ContinuStreaming: Achieving High Playback Continuity of Gossip-based Peer-to-Peer Streaming

Zhenhua Li
State Key Lab for Novel Software Technology
Nanjing University, Nanjing, P. R. China
lizhenhua@dislab.nju.edu.cn

Jiannong Cao
Internet and Mobile Computing Lab
Hong Kong Polytechnic University, Hong Kong
csjcao@comp.polyu.edu.hk

Guihai Chen
State Key Lab for Novel Software Technology
Nanjing University, Nanjing, P. R. China
gchen@nju.edu.cn

Abstract

Gossip-based Peer-to-Peer(P2P) streaming has been proved to be an effective and resilient method to stream qualified media contents in dynamic and heterogeneous network environments. Because of the intrinsic randomness of gossiping, some data segments cannot be disseminated to every node in time, which seriously affects the media playback continuity. In this paper we describe the design of ContinuStreaming, a gossip-based P2P streaming system which can maintain high resilience and low overhead while bring a novel and critical property — full coverage of the data dissemination. With the help of DHT, data segments which are likely to be missed by the gossip-based data scheduling can be quickly fetched by the on-demand data retrieval so as to guarantee continuous playback. We discuss the results of both theoretical analysis and comprehensive simulations on various real-trace overlay topologies to demonstrate the effectiveness of our system. Simulation results show that ContinuStreaming outperforms the existing representative gossip-based P2P streaming systems by increasing the playback continuity very close to 1.0 with only 4% or less extra overhead.

1 Introduction

Existing P2P streaming systems can be broadly classified into two categories: tree-based and gossip-based. Tree-based systems organize nodes into a multicast tree. The root of the tree is the media source and data segments are always delivered from parent to sons. Tree-based method can minimize redundant data segments and ensure full coverage of data dissemination, but cannot well adapt to network dynamics because the failure of a single node will partition the tree to a forest. Besides, all leaf nodes are just consumers and contribute little to other nodes. Taking these into consideration, SplitStream [1] and CoopNet [7] use multiple trees to increase the resilience and balance the load. However, multiple trees bring much higher maintenance overhead.

In recent years, gossip-based P2P streaming has been widely investigated and proved to be an effective and resilient method to stream qualified media contents in dynamic and heterogeneous network environments. In a typical gossip algorithm [3], every node maintains a limited number of neighbors and sends a newly generated or received data segment to a random subset of its neighbors. The random choice of data forwarding targets achieves high resilience to node failures and enables distributed operations. Nevertheless, gossip alone for streaming is ineffective because random push may cause significant redundancy, which is particularly severe for high-bandwidth streaming applications. As a result, existing gossip-based P2P streaming systems, e.g. CoolStreaming [13], PeerStreaming [5] and AnySee [6], adopt a smart *pull* gossip algorithm: every node periodically exchanges data availability information with its neighbors and then retrieves required data segments from a subset of its neighbors. However, because their data dissemination still bears much randomness and uncertainty, it is very difficult to guarantee that each data segment is disseminated to all the nodes before the playback deadline. Consequently, these systems cannot ensure high media playback continuity which is a critical performance metric of media streaming.

To increase playback continuity, most existing methods

978-1-4244-1694-3/08/$25.00 ©2008 IEEE

focus on designing a good data scheduling algorithm for a node to retrieve as many *good* data segments as possible from its neighbors' buffers (here *good* can be seen as rare, urgent and so on). A good data scheduling algorithm is necessary but far less than sufficient. For example, node A wants a data segment d urgently but it encounters one of the following cases : 1) none of A's neighbors has ever received d; 2) A's neighbor B has received d but d has been playbacked by B and removed from B's buffer; 3) A's neighbor C has d in its buffer but C does not have sufficient available bandwidth to send d. In each of the above cases, data scheduling algorithm does not work.

In this paper we design ContinuStreaming, a gossip-based P2P streaming system assisted by DHT, which can maintain high resilience and low overhead while bring a novel and critical property — full coverage of the data dissemination. DHT (Distributed Hash Table) is an elegant facility which provides efficient and scalable distributed data storage and retrieval in a wide-area network environment [10, 9]. With the help of DHT, every data segment would be stored in k nodes (k is a small constant). Every node continuously predicts which data segments are likely to be missed by the gossip-based data scheduling, and then triggers its on-demand data retrieval algorithm to quickly fetch the data in time so as to guarantee continuous playback. The above-mentioned mechanism is referred to as "pre-fetch" in the remaining part of this paper.

We have done theoretical analysis about the playback continuity of our system and compared the theoretical results with the simulation results. Comprehensive simulations are performed on various real-trace overlay topologies scaling from 100 to 10000 nodes under different network environments. Their results show that ContinuStreaming outperforms the existing representative gossip-based P2P streaming systems by increasing the playback continuity very close to 1.0 with only 4% or less extra overhead.

Our contributions can be summarized as follows:

1. To the best of our knowledge, we are the first to design a gossip-based P2P streaming system assisted by DHT to achieve high playback continuity of streaming media. More importantly, the extra overhead brought by our system is very low.

2. We design an *Urgent Line mechanism* for every node to dynamically and adaptively predict which data segments are likely to be missed by the data scheduling algorithm. This mechanism can efficiently avoid unnecessary pre-fetch operations.

3. We demonstrate the effectiveness of our system through theoretical analysis and comprehensive simulations on various real-trace overlay topologies.

The rest of this paper is organized as follows. Section 2 reviews related work on enhancing playback continuity of gossip-based P2P streaming. Section 3 presents the system overview. Section 4 describe in detail design of ContinuStreaming, including its P2P overlay management, data scheduling and on-demand data retrieval. We evaluate the performance by theoretical analysis and comprehensive simulations in Section 5. Finally, we conclude the paper with the discussion of future work in Section 6.

2 Related Work

Research on gossip-based P2P streaming has a quite short history. To our knowledge, Kermarrec et al. [4] are the first to provide a theoretical support for gossip-based reliable protocols. They prove if there are n nodes and each node gossips to $\log n + k$ other nodes on average, the probability that everyone gets the message converges to $e^{-e^{-k}}$. Their theoretical analysis provides guidelines for the design of practical gossip-based P2P membership protocols [3]. The coverage ratio $e^{-e^{-k}}$ is very close to 1.0 but it is an ideal theoretical result without considering the bandwidth constraint, latency and so on.

CoolStreaming [13] utilizes the above gossip-based membership protocol [3] to construct a practical and resilient gossip-based P2P streaming system. In order to enhance playback continuity, CoolStreaming designs a "rarest-first" data scheduling algorithm to assign data segments which own fewer suppliers with higher priority. Such design is useful but not sufficient to achieve high playback continuity. In this paper we propose a data scheduling algorithm which takes both rarity and urgency of data segments into consideration, and we design the on-demand data retrieval algorithm to pre-fetch the potential missed data segments.

Xu et al. [11] consider the problem of media data assignment for a multi-supplier peer-to-peer streaming session. Given a requesting peer and a set of supplying peers with heterogeneous out-bound rates, their algorithm, named OTS_{p2p}, computes optimal media data assignments for P2P streaming sessions to achieve minimum buffering delay and thus to achieve high playback continuity. But OTS_{p2p} has very strict assumptions that can hardly hold in practical gossip-based P2P streaming systems.

Zhang et al. [12] observe that *pure-pull* method in P2P streaming brings tremendous latency and thus propose a *push-pull* system called GridMedia. They classify the streaming packets into pulling packets and pushing packets. A pulling packet is delivered by a neighbor only when the packet is requested, while a pushing packet is relayed by a neighbor as soon as it is received. The main goal of GridMedia is to reduce latency and, as a side effect, improve playback continuity. However, pushing packets would bring considerable communication overhead and it cannot guarantee high playback continuity. Likewise, the P2P live streaming system AnySee [6] utilizes inter-overlay optimization to reduce source-to-end latency and meanwhile improve playback continuity.

The adaptive layered P2P streaming system PALS [8] utilizes a receiver-driven approach for quality adaptive playback of layered encoded streaming media. PALS uses a quality adaptive buffer mechanism to maximize the overall throughput and a sliding window mechanism to prevent senders from sending packets whose deadline has already passed. Its main idea is to dynamically configure the layers of media so as to maintain high playback continuity. Different from PALS, our scheme proposed in this paper is not limited to layered streaming and thus can be applied to a wider range of streaming applications. Besides, the PeerStreaming system [5] employs a "queue and buffer" mechanism for throughput control, load balancing and request indirection, but it provides no guarantee for playback continuity.

3 System Overview

The ContinuStreaming system mainly consists of three components: 1) P2P overlay management, 2) Data scheduling algorithm, and 3) On-demand data retrieval algorithm. In this section we provide an overview of the three components. Their design details will be discussed in the next section.

Gossip-based P2P streaming systems usually utilize an unstructured overlay network as its infrastructure to achieve high resilience and low maintenance cost. However, as mentioned before, the intrinsic randomness of gossiping seriously affects their playback continuity. ContinuStreaming adopts a *lightweight* hybrid P2P overlay network which combines an unstructured overlay and a structured overlay. The structured overlay is just a DHT used to provide efficient and scalable distributed data storage and retrieval. The proposed hybrid P2P overlay is *lightweight* because its DHT is loosely organized and the node state update is mainly achieved by overhearing the routing messages passing by.

The data scheduling algorithm works on the unstructured overlay. It periodically gets data availability information from the connected neighbors and then schedules the retrieval of data segments. Our data scheduling algorithm pays special attention to the rarity and urgency of data segments and greedily retrieves high-priority data segments as early as possible, which helps reduce the number of data segments missing their playback deadlines.

The on-demand data retrieval algorithm works on the structured overlay (i.e. the DHT) where every data segment would be backuped in k nodes (k is a small constant). Every node continuously predicts which data segments are likely to be missed by its data scheduling algorithm and then triggers its on-demand data retrieval algorithm to quickly fetch them in time so as to guarantee high playback continuity.

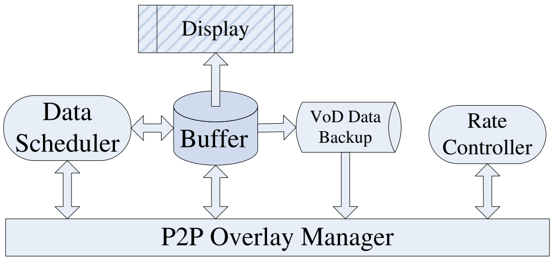

Figure 1. The software architecture of a node.

Figure 1 shows the software architecture of a node. We briefly describe the key modules. (1) *P2P Overlay Manager* behaves as the interface between the local node and overlay network. It maintains and updates the node state. (2) *Data Scheduler* gets the information about available data segments of connected neighbors and arranges where and how to retrieve required data. (3) *VoD Data Backup* stores the data segments this node is responsible to backup. Other nodes can find these data segments from this VoD Data Backup as long as this node is alive. (4) *Rate Controller* monitors and estimates the receiving rate from each connected neighbor.

4 Design of ContinuStreaming

4.1 P2P Overlay Management

Every node in our system maintains a *Peer Table* which is composed of three parts:

(1) *Connected Neighbors* contains M neighbors in the unstructured overlay. These M neighbors are connected by TCP connections and the periodical data exchange is only performed between connected neighbors. If a neighbor is found to have failed or supplied little data to the local node, it will be replaced by an overheard node which has the lowest latency.

(2) *DHT Peers* contains $logN$ DHT peers ordered in *level*. N is the maximum number of nodes the overlay can accommodate, i.e. the size of ID space. For node n, the *only* restriction of its level i DHT peer is that this peer must lie in $[n + 2^{i-1}, n + 2^i)$ in ID space (all numbers are modulo N). Therefore, node n has much freedom in choosing its DHT peers and thus the DHT is loosely organized. All the DHT peers are periodically updated by the overheard nodes for renewal.

(3) *Overheard Nodes* contains H nodes which are the latest overheard. $H = 20$ is usually enough according to our

simulation experience. Every node continually overhears the routing messages passing by and updates the overheard node list using the latest overheard nodes.

<table>
<tr><th colspan="5">Connected Neighbors</th></tr>
<tr><th>No</th><th>PeerID</th><th>IP address</th><th>Latency</th><th>Recent supply rate</th></tr>
<tr><td>1</td><td>369</td><td>210.29.131.34</td><td>5ms</td><td>50kbps</td></tr>
<tr><td>…</td><td>…</td><td>…</td><td>…</td><td>…</td></tr>
<tr><td>M</td><td>672</td><td>165.93.100.77</td><td>15ms</td><td>25kbps</td></tr>
<tr><th colspan="5">DHT Peers</th></tr>
<tr><th>Level</th><th>PeerID</th><th>IP address</th><th colspan="2">Latency</th></tr>
<tr><td>1</td><td>442</td><td>102.119.32.30</td><td colspan="2">25ms</td></tr>
<tr><td>…</td><td>…</td><td>…</td><td colspan="2">…</td></tr>
<tr><td>logN</td><td>891</td><td>219.45.128.49</td><td colspan="2">30ms</td></tr>
<tr><th colspan="5">Overheard Nodes</th></tr>
<tr><th>No</th><th>PeerID</th><th>IP address</th><th colspan="2">Latency</th></tr>
<tr><td>1</td><td>453</td><td>107.124.33.58</td><td colspan="2">15ms</td></tr>
<tr><td>…</td><td>…</td><td>…</td><td colspan="2">…</td></tr>
<tr><td>H</td><td>120</td><td>61.173.76.44</td><td colspan="2">100ms</td></tr>
</table>

Figure 2. Peer Table structure.

Figure 2 demonstrates the detailed structure of Peer Table. Clearly the Connected Neighbors and DHT Peers are both updated according to Overheard Nodes, and Overheard Nodes are updated by local overhearing which requires no extra communication overhead. Therefore, the P2P overlay we design needs low maintenance cost.

A new node A first contacts the *RP (Rendezvous Point) server* to join the overlay network. RP server holds a partial list of joining nodes and assigns a unique *ID* to node A. Then RP server gives node A a short list of several existing nodes which have close IDs as node A. Assuming A gets a list $\{B, C, D, E\}$, A will try to send them *PING* messages (e.g. in UDP packets) to detect which is the nearest alive node. The latency is approximately estimated as $\frac{RTT}{2}$ (*RTT* is the round trip time). If B, C, D are alive and B is nearest to A, then A gets B's Peer Table as the base of its own Peer Table, notifies B, C, D his joining, and tells the RP server E's failure.

The P2P overlay we proposed provides both unicast and multicast support for ContinuStreaming. Unicast function, i.e. the location of a node or a data segment, is supported by the DHT routing algorithm. It is a simple greedy algorithm: for every intermediate node, it chooses in its DHT Peers the clockwise closest peer to the destination as the next hop, until no closer peer can be found. We prove in Appendix the upper bound of routing hops for a DHT location is $\frac{logN}{log(4/3)} \approx 2.41 \times logN$. And we did simulations on DHT networks with different N and n, N is the size of ID space and n is the real number of joining nodes. Simulation results in Figure 3 show the average routing hops is very close to $\frac{\log n}{2}$ and the query success rate is very close to 1.0 even when the overlay is sparse, i.e. when n is much smaller than N.

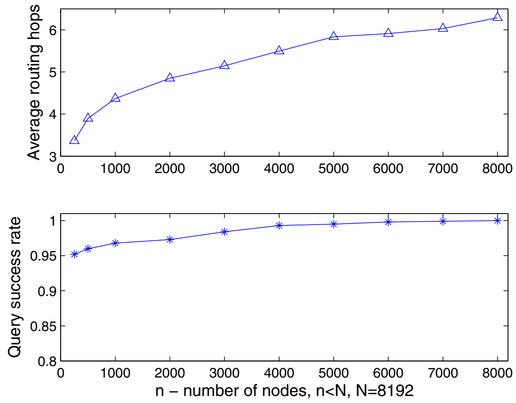

Figure 3. Average routing hops and query success rate of the DHT network.

Multicast function, i.e. the data exchange between connected neighbors, is supported by the data scheduling algorithm. The analysis in [13] shows that in a gossip-based streaming system, the coverage ratio at a given distance d is $1 - e^{-\frac{M\times(M-1)^d-2}{(M-2)\times n}}$, where M is the number of connected neighbors and n is the number of overlay nodes. However, constrained by many factors, the real coverage ratio is much less than this ideal theoretical analysis, and this is why we consider our work to be valuable.

4.2 Data Scheduling

A node periodically exchanges buffer information with its connected neighbors. The exchange period is called a *scheduling period*, denoted as τ. So for every scheduling period, the node's *Data Scheduler* first gets the information about available data segments of connected neighbors. All available data segments are not contained in the local buffer, i.e. they are all fresh to the local node. The related parameters and their descriptions are listed in Table 1.

Taking both the urgency and rarity of each data segment into consideration, the Data Scheduler computes each segment's requesting priority through equations (1) to (3).

$$R_i = \max\{R_{i_1}, R_{i_2}, \cdots, R_{i_{n_i}}\}$$

$$t_i = \frac{id_i - id_{play}}{p} - \frac{1}{R_i} \quad \text{then} \quad urgency_i = \frac{1}{t_i} \qquad (1)$$

Segment i's rarity is considered as the probability that it will be replaced in all its suppliers' buffers, which we think is more reasonable than the traditional computation $rarity_i = \frac{1}{n_i}$.

Parameter	Description
τ	Data scheduling period.
id_i	The id of data segment D_i.
n_i	How many neighbors can supply segment D_i.
I	Total inbound rate of the local node.
R_{i_j}	The receiving rate of segment D_i from the jth neighbor.
R_i	The maximum receiving rate of segment D_i.
id_{play}	id of the segment being played at this moment.
p	The number of segments being played per second.
t_i	Expected deadline left time of segment D_i.
B	Buffer size, i.e. the number of data segments Buffer can accommodate.
p_{i_j}	Segment D_i's position in the jth neighbor's buffer. The replacement strategy of Buffer is FIFO, and the position is the distance from the tail of Buffer.
$urgency_i$	Urgency of segment D_i, i.e. the probability of D_i to miss its deadline.
$rarity_i$	Rarity of segment D_i, i.e. the probability that D_i will be replaced in all its suppliers' buffers.
$priority_i$	Requesting priority of segment D_i. It takes both urgency and rarity into consideration.

Table 1. Parameters for data scheduling

$$rarity_i = \left(\frac{p_{i_1}}{B}\right) \times \left(\frac{p_{i_2}}{B}\right) \times \cdots \times \left(\frac{p_{i_{n_i}}}{B}\right) \qquad (2)$$

And finally,

$$priority_i = \max\{urgency_i, rarity_i\} \qquad (3)$$

Having got each segment's priority, the Data Scheduler sorts them in the descending order of their priority. Suppose the order is like $D_1, D_2, D_3, \cdots, D_m$. For a segment D_i, there may exist several neighbors who can supply it, and usually the neighbor who can send it earliest will become D_i's supplier. But here we encounter a conflict problem where two segments choose the same supplier, so one of them needs to wait or choose another supplier. The problem is: how to choose a proper supplier for every data segment so that the number of segments missing deadlines or being replaced can be the minimal? In fact, even a simple special case of this problem is NP-hard (known as the *Parallel machine scheduling problem* [2]), so we use a greedy algorithm trying to get high-priority segments as early as possible, see Algorithm 1. In this algorithm, the scheduler makes greedy efforts to minimize the expected receiving time t_{min} of every data segment. For segment D_i, the scheduler checks all its suppliers to find a proper supplier which can send D_i earliest.

```
Algorithm 1: Data Scheduling Algorithm
Input  : (1) Data segments $D_1, D_2, D_3, \cdots, D_m$, in
         descending order of priority; (2) Supplier set
         for each date segment: $S_1, S_2, \cdots, S_m$; (3)
         Sending rate of node $j$: R(j); (4) Queuing
         time of node $j$: $\tau(j)$, initially $\tau(j) = 0$;
Output: $supplier_i$ for each data segment $D_i$.
compute the maximum number of inbound segments
for this scheduling period: $\min(m, I \times \tau)$;
for $i = 1$ to $\min(m, I \times \tau)$ do
    set segment $D_i$'s earliest receiving time $t_{min}$=$\infty$;
    suppose $S_i$ contains $k$ suppliers $S_{i_1}, \cdots, S_{i_k}$;
    for $j = 1$ to $k$ do
        compute the expected transfer time of $D_i$ from
        $S_{i_j}$: $t_{trans} = \frac{1}{R(S_{i_j})}$;
        if $t_{trans}$+$\tau(S_{i_j})$<$t_{min}$ and $t_{trans}$+$\tau(S_{i_j})$<$\tau$
        then
            $t_{min} \leftarrow t_{trans} + \tau(S_{i_j})$;
            $supplier_i \leftarrow S_{i_j}$;
        end
    end
    if $supplier_i \neq null$ then
        $\tau(supplier_i) \leftarrow t_{min}$;
    end
end
```

4.3 On-demand Data Retrieval

For every scheduling period, the Data Scheduler predicts which data segments are likely to be missed by the data scheduling algorithm. Since it is impossible to accurately predict them, we utilize an *Urgent Line mechanism* for dynamic and adaptive prediction.

In Figure 4, the data segments in the buffer is divided into two parts by the *Urgent Line*. Only the white data segments on the left of the urgent line are predicted *missed*. Required parameters for on-demand data retrieval are in Table 2.

Then we have

$$id_{urgent} = id_{head} + \alpha \times B \qquad (4)$$

So α is a critical parameter for on-demand data retrieval algorithm and how to compute α will be explained by the end of this subsection. All the white data segments with

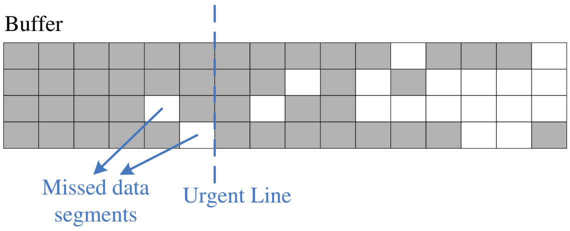

Figure 4. Urgent line of the buffer. White segments have not been got.

Table 2. Parameters for on-demand data retrieval

Parameter	Description
α	The urgent ratio of Buffer. It is dynamically computed.
B	Buffer size, i.e. the number of data segments Buffer can accommodate.
id_{head}	The id of the data segment at the head of Buffer.
id_{urgent}	The id of the data segment at the urgent line.
N_{miss}	The number of data segment predicted to be *missed*.
k	Each data segment is backuped in k nodes.
l	The on-demand data retrieval algorithm can get at most l missed data segments per time.

$id \le id_{urgent}$ are predicted *missed*, and their total number is N_{miss}. Then we have three cases to deal with:

- Case 1: If N_{miss}=0 then the on-demand data retrieval algorithm is not triggered.
- Case 2: If $N_{miss} \le l$ then the on-demand data retrieval algorithm is triggered to get all the missed data segments in parallel.
- Case 3: If $N_{miss} > l$ then the on-demand data retrieval algorithm is also not triggered to avoid too much prefetch traffic cost.

For a node with ID: n, suppose its closest DHT peer in its Peer Table is n_1, then n must store in its *VoD Data Backup* the received data segments with id satisfying

$$hash(id \times i)\%N \in [n, n_1), i = 1, 2, \cdots, k. \quad (5)$$

so that every data segment is expected to be backuped in k different nodes. $hash()$ can be any common hash function and $\%$ denotes *modulo*. The reason why we use $id \times i$ to hash is to backup a data segment into dispersed nodes so as to balance load. For example, if we use $id + i$ to hash, the data segments with close ids may aggregate in the same node to bring this node heavy load.

When node n wants to leave, it should first find the node n' which is counter-clockwise closest to n and then hand over the data segments in its VoD Data Backup to n'. When node n fails abruptly, the data handover becomes difficult. However, as time elapses, old data segments backuped by n' gradually become useless and n' takes over n's responsibility to backup new data segments, so n's failure will not bring much bad impact.

Algorithm 2 shows the on-demand data retrieval algorithm. Suppose the missed data segments are $D_1, \cdots, D_m$. For a data segment D_i, the algorithm first locates the k nodes that may have backuped D_i, and then selects the node with the highest available sending rate as D_i's on-demand supplier. When node n asks one backup node n_0 for data segment D_i, there is a probability P_{fail} that n_0 has not got D_i yet. Assume n and n_0 have the same probability to get D_i by the data scheduling algorithm, then $P_{fail} = \frac{1}{2}$ in average. Therefore, the probability n cannot get D_i from any of the k backup nodes is $(\frac{1}{2})^k$, which is quite low. $D_1, D_2, \cdots, D_m$ are downloaded directly (in UDP packets) from their on-demand suppliers in parallel to accelerate the retrieval process. When triggered to run, the on-demand data retrieval algorithm shares the inbound rate with the data scheduling algorithm.

```
Algorithm 2: On-demand Data Retrieval Algorithm
Input  : The missed data segments $D_1, D_2, \cdots, D_m$,
         $m \le l$, in ascending order of data id;
Output: on-demand $supplier_i$ for each segment $D_i$.
for $i = 1$ to $m$ do
    In order to get $D_i$, send $k$ routing messages
    targeted at $k$ nodes $n_1, n_2, \cdots, n_k$ in parallel:
    $n_i = hash(D_i \times i)\%N, i = 1, 2, \cdots, k$;
    set $D_i$'s maximum receiving rate $R_i = 0$;
    for $j = 1$ to $k$ do
        For the routing message targeted at $n_j$, the
        node $n'_j$ counter-clockwise closest to $n_j$ will
        be found;
        if node $n'_j$ has backuped $D_i$ and $n'_j$'s
        available sending rate $> R_i$ then
            on-demand $supplier_i \leftarrow n'_j$;
        end
    end
end
```

Now we discuss how to compute the critical parameter α which determines the urgent ratio of Buffer. α must be dynamically computed to properly predict how many data segments in Buffer are urgent. If α is too small, the on-demand data retrieval algorithm cannot catch the speed of playback and thus the playback continuity will be degraded. But if α is too large, many data segments predicted missed do not really need to be pre-fetched by on-demand data retrieval algorithm, i.e. N_{miss} is computed much larger than its real value, and thus much more communication cost will be incurred than required. Therefore, α will be increased or decreased as the situation changes.

Suppose t_{hop} is the average expected time to route one hop in the overlay network, then the average expected time t_{fetch} for the on-demand data retrieval algorithm to fetch a data segment is:

$$t_{fetch} = t_{locate} + t_{reply} + t_{request} + t_{retrieve} \qquad (6)$$

$$t_{fetch} \approx \frac{\log n}{2} \times t_{hop} + 3 \times t_{hop} = (\frac{\log n}{2} + 3) \times t_{hop} \quad (7)$$

n is the expected number of overlay nodes and it does not need to be configured accurately. For example, we can set $n = \frac{N}{2}$ initially. t_{hop} is also an approximate estimation from our simulation experience.

During one scheduling period τ, the on-demand data retrieval algorithm must allocate enough time for a missed data segment to be fetched before its playback deadline, so we have:

$$\alpha \times B > p \times \tau \quad \text{and} \quad \alpha \times B > p \times t_{fetch} \qquad (8)$$

that is

$$\alpha > \frac{p}{B} \times \max(\tau, t_{fetch}) \qquad (9)$$

so we get the lower bound of α and set the initial $\alpha = \frac{p}{B} \times \max(\tau, t_{fetch})$.

As time goes, situation changes. α should be updated in the following two cases:

- Case 1 - Overdue data: When there exist pre-fetched data segments arriving late, i.e. the missed data segments cannot be fetched by the on-demand data retrieval algorithm in time, then $\alpha \leftarrow \alpha + \frac{p \times t_{hop}}{B}$;
- Case 2 - Repeated data: When there exist pre-fetched data segments which can still be got by the data scheduling algorithm before their deadlines, i.e. they are repeatedly found by both the data scheduling and on-demand data retrieval algorithms, then $\alpha \leftarrow \alpha - \frac{p \times t_{hop}}{B}$;

The increment and decrement of α are both set as $\frac{p \times t_{hop}}{B}$ which is quite small to make α change smoothly. For Case 2, each pre-fetched data segment should have a *tag* which indicates this segment is got by the on-demand data retrieval algorithm, so the data scheduling algorithm can recognize repeated data segments then.

5 Performance Evaluation

5.1 Theoretical Analysis

In this section we analyze the playback continuity of ContinuStreaming. We use the *Poisson Process* to model the arrival of data segments in gossip-based streaming, because *Poisson Process* has *independent* and *stationary* increments, which is also the properties of gossip process. More specifically, suppose $N(t)$ represents the number of data segments that have arrived at a node during time t, and then we have $P\{N(t) = n\} = e^{-\lambda t}\frac{(\lambda t)^n}{n!}$, where λ is a constant parameter. It is not difficult to prove

$$E[N(t)] = \sum_{n=0}^{\infty} n \cdot P\{N(t) = n\} = \sum_{n=0}^{\infty} n \cdot e^{-\lambda t}\frac{(\lambda t)^n}{n!} = \lambda t \qquad (10)$$

and this is why the parameter λ is called the *arrival rate* of the process. For gossip-based streaming, we can approximately regard the inbound rate of the local node I as its λ.

In order to playback the media continuously, $\lambda > p$ must holds, where p is the playback rate. Now we consider the probability that the on-demand data retrieval algorithm is triggered per scheduling period. During each data scheduling period τ, if the number of arriving data segments $N(\tau)$ is smaller than the number of data segments required to playback, the on-demand data retrieval algorithm is expected to be triggered. So

$P\{$On-demand data retrieval Algorithm is triggered$\}$ =

$$P\{N(\tau) \leq p\tau\} = \sum_{n=0}^{p\tau} e^{-\lambda\tau}\frac{(\lambda\tau)^n}{n!} \qquad (11)$$

and the expected number of missed data segments is

$$N_{miss} = \sum_{n=0}^{p\tau-1} (p\tau - n) P\{N(\tau) = n\} \qquad (12)$$

$$= p\tau \sum_{n=0}^{p\tau-1} P\{N(\tau) = n\} - \sum_{n=0}^{p\tau-1} nP\{N(\tau) = n\}$$

$$= p\tau \sum_{n=0}^{p\tau-1} e^{-\lambda\tau}\frac{(\lambda\tau)^n}{n!} - \sum_{n=0}^{p\tau-1} ne^{-\lambda\tau}\frac{(\lambda\tau)^n}{n!}.$$

In the former section we have estimated that if every data segment is backuped in k nodes then the probability that a certain node cannot get a certain data segment is $(\frac{1}{2})^k$. Now there exist N_{miss} data segments requiring to be pre-fetched by the on-demand data retrieval algorithm, so the probability to successfully get all of them is $(1 - (\frac{1}{2})^k)^{N_{miss}}$. Then we can compute the old playback continuity PC_{old} and the new playback continuity PC_{new} as follows:

$$PC_{old} = 1 - P\{N(\tau) \leq p\tau\} \quad (13)$$

$$PC_{new} = 1 - P\{N(\tau) \leq p\tau\}(1 - (1 - (\frac{1}{2})^k)^{N_{miss}}) \quad (14)$$

so that

$$\Delta = PC_{new} - PC_{old} = P\{N(\tau) \leq p\tau\}(1 - (\frac{1}{2})^k)^{N_{miss}} \quad (15)$$

In order to check the validity of the theoretical analysis above, we compare it with simulation results with 1000 nodes. The detailed simulation methodology will be described in next subsection. Some major parameters are: play rate $p = 10$, average inbound rate of nodes is $I = 15$ and the scheduling period $\tau = 1s$. Every data segment is backuped in $k = 4$ nodes. The following table compares the theoretical results and the simulation results. "Homogeneous" means each node has the same inbound rate while "Heterogeneous" means the nodes have different inbound rates.

Environment	PC_{old}	PC_{new}	Δ
Theoretical result with λ=15	0.8815	0.9989	0.1174
Theoretical result with λ=14	0.8243	0.9975	0.1732
Homogeneous and static environment	0.8748	0.9979	0.1231
Homogeneous and dynamic environment	0.8520	0.9803	0.1283
Heterogeneous and static environment	0.8431	0.9726	0.1295
Heterogeneous and dynamic environment	0.8166	0.9537	0.1371

The table above illustrates the simulation results approximately lies between the theoretical results with λ=14 and λ=15, because in each simulation every node spares a small part of its inbound rate (I=15) for data pre-fetching. However, PC_{new} of simulations in heterogeneous or dynamic environments is often a little lower than the theoretical results mainly because of practical constraints and churns.

5.2 Simulation Methodology

We perform simulations on 30 real-trace unstructured overlay topologies whose data was collected from Dec. 7th 2000 to June 15th 2001 on http://dss.clip2.com (unavailable now). The data contains each node's ID, IP, port, ping time (from a central node), speed and so on, but we just use the ID, IP and ping time information. The trace topologies scale from 100 to 10000 nodes, with average node degree from less than 1 to 3.5. Because the average node degree is too small for media streaming, we add random edges into the overlay to let every node hold M=5 connected neighbors. According to our simulation experience, M=5 is usually a good practical choice and using a larger M cannot bring more benefit.

Among all the existing gossip-based P2P streaming systems, CoolStreaming [13] is the most representative. So we compare the performances of our ContinuStreaming with CoolStreaming under the same network environments. The default streaming rate is 300 Kbps and each data segment contains 30 Kbp, so the playback rate $p = \frac{300Kbp}{30Kbp} = 10$. Each node maintains a buffer of 600 data segments, i.e. 60 seconds of the streaming data. We randomly arrange inbound rate (from 300 Kbps to 1 Mbps) to each node and let the average inbound rate be both 450 Kbps, i.e. $I \in [10, 33]$ and $I = 15$ in average. The arrangement of outbound rate is alike. An exception is that the source node has zero inbound rate and much larger outbound rate, usually its $I = 100$. The data scheduling period $\tau = 1.0$ second.

The physical latency between two overlay nodes is computed as the difference between their real-trace ping times from a central node. This estimation of latency may be not accurate but reasonable for our simulation settings. The average latency of one overlay hop $t_{hop} \approx 50ms$, so the average expected time t_{fetch} for the on-demand data retrieval algorithm to pre-fetch a data segment is $t_{fetch} \approx (\frac{\log n}{2} + 3) \times t_{hop} = 8 \times 50ms = 400ms$, where $n = 1000$. Then the pre-fetch urgent ratio $\alpha = \frac{p}{B} \times \max(\tau, t_{fetch}) = \frac{10}{600} \times \max(1s, 400ms) = \frac{1}{60}$ initially. Besides, each data segment is backuped in $k = 4$ nodes. The on-demand data retrieval algorithm of every node can get at most $l = 5$ missed data segments per scheduling period. To create a dynamic network environment, we randomly let 5% old nodes leave and 5% new nodes join per scheduling period. A new joining node does not need to retrieve all the disseminated data segments from the source, and it just requests the data segments being played or will be played by its neighbors. That is to say, a new joining node starts its media playback by following its neighbors' current steps.

5.3 Metrics

We mainly use the following three metrics to evaluate the performance of our ContinuStreaming system.

1. *Playback continuity*: For every round (a round is a data scheduling period) we record the ratio of nodes that have collected sufficient data segments to playback. This ratio is so-called *playback continuity*. Some former papers define the *continuity index*, which is the ratio of segments that arrive before deadlines, to evaluate playback continuity. However, high continuity index does not guarantee continuous playback and we consider the playback continuity we define to be more accurate.

2. *Control overhead*: For every round each node exchanges buffer information with its neighbors. *Control overhead* is defined as the ratio of communication cost for buffer information exchange over the real communication cost for data segments transfer.

3. *Pre-fetch overhead*: To pre-fetch a data segment, the on-demand data retrieval algorithm should first locate k backup nodes and then choose one node as the supplier. This process includes about $k \times (\frac{\log n}{2} + 1) + 1$ routing messages on the DHT and the transfer cost for the missed data segment. So *pre-fetch overhead* is defined as the ratio of above-mentioned communication cost over the real communication cost for data segments transfer.

5.4 Simulation Results

5.4.1 Playback continuity

We first track the playback continuity of ContinuStreaming and CoolStreaming in a static network environment with 1000 nodes and a single source. Since the streaming system usually enters its stable playback phase within 30 seconds, we record the overall playback continuity of the system from 0 to 30 seconds. Figure 5 shows that CoolStreaming enters its stable phase in 26 seconds with playback continuity around 0.83, while ContinuStreaming in 18 seconds with playback continuity around 0.97. Our data pre-fetch scheme accelerates the streaming system's entering its stable phase and improves the playback continuity much closer to 1.0.

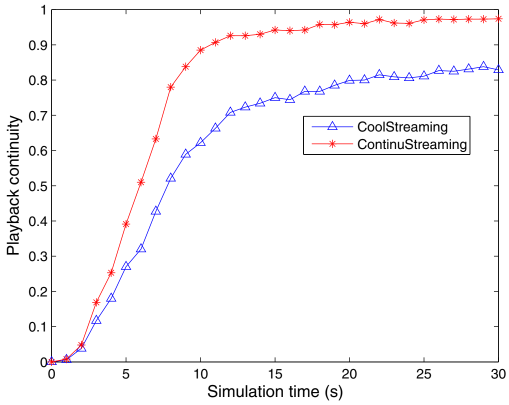

Figure 5. Playback continuity track in a static network environment.

We further track the playback continuity in a dynamic environment still with 1000 nodes. The track is illustrated in Figure 6. CoolStreaming enters its stable phase in 27 seconds with playback continuity around 0.78, while ContinuStreaming in 20 seconds with playback continuity around 0.95. Though the dynamic $PC_{new} = 0.95$ is smaller than the static $PC_{new} = 0.97$, the dynamic playback continuity increment 0.17 is larger than the static increment 0.14. From this point of view, ContinuStreaming improves playback continuity more in dynamic environments.

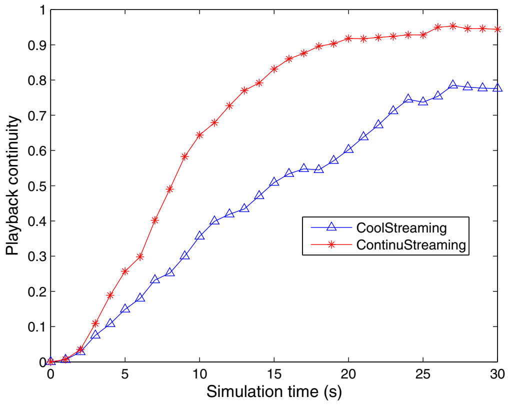

Figure 6. Playback continuity track in a dynamic network environment.

Besides, we examine the playback continuity (in stable phase) of overlay networks with different sizes, ranging from 100 to 8000, working in static and dynamic network environments. Every node has $M = 5$ neighbors. Figure 7 and Figure 8 shows the simulation results. We can see that as the network size increases, both PC_{new} and PC_{old} decrease but the playback continuity increment (Δ=PC_{new}-PC_{old}) increases, so a larger network benefits more from ContinuStreaming. Finally, we observe that using a larger M cannot bring notable increment to playback continuity, because the main constraint lies in the inbound rate of nodes.

5.4.2 Control overhead

For every round each node exchanges buffer information with its neighbors, and the control overhead comes from the communication of buffer information. We record the control overhead of networks with different sizes and different M. Figure 9 illustrates the record information. The buffer can accommodate $B = 600$ data segments, so we use 600 bits to record the data availability, with bit 1 indicating this segment is available and bit 0 indicating this segment is unavailable. The id of the first segment in the buffer is indicated by 20 bits because the source will disseminate at most $3600 \times 10 \times 24 = 864000 \in (2^{19}, 2^{20})$ data segments per hour. Therefore, getting the buffer information of

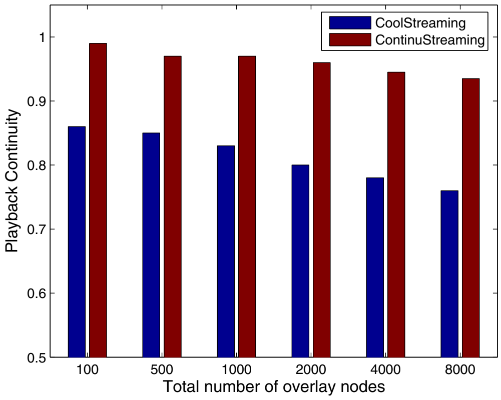

Figure 7. Playback continuity of networks with different sizes under static environments.

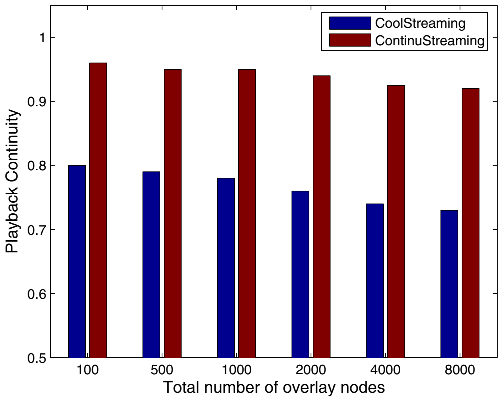

Figure 8. Playback continuity of networks with different sizes under dynamic environments.

one neighbor takes 620 bits' communication cost in total. Every data segment contains 30 Kbp data of streaming. If every node can get $p = 10$ required data segments from its neighbors per round, i.e. every node's playback continuity is 1.0, then the control overhead is about $\frac{620 \times M}{30 \times 1024 \times 10} = \frac{M}{495}$. Simulation results in Figure 9 are a little larger than $\frac{M}{495}$ because most nodes' playback continuity is smaller than 1.0. The control overhead of all networks is below 0.02, which takes only a minor part of the total communication cost. Besides, the control overhead of ContinuStreaming is similar to that of CoolStreaming because their buffer information exchange mechanisms are very alike.

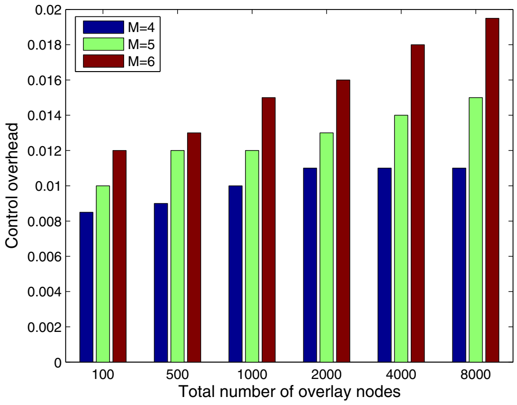

Figure 9. Control overhead of networks with different sizes.

5.4.3 Pre-fetch overhead

To pre-fetch a data segment, the on-demand data retrieval algorithm should first locate $k = 4$ backup nodes for that data segment and then choose one node as the supplier. The pre-fetch overhead consists of routing messages on the DHT and the transfer cost for the missed data segments. Each routing message costs 10 bytes, i.e. 80 bits. Because pre-fetching a data segment usually requires $k \times (\frac{\log n}{2} + 1) + 1$ routing messages, the total communication cost to pre-fetch a data segment can be estimated as about $(4 \times (\frac{\log n}{2} + 1) + 1) \times 80 + 30 \times 1024 \approx 33000$ bits ($n \leq 8000$).

We track the pre-fetch overhead of a network with 1000 nodes both in a static and dynamic environment ($M = 5$). Figure 10 shows the results. At the beginning the pre-fetch overhead is very little because the number of missed data segments of most nodes N_{miss} is larger than the threshold l (in fact most nodes did not know the existence of the media source then) and thus the on-demand pre-fetching is not triggered to run. After several seconds the pre-fetch overhead rises into a bit larger than that in the stable phase because all the nodes had known the source then. The pre-fetch overhead in the stable phase is merely 0.023 for a static environment and 0.03 for a dynamic environment.

Figure 11 shows the pre-fetch overhead of networks with different sizes working in static and dynamic environments. $M = 5$ for all the networks. Not surprisingly, for each size the pre-fetch overhead in dynamic environments is larger than that in static environments mainly because more data segments will be missed by data scheduling algorithm in dynamic networks. The pre-fetch overhead of all networks is below 0.04, which indicates that our pre-fetch scheme brings minor extra communication cost compared with the real communication cost for data segments transfer. Since the control overhead of ContinuStreaming is similar to that

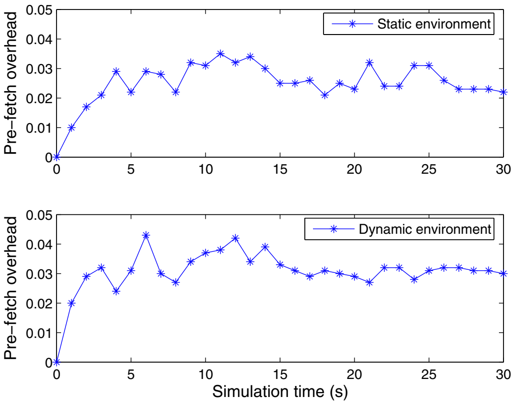

Figure 10. Pre-fetch overhead track of a network with 1000 nodes.

of CoolStreaming, the pre-fetch overhead is just the extra overhead brought by ContinuStreaming compared with CoolStreaming.

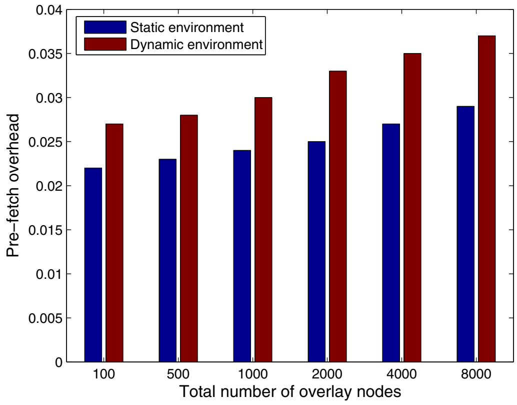

Figure 11. Pre-fetch overhead of networks with different sizes.

6 Conclusions and Future Work

Gossip-based P2P streaming has proved to be a novel and effective method for media streaming in dynamic and heterogeneous network environments. However, it suffers from the randomness of gossip-style data dissemination and thus cannot guarantee high playback continuity. In this paper we propose ContinuStreaming, a gossip-based P2P streaming system assisted by DHT, which can achieve high playback continuity for the streaming media with very low overhead. Both the theoretical and simulation results confirm the effectiveness of our system. Next step we want to validate our system in a practical and worldwide network environment. We are applying to join the "Planet-lab" and expecting to implement ContinuStreaming above this platform to check the performance.

7 Acknowledgements

The work is partly supported by Hong Kong RGC under the CERG grant PolyU 5103/06E, China NSF grants (60573131, 60673154,60721002), Jiangsu High-Tech Research Project of China (BG2007039), and China 973 project (2006CB303000).

References

[1] M. Castro, P. Druschel, A. Kermarrec, A. Nandi, A. Rowstron, and A. Singh. SplitStream: high-bandwidth multicast in cooperative environments. *Proceedings of the 19th ACM SOSP*, pages 298–313, 2003.

[2] T. Cormen, C. Leiserson, and R. Rivest. Introduction to algorithms. *MIT Press Cambridge, MA, USA*, 1990.

[3] A. Ganesh, A. Kermarrec, and L. Massoulie. Peer-to-peer membership management for gossip-based protocols. *IEEE Transactions on Computers*, 52(2):139–149, 2003.

[4] A. Kermarrec, L. Massoulie, and A. Ganesh. Probabilistic reliable dissemination in large-scale systems. *IEEE Transactions on Parallel and Distributed Systems*, 14(3):248–258, 2003.

[5] J. Li. PeerStreaming: An On-Demand Peer-to-Peer Media Streaming Solution Based On A Receiver-Driven Streaming Protocol. *2005 IEEE 7th Workshop on Multimedia Signal Processing*, pages 1–4, 2005.

[6] X. Liao, H. Jin, Y. Liu, L. Ni, and D. Deng. AnySee: Peer-to-Peer Live Streaming. *Proceedings of IEEE INFOCOM*, 2006.

[7] V. Padmanabhan, H. Wang, and P. Chou. Resilient peer-to-peer streaming. *Proceedings. 11th IEEE International Conference on Network Protocols, 2003.*, pages 16–27, 2003.

[8] R. Rejaie and A. Ortega. PALS: peer-to-peer adaptive layered streaming. *Proceedings of the 13th NOSSDAV*, pages 153–161, 2003.

[9] A. Rowstron and P. Druschel. Pastry: Scalable, distributed object location and routing for large-scale peer-to-peer systems. *IFIP/ACM Conference on Distributed Systems Platforms (Middleware)*, 11:329–350, 2001.

[10] I. Stoica, R. Morris, D. Karger, M. Kaashoek, and H. Balakrishnan. Chord: A scalable peer-to-peer lookup service for internet applications. *Proceedings of the 2001 SIGCOMM conference*, 31(4):149–160, 2001.

[11] D. Xu, M. Hefeeda, S. Hambrusch, and B. Bhargava. On peer-to-peer media streaming. *Proceedings. 22nd International Conference on Distributed Computing Systems, 2002.*, pages 363–371, 2002.

[12] M. Zhang, J. Luo, L. Zhao, and S. Yang. A peer-to-peer network for live media streaming using a push-pull approach.

Proceedings of the 13th ACM international conference on Multimedia, pages 287–290, 2005.

[13] X. Zhang, J. Liu, B. Li, and T. Yum. CoolStreaming/DONet: A Data-driven Overlay Network for Peer-to-Peer Live Media Streaming. *Proc. IEEE Infocom*, 2005.

Appendix: The upper bound of routing hops for a DHT location is $\frac{logN}{log(4/3)} \approx 2.41 \times logN$. (Section 4.1)

Proof: Suppose node S wants to find node T and the routing message should be forwarded like $S \rightarrow H_1 \rightarrow H_2 \rightarrow \cdots$. The clockwise distance from S to T $dist(S,T) \in [2^{d-1}, 2^d], d < logN$. S first chooses its clockwise closet neighbor H_1 as the next hop, then H_1 must be S's level d or d-1 neighbor.

Case 1: H_1 is S's level d neighbor, then $dist(H_1,T) \leq 2^{d-1}, dist(S,H_1) \geq 2^{d-1}$.

Case 2: H_1 is S's level d-1 neighbor, then $dist(H_1,T) < 2^{d-1} + 2^{d-2} = 3 \times 2^{d-2}, dist(S,H_1) \geq 2^{d-2}$.

In order to get the upper bound of routing hops, we adopt the worst case, i.e. Case 2, for every hop. Then we can get $dist(H_1,T) < \frac{3}{4} \times dist(S,T)$ by the following steps:

because $dist(S,T) = dist(S,H_1) + dist(H_1,T)$,
then $\frac{3}{4} \times dist(S,T) - dist(H_1,T)$
$= \frac{3}{4} \times dist(S,H_1) - \frac{1}{4} \times dist(H_1,T)$
$= \frac{3 \times dist(S,H_1) - dist(H_1,T)}{4}$;
since $dist(S,H_1) \geq 2^{d-2}$ and $dist(H_1,T) < 3 \times 2^{d-2}$, then $\frac{3}{4} \times dist(S,T) - dist(H_1,T) > 0$, i.e. $dist(H_1,T) < \frac{3}{4} \times dist(S,T)$.

Likewise, $dist(H_2,T) < \frac{3}{4} \times dist(H_1,T) < (\frac{3}{4})^2 \times dist(S,T)$, $dist(H_i,T) < (\frac{3}{4})^i \times dist(S,T) < (\frac{3}{4})^i \times N$. Let $dist(H_i,T) = 1$, i.e. H_i is sure to find target T, then $(\frac{3}{4})^i \times N > 1, i > \frac{logN}{log(4/3)}$ which is the upper bound. ■