TAG: A TREE-ASSISTED GOSSIP PROTOCOL FOR ON-DEMAND VIDEO STREAMING

by

Ming Zhou
Bachelor of Computer Science, Beijing University, 1997

THESIS SUBMITTED IN PARTIAL FULFILLMENT OF THE REQUIREMENTS FOR THE DEGREE OF MASTER OF SCIENCE

In the
School of Computing Science

© Ming Zhou 2005

SIMON FRASER UNIVERSITY
Summer 2005

All rights reserved. This work may not be reproduced in whole or in part, by photocopy or other means, without permission of the author.
APPROVAL

Name: Ming Zhou
Degree: Master of Science
Title of Thesis: TAG: A Tree-Assisted Gossip Protocol for On-Demand Video Streaming

Examin ing Committee:

Chair: Professor Andrei Bulatov
Assistant Professor of Computing Science

Professor Jiangchuan Liu
Senior Supervisor
Assistant Professor of Computing Science

Professor Qianping Gu
Supervisor
Professor of Computing Science

Professor Mohamed M. Hefeeda
Examiner
Assistant Professor of Computing Science

Date Approved: July 29, 2005
ABSTRACT

While a tree topology is often advocated for overlay video streaming due to its scalability, it suffers from discontinuous playback under highly dynamic network environments. On the other hand, gossip protocols using random message dissemination, though robust, fail to meet the real-time constraints for streaming applications. In this master thesis, I proposed TAG, a Tree-Assisted Gossip protocol, which adopts a tree structure with time indexing to accommodate asynchronous requests, and an efficient pull-based gossip algorithm to mitigate the impact of network dynamicity. It seamlessly integrates these two approaches and realizes their best features, namely, low delay with a regular tree topology, and robust delivery with smart switching among multiple paths, thus making effective use of the available bandwidth. Performance of TAG was evaluated under various settings, and the results demonstrate that it is quite robust in the presence of local and global bandwidth fluctuations.
To my family
ACKNOWLEDGEMENTS

This thesis is the result of the inspiring and thoughtful guidance and supervision of my senior supervisor, Professor Jiangchuan Liu at School of Computing Science, Simon Fraser University. But for him this work could not have been completed. I also thank him for providing the necessary equipments for the research work and experiments.

I would also like to thank Natural Sciences and Engineering Research Council of Canada for financial support of my whole study period so that I could concentrate on my study and research.

Thanks go to the Network Modeling Research Group at School of Computing Science, Simon Fraser University, for the series of inspiring presentations and discussions. I am also grateful for the good time and friendship I enjoyed while I worked in the research group.

I would also like to express my gratitude to my wife, Lan Zhang, for all her encouragement and support during this time.
# TABLE OF CONTENTS

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Approval</td>
<td>ii</td>
</tr>
<tr>
<td>Abstract</td>
<td>iii</td>
</tr>
<tr>
<td>Dedication</td>
<td>iv</td>
</tr>
<tr>
<td>Acknowledgements</td>
<td>v</td>
</tr>
<tr>
<td>Table of Contents</td>
<td>vi</td>
</tr>
<tr>
<td>List of Figures</td>
<td>viii</td>
</tr>
<tr>
<td>List of Tables</td>
<td>ix</td>
</tr>
<tr>
<td>Glossary</td>
<td>x</td>
</tr>
<tr>
<td>Chapter One: Introduction</td>
<td>1</td>
</tr>
<tr>
<td>Chapter Two: Related Works</td>
<td>4</td>
</tr>
<tr>
<td>2.1 Overlay Multicast</td>
<td>4</td>
</tr>
<tr>
<td>2.2 Gossip Protocols</td>
<td>6</td>
</tr>
<tr>
<td>2.3 On-Demand Media Streaming</td>
<td>8</td>
</tr>
<tr>
<td>2.3.1 Sequential Access</td>
<td>9</td>
</tr>
<tr>
<td>2.3.2 Nonsequential Access</td>
<td>10</td>
</tr>
<tr>
<td>Chapter Three: TAG Overview</td>
<td>12</td>
</tr>
<tr>
<td>3.1 Data Asynchrony</td>
<td>12</td>
</tr>
<tr>
<td>3.2 Loss Multiplicity</td>
<td>13</td>
</tr>
<tr>
<td>3.3 TAG Overview</td>
<td>14</td>
</tr>
<tr>
<td>Chapter Four: Protocol Operations</td>
<td>17</td>
</tr>
<tr>
<td>4.1 Timing Condition and List</td>
<td>17</td>
</tr>
<tr>
<td>4.2 Construction of TAG Overlay</td>
<td>20</td>
</tr>
<tr>
<td>4.3 Maintenance of TAG Overlay</td>
<td>21</td>
</tr>
<tr>
<td>4.4 Data Scheduling and Dissemination</td>
<td>22</td>
</tr>
<tr>
<td>Chapter Five: AVL Tree Based Indexing</td>
<td>26</td>
</tr>
<tr>
<td>5.1 Joining Operations with Playback Offset</td>
<td>27</td>
</tr>
<tr>
<td>5.2 Failure Recovery</td>
<td>30</td>
</tr>
<tr>
<td>Chapter Six: Performance Evaluation</td>
<td>33</td>
</tr>
<tr>
<td>6.1 System Configurations</td>
<td>33</td>
</tr>
<tr>
<td>6.2 Control Overheads</td>
<td>34</td>
</tr>
<tr>
<td>6.2.1 Overheads of Join and Failure Recovery</td>
<td>34</td>
</tr>
<tr>
<td>6.2.2 Overhead Distribution</td>
<td>37</td>
</tr>
</tbody>
</table>
6.3 Streaming Quality .............................................................................................................38
  6.3.1 Streaming Quality with Bandwidth Fluctuations ..................................................39
  6.3.2 Streaming Quality with Node Failures .................................................................42
  6.4 Sensitivity to Parameter Settings ...............................................................................43

Chapter Seven: Conclusion and Future Work ....................................................................46

Appendix .............................................................................................................................47

Bibliography .........................................................................................................................49
<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Pure tree overlay</td>
<td>12</td>
</tr>
<tr>
<td>2</td>
<td>Parent/child relationship for on-demand streaming of asynchronous requests</td>
<td>13</td>
</tr>
<tr>
<td>3</td>
<td>Tree assisted gossip overlay</td>
<td>15</td>
</tr>
<tr>
<td>4</td>
<td>Buffer status at nodes ( j ) and ( i ) at time ( t )</td>
<td>18</td>
</tr>
<tr>
<td>5</td>
<td>An illustration of the index list structure (dashed line), which facilitates the construction of the delivery tree (solid line) and gossip partnerships</td>
<td>19</td>
</tr>
<tr>
<td>6</td>
<td>Parent search algorithm for node ( i )</td>
<td>21</td>
</tr>
<tr>
<td>7</td>
<td>Gossip partner search algorithm for node ( i )</td>
<td>21</td>
</tr>
<tr>
<td>8</td>
<td>Fields of message Data Offer</td>
<td>23</td>
</tr>
<tr>
<td>9</td>
<td>Pseudo code for the scheduling algorithm at a node</td>
<td>24</td>
</tr>
<tr>
<td>10</td>
<td>An illustration of right-rotation (case 1)</td>
<td>29</td>
</tr>
<tr>
<td>11</td>
<td>An illustration of left-right rotation (case 2)</td>
<td>30</td>
</tr>
<tr>
<td>12</td>
<td>An illustration of failure recovery (failure recovery case 1)</td>
<td>32</td>
</tr>
<tr>
<td>13</td>
<td>Maximum node cost for node join</td>
<td>35</td>
</tr>
<tr>
<td>14</td>
<td>Overall system cost for node join</td>
<td>35</td>
</tr>
<tr>
<td>15</td>
<td>Maximum node cost for failure recovery</td>
<td>36</td>
</tr>
<tr>
<td>16</td>
<td>Overall cost for failure recovery</td>
<td>37</td>
</tr>
<tr>
<td>17</td>
<td>Distribution of control overheads (TAG_S)</td>
<td>38</td>
</tr>
<tr>
<td>18</td>
<td>Distribution of control overheads (TAG_NA)</td>
<td>38</td>
</tr>
<tr>
<td>19</td>
<td>Quality with local bandwidth fluctuations (0.975 of base setting)</td>
<td>39</td>
</tr>
<tr>
<td>20</td>
<td>Quality with local bandwidth fluctuations (0.95 of base setting)</td>
<td>40</td>
</tr>
<tr>
<td>21</td>
<td>Quality with local bandwidth fluctuations (0.8 of base setting)</td>
<td>40</td>
</tr>
<tr>
<td>22</td>
<td>Quality with different overall network bandwidths</td>
<td>41</td>
</tr>
<tr>
<td>23</td>
<td>Segment missing rate vs. node failure rate</td>
<td>42</td>
</tr>
<tr>
<td>24</td>
<td>Streaming quality as a function of position reseeking rate</td>
<td>43</td>
</tr>
<tr>
<td>25</td>
<td>Segment loss rate as a function of the number of gossip partners</td>
<td>44</td>
</tr>
<tr>
<td>26</td>
<td>Segment loss rate as a function of the candidate set size</td>
<td>44</td>
</tr>
<tr>
<td>27</td>
<td>Segment loss rate as a function of buffer size under different overlay sizes (Normalized buffer size = buffer size / video length)</td>
<td>45</td>
</tr>
</tbody>
</table>
LIST OF TABLES

Table 1  Fields in each node for overlay construction. ..................................................20
Table 2  Fields at each node for the AVL indexing tree. ..................................................27
GLOSSARY

VOD: Video on Demand

IP: Internet Protocol

P2P: Peer to Peer

ALM: Application Layer Multicast

TAG: Tree Assisted Gossip Protocol

TAG-S: TAG with sequential accesses

TAG-N: TAG with non-sequential access

TAG-NA: TAG with non-sequential accesses and AVL indexing
CHAPTER ONE:
INTRODUCTION

With exponential expansion of the Internet’s resources and users, Video-on-Demand (VoD) has become one of the most attractive networked services. Nevertheless, for VoD with hundreds or thousands of clients, the traditional client/server model generally fails, due not only to the network bandwidth constraints but also to the unbearable stresses at the server. There have been many proposals on providing scalable on-demand streaming through multicast or proxy caching; yet the deployment of IP multicast and dedicated proxies remains limited nowadays. Recently, application-layer overlays have been considered as a readily deployable and thus promising alternative solution. In this scenario, an overlay network is built out of unicast tunnels across cooperative nodes with certain buffering capabilities [20-25]. Each overlay node acts as an application-layer proxy, and caches a certain amount of the video data it receives; the data are then relayed among the active nodes in the overlay to realize multicasting.

Initially as remedies to IP multicast, many overlay construction algorithms also advocate a tree structure for data delivering (eg. [4,5,7-12]). While this works well with dedicated infrastructure routers as in IP multicast, it often mismatches an application-level overlay with dynamic nodes. First, each node relies on a single path from source to retrieve data; bandwidth fluctuations at this specific path may result in highly unstable video quality. Second, only the paths in the tree are used to deliver data; the paths close to the root thus often become bottlenecks, while all other potential paths are untouched, leading to poor resource utilization. Last, the leave of an internal node, especially those close to the root, may cause buffer underflow at a large population of downstream nodes; such situations are not uncommon as each overlay...
node can join or leave at will. For on-demand streaming, the asynchrony among client requests further aggravates the above problems.

Opposite to a tree-based protocol, gossip protocols enable random data dissemination with no support from a regular overlay structure [15-17]. In a typical gossip process, a node randomly selects a subset of target nodes to deliver recently available data segments, and meanwhile, receives segments pushed from these nodes. It is known that gossip algorithms achieve highly robust data distribution. Nevertheless, it is not straightforward to apply gossiping in on-demand streaming, for it often fails to achieve a timely delivery. Furthermore, the push-based gossip could cause excessive data duplications, which is particularly severe for high-bandwidth videos.

In this master thesis, I present TAG, a Tree-Assisted Gossip protocol for on-demand media streaming. TAG constructs and maintains two overlays, namely, a tree overlay and a gossip overlay, which collectively deliver video streams to clients. I also design intelligent and efficient overlay construction and data scheduling algorithms for this hybrid system. They seamlessly integrate the two distinct approaches and realize their best features: low delay with a regular tree topology, and robust delivery with smart switching among multiple paths, thus making effective use of the available bandwidth in the network. A timing listing is used to accommodate the asynchronous requests in an on-demand streaming system. I also substitute the push-based delivery by a pull process, which greatly eliminates the massive redundancy due to random disseminations. Finally, I enhance the TAG system by introducing AVL tree based indexing, which facilitates non-sequential accesses.

Performance of TAG is evaluated under various network configurations. The results demonstrate that it is highly robust when facing local and global bandwidth fluctuations. As compared a pure tree-based overlay VoD system, it achieves much lower and stable segment
missing rates (<10%) under dynamic network environments. Meanwhile, its control overhead is kept at low levels, suggesting that TAG scales well to large overlay networks.

The basic idea of using a hybrid overlay structure for video distribution and some preliminary experimental results are shown in [35]. A fully distributed overlay construction/maintenance algorithm, protocol operation details and extensive experimental results are presented in [36]. These two piece of work together lead to this master thesis.

The rest of the thesis is organized as follows. The related works are summarized in Chapter 2. The motivations for designing TAG together with an overview of TAG are presented in Chapter 3. Detailed protocol operations are presented in Chapter 4. In Chapter 5, I further enhance TAG by introducing AVL tree based indexing. The performance of TAG is evaluated in Chapter 6. Finally, Chapter 7 concludes the thesis and offers some future research directions.
CHAPTER TWO:
RELATED WORKS

Three categories of work are related to my research: overlay multicast, gossip protocols, and on-demand streaming. I will show how my research differs from these previous work, the underlying problems and how they motivate this research work.

2.1 Overlay Multicast

A substantial portion of today’s Internet bandwidth is used to deliver multimedia content. In a traditional client-server system, the server often becomes the bottleneck due to the high bandwidth and long duration of multimedia streams. Scalability is thus the key issue. IP multicast would appear to solve the scalability issue by alleviating bandwidth demand at the source. There have been significant studies on video over IP multicast as well in the past decade; see a survey in [14]. Unfortunately, IP multicast has not been widely deployed in the current Internet due to its requirement to upgrade the Internet infrastructure. Other issues such as per-group state maintenance, security and address allocation further aggravate the problem.

Researchers then shift their focus to application-layer multicast (ALM) [2], or overlay multicast. Unlike IP multicast, overlay multicast does not assume multicast support at the IP layer. On the other hand, an overlay multicast tree or mesh consisting of only end-hosts is constructed, which is used to deliver media streaming files. Those proposed overlay multicast systems can be broadly classified into two categories [9]: proxy-assisted and peer-to-peer based. In the former, a set of servers or application-level proxies are strategically placed, and a high-quality overlay can then be constructed with the assistance of these anchor nodes [28,29,31]. The latter does not rely on dedicated nodes, but build an overlay out of self-organized autonomous
nodes. TAG belongs to the second category, which enables readily deployable solutions. In this section, I give a brief overview of the existing overlay streaming protocols for both real-time streaming and on-demand streaming.

A typical example of proxy-assisted overlay multicast systems is Overcast [31]. In Overcast, a collection of nodes are placed at strategic locations in an existing network fabric. These nodes are supposed to implement a network abstraction on top of the network provided by the underlying substrate network. For all the nodes participating in the system, Overcast builds a single source multicast tree. The goal of the tree construction algorithm is to maximize the bandwidth to the root for all nodes. At a high level, the algorithm proceeds by placing a new node as far away from the root as possible without sacrificing bandwidth to the root. This leads to deep distribution trees in which the nodes nonetheless observe no worse bandwidth than obtaining the content directly from the root. By choosing a parent that is nearby in the network, the distribution tree will form along the lines of the substrate network topology.

On the other hand, End-system multicast [2] is one of the early proposed peer-to-peer based overlay multicast systems. The overlay construction protocol is called Narada. In Narada, the participating nodes first self-organize into a mesh structure. Based on the mesh structure, a shortest path multicast tree is then computed for the system. The cost for any two nodes is computed using end-to-end measurement. Another example of peer-to-peer based overlay multicast system is Yoid [32]. Yoid includes a full framework for overlay multicast implementation addressing applications such as netnews, streaming broadcast, and bulk email distribution. The core of Yoid is its topology management protocol name YTMP (Yoid Tree Management Protocol). According to YTMP, participating nodes auto-configure into two separate topologies: a mesh and a tree. The tree is optimized for delivery efficiency while the mesh is constructed for system robustness.
As we can see, all the above-mentioned systems adopt a tree structure. Many other overlay streaming systems also employ a tree structure, stemmed from IP multicast. Constructing and maintaining an efficient distribution tree among the overlay node is a key issue to these systems. In CoopNet [5], the video source, as the root of the tree, collects the information of all the nodes for tree construction and maintenance. Such a centralized algorithm can be very efficient, but relies on a powerful and dedicated root node. To the contrary, distributed algorithms, such as SpreadIt [8], NICE [4], and ZIGZAG [9], perform the constructing and routing functions across a series of nodes. For a large-scale network, these algorithms adopt hierarchical clustering to achieve minimized transmission delay (in terms of fanout degree). Still, an internal node in the tree has a higher load and its leave or crash often causes buffer underflow in a large population of descendants. Several tree repairing algorithms have been devised to accommodate node dynamics [4, 7, 9]; yet the tree structure may still experience frequent breaks in the highly dynamic Internet environment.

There are also many other solutions addressing the vulnerability of the tree structure. Examples include building mesh-based tree (Narada [2] and its extensions [14], Yoid [32], and Bullet [20]), maintaining multiple distribution trees (SplitStream [19]), and leveraging layered coding (PALS [29]) or multiple description coding CoopNet [3]). I note that the scalability and efficiency of such complex structures and coding schemes remains debatable. Thus, TAG dose not relies on an advanced coding scheme, though it might be helpful in the system as well.

2.2 Gossip Protocols

Gossip (or epidemics) algorithms have recently gained popularity as a robust and scalable way of propagating information in distributed systems [34, 13, 15]. In a gossip algorithm, a process that wishes to disseminate a new piece of information to the system does not send it to a server, or a cluster of servers, in charge of forwarding it, but rather to a set of other peer processes, chosen at random. In turn, each of these processes does the same, and also forwards
the information to randomly selected processes, and so forth, until all processes in the system received the piece of information. As we can see, gossip dissemination algorithms are simple and easy to deploy. In addition to their attractive scalability promises, gossip algorithms exhibit a very stable behaviour even in the presence of high rate of link and/or process failures. There is no single point of failure and the reliability degrades gracefully with the number of failures. Due to such properties, gossip algorithms prove themselves suitable for the design of scalable peer-to-peer application layer multicast systems.

In the original gossip broadcast algorithms [1], it is assumed that every process knows every other process. That is, every process has a list of all other processes in the system, and therefore is able to communicate with every such process. This assumption, however, is unrealistic in a large system. The requirement of scalability imposes to use a decentralized protocol providing each process only with a partial view of the system (that is, a subset of other processes' identifies), on which a gossip dissemination algorithm can rely. Such an approach was proposed in [33]. In the proposed system, a new process to join the system by sending a join request to an arbitrary process, called a contact or a bootstrapping process. The newcomer will then initialise its partial view with the contact process. The latter process then propagates the request to all the processes present in its own partial view. Each of these processes either keep the new process in its own partial view, or forwards the request to some processes randomly chosen from its local view. This simple mechanism ensure that the system configures itself towards partial views of size \((c+1)\log(n)\) on average, where \(c\) is a design parameter, selected to ensure a high reliability for a target transmission failure probability. It is proved that the data loss rate is exponentially low in such a gossip system, given enough time.

Gossip algorithms have also inspired some research works in media streaming. In DONet [17], a data-centric streaming overlay is designed, where a node always forward data to others that are expecting the data, with no prescribed roles like father/child, internal/external, or
upstreaming/downstreaming, etc. In other words, it is the availability of data that guides the flow directions, while not a specific overlay structure that restrict the flow directions. Each node maintains a set of partners, namely, a partial view of the system. Every node periodically exchange data availability information with a set of partners, and receive unavailability data from one or more partners, or supplies availability data to partners. DONet have been extensively evaluated over Planetlab, and a public Internet-based DONet implementation called coolStreaming was released, which was popularly used to lively broadcast sports program.

Nevertheless, the use of gossip for on-demand media streaming is not straightforward because its random push may cause significant redundancy, which is particularly severe for high-bandwidth streaming applications. In TAG, I devised smart partners selections algorithm and a low-overhead scheduling algorithms to intelligently pull data from multiple partners, which greatly reduces redundancy. Such design is partly motivated by the data-driven concept in DONet[17], which targets live video streaming. The asynchronous nature in on-demand streaming, however, calls for new solutions as addressed in TAG.

2.3 On-Demand Media Streaming

On-demand media streaming is a problem that poses special challenges different from those by live streaming. The fundamental challenge is the unpredictability of user requests in the following aspects: 1) Asynchrony: users may request the same media object at different time. 2) Nonsequentiality: users' stream access pattern is VCR-type, instead of sequential (from the beginning to the end). 3) Burstiness: the request rate for a certain media object is highly unstable over time. On-demand media streaming techniques based on IP multicast, such as Batching, Patching, and Periodic Broadcast were proposed and extensively investigated in the 1990s. The basic approach of those solutions is to repeat the same media content on different multicast channels over time. Clients are either forced to synchronize at the price of service delay, or required to participate in several multicast sessions simultaneously. Furthermore, these techniques
are yet to be applicable because they assume the existence of IP multicast as the underlying network infrastructure.

The above-mentioned challenges, however, can be much better solved in the context of overlay network. In an overlay network system, each node has a buffering capability. Each node can buffer its played back media content, and relay it to other nodes. The temporal correlations of asynchronous requests of different nodes can be explored to decide a media content source for a newly joined node. In this way, requests at different times are satisfied by the same stream, thus achieving efficient media delivery.

Previous work address on-demand media streaming in the way of overlay network can be divided into two categories: those assuming sequential user access pattern (from beginning to end) and those assuming nonsequential user access pattern.

2.3.1 Sequential Access

Chaining [30] is probably the first overlay on-demand streaming system although it was originally proposed as a new “batching” mechanism to solve the problem of unfair delays in conventional batching. It is assumed that every node start viewing a video from the first data segment, namely sequential access only. Each client caches data for later multicasting to late-comers, resulting in a transmission chain starting from the content server. Once a new node cannot use the previously joined node as its source, a new chain is started from the content server. The problem for chaining is that it does not consider recovery from node failures, which is common in overlay multicast systems.

P2Vod [22] is an extension to chaining to solve the failure recovery problem. It also assume that each node start view the video from the first data segment. Clients in P2Vod can forward the video stream to a new client as long as they have enough out-bound bandwidth and still hold the first data segment of the video file in the buffer. This caching scheme allows a group
of clients, arriving to the system at different time, to store the same video content in the prefix of their buffers. Such a group forms a generation. When a member of a generation leaves the system (due to failure, early departure, and etc.), any remaining member of that generation can provide the video stream to the abandoned children of the leaving member provided that out-bound bandwidth is sufficient. In this way, failures are handled locally without involving the content server.

2.3.2 Nonsequential Access

The problem of nonsequential access addressed in later works [20,25]. In oStream [20], clients also buffer the most recent video content they received for later comers. A client that can relay stream to another client is called a buffer-constrained predecessor of the latter. Such relationships among the participating clients form a directed graph, which is called a media distribution graph. Out of the directed graph, a minimum spanning tree in term of transmission cost is computed, which is called the media distribution tree. A directory server is used to compute and maintain such a tree. When a client wants to join the system, it contacts the directory server with the position of the stream it wishes to start receiving. The directory server updates the tree, and responds with a position (the parent of the new client in the media distribution tree) where the new client can join the tree. Such a process also happens when a client in the system wants to move to a different position of the stream.

In another work, P2PStream [25], a similar tree structure is advocated to distribute media data. An index server is used not only to compute the tree, but also to tell each client which segment to cache. Nodes with close IP prefixes are grouped into a cluster. Data segment allocation is computed in the index server to ensure that enough copies of the any data segment in a cluster to serve all requests from within the cluster. In this way, the distribution tree is roughly adjusted to reflect the underlying network topology.
In summary, pioneering works in overlay on-demand media streaming area most advocate a general tree structure [30,20,22,25] (note that a chain can be viewed as a special type of tree). A directory server [20] or an index server [25] is employed to compute the parent/children relationship in time of node joining, reseeking, departure and failure. These schemes, though scales well, may suffer from the vulnerability of the tree structure as I will discuss in the next chapter.
CHAPTER THREE: 
TAG OVERVIEW

In this section, I discuss the inherent challenges for a tree structure to support on-demand streaming, and the weaknesses motivate my design of a tree-assisted gossip protocol. Then I present the overview of the proposed solution.

3.1 Data Asynchrony

In a dynamic overlay, a node can join or leave the system at will, or fail without noticing others. There have been many studies on tree-based joining and repairing for live media streaming [2,5,10,11,12]. Yet these operations for on-demand streaming are generally more complex due to data asynchrony.

Figure 1  Pure tree overlay
As shown in Fig. 1, there are 4 client nodes in the tree rooted at the content server. Node 1 retrieves data segments from the server, and forwards to node 2, which further relays to nodes 3 and 4. For live media streaming, since the nodes are isochronous, i.e., expecting the same or almost the same data segment at a time instance, any available node can substitute for a failed node as long as its outbound bandwidth is enough.

This is, however, not the case for on-demand streaming with asynchronous requests. In this scenario, different nodes may expect quite different data segments at any given time instance, in Fig. 2. For illustration, consider a failure of node 2. Obviously, node 4 cannot serve as substitute for node 2; otherwise, since the data segment requested by node 3 is not in node 4's buffer, node 3 would suffer from severe segment losses. Similar challenges exist for a new node to locate a proper position and join the tree. In general, there are certain conditions for two nodes to form a parent/children relationship besides the outbound bandwidth constraints. In this thesis, I derive such conditions and, accordingly, devise a novel timing list to deal with the asynchrony.

3.2 Loss Multiplicity

Besides node failures that may result in data losses, a greater challenge is the loss multiplicity problem in any tree structure. As shown in Fig. 1, suppose the link between node 1 and node 2 encounters congestion such that 20% data segments are lost. These data are absent not only in node 2, but also at any node in the sub-tree rooted at node 2. Even worse, low-rate losses
can accumulate and cause severe problem. Consider a leaf node with a path of 5 hops to the root, and each link in the path has loss rate of 5%. The loss rate at the leaf node could be as high as 23%. Note that a tree of 5 levels is not uncommon in an overlay with hundreds of participating nodes, especially considering their fanout degrees are restricted by outbound bandwidths. In this thesis, I introduce gossip-style random data dissemination to complement the tree structure, which, as suggested by MY experimental results, effectively mitigates the loss multiplicity problem.

3.3 TAG Overview:

A TAG system consists of a content server, which stores a repository of media files, and a set of autonomous nodes, which can join or leave the system at will. It is assumed that the address of the content server is publicly available through an advertising protocol, such as SAP; thus, a node can always retrieve the media stream from the server; yet a scalable solution is expected given the limited server resources. To this end, each node in the TAG system contributes a certain buffer space, which caches the recently received data at the node, and a node thus can retrieve data not only from the server, but also from other active nodes with expected data in their buffers.
TAG adopts a tree assisted gossip protocol to organize the nodes, locate partners with cached data, and schedule the data fetching. Fig. 3 shows such an overlay structure, where a tree organizes all the nodes, and these nodes also form gossip partners to exchange data with each other. I divide buffer at every node $i$ into two parts, namely, a forward buffer of size $b_i^+$, and a backward buffer of size $b_i^-$. A node stores data segments pre-fetched from its parent or partners in its forward buffer, and caches played out segments in its backward buffer, both of which can be used to supply its children or partners upon requests.

I show an example the gossip partnership for node 7 in Fig. 3, and stress three salient features of this hybrid design: (1) Adaptive, as a receiver can intelligently switch among multiple suppliers (parent and gossip partners), and the fanout constraint for tree nodes can be relaxed; (2) Efficient, as the availability at different paths/nodes can be explored; and (3) Robust, as the bandwidth fluctuation or node failure at a particular path has less impact.

The experimental results suggest that most of these features are enabled by the gossip algorithm; yet the tree structure is indispensable to meet the real-time constraints. It is, however, not straightforward to employ a tree structure or a gossip algorithm for on-demand streaming, not to mention integrating them. There are several challenges to be addressed, in particular:
1. How is a newly joined node inserted to the tree and assigned to gossip partners? Note that the nodes are with asynchronous join times and limited buffer spaces. Similar issues have to be addressed when node fail or leave the system.

2. For each expected data segment, where and when to fetch it? There are multiple suppliers with non-uniform bandwidth and data availability, and the playback deadline has to be met.

I detail the TAG operations in the next two chapters, which offer efficient solutions to the above issues in this hybrid system.
CHAPTER FOUR:
PROTOCOL OPERATIONS

For ease of exposition, I focus on the distribution of a single video stream only, and the solution can be easily extended to the multi-stream case. I assume that the stream is divided into equal-sized data segments, each with a unit playback time. The buffer size is measured as the total number of segments it can accommodate. I also assume that each segment has a sequence number, and video playback at a node always starts from the first segment. Extensions to support non-sequential accesses will be addressed in the next section.

4.1 Timing Condition and List

Due to data asynchrony in on-demand streaming, a parent-child relationship or gossip partnership cannot be directly set up between any two nodes, even without the outbound bandwidth constraint. I now derive the conditions for two nodes to form a parent-child or gossip relation, which will serve as a foundation for overlay construction and maintenance.

Fig. 4 depicts a snapshot of the buffers at nodes $i$ and $j$, respectively, at time $t$. Suppose $t - t_i$ is the currently played segment for node $i$, which joins the system at time $t_i$; the maximum sequence number of the data segments in its buffer is thus $t - t_i + b_i^+$, and the minimum one is $t - t_i - b_i^-$; so is node $j$. 
From Fig. 4, the necessary condition for \( j \) being the parent of node \( i \) should be

\[
\begin{align*}
\{ & t - t_i < t - t_j \\
& t - t_i > t - t_j - b_j^- \\
\}
\end{align*}
\]

which is equivalent to

\[
t_i - b_j^- < t_j < t_i.
\]

That is, the join time of node \( j \) should be earlier than that of node \( i \), and their difference should be less than the draining time of the backward buffer of node \( j \).

Opposite to the parent-child relation, data delivery is bidirectional with a gossip partnership. From Fig. 4, for node \( i \) to forward data to node \( j \), the following condition should be met:

\[
i \rightarrow j : \begin{cases} 
  t - t_i - b_i^- < t - t_j + b_j^+ \\
  t - t_i + b_i^+ > t - t_j
\end{cases}
\]

which basically states that at least part of the buffer (backward buffer plus forward buffer) of node \( i \) should overlap with the forward buffer of node \( j \). Similarly, the condition for node \( j \) to forward data to \( i \) is

\[
j \rightarrow i : \begin{cases} 
  t - t_j - b_j^- < t - t_i + b_i^+ \\
  t - t_j + b_j^+ > t - t_i
\end{cases}
\]
Combining Eq. (3) and (4), we have

\[
j \leftrightarrow i : \begin{cases} 
t_i < t_j + b_j + b_i^- \\
t_i > t_j - b_j^+ \\
t_i > t_j - b_j^+ - b_i^+ \\
t_i < t_j + b_i^+ 
\end{cases}
\]

which follows that

\[
t_i - b_i^+ < t_j < t_i + b_j^+ \tag{6}
\]

To efficiently examine the above timing conditions in TAG, I link all the active nodes into a timing list, sorted according to their joining times. In this list, node \( j \) is the predecessor of node \( i \) if node \( j \) joined system immediately before node \( i \), and, accordingly, \( i \) is referred to as \( j \)'s successor. A bidirectional link is then added between the predecessor and the successor. Fig. 5 depicts such a timing list structure for the nodes.

Figure 5 An illustration of the index list structure (dashed line), which facilitates the construction of the delivery tree (solid line) and gossip partnerships.

Note that, to construct and maintain the timing list, the content server needs to keep track of the latest joined node only. In the bootstrapping stage, the content server itself is such a node.
Each newly joined node first contacts the content server, which then redirects the node to the existing latest joined node, and a predecessor and successor relation can then be formed, as will be detailed next.

4.2 Construction of TAG Overlay

<table>
<thead>
<tr>
<th>Field</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Parent</td>
<td>Parent of a node in the delivery tree</td>
</tr>
<tr>
<td>ChildrenList</td>
<td>List of children in the delivery tree</td>
</tr>
<tr>
<td>Predecessor</td>
<td>Predecessor in the timing list</td>
</tr>
<tr>
<td>Pre-Predecessor</td>
<td>Predecessor of predecessor in the timing list</td>
</tr>
<tr>
<td>Successor</td>
<td>Successor in the timing list</td>
</tr>
<tr>
<td>Suc-Successor</td>
<td>Successor of successor in the timing list</td>
</tr>
<tr>
<td>PartnerList</td>
<td>List of gossip partners</td>
</tr>
</tbody>
</table>

A TAG system is constructed with nodes joining the overlay asynchronously. To facilitate the join process, each node maintains a set of status information, as shown in Tab.1, and a new node $i$ performs the following join operations:

1) Node $i$ sends message $\text{Join } < i >$ to the content server;

2) The content server records the join time of node $i$, and redirects it to nodes $L$, which is the latest joined node so far, i.e., the one immediate before node $i$;

3) Node $L$ sets node $i$ as its successor, and node $i$ sets node $L$ as its predecessor. The predecessor's predecessor and successor's successor relation is also set between node $i$ and the predecessor of node $L$;

4) Node $i$ invokes a parent search and a partner search algorithm to locate its parent and gossip partners, and then sets the corresponding relations. Both algorithms rely on the timing list to check the timing conditions, as shown in Figs. 6 and 7, respectively.
Figure 6  **Parent search algorithm for node $i$.**

1) Traverse the timing list, staring from the predecessor of node $i$;
2) Test condition (2) for each encountered node, until the second node violating the condition is found, or $K$ nodes have been visited;
3) For all the nodes that satisfy the condition, select the one with the maximum bandwidth to node $i$ as its parent.

Figure 7  **Gossip partner search algorithm for node $i$.**

1) Traverse the timing list, staring from the predecessor of node $i$;
2) Test condition (6) for each encountered node, until the second node violating the condition is found, or $K$ nodes are visited;
3) Among all the nodes that satisfy the condition, randomly select $k$ nodes as gossip partners.

For both search algorithms, the number of nodes involved is bounded by $O(K)$, and I will show through experiments that a relatively small $K$ (say less than 12) is enough in most cases. The number of gossip partners, $k$, is also an important factor, whose impact will be investigated in my experiments as well. Note that I also make each node linked to its predecessor’s predecessor and successor’s successor in the timing list, which helps with recovering from node failures. The predecessor for the list head (the content server) and the successor for the list tail (the latest joined node) are two special cases, in which the predecessor and the successor are set as the head itself and tail itself, respectively.

### 4.3 Maintenance of TAG Overlay

I use a heartbeat protocol to maintain the parent-child and partner relationships. Each node periodically sends an *Echo* message to its related nodes, namely, parent, children, and partners, as well as successor and predecessor in the index list. The leave of a node, due either to an intended departure or abrupt failure, can thus be easily detected. The following failure recover operations will then be executed at the affected nodes:
• Predecessor/Successor:

The predecessor and successor of the failed node contact each other and form a direct predecessor-successor relationship; this is viable because each node records its predecessor and successor as well;

• Parent/Gossip Partners:

Removes the failed node from its children list or gossip partner list;

• Children:

Each child invokes the parent search algorithm to locate a new parent. The starting node will be the predecessor of the child, or the pre-predecessor if its predecessor is just the failed node.

In a dynamic network, the above operations can as well be periodically invoked by a node to refine its parent-child relationship or gossip partnership.

4.4 Data Scheduling and Dissemination

In TAG, a data segment could be available at multiple suppliers, and a commonly used push mechanism for data delivering may cause excessive redundancy. I thus resort to a pull mechanism, in which a node with data available first sends a Data Offer message to a target node, namely, a child or a gossip partner. The target node will then send back a Data Request if it decides to fetch a data segment.
The fields included in a Data Offer are shown in Fig. 8. Note that their sizes are relatively small, as the availability for each segment is indicated by one bit only. To further reduce the overhead, the data offer and request can both be piggyback by the Echo messages, and the requests for a set of segments from the same supplier can be batched together as well.

Since a node will collect a set of Data Offers from its parent and gossip partners during an exchange period, a key issue is thus to decide which unavailable data segments should be fetched from which node. In other words, a scheduling algorithm is necessary for data fetching in TAG. There are two constraints for the scheduling algorithm: 1) each data segment should be fetched before its playback deadline; 2) the number of data segment fetched from a partner should be within its delivery capability, i.e., the outbound bandwidth. The objective is thus to fetch the maximum number of data segments ahead of their deadlines, subject to the bandwidth constraints. This is variation of the Parallel Machine Scheduling problem, which is know NP-hard[27]. It is thus not easy to find an optimal solution, particularly considering that the algorithm must quickly adapt to the highly dynamic network conditions. Therefore, I resort a simple heuristic of fast response time.

The heuristic algorithm starts from examining the segment with the earliest deadline, and then the second earliest, and so on. In case multiple suppliers are available for a segment, the algorithm selects the supplier that offers the least number of unavailable data segments. For example, suppose the segment has two suppliers, one offers ten unavailable segments, while the other does not have any other unavailable segment but the expected segment; the latter is then selected, because the former is more flexible in supplying data and can potentially be use to fetch
other unavailable segments if needed. In addition, fewer suppliers also imply that the segment could be relatively new, and thus should be gossiped as soon as possible to minimize delay.

Figure 9  Pseudo code for the scheduling algorithm at a node.

Input:

\[ \text{expect} \_\text{segm}(n) : \text{set of expected data segments at the node; } \]
\[ \text{data} \_\text{set}(k) : \text{set of data segments offered by partner } k; \]
\[ \text{partner} \_\text{set} : \text{set of partners of the node; } \]
\[ \text{num} \_\text{partners} : \text{number of partners of the node; } \]
\[ \text{avail} \_\text{band}(k) : \text{available bandwidth from partner } k \text{ to the node; } \]
\[ \text{deadline}(i) : \text{available time before playback deadline of segment } i \]

Scheduling:

\[ \text{for } k \in \text{partner} \_\text{set} \text{ do //initialize the number of expected segment available at partner } k \]
\[ \text{num} \_\text{exp} \_\text{seg}(k) = | \text{data} \_\text{set}(k) \cap \text{exp} \_\text{seg} \_\text{set} |; \]
\[ \text{for } i \in \text{exp} \_\text{set} \text{ do //first pass, single supplier available among gossip partners } \]
\[ \text{supplier} \_\text{set}(i) \leftarrow \emptyset ; \text{initialize the set of suppliers for segment } i \]
\[ \text{for } k \in \text{partner} \_\text{set} \text{ do } \]
\[ \text{if } i \in \text{data} \_\text{set} \text{ then } \]
\[ \text{num} \_\text{suppliers}(i)++; \text{supplier} \_\text{set}(i) \leftarrow \text{supplier} \_\text{set}(i) \cup \{k\}; \]
\[ \text{end if } i; \]
\[ \text{end for } k; \]
\[ \text{if num} \_\text{suppliers}(i) = 1 \text{ then } \]
\[ k \leftarrow \text{member}(\text{supplier} \_\text{set}(i)); \]
\[ \text{if avail} \_\text{band}(k) \geq \text{seg} \_\text{size} / \text{deadline}(i) \text{ then } \]
\[ \text{supplier}(i) \leftarrow k; \text{avail} \_\text{band}(k) = \text{seg} \_\text{size} / \text{deadline}(i); \]
\[ \text{for } j \in \text{partner} \_\text{set} \text{ do } \]
\[ \text{if } i \in \text{data} \_\text{set}(j) \text{ then num} \_\text{exp} \_\text{seg}(j)--; \]
\[ \text{end if } ; \]
\[ \text{end if } ; \]
\[ \text{end for } i; \]
\[ \text{for } i \in \text{exp} \_\text{set} \text{ do //second pass, multi-supplier available among gossip partners } \]
\[ \text{if num} \_\text{suppliers}(i) > 1 \text{ then } \]
\[ k \leftarrow \text{arg} \min_{r \in \text{num} \_\text{exp} \_\text{seg}(r)} \{ i \in \text{data} \_\text{set}(r), \text{avail} \_\text{band}(r) \geq \text{seg} \_\text{size} / \text{deadline}(i) \}; \]
\[ \text{if avail} \_\text{band}(k) \geq \text{seg} \_\text{size} / \text{deadline}(i) \text{ then } \]
\[ \text{supplier}(i) \leftarrow k; \text{avail}_\text{band}(k) = \frac{\text{seg}_\text{size}}{\text{deadline}(i)}; \]

\text{for} \ j \in \text{partner\_set} \ \text{do}

\begin{center}
\text{if} \ i \in \text{data\_set}(j) \ \text{then} \ \text{num\_exp\_seg}(j) \leftarrow; \\
\end{center}

\text{end if;}

\text{end for} \ i; \]

\text{for} \ i \in \text{exp\_set} \ \text{do} \ //\text{Retrieve remaining segments from the parent in the delivery tree}

\text{if} \ \text{supplier}(i) = \text{nil} \ \text{then} \ \text{supplier}(i) = \text{TreeParent} ;

\text{end if;}

\text{end for} \ i;

\textbf{Output:}

\[ \text{supplier}(i) : \text{supplier for expected data segment} \ i \]

Since the number of expected segments is less than the total size of the buffers \((b^+ + b^-)\), the complexity of the scheduling algorithm is bounded by \(O[k(b^+ + b^-)]\), where \(k\) is the number of partners. My implementation in a Pentium III, 1GHz computer suggest that this algorithm reasonably fast (<20ms) under various combinations of \(k\), \(b^+\) and \(b^-\), and is thus suitable for real-time scheduling.
CHAPTER FIVE:
AVL TREE BASED INDEXING

In the basic TAG system, I assume sequential accesses that always starts playback from
the initial segment of a stream. For implementing VCR-like operations, such as forward,
backward, and random seek, however, non-sequential access from arbitrary starting position
become necessary. In this section, I present effective enhancement to the basic TAG system to
support non-sequential accesses.

Suppose a new node \( i \) joins the overlay at time \( t_i \) with a playback offset \( o_i \); at time \( t \),
the node expects to play out segment \( (t - t_i + o_i) \). Since the conditions to form parent/children
and gossip partners still hold if we replace \( t_i \) by \( (t_i - o_i) \), a naive solution is to search the timing
list until candidates satisfying the revised condition are found. Unfortunately, in the worse case,
this may result in a traverse across all the nodes in the sorted timing list, yielding unacceptably
high cost. Earlier studies on this issue \([20,24,25]\) have suggested that a centralized server
maintains a global tree structure for both timing and data delivering. While this solution is easy to
implement, it is often not scalable, and the delivery tree itself is not an ideal indexing structure
given that its height is unbounded.

To this end, I introduce an AVL index tree to assist the search in the timing list, which is
implemented in a distributed way as described later. An AVL tree is a binary search tree with the
following balance property: for any node in the tree, the height of the left and the right sub-tree
can differ by at most 1. It is known that, for an AVL tree with \( N \) nodes, its height \( H \) satisfies:

\[
H < 1.44 \log(N + 2) - 1.328.
\]
Hence, the cost of locating an proper insertion point is $O(\log N)$, implying that the joining and failure recovery costs would be greatly reduced for non-sequential accesses. It is worth noting that the AVL indexing tree is a complement to the timing list, and is independent of the data delivery tree; hence, the list construction and maintenance, as well as the data scheduling and dissemination algorithms, remain unchanged.

I now detail the operations of the AVL index tree for non-sequential accesses. Tab. 2 lists the related information kept at each node.

### Table 2  Fields at each node for the AVL indexing tree.

<table>
<thead>
<tr>
<th>Field</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>avlParent</td>
<td>Parent in the AVL tree</td>
</tr>
<tr>
<td>avlLeftChild</td>
<td>Left child in the AVL tree</td>
</tr>
<tr>
<td>avlRightChild</td>
<td>Right child in the AVL tree</td>
</tr>
<tr>
<td>avlLeftHeight</td>
<td>Height of the left subtree in the AVL tree</td>
</tr>
<tr>
<td>avlRightHeight</td>
<td>Height of the right subtree in the AVL tree</td>
</tr>
<tr>
<td>Virtual join time</td>
<td>Value $(t_i - o_i)$ for node i</td>
</tr>
<tr>
<td>avlGrandParent</td>
<td>Parent’s parent in the AVL tree</td>
</tr>
</tbody>
</table>

### 5.1 Joining Operations with Playback Offset

The AVL index tree is constructed with the growth of the timing list. For a newly joined node with playback offset $o_i$, the following operations are performed:

1) Node $i$ sends message $\text{Join} \ < \ i, o_i >$ to the content server;

2) The content server records the virtual join time $(t_i - o_i)$ of node $i$, and redirects it to nodes $R$, which is the root of the AVL index tree;

3) If the virtual join time of node $i$ is less than that of node $R$, $R$ redirects $i$ to its left child in the AVL index tree, or otherwise to its right child. The above operations are repeated until the corresponding child is empty, and node $i$ is then inserted to this position as a leaf node;
4) If \( i \) is inserted as left child of its \( \text{avlParent} \), it will be the predecessor of \( \text{avlParent} \) in the timing list, or else its successor. Similar operations for a new node to join the timing list and data delivery tree are performed (steps 3 and 4 in Section 4.B) with this insert position;

5) Node \( i \) sets its height to 0, and sends a \textit{HeightReport} message to its \( \text{avlParent} \). Upon receiving the report, the parent resets its \( \text{avlLeftHeight} \) or \( \text{avlRightHeight} \), depending on which branch the report comes from, and then calculate its own height as

\[
\max(\text{avlLeftHeight, avlRightHeight}) + 1.
\]

If the height is changed, the node reports as well to its own \( \text{avlParent} \) until the root of the AVL tree is reached;

6) If unbalance is detected after update the height, a subtree rotation should be performed, and the root of the AVL, if updated, is then reported to the content server.

A node detects a violation of the balance if the difference between its \( \text{avlLeftHeight} \) and \( \text{avlRightHeight} \) is greater than 1, which would occur in the following four cases:

1) An insertion into the left subtree of its left child;

2) An insertion into the right subtree of its left child;

3) An insertion into the left subtree of its right child;

4) An insertion into the right subtree of its right child;
Fig. 10 shows that an unbalance occurs when a new node is inserted into the sub-tree $A$, i.e., case 1. To re-balance the AVL tree, node $x$ initiates a right-rotation as follows:

1) Node $x$ sends message $\text{RotateRequest}<$RIGHT, $avlParent_x,$> to node $y$;

2) Node $y$ sets node $x$ as its own $avlRightChild$ in the AVL tree, and the current $avlParent$ of node $x$ as its $avlParent$. It then sends message $\text{RotateReply}<$avlRightChild,$y,$> to node $x$;

3) Node $x$ sets node $y$ as its own $avlParent$, and the $avlRightChild$ of $y$ as its $avlLeftChild$;

4) Nodes $x$ and $y$ respectively notify the roots of sub-trees $A$, $B$ and $C$ to update their $avlParent$ fields.

Similar operations can be performed for a left-Right double rotation, as shown in Fig. 11. It solves the second case, and the solutions to the remaining two cases are symmetric to the previous two. In all the four cases, it is known that one rotation is enough to re-balance the AVL tree. Since the height of the AVL tree is $O(\log N)$, the cost for a joining operation is thus bounded by $O(\log N)$. 

![Figure 10 An illustration of right-rotation (case 1).](image-url)
5.2 Failure Recovery

I assume that each node also maintains its relation with its `avlParent`, `avlLeftChild` and `avlRightChild` through the heartbeat protocol, and its failure can thus be detected by these nodes. The following recovery operations will then be performed (for ease of exposition, I denote the failed node as node `F`, and its predecessor and successor in the timing list as `P` and `S`, respectively):

1) `F`'s `avlParent` removes `F` from its children list; `F`'s `avlLeftChild` and `avlRightChild` respectively mark their links to `F` as broken;

2) `P` and `S` respectively send a probe, which is forwarded toward the root in the AVL tree, until the root or a link marked as broken is encountered;

3) Assume `W_P` is the last node traversed by `P`'s probe and `W_S` is that by `S`'s probe. There are three different cases to be addressed:
   
   - Case 1: Both probes stop after encountering a broken link.
I can prove (see Appendix) that \( W_P \) and \( W_S \) must respectively be the \textit{avlLeftChild} and the \textit{avlRightChild} of \( F \) in the AVL tree. Furthermore, \( S \) must be a leaf node or a node with only right child in the AVL tree. The following operations are then performed:

a) If \( S \) has \textit{avlRightChild}, it will be connected to the \textit{avlParent} of \( S \) as a right child;

b) \( S \) sets \( W_P \) as its \textit{avlLeftChild}, and \( W_S \) as \textit{avlRightChild};

c) \( S \) sets the \textit{avlGrandParent} of \( W_P \) (which is \( S \)'s \textit{avlLeftChild} now) as its own \textit{avlParent};

- Case 2: Only one probe stops after encountering a broken link; the other stops after reaching the root, or there is no probe sent in that branch at all. I can prove (see Appendix) that \( F \) must have either \textit{avlLeftNode} or \textit{avlRightNode}, while not both. This child is then directly connected to its \textit{avlGrandParent} to substitute the failed node;

- Case 3: Neither probe encounters a broken link. I can prove (see Appendix) that \( F \) in this case must be a leaf node in the AVL tree, and thus no further operations are needed;

4) Both the \textit{avlLeftChild} and the \textit{avlRightChild} of \( F \) report their tree height to their new \textit{avlParent}, and, if necessary, perform re-balancing operations as in Step 4 of the joining process.

5) The timing list is recovered following the steps described in Section 4.B.

Fig. 12 shows an example of the recovery process for failed node 5. Suppose in the timing list its predecessor (\( P \)) is node 4 and successor (\( S \)) is node 6. According to the AVL tree construction algorithm, they should be respectively in the left subtree and the right subtree of node 5. In Step 2 of the recovery algorithm, nodes 4 and 6 probe toward the AVL root and stop nodes 2 (\( W_P \)) and 7 (\( W_S \)), which are respectively the \textit{avlLeftChild} and the \textit{avlRightChild} of

31
failed node 5. Node 6 then serves as a substitute for node 5 (Fig. 12b), and a double rotation is then performed to re-balance the AVL index tree (Fig. 12c).

Figure 12  An illustration of failure recovery (failure recovery case 1).

(a) AVL tree with node 5 failed.

(b) AVL tree after node substitution.

(c) AVL tree after rebalancing
CHAPTER SIX: PERFORMANCE EVALUATION

I evaluate the performance of TAG under various network settings, with a focus on the following two important measures: control overhead and streaming quality, as well as their sensitivity to parameter settings. I also compare TAG with other overlay on-demand systems, in particular, oStream, a pure tree-based system.

6.1 System Configurations

Unless otherwise specified, the results presented in this section are based on the following default configurations; yet, similar results have been observed with other configurations, and the impact of several key parameters will be further investigated in the end of this section.

The content server has 10 videos for streaming, each with 256 Kbps rate and 2-hour length. The length of a segment (or a time unit) is 1 second, and the buffer at a node can accommodate $1080 = 15\%$ of a video stream, which is equally split into the forward and backward buffers. The size of the candidate set for parent or gossip partner search is 12, and each node has 5 gossip partners.

The underlying network topology is generated using the GT-ITM package [26], which emulates the hierarchical structure of the Internet by composing interconnected transit and stub domains. The network topology for the presented results consists of 10 transit domains, each with 7 transit nodes, and a transit node is then connected to 6 stub domains, each with 7 stub nodes. The total number of nodes is thus 3010. I assume that each node represents a local area network with plenty of bandwidth, and routing between two nodes in the network follows the shortest
path. The initial bandwidth assigned to the links is as follows: 1.5 Mbps between two stub nodes, 6 Mbps between a stub node and a transit node, and 10 Mbps between two transit nodes. I will also inject cross traffic in the experiments to emulate dynamic network conditions.

To mitigate randomness, each result presented in this section is the average over 10 runs of an experiment.

6.2 Control Overheads

I first consider the control overhead of TAG, in particular, the overhead for node joining, leaving, or failing in a dynamic overlay. I am interested in both local and global overheads and thus adopt two measures: the maximum node cost, which represents the maximum possible overhead at each node, and the overall cost, which represents the total control overhead of the system per operation. The costs are measured in terms of the number of messages exchanged per operation, thus reflecting both the bandwidth consumption and the execution time.

6.2.1 Overheads of Join and Failure Recovery

Fig. 13 shows the maximum node cost for a joining operation in the three variations of TAG, namely, basic TAG with sequential accesses (TAG-S), basic TAG with non-sequential access (TAG-N), and TAG with non-sequential accesses and AVL indexing (TAG-NA). I assume that the content server is the only initial node in the system, and other nodes then join the system following a Poisson arrival with an inter-arrival time of 2 seconds. In TAG-N, the naive timing list searching algorithm is employed.
Intuitively, the joining node itself incurs the maximum node cost, which is mainly for joining the timing list and initiating the search for parent and gossip partners. As shown in Fig. 13, the cost monotonically increases with increasing the overlay size in the initial part, and becomes almost a constant when the overlay size is greater than 100 nodes. Since TAG-NA incurs extra overhead to maintain the AVL index tree, its maximum node cost is higher than that of the other two.

Nevertheless, as shown in Fig. 14, the overall join cost of TAG-NA can be much lower than that of TAG-N. Since the overall cost is calculated across all the affected nodes in a join operation, it is related not only to the individual node cost but also the number of affected nodes.
For TAG-N, the overall join cost is almost a linear function of the system size, for the number of involved nodes is proportional to the overlay size in the naive searching algorithm. For TAG-NA, this becomes a logarithmic function (note that the y-axis in Fig. 13 is log-scaled), suggesting that the joining operation with AVL indexing is scalable, and the cost for maintaining the AVL tree can be ignored for large networks. On the other hand, for TAG-S, the overhead is almost a constant, as only a limited number of tail nodes in the timing list are affected.

The maximum node costs and the overall costs for a failure recovery operation are shown in Fig. 15 and 16, respectively. The general trends are quite similar to that of joining operations, and the overall costs for failure recovery are slightly higher in all the three TAG variations. This is because more nodes are affected, in particular, all children of the failed node have to re-locate parents.

Figure 15  Maximum node cost for failure recovery
In summary, the joining/failure recovery operations are efficient for both TAG-S and TAG-NA, while that for TAG-N might suffer from high cost in large overlay networks. I thus focus only on the performance of TAG-S and TAG-NA in my following experiments.

6.2.2 Overhead Distribution

To better understand the control overheads, I present a typical control traffic distribution with different overlay sizes in Fig. 17. Here, the control traffic volume is normalized by the total traffic volume (control + video traffic). The results suggest that the Echo messages for heartbeat protocol contribute to most of the control traffic. Since Echo messages are only locally exchanged, its ratio is almost constant with different system sizes. More importantly, the total control traffic, as compared to video traffic, is essentially minor, i.e., less than 2%. In other words, the TAG system is scalable from its control traffic point of view.
6.3 Streaming Quality

Given that playback continuity is critical for streaming applications, I adopt the Segment Missing Rate (SMR) as the major criterion for evaluating streaming quality. A data segment is considered missing if it is not available at a node till the play-out time, and the SMR for the whole system is the average ratio of the missed segments at all the participating nodes during the simulation time. As such, it reflects two important aspects of the system performance, namely, delay and capacity.
For comparison, I also simulate an existing on-demand overlay streaming system, $oStream$, with the same network and buffer settings. $oStream$ employs a pure tree structure, in which each node caches played out data and relays to its children of asynchronous playback times. A centralized directory server is used to maintain the global information of the overlay, and facilitates node join or failure recovery. Detailed about $oStream$ can found in [20].

6.3.1 Streaming Quality with Bandwidth Fluctuations

I first investigate the performance of TAG under dynamic network environments with local and global bandwidth fluctuations.

To emulate local bandwidth fluctuations, I randomly inject traffic to the network links such that the available bandwidth at each link various over time, yet the total available bandwidth of the network remains constant, which is 0.995, 0.985 and 0.8 of the base setting respectively (with no cross traffic).

Figure 19  Quality with local bandwidth fluctuations (0.995 of base setting).
Fig. 20 Quality with local bandwidth fluctuations (0.985 of base setting).

![Graph showing quality with local bandwidth fluctuations (0.985 of base setting).]

Fig. 21 Quality with local bandwidth fluctuations (0.8 of base setting).

![Graph showing quality with local bandwidth fluctuations (0.8 of base setting).]

Fig. 19-21 shows the segment loss rates (SMRs) for TAG and oStream over time in different network settings. As we can see from the above figures, the loss rates of TAG and oStream both increase as the network bandwidth decreases from 0.995 to 0.8 of the base setting. When there is enough bandwidth (Fig. 19), the average SMR of oStream is slightly better than that of TAG, though it has more fluctuations. When the network capacity decreases, TAG outperforms oStream however. Specifically, in the bandwidth constrained case (Fig. 21), the segment missing rate of TAG is generally around 0.05 to 0.1. Note that we have not applied channel coding here, e.g., FEC, which would further reduce the loss rate. From a video decoding point of view, such losses can be effectively masked by interleaving or error-concealment techniques. On the other hand, the loss rate of oStream greatly fluctuates over time, and the peak
value can be as high as 0.7, resulting in poor video quality. This is because oStream relies on a specific tree structure for streaming, and the bandwidth reduction at an internal link of the tree, especially those close to the root, could result in the loss multiplicity problem.

Figure 22  Quality with different overall network bandwidths.

It is known that not only the available bandwidth of local links dynamically changes, but also the overall available bandwidth of a network changes over time on an hour or daily basis, e.g., working and sleeping hours, working days and weekends. Hence, in the second set of experiments, I compare the performance of TAG and oStream under different global network bandwidths. Their segment loss rates are depicted in Fig. 22, where the overall available bandwidth of the network is gradually reduced from 100% to 60% of the base setting.

Not surprisingly, for both TAG and oStream, SMR increases with decreasing the overall bandwidth. However, the increasing rate for TAG is generally lower than that of oStream, especially when the reduction is less than 25%. As an example, for a reduction of 25%, the SMR of oStream has reached 0.35, or 35% of the segments are lost or missed the playback deadline; yet the SMR of TAG is still close to 0.1. This is because oStream explores the available bandwidth at a small subset of network links only, i.e., those tree links, while TAG makes more effective use of the available bandwidth across much more paths. In addition, as explained before, once a segment is lost at a high level node in an oStream tree, it will be lost at all downstream
nodes. This is, however, not the case in TAG for each segment has multiple potential suppliers. As a matter of fact, I have observed that over 90% of the data segments are delivered through the gossip process in my experiments, which confirms my intuition that gossip greatly enhances the robustness of the system.

6.3.2 Streaming Quality with Node Failures

In this set of experiments, I consider dynamic node failures. I assume that there is no global bandwidth reduction, so as to focus on the impact of node failures. Fig. 23 presents the segment missing rates as a function of node failure rate for oStream, TAG-S, and TAG-NA. It can be seen that, when there is no failed node, all the systems work well in this stable scenario. For TAG-S, the segment missing rates slightly increase with increasing the failure rate, but are generally less than 6%. The missing rate of TAG-NA is only a little higher than that of TAG-S. On the other hand, when 10% nodes fail, the segment missing rate for oStream can be as high as 25%.

**Figure 23 Segment missing rate vs. node failure rate.**

I next investigate the effect of random seeking, a key operation toward supporting interactive streaming. For both oStream and TAG-NA, random seeking can be implemented by letting the node leave the system and then re-join with the new playback offset. Fig. 24 compares
the streaming quality of oStream and TAG-NA in this scenario. Obviously, the tree-assisted gossip enables a quite robust delivering structure, making the re-seeking operation in TAG-NA much smoother than that in oStream. When 10% nodes perform reseeking, the SMR of TAG-NA is still lower than 10%, while that of oStream has reached 35%, which is difficult to mask at the receiver’s end.

Figure 24 Streaming quality as a function of position reseeking rate.

6.4 Sensitivity to Parameter Settings

In the last set of experiments, I study the sensitivities of the key parameters in the TAG system, in particular, the number of gossip partners, the size of candidate set, and the size of buffers.

Fig. 25 depicts the streaming quality as a function of number of gossip partners for TAG-S and TAG-NA under different system bandwidths. It can be seen that the segment missing rate reduces when increasing the number of gossip partners. This is consistent with my intuition that the system is more robust when increasing the number of suppliers. However, the improvement with over 5 partners is marginal. Since the computation and transmission overhead
of maintaining a large number of partners can be excessive, I believe that 5 is a reasonable choice, which is used in my default setting. Similarly, from Fig. 26, I choose 12 as the default value for $K$, the size of the candidate set in parent or gossip partner searching. As shown in my previous results, these default settings lead to reasonably low control overhead and quite good streaming quality under various network configurations.

**Figure 25** Segment loss rate as a function of the number of gossip partners.

**Figure 26** Segment loss rate as a function of the candidate set size.

Regarding buffer size, though it would be desirable if every overlay node caches all the video streams, it is often impractical given the large size of video streams. The choice of buffer size is also closely related to the number of active nodes in the overlay. As shown in Fig. 27,
when there are enough active nodes, even a small buffer can enable reasonably good streaming quality with node collaborations. Considering these factors, I set the buffer size as 20% of the video stream size in my experiments, which is sufficient to achieve low segment loss rates and, with this setting, the computation time for the scheduling algorithm is less than 20 ms, which is suitable for real-time streaming.

Figure 27  Segment loss rate as a function of buffer size under different overlay sizes (Normalized buffer size = buffer size / video length).
CHAPTER SEVEN:  
CONCLUSION AND FUTURE WORK

In this thesis, I have presented TAG, a tree-assisted gossip protocol for on-demand streaming. TAG has combined the best features of tree structure and random message dissemination: low delay with a regular tree topology, and robust delivery with random switching among multiple paths, which make effective use of the available bandwidth in the network. The performance of TAG has been extensively evaluated under various network configurations. The results demonstrated that it is highly robust in the presence of local and global bandwidth fluctuations. As compared pure tree-based overlay VOD system, TAG achieves much lower and stable segment missing rates, even under highly dynamic network environments.

The next step of this project is to build an Internet-based prototype for TAG, and conduct experiments over the PlanetLab. Possible further research avenues include optimizing the scheduling algorithm and overlay organization, dealing effectively with failure of multiple related nodes, and incorporating advanced coding techniques, such as layered or multiple-description coding. I expect more issues can be identified and solved in this prototyping process.
APPENDIX

In the failure recovery algorithm for AVL index tree, assume that the predecessor and successor in the timing list for the failed node $F$ are $P$ and $S$, respectively, and $W_P$ is the last node traversed by $P$'s probe and $W_S$ is that by $S$'s probe. I have the following observations:

Case 1: Both probe stop after encountering a broken links. In the AVL index tree, $W_P$ and $W_S$ must be the $avlLeftChild$ and the $avlRightChild$ of $F$, respectively. Furthermore, $S$ must be a leaf node or a node with only the right child in the AVL tree.

Case 2: Only one probe stops after encountering a broken link; the other stops after reaching the root, or there is no probe message sent in that branch at all. In this case, $F$ must have either $avlLeftChild$ or $avlRightChild$, while not both;

Case 3: Neither reaches a broken link. The failed node in this case must be a leaf node in the AVL tree.

Proof:

Case 1: In this case, obviously, both the $P$ and $S$ are non-empty. Moreover, according to the AVL tree construction algorithm, then $P$ must be in the left subtree of $F$, and $S$ in the right subtree. It follows that, in the AVL index tree, $W_P$ and $W_S$ must respectively be the $avlLeftChild$ and the $avlRightChild$ of $F$, the failed node.

Suppose $S$ has a left child, whose virtual join time should be less than that of $S$, but greater than that of $F$. That is, in the timing list, this left child should be the successor of $F$, which contradicts the fact that $S$ is the successor. Hence, $S$ must be a leaf node or a node with only the right child;
**Case 2:** I first assume that only $P$'s probe reaches a node with a broken link, which must be the $avlLeftChild$ of the failed in the AVL tree, as proved in case 1.

In this case, if $F$'s successor $S$ is empty, i.e., there is no probe sent in the right branch at all, $F$ cannot have a right child in the AVL tree; otherwise, one of the nodes in $F$'s right subtree will become its successor in the timing list.

On the other hand, suppose $S$ is non-empty and $F$ has a right child. Since $S$'s probe does not reach the $avlRightChild$ of $F$, $S$ cannot be in the right subtree of $F$. Assume $R$ is the root of the minimum subtree that covers both $F$ and $S$. Then, $S$ must be in the left subtree of $R$, while $F$ must be either $R$ itself or a node in the right subtree of $R$; otherwise, $S$'s probe will reach a broken link as well. It follows that the right child of $F$ has a virtual join time greater than that of $F$, but less than that of $S$. This contradicts my assumption that $S$ is the successor of $F$, and hence, the failed node $F$ does not have right child.

Similarly, I can prove that $F$ does not have a left child if only $S$'s probe reaches a broken link (Note that, we can ignore the case that $P$ is empty in the proof given that content server persists). In summary, the failed node has a single child in this case;

**Case 3:** Suppose $F$ has a non-empty $avlRightChild$. Since the virtual join time of this $avlRightChild$ is greater than that of $F$, $F$ must have a non-empty successor according to the AVL tree construction algorithm. As proved in Case 2, if $S$ is non-empty and $F$ has a right child, $S$ must be in the right subtree of $F$. Hence, $S$'s probe will encounter the broken link in the right branch, which contracts the fact that no broken link is encountered. Similarly, I can prove that $F$ does not have a left child, and it thus must be a leaf node.
BIBLIOGRAPHY


