

P2PCAST: A Peer-to-Peer Multicast Scheme for Streaming Data

Antonio Nicolosi*
nicolosi@cs.nyu.edu

Siddhartha Annapureddy*
siddhart@cs.nyu.edu

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 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, and sketch preliminary results obtained with a prototype implementation.

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.

1.2 Goals, Settings and Assumptions

We now state the goals 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 contacting a random node already in the system. The overhead of joining 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 within 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 data even if one of the “parents” from which it receives the data fails.

We want a distributed algorithm for structuring the nodes in the system, as a centralized manager would be a single point of failure, susceptible to malicious attacks. A desirable property of the 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.

Ideally, malicious users 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. Malicious behavior should be detected, and the responsible node expelled from the system.

We now state the setting in which our system operates and the key assumptions 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 system operates in a peer-to-peer (P2P) setting and participants are also referred to as “users”, “peers” or “nodes”. 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. Each peer donates just as much bandwidth as it obtains from the system. P2PCAST doesn’t currently address security issues; instead, we use a simple fail-stop model for the peers i.e., the peers honestly adhere to the protocol until they fail, at which point they stop interacting with the system. Peers can enter and leave the system arbitrarily. We assume the existence of a routing overlay (like Chord or Pastry) only for finding a random node to enable us to join.

1.3 Related Work

Application level multicast has recently received a lot of attention, and several systems have been proposed. For want of space, we briefly compare P2PCAST only against SplitStream [1], which 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. However, SplitStream has some limitations. It assumes that users would be willing to let it use their “spare” bandwidth, and that the aggregate out-bandwidth of the system is greater than its aggregate in-

*Courant Institute of Mathematical Sciences, New York University, USA.

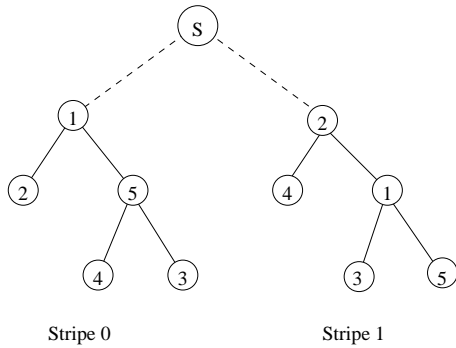


Figure 1. All nodes (except 3) are internal in one tree and a leaf in the other

bandwidth—an assumption that makes SplitStream ill-suited to the case of users with common cable or DSL connectivity. 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. In contrast, P2PCAST handles any request by making a small number of local changes.

We husband our bandwidth resources more carefully and rely much less on randomness to ensure balanced trees. Also, SplitStream is built on top of Scribe and Pastry, and relies heavily on the use of “rendezvous” points, which entails the problematic assumption that peers will be willing to join the system even if they’re not interested in the service (in the case that their *nodeId* shares a long prefix with the id of the group). Moreover, the number of stripes in SplitStream is bound to the configuration parameter b in Pastry. In contrast, P2PCAST makes absolutely no assumptions about the underlying routing overlay.

2 Design Guidelines

In this section, we describe the design decisions we made to achieve the proposed goals and the rationale behind them.

Disseminate the content through a forest of multicast trees.

The most common choice for multicast schemes is to organize nodes in one *multicast tree*, which is inherently unfair [1]. To balance the forwarding load among all the peers, the stream of data to be multicast is first broken 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).

The aggregate out-bandwidth required by the system is not greater than its aggregate in-bandwidth. For most users with low-bandwidth connectivity (such as cable modem or DSL), upload bandwidth is often a much more precious resource than download bandwidth. Consequently, P2PCAST ensures that each node contributes exactly as much bandwidth as it consumes, and that all the forwarding capacity available in carefully distributed

to match the system aggregate in-bandwidth requirement.

Participants join by contacting a random node already in the system. The system changes dynamically and hence it is essential that common operations do not rely on perfect information, especially for the JOIN operation, since that is the only circumstance under which the system obtains 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 3.1) and/or security (see. Section 4).

Correlate node reliability with the importance of the role played. In the P2P multicast setting, the Orwellian truth that “All peers are equal, but some peers are more equal than others” holds. Indeed, although any two internal nodes of a complete f -ary tree contribute the same amount of bandwidth to the system, they may well not be equally important: in fact, failure of a peer serving as internal node high 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 makes it possible to distribute 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 a design choice would be dubious. Thus, it is important that trees in the forest be “locally-repairable” (i.e. requiring at most logarithmic amount of work plus $O(1)$ work in other trees).

Each multicast tree has a regular 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 obtain 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 splits 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 $n = f \cdot k + 1$ (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 three categories:

- **Incomplete nodes:** A node with exactly one child in each stripe (such that no two children are the same);
- **Complete nodes:** A node with exactly f children in its proper stripe (and none in the remaining $f - 1$ stripes);
- **Special nodes:** A node which is a leaf in all multicast trees of the forest;

Notice that allowing nodes to be incomplete doesn’t compromise the requirement that each peer contributes as much bandwidth as it takes, as any incomplete node overall has f children across the multicast forest, with all its children being leaves. However, it is important to insist on these f children being *distinct*. This 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, preventing a peer from 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

Since incomplete nodes have a child in each stripe, it follows that no incomplete node can be a leaf in any tree of the multicast forest. This means that (ignoring the small set of special nodes) the set of leaves of any multicast tree is made up entirely of complete nodes. In other words, if v is a leaf in stripe i , then v must be a complete node, with proper stripe $j \neq i$. On the other hand, the node z that is v ’s parent on stripe i , being at the next-to-last level, could be either a complete or an incomplete node. We say that v is a **regular leaf** on stripe i if z is a complete node; and that v is a **lonely leaf** on stripe i if z is an incomplete node.

In P2PCAST, no peer takes up the role of a special node, as this would violate the requirement that users should contribute as much bandwidth as they consume. Instead, special nodes are mere place-holders maintained by the source. Were there many special nodes in the multicast forest, this would result in a waste of bandwidth, again violating our design guidelines. Luckily, special nodes are usually very few; in particular, under stable conditions, there is at most one special node, as we now argue.

¹We actually allow an exception to this i.e., special nodes can be the child of the same peer in more than one tree. This does not cause any problem as special nodes are just place-holder.

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 $\sum_{i=0}^{k-1} f^i = \frac{f^k - 1}{f - 1}$ 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 has $\frac{f^k - 1}{f - 1}$ internal nodes, there are overall $(f - 1) \frac{f^k - 1}{f - 1} = f^k - 1$ “internal node” spots in the rest of the forest, so that one peer will serve as a leaf in *all* trees.

3.1 The Insertion Algorithm

A new user joins P2PCAST by contacting a random peer already in the system. We assume that none of the trees in the multicast forest is empty, as the case in which less than f nodes are currently in the multicast group can be dealt with easily.

To add a new user to the multicast forest, each tree has to be ‘stretched’ a bit to make room for the new peer. To minimize the number of already-settled peers that must be rearranged each time a new user joins the system, the new peer is inserted in the system as an incomplete node. Besides its conceptual simplicity, this choice has a couple of additional advantages.

First, it can improve the robustness of the system, for a reason that 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 [4] shows that most users come and go after a short time, whereas nodes that have been in the system for a while are likely to remain in the system even 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. Instead, our insertion algorithm places the new peer in the next-to-last level in each tree of the multicast forest.

Second, adding new nodes as incomplete nodes can improve the performance of the system when users leave the system. As described in Section 3.2, the deletion algorithm 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 that this node will fail shortly after joining.

To insert a new peer u as an incomplete node, we need to find f distinct nodes v_1, \dots, v_f such that v_i is a regular leaf in stripe i , for $i = 1, \dots, f$. Then, we can ‘stretch’ the link between each v_i and its parent, and place u in the middle, as shown in Figure 2.

However, this approach ‘consumes’ regular leaves: to add u , f regular leaves were changed into lonely leaves, so that eventually all the leaf in some stripe will be lonely leaves, preventing further insertion from happening.

Luckily, once we have f incomplete nodes x_1, \dots, x_f (which entails the presence of f^2 lonely leaves), it is possible to *consolidate* them, changing them all into complete nodes (see Figure 3). Basically, consolidation consists of arranging the nodes x_1, \dots, x_f as a small subtree, and then rotating the root, so that,

for $i = 1, \dots, f$, node x_i becomes a complete node for stripe i .² After consolidation, all the lonely leaves have been changed into regular leaves; furthermore, the x_i 's are now complete nodes, and thus we have ‘generated’ some more regular leaves.

The only detail left uncovered is how the new peer u finds the f distinct regular leaves that it needs to insert itself as an incomplete node. The new peer maintains three sets of nodes, Visited, Useful_Leaves and Incomplete, initially empty, and a set of stripes Needed := $\{1, \dots, f\}$. Then u begins a visit of the multicast forest, starting at a random node³ v .

When a node is visited, it replies to u with its entire state information, including its complete/incomplete status, all its f children, and a summary of the state of each of its f parents (the summary doesn’t contain information about the grandparents). Upon receiving this data, u first adds v to the set Visited, and then:

1. if v is an incomplete node, then u also adds v to Incomplete;
2. otherwise, if v is a regular leaf in a stripe $i \in$ Needed, then u also adds v to Useful_Leaves and removes i from Needed;
3. otherwise (i.e., v is a lonely leaf in all stripes $i \in$ Needed), do nothing.

Then u contacts one of the nodes listed in v 's state that hasn’t been visited yet and repeat the above process, until it discovers: either f distinct incomplete nodes (in which case u will be able to join after the incomplete nodes run the consolidation algorithm); or f distinct nodes v_1, \dots, v_f such that v_i is a regular leaf in stripe i , for $i = 1, \dots, f$ (in which case u joins as shown in Figure 2).

As a final remark, we notice here that the above algorithm would still work if the contact node is selected ad hoc (e.g., according to some metric such as round-trip latency), 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 unbalanced. This is important because the deletion mechanism (discussed in the next subsection) 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.

3.2 The Deletion Algorithm

In P2PCAST, nodes may leave either because they fail, or because they are no longer interested in the data. In either case, we refer to the event of a node leaving the system as “deletion”.

The case when the source fails is not interesting, as then there’s no content to be streamed. Special nodes are mere place-holders; thus, the failing node must be either complete or incomplete.

Adjusting the forest of multicast trees when the node u being deleted is an incomplete node is quite straightforward (see Figure 4). It’s enough to ‘reverse’ the actions of an insertion: the node u is by-passed in all the stripes, and we’re done.

²We are assuming here that a total order over nodes (based on their IP address or Chord-like *nodeId*) as been specified as a system-wide parameter.

³This could be achieved, for example, using an underlying routing layer, such as Chord or Pastry.

When the failed node u is a complete node, things are a little bit more complicated. Removing u from the picture leaves a “hole” in each stripe of the multicast forest. The empty spots in the $f - 1$ stripes where u was a leaf are of little harm; the biggest problem is to accommodate the f nodes that u orphaned in its proper stripe i . To solve this problem, we proceed as follows: first, place an imaginary “bubble” in the position in stripe i left empty by the failed node. We want to “push” this bubble as low in the stripe as possible. To this aim, we repeatedly swap the bubble with one of its complete children, until all f of the bubble’s children are either incomplete nodes or regular leaves.

If at least one of the children is an incomplete node (see Figure 5, where the incomplete child is x_1), then such child takes the place of the bubble in stripe i (thus becoming a complete node for stripe i), and it is swapped with u in all the other stripes. After these changes, the failed node u has become an incomplete node, and hence it can easily be deleted (see Figure 4 again).

Otherwise, if u was a lonely leaf in at least one stripe i.e., one of u 's parent in a stripe $j \neq i$ is an incomplete node, then such parent takes the place of the bubble in stripe i , and it is swapped with u in all the other stripes, and again we are back to the easy case of deleting an incomplete node.

Otherwise, all f children of the bubble are regular leaves i.e., they are all complete nodes, and all parents of u are complete nodes (see Figure 6, where u 's proper stripe is assumed to be $i = 1$). We would like to swap u 's parent on stripe j (call it $p_{j,1}$) with the bubble’s j^{th} child, for $j = 1, \dots, f, j \neq i$. This is only possible if $p_{j,1}$ is a regular leaf on stripe i —but again, if this is not the case, then we found an incomplete node that we can swap with u in all stripes and then easily delete. Continuing in this fashion (see Figure 6), eventually we will be able to reconstruct exactly the situation that arises *after* a consolidation (see Figure 3). At this point, we can apply the steps of consolidation in exact reverse order—call this operation *expansion*. After the expansion, we generated plenty of incomplete nodes, that we can use to terminate the process.

3.3 Implementation

We have built a prototype implementation of the algorithm for the case $f = 2$, and preliminary results are encouraging. In this section, we briefly present the architecture of our system. Refer to [3] for more details on the P2PCAST project.

The content provider uses a module called “encode” to divide the data to be sent into $f - 1$ stripes and uses a simple xor-based forward 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.

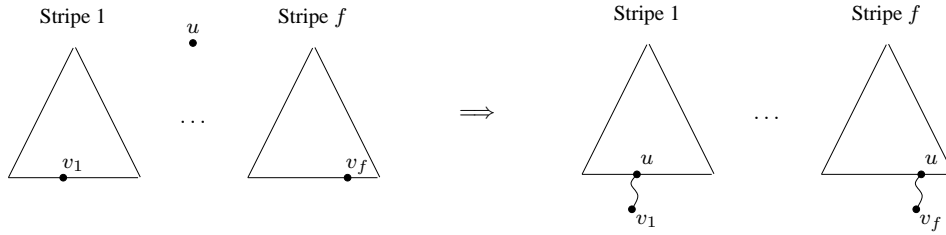


Figure 2. Inserting a new node u

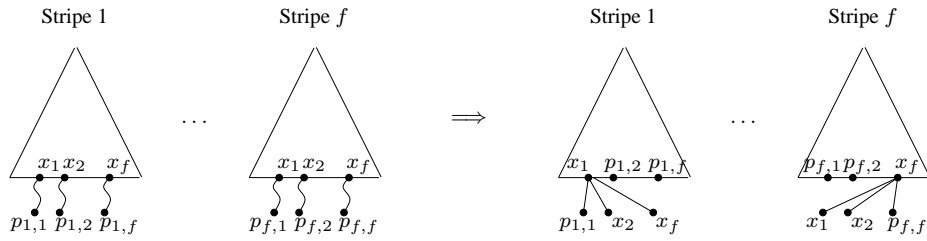


Figure 3. Consolidating f incomplete nodes x_1, \dots, x_f

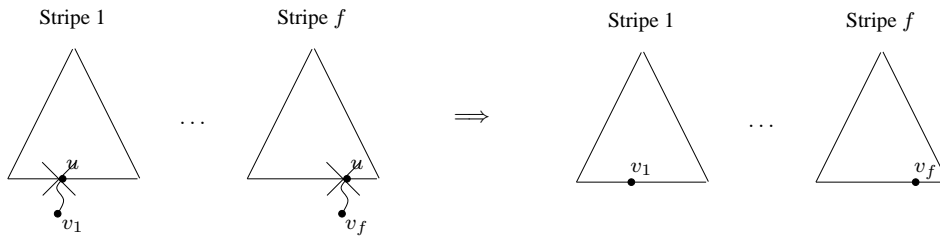


Figure 4. Deleting an incomplete node u

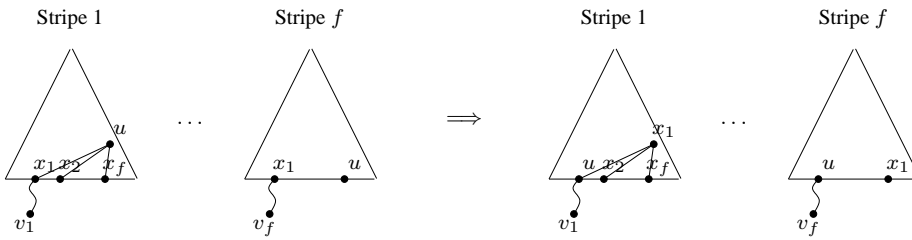


Figure 5. Swapping the failed node u with an incomplete node x_1

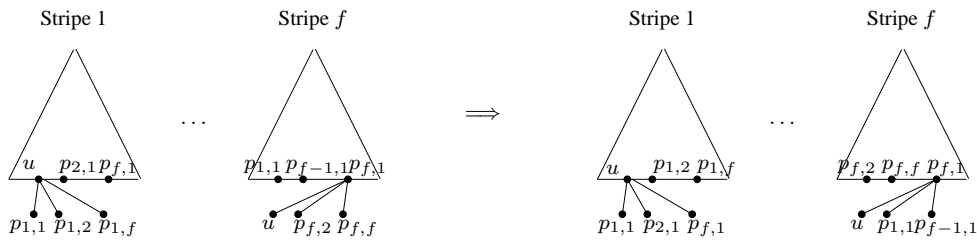


Figure 6. Preparing $u, p_{1,1}, \dots, p_{f-1,1}$ for expansion

4 Conclusions

We have established goals for multicasting streaming data in a P2P setting. We then described a novel algorithm which meets these requirements and briefly sketched the architecture of our prototype implementation.

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 mount a Sybil attack [2] on the system: in fact, such attacks could be preventable in the multicast setting, under the realistic assumption that the attacker only has a small fraction of the bandwidth of the entire system.

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.

5 Acknowledgements

We would like to thank David Mazières for suggesting the problem and for his useful libraries which greatly simplified the implementation. We would also like to thank the anonymous referees of the IRIS Student Workshop '03 for their detailed comments on improving the presentation.

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] J. Douceur. The Sybil attack. In *IPTPS '02*, Cambridge, MA, USA, 2002.
- [3] A. Nicolosi and S. Annapureddy. P2PCAST: A peer-to-peer multicast scheme for streaming data. Project Report. Available at: www.cs.nyu.edu/~nicolosi/P2PCast-PR.ps, 2003.
- [4] S. Saroiu, P. Gummadi, and S. Gribble. A measurement study of peer-to-peer file sharing systems. In *Proc. of Multimedia Computing and Networking 2002 (MMCN '02)*, San Jose, CA, USA, 2002.