $v_p \neq v_c$, it holds that $\ell(v_c, v_2) \geq \pi_v + 1$, due to the fact that either v_p found peer v in its routing table, implying that $\ell(v_c, v_2) = \pi_v + 1$, or the procedure *getNext* has been invoked which by definition means that $\ell(v_c, v_2) > \pi_v + 1$. As p sends the value $\ell(v, v_c)$ to v, it holds at peer v that $\pi_v = \ell(v, v_c)$, again leading us to the conclusion that $\ell(v, v_2) = \pi_v$.

This concludes the proof that the resulting peer sets are always disjoint. Since peers might forward push messages back to a predecessor, we cannot yet conclude that no duplicates are produced. Let $v^{(0)} \rightsquigarrow v^{(1)} \rightsquigarrow \ldots \rightsquigarrow v^{(k)}$, where $k \leq \pi_v$, denote the path from peer $v^{(0)} := v$ back to the source $v^{(k)}$. Note that the value π steadily increases downwards, implying that $\pi_{v^{(0)}} > \pi_{v^{(1)}} > \ldots > \pi_{v^{(k)}}$. Let us first assume that $v_c \neq \emptyset$ on the entire path. If $v_c = v^{(1)}$ then it holds that $\ell(v, v^{(1)}) = \pi_v$ according to the algorithm. In the case $v_c \neq v^{(1)}$, then by definition $\ell(v, v_c) = \pi_v$ and as $\ell(v^{(1)}, v_c) = \pi_{v^{(1)}} < \pi_v$, we get that in all cases $\ell(v, v^{(1)}) \leq \pi_v$. Inductively, the same argument can be applied to the maximal prefix length between $v^{(1)}$ and $v^{(2)}$ which is bounded by $\pi_{v^{(1)}}$ etc. Using Fact (1), we have that $\ell(v, v^{(i)}) \leq \pi_v$ for all $1 \leq i \leq k$. If for some $i^* \in \{0, \ldots, k-1\}$ a peer is reached that received $v_c = \emptyset$ from $v^{(i^*+1)}$, it holds according to the definition of $\mathcal{ALG}_2$ that all other peers closer to the source on this path also received $v_c = \emptyset$. This entails that $\ell(v^{(k)}, v^{(k-1)}) = 0$, $\ell(v^{(k-1)}, v^{(k-2)}) = 1$ and so on, down to $\ell(v^{(i^*)}, v^{(i^*+1)}) = k - i^* \leq \pi_v$. Applying the same inductive argument as before, we can conclude that $\ell(v, v^{(j)}) \leq \pi_v$ also for all $j > i^*$ if such an i^* exists. Since the first π_v bits are not changed at peer v when forwarding the message to other peers, it is impossible for v to send the push message to a predecessor as all predecessors' identifiers differ in at least one bit among the first π_v bits, which concludes the proof that no duplicates can occur and the resulting structure is a spanning tree.

(b) Complete: It remains to be shown that all peers are reached using this procedure. Using $\ell(v, v_1) \geq \pi_v + 1$ and $\ell(v, v_2) = \pi_v$ at any peer v, it follows that, when forwarding push messages, the current prefix is extended with a 0 and a 1, and the value π is increased. Note that care has to be taken only if identifiers with certain prefixes do not exist. If no peer v_1 such that $\ell(v_1, v) = \pi + 1$ exists, the next bit can be tried by choosing v_1 such that $\ell(v_1, v) = \pi + 2$ and so on. Given that v_1 is chosen among all peers in $\mathcal{S}$ such that the $\ell(v_1, v)$ is minimal, it is guaranteed that no peer is left out. Similarly, if there is no peer v_2 such that $\ell(v_2, v_c) = \pi_v$, the next bit is tried by calling the function *getNext* at peer v_c which chooses v_2 the same way as peer v chooses v_1. This means that prefixes are only left out if no peer's identifier has this particular prefix and thus every peer can be reached. □

Observe that at any time, at most one predecessor is critical and has to be included in a push packet. A disadvantage of $\mathcal{ALG}_2$, compared to $\mathcal{ALG}_1$, is that peers have to maintain twice as many connections to other peers. Since all peers ideally communicate regularly with all their neighbors, it is best to keep the set of neighboring peers small.

However, both algorithms are not sufficient to quickly disseminate data to all peers in dynamic environments such as the Internet. Due to the perpetual arrival and departure of peers, which results in inaccurate routing tables, only a certain fraction of all peers are effectively reached through pushing. Thus, a second mechanism has to be used where peers having received new data blocks *notify* their neighbors about the corresponding sequence numbers. A peer can then obtain data blocks it is interested in by explicitly requesting them from a neighbor (*pull operation*). Hence, a data block is never forwarded twice to the same peer, and there are no redundant transmissions. The initial distribution of new data blocks through pushing ensures that almost every peer has at least one other peer in its vicinity that offers the missing data block.

4 Evaluation

Our protocol has been evaluated in several respects. We have performed extensive *emulations* (simulations using the real code base of the *Pulsar* system in a simulated network) with up to 100,000 peers on a single Core2 Quad personal computer with 4GB of RAM. Our emulation results have also been confirmed in tests on PlanetLab. Finally, a real-world beta test has shown that the protocol manages to cope well with the peculiarities of the Internet and to distribute the content reliably. Due to space constraints, we only present results concerning the key concepts introduced in this paper, namely the neighbor selection and the push- and pull-based data dissemination policy.

4.1 Topology and Neighbor Selection

First, we have evaluated the properties of the streaming topology itself. We have streamed data over a network of 100 to 100,000 peers and counted the total number of hops taken by each data packet. Figure 2 shows that, as expected, the hypercubic structure induced by the neighbor selection results in a logarithmic network diameter.

As described in Section 3, the flexibility of our topology allows for locality awareness, i.e., for the choice of close peers as neighbors. This indeed helps to reduce the round trip times significantly compared to a random neighbor selection strategy, as Figure 3 clearly suggests. Figure 3 depicts the number of neighbors that the average peer maintains for any given round trip time. In this emulation, peers are distributed uniformly on a square with a minimum delay of 10 ms and maximum delay of 200 ms which corresponds to the square's diagonal.

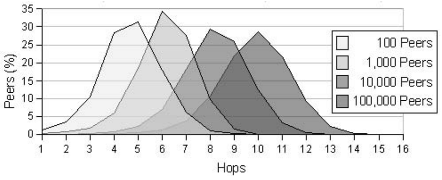

Fig. 2. Number of hops taken by each data packet to reach the destination peer. The network diameter scales logarithmically with the total number of peers.

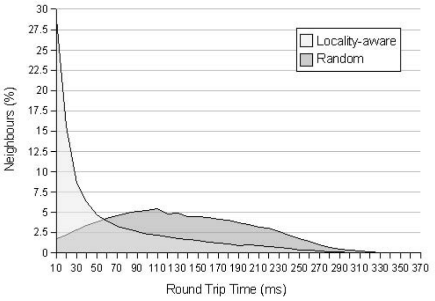

Fig. 3. Effect of locality awareness with 10,000 peers: The average round trip times to all neighboring peers are significantly smaller in the network constructed using our protocol than in a network where neighbors are chosen at random

4.2 Push-to-Pull Data Distribution

Figure 4 compares the two push strategies $\mathcal{ALG}_1$ and $\mathcal{ALG}_2$ introduced in Section 3 with a pull-only strategy like the one adopted by Chainsaw.

The figure shows that, compared to pull-only protocols, pushing considerably speeds up the distribution of new data blocks and thereby reduces the playback delay. Once a sufficient number of peers have received a block, the remainder of the peers can retrieve the fresh data block using the pull mechanism with a moderate additional overhead. It is evident from Figure 4 that $\mathcal{ALG}_1$ is almost as fast as $\mathcal{ALG}_2$, although only about one third of all peers obtain the new data block through pushing, while almost all peers are reached using $\mathcal{ALG}_2$ in this test.

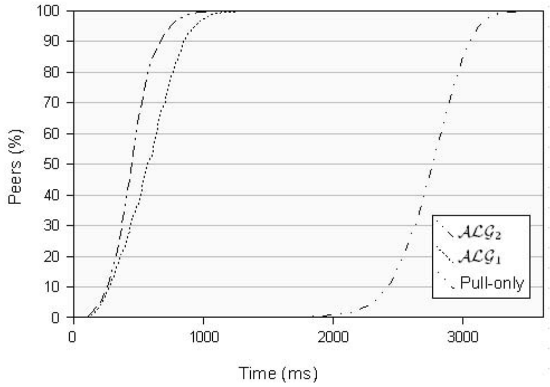

Fig. 4. Time required until the push strategies $\mathcal{ALG}_1$, $\mathcal{ALG}_2$, and a pull-only strategy reach a given fraction of all peers in a network of 10,000 peers

Figure 5 depicts the percentage of all data packets received through pushing for both algorithms $\mathcal{ALG}_1$ and $\mathcal{ALG}_2$ for increasing network sizes. Independent of the chosen algorithm, less packets are received through pushing as the network grows. This decline is due to the increased chance of branches of the distribution trees being cut off, because of inaccurate routing tables, before a substantial number of peers is reached. As expected, the fraction of pushed packets decreases much more rapidly when $\mathcal{ALG}_1$ is used. However, it is sufficient to reach only a fraction of all peers in the pushing phase, as the subsequent pull operations can be performed efficiently and with a small additional delay. Note that both pushing strategies greatly benefit from the locality awareness which not only decreases the chances of packet loss but also allows the use of short timeouts for acknowledgments.

A second test studies the scalability of the $\mathcal{ALG}_1$ pushing algorithm. Figure 6 indicates that the network scales well with the number peers, as exponentially more peers merely results in a linear increase of the delays. Moreover, all peers experience a delay of not more than 1.5 seconds.

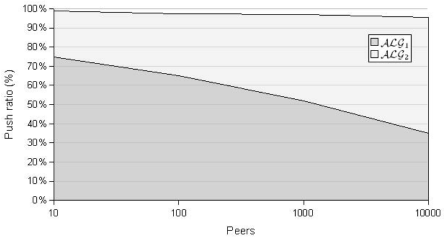

Fig. 5. As the network grows, less data is received in the pushing phase. The fraction of data obtained through pushing decreases considerably faster when algorithm $\mathcal{ALG}_1$ is used.

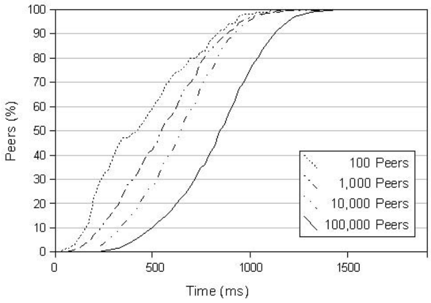

Fig. 6. Given an exponential increase of the number of peers, the delays increase only linearly

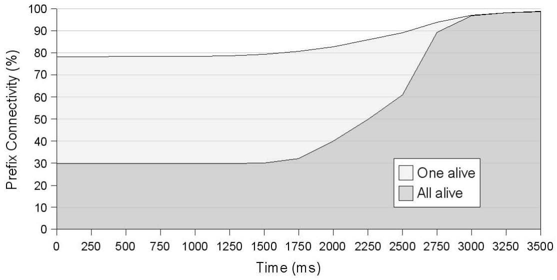

Fig. 7. Effect of 50% simultaneous random crashes in a network of 5,000 peers. “One alive” shows the percentage of prefixes for which at least one connection is present, while “All alive” depicts the percentage of prefixes for which all connections are still alive. In both cases, already after 3 seconds, the peers are again fully connected.

4.3 Robustness to Churn

The high connectivity of our hypercubic network topology and the flexible choice of neighbors allows to build up and fix routing tables quickly. Several scenarios have been considered in which a large fraction of peers leaves simultaneously. It turns out that it is easy to maintain the topology and to recover even from such massive concurrent network changes.

Figure 7 shows a network where a random set of 50% of the 5,000 peers leave simultaneously. A severe network failure is assumed where all the peers crash without notice (no “leave message”). For each prefix stored in the routing table, a peer maintains roughly 2 to 3 connections to other peers whose identifiers match the specific prefix. Immediately after the network failure, for approximately 80% of the stored prefixes, at least one connection to a peer that is still alive is retained. After roughly 3 seconds, the routing table is again almost completely repaired. The figure also depicts the percentage of prefixes for which all connections are still alive. This short loss of connectivity is only due to the lack of a proper leave message. In case disconnecting peers are able to send a leave message, which is certainly the normal case, the network is hardly affected if as many as 50% of the peers leave, and the prefix connectivity does not drop noticeably, as peers immediately search for suitable replacements.

Due to the fast repairing process, our system also copes well with membership changes occurring continuously over time.

5 Conclusions

Given the growing number of radio stations and TV channels available online, peer-to-peer live streaming is able to overcome the limitations of traditional, centralized approaches, and it enables content providers to both increase playback quality and to reduce costs. Thus, the p2p paradigm has the potential to democratize the streaming world in that it enables everyone to broadcast her own media content—similarly to how the world wide web revolutionized the distribution of information more than a decade ago: Nowadays, everyone can publish her thoughts on her own blog or website at virtually no cost.

By combining pull-based and push-based techniques, our push-to-pull protocol for live streaming achieves high efficiency and robustness, both essential features of a reliable p2p streaming service. As a second central ingredient, our protocol makes use of the lessons learnt from distributed hash tables by structuring the overlay topology while still maintaining a large degree of flexibility. The resulting system is locality-aware and has a guaranteed logarithmic diameter. Moreover, it enables the source to push new data blocks to speed up data dissemination. Having a push mechanism allows to reduce the notification frequency, which leads to a substantially smaller overhead.

References

1. Castro, M., Druschel, P., Kermarrec, A.-M., Nandi, A., Rowstron, A., Singh, A.: SplitStream: High-bandwidth Content Distribution in a Cooperative Environment. In: Kaashoek, M.F., Stoica, I. (eds.) IPTPS 2003. LNCS, vol. 2735, Springer, Heidelberg (2003)
2. Chu, Y., Rao, S., Zhang, H.: A Case For End System Multicast. In: Proc. Int. Conference on Measurement and Modeling of Computer Systems (SIGMETRICS), pp. 1–12 (2000)
3. Goyal, V.K.: Multiple Description Coding: Compression Meets the Network. IEEE Signal Processing Magazine 18(5), 74–93 (2001)
4. Jannotti, J., Gifford, D.K., Johnson, K.L., Kaashoek, M.F., O'Toole, J.W.: Overcast: Reliable Multicasting with an Overlay Network. In: Proc. 4th Symposium on Operating System Design and Implementation (OSDI) (2000)
5. Karp, R., Schindelhauer, C., Shenker, S., Vocking, B.: Randomized Rumor Spreading. In: Proc. 41st Annual Symposium on Foundations of Computer Science (FOCS). IEEE Computer Society Press, Los Alamitos (2000)

6. Kosti, D., Rodriguez, A., Albrecht, J., Vahdat, A.: Bullet: High Bandwidth Data Dissemination Using an Overlay Mesh. In: Proc. 19th ACM Symposium on Operating Systems Principles (SOSP), pp. 282–297. ACM Press, New York (2003)
7. Padmanabhan, V.N., Sripanidkulchai, K.: The Case for Cooperative Networking. In: Druschel, P., Kaashoek, M.F., Rowstron, A. (eds.) IPTPS 2002. LNCS, vol. 2429, pp. 178–190. Springer, Heidelberg (2002)
8. Pai, V., Tamilmani, K., Sambamurthy, V., Kumar, K., Mohr, A.: Chainsaw: Eliminating Trees from Overlay Multicast. In: Castro, M., van Renesse, R. (eds.) IPTPS 2005. LNCS, vol. 3640, Springer, Heidelberg (2005)
9. Plaxton, C.G., Rajaraman, R., Richa, A.W.: Accessing Nearby Copies of Replicated Objects in a Distributed Environment. In: Proc. 9th Annual ACM Symposium on Parallel Algorithms and Architectures (SPAA), pp. 311–320. ACM Press, New York (1997)
10. Rowstron, A., Druschel, P.: Pastry: Scalable, Decentralized Object Location, and Routing for Large-Scale Peer-to-Peer Systems. In: Proc. International Conference on Distributed Systems Platforms (Middleware), pp. 329–350 (2001)
11. Stoica, I., Morris, R., Karger, D., Kaashoek, F., Balakrishnan, H.: Chord: A Scalable Peer-To-Peer Lookup Service for Internet Applications. In: Proc. ACM SIGCOMM Conference, pp. 149–160. ACM Press, New York (2001)
12. Venkataraman, V., Francis, P., Calandrino, J.: Chunkyspread: Multi-tree Unstructured Peer-to-Peer. In: Proc. Int. Workshop on Peer-to-Peer Systems (IPTPS) (2006)
13. Wang, W., Helder, D.A., Jamin, S., Zhang, L.: Overlay Optimizations for End-host Multicast. In: Networked Group Communications (2002)
14. Zhang, M., Tang, Y., Zhao, L., Luo, J.-G., Yang, S.-Q.: Gridmedia: A Multi-Sender Based Peer-to-Peer Multicast System for Video Streaming. In: IEEE Int. Conference on Multimedia and Expo (ICME), pp. 614–617. IEEE Computer Society Press, Los Alamitos (2005)
15. Zhang, X., Liu, J., Li, B., Yum, Y.: CoolStreaming/DONet: A Data-Driven Overlay Network for Peer-to-Peer Live Media Streaming. In: Proc. Annual IEEE Conference on Computer Communications (INFOCOM), pp. 2102–2111. IEEE Computer Society Press, Los Alamitos (2005)