P2PCast: A Peer-to-Peer Multicast Scheme for Streaming Data

Antonio Nicolosi Siddhartha Annapureddy
{nicolosi,siddhart}@cs.nyu.edu

May 9, 2003

Abstract

Currently, the only way to disseminate streaming media to many users is to pay for lots of bandwidth. A more democratic alternative would be for interested users to donate bandwidth to help disseminate the data further. In this paper we discuss the design of P2PCast, a completely decentralized, scalable, fault-tolerant self-organizing system aimed at being able to stream content to thousands of nodes from behind a relatively low-bandwidth network.

Our system leverages the full bandwidth that has been committed by its users by striping the data which also enhances fault-tolerance. We propose a novel algorithm for managing these stripes as a forest of multicast trees in a systematic fashion under stress conditions. Finally, we discuss a prototype implementation of our system using *libasync* and sketch some preliminary results.

1 Introduction

1.1 Motivation

The dissemination of streaming media to a number of users has many interesting applications, spanning from leisure and entertainment (e.g. radio stations) to journalism and mass communication (e.g. news channels or even single users broadcasting their own live web-camera coverage). But as the number of subscribers grows, content providers would currently need to pay for lots of bandwidth. A more popular alternative would be to empower anyone with meagre resources to distribute data to an interested set of subscribers. Instead of paying for bandwidth, the trade-off is to ask the users interested in the content to help in the further dissemination of data. In such a *cooperative environment*, the users contribute some of their resources in exchange for using the service.

1.2 Goals

We now state the goals that we have set for the system. These represent the properties that are desirable in such a system.

The system should scale easily to thousands of nodes. A user must be able to join the system by sending a message to any random node that is already a part of the system. The overhead of joining the system should be minimal and a node should be able to leave without any intimation. The number of "management" messages also should be kept to a minimum. A user must be responsible for providing data to a moderate number of users with in the limits of its bandwidth. In particular, a user might not want to expend more bandwidth than he is getting. On the other hand, the system must be able to leverage the full bandwidth that has been committed to it by a user. A user must be able to recover

---

This paper reports on the final project for the class **G22.3250-001 — Honors Operating Systems** (Instructor: *Prof. David Mazières*, Term: *Spring 2003*.)

data even if one of the "parents" from which it receives the data fails.

We want the algorithm for structuring the nodes in the system to be decentralized, as a centralized manager would become a bottleneck. It is potentially a single point of failure and is very susceptible to malicious attacks.

A desirable property of such a system would be to conglomerate nodes which are stabilized into a small "subsystem" so that they are not affected by the vagaries of the other components.

In terms of security, a malicious user should not be able to corrupt the system and bring it to a halt. Such a node should not be able to cut off some user who is part of the system from availing the service. Ideally, such a malicious node should be detected as soon as possible and expelled from the system.

1.3 Setting & Assumptions

In this section, we state the setting in which our system operates and the key assumptions that we make about the participants and the system as a whole.

In the rest of the paper, the content provider is referred to as the *source*. The source is a node distinct from the rest of the participants involved in the distribution of data. All the participants are on an equal footing in terms of their responsibilities. The systems thus operates in a peer-to-peer (P2P) setting and the participants are henceforth also referred to as "peers" or "users". We assume that all the nodes in the system are connected by an asynchronous network like the Internet.

The source and the peers are assumed to have only a limited amount of bandwidth available for this task. Each peer donates just as much bandwidth as it obtains from the system. Specifically, we *don't* assume that a peer which has bandwidth to spare is willing to donate it for our system.

We use a simple fail-stop model for the peers i.e., the peers honestly adhere to the protocol and don't behave maliciously. Any given peer is trusted to deliver the data to the other peers it is responsible for. Further, a peer is assumed to stop interacting with the system once it fails and not introduce any corrupted messages into the system.

The peers can enter and leave the system arbitrarily. We only assume the existence of a routing overlay for finding a random node in a P2P setting. In particular, no assumptions are made about the routing infrastructure that is used. We only need the routing mechanism to find an arbitrary node already in the system to enable us to join. Furthermore, no explicit notification is required when a peer wants to leave the system.

2 Design Guidelines

In designing P2PCast, we first established few design guidelines to help us achieve the proposed goals. In this section we describe these guidelines, and for each of them we provide a brief rationale for adopting it.

- **Dynamic membership—Low stability requirements**.
  One of the most appealing features of peer-to-peer systems is that they do not impose a permanent commitment of resources on their users. Participants are expected (sometimes encouraged or compelled) to contribute to the system an amount of resources proportional to the benefit the gain from using the system itself. This has the advantage of attracting a potentially very large pool of users, but it poses the challenging problem of achieving robustness out of unreliable (or even untrusted) components.

  In the design of our multicast scheme, we made the (very common) simplifying assumption that participants exhibit a fail-stop behavior, and that node's failures are independent from each other. Such assumption has serious security implications, that we will discuss later (10.2).

- **Participants join by contacting a random node already in the system**.
  Allowing the system to have a (highly) dynamically changing set of users also entails that information about the system is almost always incomplete and only partially correct, as the rate at which the system evolves is too high for changes to be propagated in an exact and timely manner. Thus, for the resulting system to be robust, it is essential that common operations do not rely on perfect information, and this is even more important for the case of the Join operation, since that is the only circumstance under which the system obtain new resources.

  Having new users joining by contacting a random node already in the system can also turn out to be handy for scalability (see. Section 4) and/or security (see. Section 10.2).

- **Disseminate the content through a forest of multicast trees**.
  The most common choice for multicast schemes is to organize nodes in one *multicast tree*. As already observed in [1], this structure is not well-fitted for cooperative environments, because of its inherent unfairness: the work of forwarding the content would be limited to a minority of users (the internal nodes of the multicast tree), whereas more than half of the users of the system would benefit from it without contributing any resource.

  To balance the forwarding load among all the peers, the stream of data to be multicast is first broken down into f parts referred to as "stripes". Each stripe is then transmitted down its own multicast tree, giving rise to a forest of multicast trees, each of arity f (see Figure 1. By having each node being internal in only one tree, whilst being a leaf for all the others, it is possible to meet the bandwidth constraint at each node.

  In our system, we insist on each node contributing the *exact* same amount of bandwidth it receives from the system. This differs from the work of [1], where it is assumed that the participants will have different in- and out- bandwidth.

- **"All peers are equal, but some peers are more equal than others"**.
  For a peer-to-peer system to be robust, peers must play symmetric roles, preventing any particular peer from becoming "essential". In our multicast setting, this means that no peer should contribute more bandwidth than the others, a consideration already established in the point. However, even though any two internal nodes of a complete f-ary tree do contribute the same amount of bandwidth to the system, they may well not be equally important: in fact, failure of a peer serving as an internal node higher in the tree is potentially more harmful than failure of an internal node in the bottom levels. Hence, even though all peers play a symmetric role ("are equal"), some of them are somewhat more important (i.e. "more equal") than others.

  From previous experience with peer-to-peer systems, it is well-known that peers with a high enough up-time are more likely to stay up longer. This suggests peers that have been in the system for a while as natural candidates for the "more equal" role.

- **Trees are repaired locally upon failure**.
  Having a forest of multicast trees entails the advantage of distributing the forwarding load evenly across peers. Were this advantage to come at the cost of an increased run-time or communication complexity upon node failure, such design choice would be dubious. Thus, it is important that each tree in the forest be "locally-reparable" (i.e. requiring at most logarithmic amount of work plus $O(1)$ work in other trees).

- **Each multicast tree has a regular**

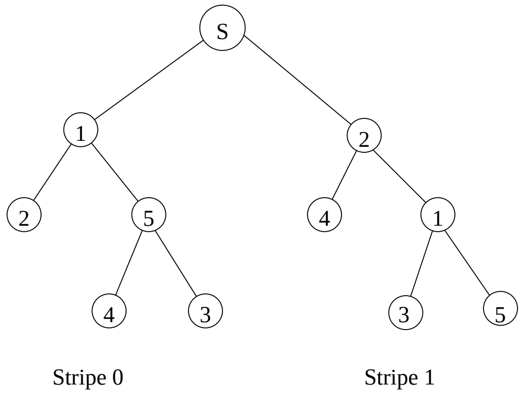

Figure 1: All nodes (except 3) are internal in one tree and a leaf in the other

structure.
The point of this seemingly vague guideline is that enforcing some kind of regularity on the structure of the forest of multicast trees is essential if we hope to ever being able to obtain any security guarantees from the multicast scheme. Indeed, in order to be able to prove properties about the flow of information in the system (for example, to certify proper delivery of each stripe's content), we ought to be able to reason about the structure of each multicast tree.

3 The P2PCast Protocol

The set of guidelines discussed above suggests the use of a forest of multicast trees to distribute the multimedia data to the participants. In particular, the content provider split the stream of data into f stripes. Each tree in the forest of multicast trees is an (almost) full tree of arity f. These trees are conceptually separate: every node of the system appears once in each tree, with the content provider being the source in all of them. To ensure that each peer contributes as much bandwidth as it receives, every node is a leaf in all the trees except for one, in which the node will serve as an internal node. We call the tree on which the peer is not a leaf the *proper tree* of that node. Since each node has a parent in all trees, every peer receives data from f different nodes. On the other hand, each peer has to forward the data that it gets on its proper tree to its f children. Thus, each node forwards exactly as much data as it gets from the system.

Ideally, we would like each tree in the forest to be a full f-ary tree, meaning that each node of the tree has either exactly f or exactly 0 children. Unfortunately, this is only possible when the total number n of nodes in the system is of the form $n = f * k + 1$ (for some positive integer k). Hence, we slightly weaken the "fullness" requirement by allowing *incomplete* nodes, but only in the next-to-last level. More precisely, the nodes in our forest of multicast trees can be classified in the following four categories:

- **Incomplete nodes**: A node with less han f children in its proper stripe;
- **Only-Child nodes**: A node whose parent (in any of the multicast trees) is an incomplete node;
- **Complete nodes**: A node with exactly f children in its proper stripe (if the above Only-Child case doesn't apply);
- **Special node**: The unique node which is a leaf in all multicast trees of the forest (discussed below);

Each tree in the multicast forest must satisfy the constraint that all the children of an incomplete node must be leaves. This is similar in some regards to the notion of *f-ary heap*, the main difference being that in an f-ary heap there can be only one incomplete node, and it must occur in the rightmost bottommost non-leaf position of the tree. In designing P2PCast, we discarded the idea of f-ary heap, as they go against the guideline saying that peers should be able to join by contacting a random node in the system. In fact, in an f-ary heap there is a single spot where new node can be attached, so that the joining process couldn't have involved any randomness. In contrast, imposing our milder constraint allows new users to be added in the system by contacting a random node; still, a satisfactory placement for the new node can be easily found in all the multicast trees of the forest, as explained in the next section.

Allowing nodes to be incomplete doesn't imply giving up the requirement that each peer contributes as much bandwidth as it takes, though. In fact, incomplete nodes uses the bandwidth that they are not consuming in their proper tree to "adopt" leaves in the other multicast trees. In other words, incomplete nodes make up for the children they are missing in their proper tree by (temporarily) having children in other trees. In this way, any incomplete node will also have f distinct children across all the trees, with all its children being leaves. Notice that insisting on incomplete nodes to only have distinct leaves as children suffices to ensure that no peer will be a descendent of the same node in more than one tree, since the only peers with children in more than one tree are the incomplete nodes. In turn, avoiding a peer to be depending on another for more than one stripe is essential to guarantee that, given a suitable encoding of the multimedia content as f separate stripes, every peer will be able to recover the streamed data even during maintenance condition involving the recovery from a node failure.[1]

[1] We actually allow an exception to this i.e., the special node can be the child of the same peer in more than one tree. This does not cause any problem since the special node is a just a "dummy" peer being maintained by the source itself.

Finally, here we notice another implication of our organization of peers as nodes in a forest of f-ary trees. For simplicity, suppose that the number of peers is $f^k + \frac{f^k-1}{f-1}$ for some k. This means that peers can be arranged to form a complete f-ary tree of height k, with f^k nodes being leaves and $\sum_{i=0}^{k} f^i = \frac{f^k-1}{f-1}$ internal nodes. We would like to have each node being an internal node in exactly one tree, which means that the f^k leaves of this tree should find a spot as internal node in one of the remaining $f-1$ trees of the forest. But since each of these $f-1$ trees can have at most $\frac{f^k-1}{f-1}$ internal nodes, there are overall $(f-1) * \frac{f^k-1}{f-1} = f^k - 1$ "internal node" spots available in the rest of the forest, so that one peer will serve as a leaf in *all* tree.

We exploit this property by having a *special node* playing the role of the "always-a-leaf" peer. At the present, such node is only used in bootstrapping the system: when a new multicast group is created, the special node is designated as root in all the f multicast trees in the forest. As new users join the system, the insertion algorithm will push the special node down to the bottom level of all multicast trees, always maintaining its "always-a-leaf" invariant.

However, we remark that the presence of such a special node could potentially turn out to be valuable to collect statistics about the multicast trees, or to help in securing the multicast scheme.

4 The Insertion Algorithm

For the sake of concreteness, in the rest of this paper we will assume $f = 2$. This will make the discussion of our algorithms easier, but the technique generalizes to handle any number of stripes.

4.1 The Idea

Before describing how the algorithm to insert new peers in the system works, we outline the basic underlying ideas, referring to the design guidelines discussed in Section 2.

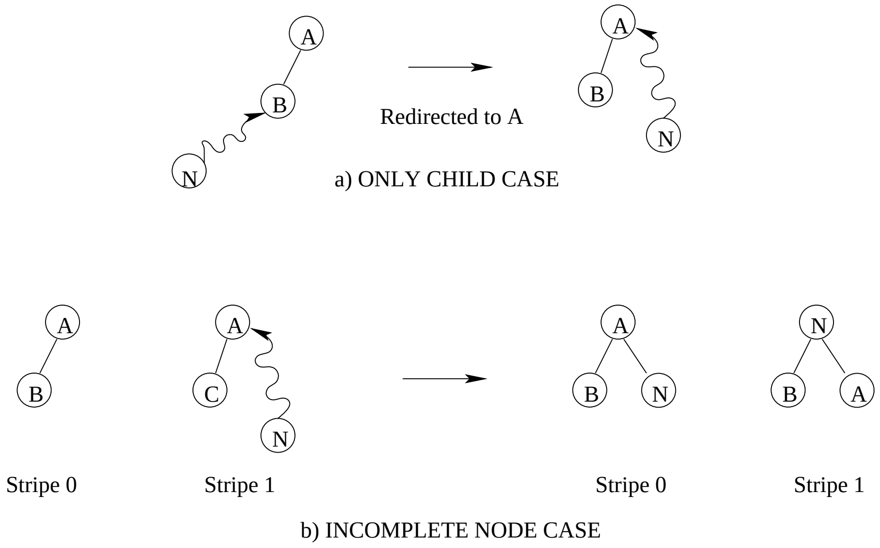

Figure 2: The Only-Child case and Incomplete node case for Insertion

Our design choices already limit the tasks of the insertion algorithm: the only details left uncovered are:

- the multicast tree in which the new peer will be serving as an internal node;
- the exact internal location that the new peer will take in its proper tree;
- the exact leaf location that the new peer will take in the other tree.

The choice of the proper tree for the new peer is based on the random node that was contacted when the Join process start. In that tree, the incoming user will take a place in the next-to-last level. There are basically two reasons why this is done so.

The first reason has to do with the expected uptime of peers in the system. When a new user joins, we do not know in advance how long it is going to stay around. However, experience with peer-to-peer systems [5] shows that new users come and go after a short time, whereas node that have been in the system for a while are likely to remain in the system somewhat longer. Hence, it makes little sense to place the new peer high in the hierarchy of its proper tree, since this would make it responsible for a large fraction of the users of the system. Moreover, the deletion algorithms (see Section 5) has communication complexity proportional to the height of the failed node, so that placing new peers near the bottom of the tree minimizes the damage in the case this node will fail shortly after joining.

Another consideration in favor of placing a prospective peer in the next-to-last level of its proper tree is that, according to our constraints on the structure of the multicast trees, this is the only level where incomplete nodes are allowed. As we discussed above, incomplete nodes are unavoidable, and in the case $f = 2$ that we are discussing, even the best[2] possible sequence of Join operations will cause incomplete nodes to be generated at every other Join. If we want to minimize the number of already-settled peers that must be rearranged per Join operation, the best choice is to have the new peer starting as an incomplete node, which is possible only if it joins in the next-to-last level.

Placing the new peer in the other tree, where it will be only receiving data as a leaf, does not

[2] Here we use “best” in the sense of minimizing the total number of occurrences of incomplete nodes in the forest of multicast trees.

entail any subtlety. Indeed, a node running as a leaf can do little harm (as far as security is not being addressed), so that any leaf-location in the tree is essentially as good as the others. However, we exploit such freedom to eliminate incomplete nodes from the picture when the chance arises.

4.2 Details of the Algorithm

We now describe in more details the Join process. The insertion algorithm begins with the new peer contacting a random node in the system. The algorithm then proceeds in different ways according to the status of the contacted node:

- **Incomplete node**: If the new user N contacted an incomplete node A, then A will arbitrarily select for which tree it wants to serve as an internal node, and adopt N in that tree. Before being able to adopt N, A must drop the child C that it is currently maintaining in the other tree. Node C will then be taken on by the new peer, which will also adopt A, thus becoming an internal node in the stripe that A didn't choose.

  This process is exemplified in Figure 2a, where we assume A decided to become an internal node for Stripe 0. At the N becomes an internal node in Stripe 1.

- **Only-Child nodes**: Recall that an Only-Child node is a peer having as parent an incomplete node (called respectively B and A in Figure 2b). In such case, the incoming user can simply be redirected to the parent (i.e. to A), which will then handle the Join operation as explained above;

- **Complete nodes**: If the new user N contacted a complete node A, then A must be a leaf in the other tree. For concreteness, assume that A is a complete internal node in Stripe 0 and a leaf in Stripe 1, as sketched in Figure 3. The incoming peer N will then become an internal node in Stripe 1, where it will take the place of A, adopting it at the same time (see Figure 3). To find a place for itself in stripe 0, N starts a random walk down the subtree rooted at the sibling of A.[3], which ends as soon as N finds an incomplete node Z (Figure 4a) or a leaf B (which must have a sibling, because otherwise its parent Z would have been an incomplete node; see Figure 4b). We consider this two subcases separately.

  - **Incomplete node subcase**: If at the end of its random walk N hits an incomplete node Z (Figure 4a), then N will be adopted by Z. Before being able to adopt N, Z must drop the child C that it is currently maintaining in Stripe 1. Node C will then be taken on by N, which will also adopt Z, thus becoming an internal node in Stripe 1.
  - **Leaf subcase**: If at the end of its random walk N hits a leaf B (Figure 4b), then N will take the place of B, adopting it at the same time (see Figure 4b). Notice that in this case the incoming peer ended up being an incomplete node.

- **Special node**: This case is only being executed if the Only-Child case didn't apply. Recall that the special node is the single peer in the system that is a leaf in all the trees. The new peer N can then simply join by adopting the special node in both trees.

As a final remark, we notice here that the above algorithm would still work if the contact node is selected ad hoc, rather than at random. However, the random selection of the contact node gives us a probabilistic guarantee that none of the multicast trees in the forest will grow too

[3]The case in which A is one of the root of the multicast trees, and hence it doesn't have any sibling, is dealt with specially by sending N directly to a leaf in the tree rooted at A.

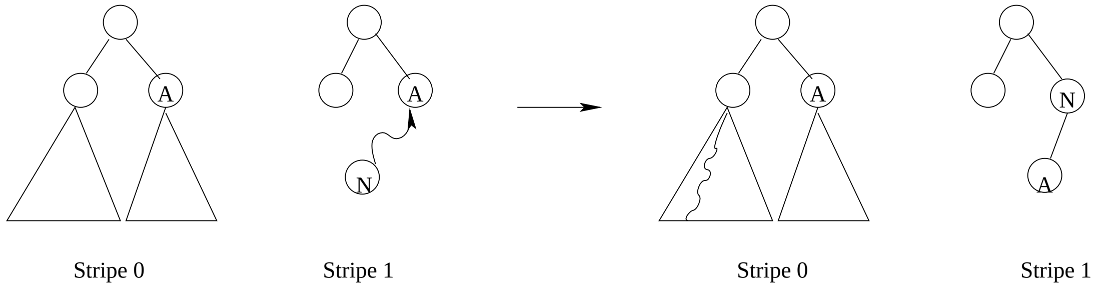

Figure 3: Complete node case for Insertion: part I

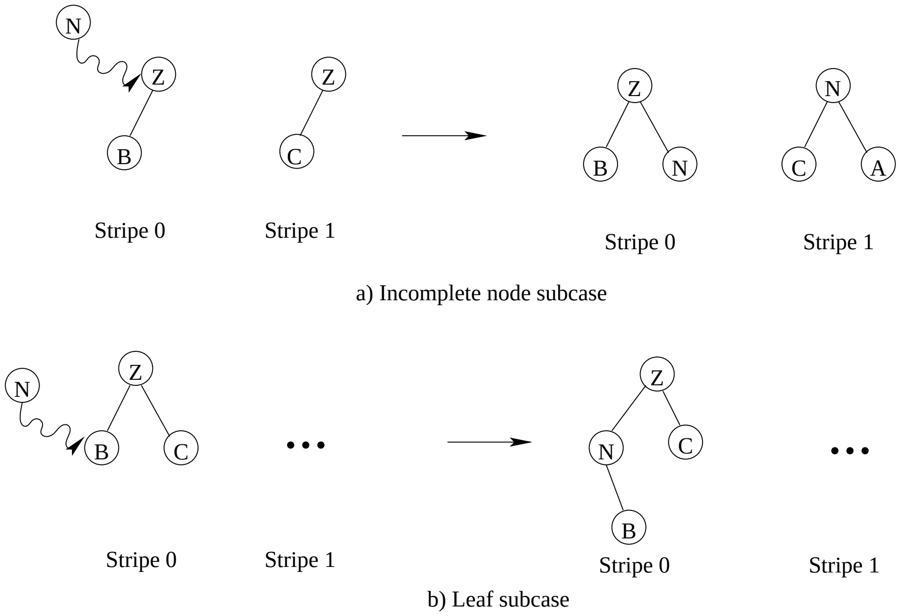

Figure 4: Complete node case for Insertion: part II

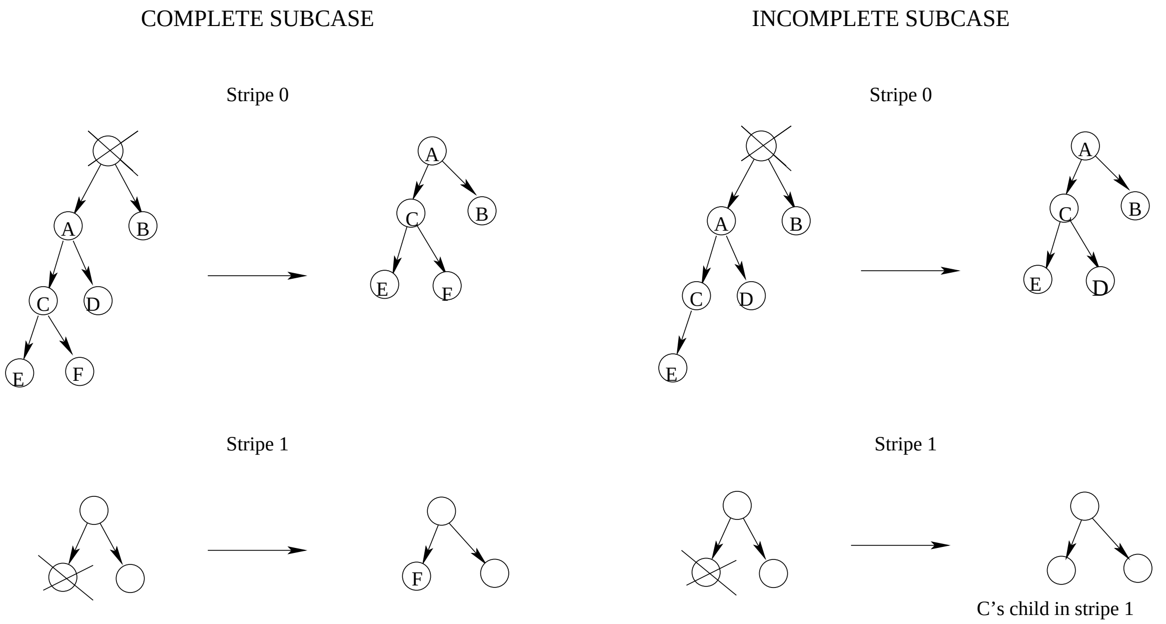

Figure 5: Illustration of the deletion algorithm

unbalanced. This is important because the deletion mechanism (discussed in the next section) has complexity proportional to the height of the failed node, which in the case of a balanced tree is logarithmic in the total number of nodes. On the other hand, an ad hoc selection of the contacted node can easily cause the multicast trees to assume the shape of a linear chain of length half the total number of peers—causing each subsequent node failure to take an excessive amount of time, and thus making the system unusable for large multicast groups.

5 The Deletion Algorithm

In this section, we describe the algorithm that is used to adjust the trees when a node leaves the system. A node may leave the system either because it fails or it's no longer interested in the data. We will refer to the event of a node leaving the system as "deletion".

We will not consider the case when the source (and the special node) fails as then there's nothing to be disseminated. Thus, the failing node must be either complete or incomplete (Only-Child nodes are treated according to their complete/incomplete status in their proper tree).

If the failed node is incomplete, then we simply replace the node with the corresponding child in each stripe tree. Thus, the incomplete node case is easy to handle.

If the failed node is complete, then a "bubble" is created at that place. We would now need to adjust the trees so as to fill in the "bubble". One alternative would be to search down the tree and replace it with a node at the bottom level i.e., a simple deletion algorithm for a binary tree would replace the "bubble" with the rightmost descendant of the left child of the deleted node. But in this process, a malicious node at the bottom level would have been pushed up to a much more responsible position. Another disadvantage with this scheme is that it might take a long time to find the replacement and the peers might time out. This might potentially lead to cascading deletions. To avoid these problems, we take the following approach.

When a node is deleted, the bubble is filled by a random child (see Figure 5). In this case, node A fills the bubble. But as A already has children

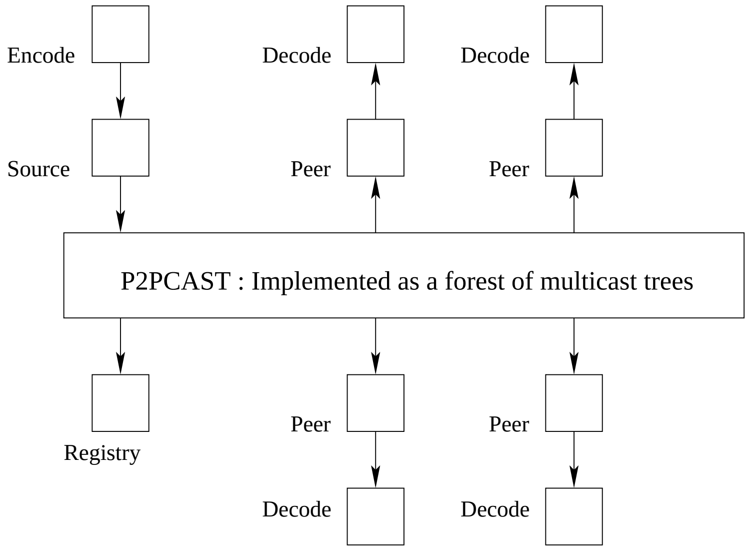

Figure 6: Architecture of P2PCast

C and D, it needs to shed a child. It sheds a random child, in this case D. In the figure, we have just shown a node D, but there might be a whole subtree below it. So, D is "directed" to C which sheds off a random child, in this case F. (In general, this process ripples down the tree until the next-to-last level is reached, which in the figure is represented by C). This node F is then taken as a child by the parent of the deleted node in the other stripe. (Note that this parent of the deleted node now becomes an incomplete node). This is the complete case as illustrated in the figure. On the other hand, when D is directed to C, C might be incomplete in which case it takes on the node that was "directed" to it. Then, C sheds off its child in the other stripe which is taken over by the parent of the deleted node in that stripe, as shown in the figure.

6 Architecture

In this section, we present the architecture of our system and where it fits into a real-world content distribution mechanism (see Figure 6).

The content provider uses a module called "encode" to divide the data to be sent into $f-1$ stripes and uses an error correcting code to generate the f^{th} stripe. This adds to fault tolerance when a stripe tree is "under adjustment". Here, we trade-off bandwidth for increased fault tolerance. The encoder then passes the f stripes to the source which uses P2PCast to multicast the streaming media to the users.

Each of the peers receives f stripes from the P2PCast system. The peer then passes these f stripes to the module "decode" which then recovers the data.

7 Implementation

In this section, we describe the various aspects of our implementation in more details. This includes a brief description of the registry, the P2PCast protocol and the data transfer mechanism. In our implementation, we use the asynchronous callback mechanism provided by libasync.

The registry is used to simulate the routing overlay in our implementation. We need the routing overlay simply to reach an arbitrary node in the system. Instead, a node that intends to join the system contacts the registry which returns a random node. The new node then contacts this node and the normal protocol ensues.

This is implemented as an RPC protocol with JOIN and DELETE functions. JOIN returns a random node while a DELETE can be sent by the child of a dead parent (when the child detects that its parent is dead) so that it may be deleted from the registry.

The P2PCAST protocol manages the algorithm for insertion and deletion and is the heart of our implementation. The RPC protocol supports the following functions:

- TAKEON — To take another peer as a child
- SUBSTITUTE — To take the place of a child of some peer
- SEARCH — To obtain the child of a node in a particular stripe
- REPLACE — Different from substitute in that the node which makes us its child sheds off a random child
- DIRECT — To direct a node to its "would-be" parent
- UPDATE — A node sends its updated state to its children

In the above calls, TAKEON, SUBSTITUTE, SEARCH are used for implementing insertion, whereas REPLACE, DIRECT are used for implementing deletion. UPDATE is used to ensure that each node knows enough about the state of all its parents to be able to start the deletion algorithm in case of failure of one of them.

For example, in the case of insertion (the complete case, incomplete subcase, see Figure 4a), N sends TAKEON to A, which returns its sibling in the other stripe tree. Then N uses SEARCH to reach the next-to-last level in stripe 0, and it sends SUBSTITUTE to Z in stripe 0; finally, C sends a SUBSTITUTE to N.

In the case of deletion (the complete case, see Figure 5), A sends a REPLACE to the parent of the deleted node (called the "grandparent"). When B also tries to do so, the grandparent sends a DIRECT to B asking it to contact A. B then sends a REPLACE to A which sheds off D and so on. Finally, F sends a REPLACE to the parent of the deleted node in the other stripe and the adjustment is done.

The data transfer mechanism has been implemented using asynchronous TCP sockets. Each peer listens on a TCP data port. After completing the Join operation, a new peer open a TCP connection to each of its parents' data ports, specifying which stripe it wishes to receive. A contacted parent only accepts connections by those peers that are currently registered (according to its view of the state of the system) as its children for the stripe in question. Once a connection is established, the parent peer employs a best-effort, one-shot attempt approach to deliver the stripe's content to the child. Should the child fail, the parent would just close the connection, without initiation any failure recovery mechanism. On the other hand, when the parent fails, the children reattempts to connect, and if no response is obtained, the deletion algorithm is started. Eventually, the orphaned child will be adopted by another peer: it will then open a TCP connection to the new parent, repeating the above process.

8 Evaluation

In this section, we do a brief evaluation of our implementation. We tested our system on OpenBSD 3.3. We were able to achieve the insertion of ten peers (running on the same machine) and observed that the system was working according to the specification. We expect this behavior to scale as new nodes are added, because all adjustments done during insertion are local. With the testing of ten peers, all the cases of the insertion algorithm have been exercised. We haven't been able to test the insertion algorithm on PlanetLab and the deletion algorithm (though it has been fully implemented) due to lack of time.

9 Related Work

There has been a lot of work on application level multicast for streaming media. Of the several systems proposed, the most related to the present work are SplitStream, SpreadIt, CoopNet. We will presently compare P2PCast with these systems.

Earlier systems like SpreadIt [2] used only a single multicast tree and hence weren't able to leverage the full bandwidth that was committed to the system by the users. We have made sure that any user who is part of the system must contribute exactly as much bandwidth as he gets.

Systems like CoopNet [4] differ fundamentally in the kind of algorithm that is used to build the multicast trees. While CoopNet uses a centralized algorithm which has obvious disadvantages, we use a completely decentralized algorithm which makes it more scalable and less prone to malicious attacks. Also, CoopNet doesn't explicitly manage the bandwidth resources contributed by the users.

Of all the systems mentioned above, SplitStream [1] comes closest to our approach. Both P2PCast and SplitStream use a decentralized algorithm and are based on the idea of stripes and a forest of multicast trees, with each tree representing the distribution of a stripe. But SplitStream has several disadvantages. It assumes that users would be willing to let it use their "spare" bandwidth. Also, SplitStream falls back on the "spare" capacity group whenever the random node it contacts is a complete node or the complete node rejects one of its children which joins the system anew. This would probably often result in falling back onto the spare capacity tree. Even with this spare capacity group, there is no guarantee that the trees don't grow unbalanced. On the other hand, P2PCast accommodates any request by making a very small number of local changes. We husband our bandwidth resources more carefully and rely much less on randomness to ensure balanced trees. Also, Splitstream/Scribe are intimately bound to pastry and they assume "rendezvous" nodes who might not be willing to avail the service. And the number of stripes in SplitStream is bound to the configuration parameter b in Pastry. On the contrary, P2PCast makes absolutely no assumptions about the underlying routing overlay.

10 Conclusions

10.1 Lessons Learnt

In this section, we discuss the lessons that were learnt during the course of the project and some features which we would like to reimplement. The implementation has been rendered very easy because of libasync. The asynchronous callback mechanism has proved to be very powerful in chaining callbacks and implementing the automata for the insertion and deletion algorithms.

The registry has proved to be very useful. In fact, we are now considering using it without any recourse to the routing overlay except possibly as a fallback mechanism. The registry is implicitly trusted (because it is maintained by the source) and as UDP is being used for the registry protocol, the overhead even when a number of users join simultaneously isn't significant and we might always refuse to accept connections if the number exceeds certain limits. Also, the registry saves the peers a costly routing lookup and enables the source to collect user statistics as a fringe benefit.

Finally, in our current implementation, data transfer is done directly on top of TCP. In retrospect, we feel that using an RPC protocol for data transfer would have considerably reduced the effort put into development.

10.2 Future Work

Working on the P2PCast project has been a great opportunity to get acquainted with the issues involved in designing, implementing and evaluating multicast protocols in the Peer-to-Peer setting, and future work is being planned in the area.

The most challenging goal left uncovered by our prototype P2PCast implementation is to achieve security. Defining the appropriate security requirements for such a P2P multicast protocol is an interesting research direction on its own. At the very least we would like our system to be able to withstand malicious peers behaving in a Byzantine manner. Also, users should not be able to get any benefit by joining the system more than once, so as to render a Sybil attack [3] harmless.

On a different note, we plan on experimenting with concrete applications of our system, such as an on-line radio station and/or a web-cam viewer. Developing such applications would allow us to evaluate our system on a concrete setting, based on an end-to-end argument. This would also provide valuable experience to improve the Application Programming Interface of P2PCast.

References

[1] M. Castro, P. Druschel, A.-M. Kermarrec, A. Nandi, A. Rowstron, and A. Singh. Splitstream: High-bandwidth content distribution in a cooperative environment. In *IPTPS'03*, Berkeley, CA, USA, 2003.

[2] Y. Chu, S. Rao, and H. Zhang. A case for end-to-end multicast. In *ACM SIGMETRICS*, pages 1–12, 2000.

[3] J. Douceur. The Sybil attack. In *IPTPS'02*, Cambridge, MA, USA, 2002.

[4] A. Mohr, E. Riskin, and R. Ladner. Unequal loss protection: Graceful degradation of image qaulity over packet erasure channels through forward error correction. *IEEE JSAC*, 18(5):819–828, 2000.

[5] S. Saroiu, P. Gummadi, and S. Gribble. A measurement study of peer-to-peer file sharing systems. In *Proceedings of Multimedia Computing and Networking 2002 (MMCN '02)*, San Jose, CA, USA, 2002.