Gotan: A Small-Scale Application-Level Multicast Protocol

Jason Lee
jaewook@cs.ucla.edu

Lih Chen
lihchen@cs.ucla.edu

Prabash Nanayakkara
etothex@cs.ucla.edu

ABSTRACT
As part of the hybrid overlay multicast architecture, small-scale application-level multicasts must handle the construction and maintenance of the stub domains into a multicast tree. Whereas the overlay infrastructure allows for more specific optimizations in multicasting efficiently, the end users attached to one of the overlay’s proxy nodes follow much more dynamic behavior. Thus, the hybrid overlay multicast utilizes a two tier system that has different multicast strategies, an overlay protocol for the backbone domain and an application-level one for the stub domain. In this paper, we present a small-scale application multicast protocol for the end users and proxy node that form a stub domain. End users must connect to the local proxy to receive their multicast tree parent information as well as give their neighboring link information. The proxy uses two a two tier approach to either construct an un-optimized balanced tree or a hierarchy of clusters that efficiently groups nodes by their link information. We have implemented this algorithm in C++ and early results show promising improvements in efficiency optimizations as well as reliability.

1. INTRODUCTION
The rapid growth of streaming multimedia applications needs a multicast solution to handle the issues of scalability and efficiency. Although IP Multicast solutions have been available, the prohibitive costs of setting up their infrastructure have limited their use. Thus, application-level multicasts have become popular alternatives because of they do not need wide-scale deployment nor standardization on group membership.

In this paper, we differentiate these application multicast algorithms from multicast protocols developed for overlay networks. We use the term overlays here refer to the network formed by a set of fixed proxy nodes within the backbone domain. Because this system tends to have high bandwidth links and a network topology that doesn’t rapidly fluctuate as much as the end users, an overlay multicast therefore capitalizes on this property to offer efficient resource usage and multicast performance (in terms of latency, bandwidth, etc). The proxy nodes mentioned earlier are also responsible for the creation of these overlay multicast trees.

Figure 1: Backbone and Stub Domains. The servers within the ISP regions form the backbone (or transit) domains whereas the end users and the associated proxy node form a stub domain.

On the other hand, an application-level multicast has more flexibility and can adapt to network dynamics, particularly the behavior of end users. In a network of backbone (or transit) domains and stub domains as shown in Figure 1, the end users and one proxy node would therefore form application-level multicast trees in the stub domain.

By creating such a two-tier infrastructure, multicasts using both multicast and application-level multicast trees are more scalable to large group sizes that have two distinct topologies. Control messages between end users are thus limited to the local scope. Furthermore, we can optimize each multicast tree more efficiently. The characteristics of the overlay’s proxy nodes would tend to be more similar with each other, whereas the stub domains must be more accommodating to the potentially diverse set of end users.

In this paper, we present the “small-scale” application level multicast protocol Gotan as a multicasting solution for the stub domain of such a network. Section 2 details the architecture of the system, followed by the description of the implementation, and preliminary results available from experiments. Related work is covered in Section 5 along with future work before concluding in Section 7.
2. ARCHITECTURE

2.1 Two-Tier Approach
Gotan uses a two-tier approach. First, it uses a simple, but an un-optimized, balanced tree for simplicity and fast insertions of nodes. The readers can think of this sub-optimal tree as a buffer for the newly joined clients. Since it is reasonable to assume that there might be a case of a large influx of client nodes requesting to join a given proxy node at roughly the same time, especially when our system first comes up, we decided that we are not going to incur any significant overhead on a per node-basis during joining. Instead of dealing with the issue of optimization right from the start, we defer it to later. Hence, we propose our second approach.

The second approach is the core of our project. We use a progressively-optimized cluster tree. It is progressively-optimized in a sense that as the proxy node learns more about the round trip time latency among different client node pairs gradually over time. In order to acquire the inter-node latency knowledge, the proxy periodically chooses a set of pings to be done by the client nodes in a randomized fashion. Once the client nodes fulfill the proxy's ping request, they forward the RTT results back to the proxy. As a result, it can progressively make a better decision on clustering based on the gradual accumulation of latency information. In order to make a smooth transition to this cluster tree from the sub-optimized tree mentioned above, we introduce each freshly-joined client a notion of seniority. If a client node has been with our un-optimized balanced tree over a specified amount of time, we grant it the privilege of joining our optimized cluster tree.

It should be obvious to the readers that we can easily replace our simple un-optimized scheme with some other strategies of handling newly joined clients, such as using a minimum spanning tree. We, then, can do further benchmark measurements against our cluster algorithm. However, it is important to keep in mind that we do not want to incur too much overhead for each client join request to a point that our system becomes un-scalable.

2.2 Basic Balanced Tree
From this point on, we will refer to the basic balanced tree as BBT, unless specified. The purpose of having a BBT is to minimize the optimization overhead during the joining of clients. Since the BBT acts as a buffer to our optimized tree, we expect that this scheme would scale well during a scenario where the activity of joining and leaving is at a high frequency.

2.2.1 Joining the basic balanced tree
Figure 2 shows a BBT with a maximum fan out of 2 (each node can have a maximum number of 2 children). As shown, we encode each path with a string of $2^{|l|}$ digits, where $l$ denotes (current level - 1). The basic balanced tree keeps track of the path of the next insertion. After insertion, the basic balanced tree updates the path string to refer to the next insertion path. Needless to say, the BBT is aware when a new level needs to be established. This case is encountered whenever all digits in a given level is equal to the maximum fanout - 1.

Figure 2. We add the newest Node (the white circle) to the first available position. The numbers in the bracket represents the order of traversal.

2.2.2 Leaving the basic balanced tree
Again, we assume that we have a BBT with a maximum fan out of 2. If a node either decides to leave the BBT or is ungracefully terminated, the BBT will be notified. If

Figure 3. Node B replaces the dead Node A. We use the most recently added node because it is guaranteed to have no children.
the existing node is not the most recently added node, its departure will create a hole in the BBT. To handle the removal of an “existing” node, the BBT simply finds the most recently added node to replace the hole. For example, in Figure 3, Node A decides to leave. Since Node B is the most recently added node, the BBT uses it to take the place of Node A in order to fill the hole.

2.3 Optimized Cluster Tree

Unless specified, we will refer to the Optimized Cluster Tree as OCT. Unlike the BBT, the role of OCT is to provide a close to optimal tree construction based on an Euclidean Cluster Algorithm. Though finding the most optimal clusters is a NP-complete problem, this algorithm finds a solution that is close to optimal. Overtime, the proxy node will gradually gather more information about the inter-node latency. With more information, the proxy would be able to make a more realistic cluster.

2.3.1 Euclidean Cluster Algorithm

In order to calculate a Euclidean distance between any pairs of two nodes, we must design a systematic way to assign a coordinate to each node. We define the position of a node in an n-dimensional space, where n is the number of existing clients that are either currently members of the BBT or members of the OCT. A coordinate of a node is comprised of the distances from itself to all other existing nodes. For example, let's assume that we would like to define the coordinate for UCLA and that the other n-1 nodes are Berkeley, UCSD, and USC. The latency between UCLA-Berkeley is 1.73; the latency between UCLA and UCSD is 1.24; the latency between UCLA and USC is 0.57, and of course, the latency between UCLA and itself is 0. We, furthermore, define the ordering in a coordinate as (UCLA-UCLA, UCLA-UCSD, UCLA-USC, UCLA-Berkeley). Consequently, the coordinate for UCLA is (0, 1.24, 0.57, 1.73). We chose latency as a key component of our coordinate system because the latency between any given two nodes is relatively static. If we currently do not have the latency information between a node pair, it will be assigned the maximum value temporarily until the latency knowledge comes to light. Once we have the coordinates, \((x_1, x_2, x_3, ..., x_N)\) and \((y_1, y_2, y_3, ... , y_N)\), we can easily compute the Euclidean distance.

\[
\text{Euclidean Distance} = \sum x_i - y_i, \text{ where } i = 1...n.
\]

Our Euclidean Cluster Algorithm clusters nodes based on the Euclidean distance mentioned above.

```plaintext
vector of vectors Euclidean Dist. v.
for(int i; i < size of v; ++i )
{
    Vector curr_cluster = v[i];
    CH = first element in curr_cluster;
    NEW_C = maximum(EU-dist(CH, vi));
    v[i+1][0] = NEW_C;
    foreach element in curr_cluster
    {
        if( ED(CH, vi) > ED(C, vi) )
        {
            join the C cluster.
        }else
        {
            stay in CH cluster.
        }
    }
}
```

The pseudo-code of the Euclidean Cluster Algorithm is shown above. We use a 2-dimensional dynamic data structure, such as a vector of vectors, to do the bookkeeping of which node belongs to which cluster group. The first dimension keeps track of each cluster, and the second dimension keeps track of each node in a cluster. The size is of the first dimension of the 2-dimensional data structure is determined by a predefined number of clusters.

We introduce the notion of cluster heads. A cluster head can be thought of as the leader of a particular cluster. During the initial stage of the algorithm, all nodes are assigned to the first cluster group. We, then, randomly choose a node as a cluster head (CH). The algorithm goes through each elements in the cluster group to find an element(NEW_CH) with the maximum Euclidean Distance from the cluster head. The NEW_CH element is removed from the current cluster and is assigned to a new cluster as the cluster head for the new cluster. The algorithm will go through each element(vi) to compare the distance between CH and vi and the distance between NEW_CH and vi. If the distance of NEW_CH is less than the distance of CH and vi, then vi is reassigned to the new cluster, where NEW_CH is the
cluster head. The algorithm performs the procedure described above iteratively.

When the algorithm terminates, what we have is a number of clusters, where the members of a cluster tend to be close to one another in terms of Euclidean distance.

The algorithm takes \( O(\log n) \) to sort the elements in a cluster. It takes \( O(n) \) to find the element with the maximum Euclidean distance from the cluster head. It takes another \( O(n) \) to determine whether an element is reassigned to the newly established cluster group. The algorithm repeats the same procedure iterative for \( m \) number of clusters. As a result, the run time of our cluster algorithm is \( O(m^*(2n+\log n)) = O(mn) \).

2.4 Proxy Node
The proxy node handles all incoming join requests and assigns the parents to each client node. Though not part of this implementation, the proxy node will also handle overlay messages as part of the hybrid overlay multicast protocol. Because the targeted receiver group is intended to be on a small-scale, the centralized role played by the proxy allows for its total awareness of every connected end user without impacting scalability. Other proxy nodes in the backbone domain overlay will also perform the same functions to other end users in their own multicast trees.

2.4.1 Joins
When a client requests to join the multicast tree, the proxy immediately places the node onto its BBT. If the client node is still on the BBT when the proxy node begins a new round of clustering, this client node will then be notified of its new parent within the cluster.

2.4.2 Pings
Upon joining the multicast tree, the proxy creates a list of TupleRTT’s, i.e. a tuple of two nodes and their round trip time, whose link measurements are not yet known. After a certain user-specified interval, the proxy node dispatches a subset of these tuples to every node and asks them to perform the requested measurements. The client node can then construct the \( N \times N \) matrix of latencies with which the clustering algorithm uses in its calculations.

2.4.3 Clustering
Clustering rewards end users that do not immediately join and leave our multicast tree. After the proxy runs the algorithm mentioned in Section 2.3, the proxy then informs every end user of their parent in the cluster. However, if no new link measurements have been received, the proxy node does not re-cluster the entire receiver set.

2.4.4 Heartbeats
A proxy node initiates the heartbeat of its children (that is, the nodes directly connected to the proxy), who likewise heartbeat their children. We use these tests for detecting node failures and if a parent notifies the proxy of a dead child, we use the dynamic leave algorithms mentioned in the previous Section.

2.4.5 Multicasting
We dispatch multicast message with our udp socket, and by flooding the network with multicasts, we can effectively test our implementation. In our tests, we place a unique number and timestamp in each message for every round of multicasting sent by the proxy. The clients check the identifying numbers to determine whether any previous multicast messages have been dropped. The timestamp gives the amount of time it took for that given packet to reach the end user. These two measurements provide us the latency and reliability.

2.5 Client Node
The complexity of a client node is much simpler than that of a proxy node. Basically, it operates according to the proxy’s messages. It also keeps a record of its child nodes and its parent node.

3. IMPLEMENTATION
Gotan was implemented in C++ with over 10,000 lines of code. We have written our own serialization and socket libraries as well as making the software easily modifiable with different algorithms. Therefore this software base simplifies the implementation of future version of Gotan or any other multicasting algorithm.
3.1 Serialization Library
In order to communicate among themselves, a source node has to transmit packets across the pipe to a destination node using either TCP or UDP. Since a packet is simply an array of bytes, we could simply design a scheme to provide each node the knowledge of what each byte in a specific array position represents logically. For example, we can specify the first byte as the message type, the second as the source id, the third as the destination id, and so on. This approach, though, is simple to implement, it has many pre-existing limitations on the extensibility for an ongoing development, as well as, any future modifications.

Consequently, we decided that it’s advantageous for us design our own serialization library. With this serialization library, we are able to serialize a C++ serializable object into an array of bytes to be sent across a pipe. On the receiving end, we also provide the mean of un-serializing the bytes into the original C++ object. As a result, no long do we need to worry about the byte organization of our packets, instead we focus on designing messages with meaningful information. Another good benefit we get from this scheme is its robustness and maintainability.

3.2 Socket Library
The existing C socket library is not very easy to use. The programmer is responsible of dealing with the little endian/big endian problem, as well as a numerous number of other issues. The goal of the socket library is to provide a well-defined interface to abstract the programmer from the mundane details mentioned above. Moreover, our socket library also integrates the use of the serializable objects. Simply after constructing a serializable object and invoking a socket’s send method, the serializable object will be well on its way to the destination.

3.3 Messages
All the messages used are serializable and are dispatched with our socket library. Abstract node handles the sending and receiving of these objects for the proxy and the client. The following is a list of the messages used:

1. Child_Proxy_DeadParent_Msg – Child informs the proxy that its parent is no longer sending heartbeats to the child.
2. Client_Join_Request_Msg – Client dispatches a join request to the proxy.
3. Client_Parent_Msg – Client registers itself with the assigned parent node.
4. Client_Unregister_Child_Msg – Client unregisters itself as a child of its former parent node.
5. Heartbeat_Msg – Periodic probe to check consistency of links.
6. Multicast_Msg – We use this message for testing our multicast performance.
7. Parent_Proxy_DeadChild_Msg – The parent node informs the proxy that its child is no longer responding to heartbeats.
8. Ping_Request_Msg – A node sends this message to ping its neighbor. The response would contain the timestamp of the source node.
9. Probe_Request_Msg – Proxy requests a client to send pings to the listed nodes.
10. Probe_Response_Msg – Client returns its ping results back to the proxy.
13. Proxy_Repair_Msg – Proxy tells the node to serve as the replacement of a dead node. The client must then un-register itself from its current parent before notifying its new parent.

3.4 Abstract Node
The abstract node class is a parent class for both proxy and client node. Its primary responsibilities include defining the common interface methods across for both proxy and client, and handling the major of the synchronization issues with multithreading. The threads in the abstract node can be categorized into three distinct categories.

- Receiving Incoming Message Thread
- Handling Incoming Message Thread
- Outgoing Message Thread

The Receiving Incoming Message Thread accepts all incoming messages and push the incoming messages to a queue for processing in the future. Once a message is
pushed on the queue of incoming messages, the Receiving Incoming Message Thread signals the Handling Incoming Message Thread to start processing.

The Handling Incoming Message process the incoming message. It calls the appropriate handling routines, according to the message type.

There is a number of pre-instantiated Outgoing Message Threads. They are responsible of sleeping for a period of time specified by our simulator before sending the outgoing messages. Since all nodes are run on the same machine, we need to fabricate latency information for all node pairs.

The threads in abstract nodes work in conjunction to avoid any unnecessary blocking of the node process, resulting in a highly scalable framework.

4. EXPERIMENTS

4.1 Simulator

In order to evaluate our implementation on a single machine, we created a simulator that delays the transmission of packets among the client nodes. Specifically, the perl script configbuilder.pl generates a set of random latencies for every client to another client before launching the client node processes to use this network topology. The abstract node, from which proxy and client both inherit from, notices the availability of these link latencies and upon receiving a request to send a packet, will wait for the appropriate amount. However, the rest of the implementation does not realize that every node exists in the localhost without making possible modifications to reduce the computing costs.

Thus, this simulator did not hide the other systems calls, especially the calls to open sockets and threads for monitoring incoming and outgoing messages. Such a stimulator in hind sight turned out to be inappropriate for resource-limited test-beds, such as the single P3 700 MHz computer used. The results must be considered against this limitation and the implementation of a simulator after having programmed the protocol for a real-system consisting of one node per computer. For example, the clustered algorithms experience heavy UDP packet loss during cluster because of the sudden surge in control packets without altering the rate of multicast packets also being dispatched. When we are running 100 clients, each with their own threads, open sockets and (during clustering) TCP connections to the proxy, we were more amazed that the computer used for a test-bed still managed to return results.

4.3 Average Latencies for 50, 75, 100 Nodes

Running the experiment with 50 and 75 nodes differed from the simulation with 100 nodes. At 100 nodes, the computer could barely handle the load and thus, there was a considerable latency between the times each client registered with the proxy. For 50 and 75 nodes, all the clients registered within approximately 15 seconds. Thus, the 100 node experiment already began to cluster the nodes before half of them even joined the proxy, even though we set the cluster rate at one per three to five minutes. But for all three experiments, the clustered tree clearly optimizes on delay. Multicast trees that have high latencies to the leaf nodes were penalized further because each multicast round will overload the queues of every node and thereby causing increasing latencies. However, because we used a random network topology generator for each experiment, results may vary for the un-optimized balanced tree, especially when the links with high latencies happen to be added last. Also, the beginning of each experiment shows relatively low latencies, even when compared to the end values obtained by the clustering algorithm. Because the nodes closest to the root of the multicast tree receive the multicast the earliest, they report their times first before the leaves (who have the highest latencies) respond with their measurements.

4.4 Experiment with 75 Nodes

Due to the limitations of the test-bed used, clustering introduced an overhead that lead to dropped UDP packets. We believe that such a result would not occur in a real system as long as each node isn’t running an unusually high number of nodes, and the proxy has the

<table>
<thead>
<tr>
<th>Num of Nodes</th>
<th>Only Basic Balanced Tree</th>
<th>Optimized Multicast Tree</th>
</tr>
</thead>
<tbody>
<tr>
<td>50</td>
<td>36.263</td>
<td>7.494</td>
</tr>
<tr>
<td>75</td>
<td>16.247</td>
<td>10.12</td>
</tr>
<tr>
<td>100</td>
<td>30.578</td>
<td>8.14</td>
</tr>
</tbody>
</table>

Table 1. The values shown are the average latencies for the given number of nodes.
resources available for handling a high number of clients. Thus, minute three shows an increase in average latency as the first cluster tree turns out to not be the optimal solution. However, as the leaves report their times to the un-optimized tree, the clustered tree progressively improves, and there is a drastic improvement between minutes 7 and 10. In Figure the probability of drops increases during clustering for the reasons mentioned above. But as the clustered groups settle into relative stability, this number drops to zero (not shown in the graph).

Figure 4. The clustered group has its first clustering around minute three. It eventually approaches to the optimal solution at the 11th minute.

Figure 5. Percentage of dropped packets for 75 nodes.

4.5 Experiment with 100 Nodes
As mentioned in Section 4.2, the 100 nodes experiment began clustering before every node managed to register itself with the proxy node. Thus there is only a very slow increase in latencies before leveling off below an of 10 “time intervals” as the average latency. The earlier clusters are not as expensive as the later cluster when the proxy must dispatch the new parent assignment messages to a hundred nodes.

Figure 6. Similar to the 75 node scenario, the clustered multicast tree has significantly better latency performance than the un-optimized balance tree. Clustering began before all hundred nodes joined the network.

Figure 7. Percentage of dropped packets for 100 nodes.

5. RELATED WORK
Other application-level multicasts tend to be system-wide instead of the two-tier infrastructure. Of course others may develop hierarchies, but the approach would be consistent throughout the multicast tree.

ALMI [1] could limit the amount of monitoring per node to O(n) but that would create a sub-optimal tree. Narada [2] and Nice [3] are two other application-level multicasts but, like ALMI, that could replace Gotan but do not take advantage of the centralization of the proxy node. However, they would provide good comparison benchmarks in future experiments.

Bayeux [4] utilizes the Tapestry [5] overlay network to construct its own architecture for multicast. However, this system is more appropriate for the overlay multicast rather than the stub domains, especially since it requires Tapestry’s routing protocols. Another such
decentralized application-level multicast infrastructure is SCRIBE [6], but it targets large-scale networks and would not utilize the centralization offered by the proxy node.

Instead of these more traditional tree construction schemes, epidemic style protocols [7] offer a considerable advantage in scalability though only probabilistic guarantees in delivery. The individual node’s overhead is considerably reduced, but efforts must be made to avoid overwhelming to crucial network links and posses some form of adaptivity.

6. FUTURE WORK

7. Gotan needs to be tested on a more representative test-bed, such a cluster of machines on a local and wide area network. The results here, especially for the dropped packets, might not be relevant if a more powerful computer and an optimized simulator were used instead.

More comprehensive benchmarking must be done with simulations stress testing the fault tolerance and performance of the tree. One possible experiment would have nodes constantly joining and leaving the network, and the end measure of success would depend on the reliability of multicast messages being successfully received. The baseline would simply be the balanced tree’s performance. Traffic conditions should also be measured to compare Gotan’s performance to other multicast protocols. In particular, we would be interested in latency and bandwidth, of which bandwidth will be implemented in a future version.

8. CONCLUSION

We have implemented one part of the two-tier hybrid overlay multicast protocol. This application-level multicast tree utilizes the squared Euclidean distance clustering algorithm to effectively partition the nodes into clusters and progressively improve the clustering. Yet the construction of these clusters do not require the end users to immediately ping all other neighbors, but can be done in parts. The un-optimized basic balance tree, though not necessarily the best choice to complement the clustered tree, does provide a baseline for measurements and can be easily replaced.

In conclusion, we have presented Gotan, a small-scale application-level multicast protocol. Preliminary results from the simulations are promising, though run on a flawed test-bed. Gotan will be further developed to optimize other link metrics such as bandwidth and other algorithms to handle both un-optimized and optimized groups of nodes, but currently its scalable, fault-resilient multicast delivery makes a good match with an overlay multicast protocol in this hybrid scheme.

9. REFERENCES


