# SplitStream: High-bandwidth content distribution in cooperative environments*

Miguel Castro[1] Peter Druschel[2] Anne-Marie Kermarrec[1] Animesh Nandi[2]
Antony Rowstron[1] Atul Singh[2]

[1]Microsoft Research, 7 J J Thomson Avenue, Cambridge, CB3 0FB, UK.
[2]Rice University, 6100 Main Street, MS-132, Houston, TX 77005, USA.

## Abstract

In tree-based multicast systems, a relatively small number of interior nodes carry the load of forwarding multicast messages. This works well when the interior nodes are dedicated infrastructure routers. But it poses a problem in cooperative application-level multicast, where participants expect to contribute resources proportional to the benefit they derive from using the system. Moreover, many participants may not have the network capacity and availability required of an interior node in high-bandwidth multicast applications. SplitStream is a high-bandwidth content distribution system based on application-level multicast. It distributes the forwarding load among all the participants, and is able to accommodate participating nodes with different bandwidth capacities. We sketch the design of SplitStream and present some preliminary performance results.

## 1 Introduction

End-system or application-level multicast [2, 11, 21, 6, 18, 14, 1] has become an attractive alternative to IP multicast. Instead of relying on a multicast infrastructure in the network, which is not widely available, the participating hosts pool their resources to route and distribute multicast messages using only unicast network services. In this paper we are particularly concerned with application-level multicast in *cooperative* environments. In such environments the participants contribute resources in exchange for using the service and they expect that the forwarding load be shared among all participants.

Unfortunately, conventional tree-based multicast is inherently not well matched to a cooperative environment. The reason is that in any efficient (i.e. low-depth) multicast tree a small number of interior nodes carry the burden of splitting and forwarding multicast traffic, whilst a large number of leaf nodes contribute no resources. This conflicts with the expectation that all members should share the forwarding load. The problem is further aggravated in high-bandwidth applications like video or bulk file distribution, where many nodes may not have the capacity and availability required of an interior node in a conventional multicast tree. SplitStream is designed to address these problems.

SplitStream enables efficient cooperative distribution of high-bandwidth content, whilst distributing the forwarding load among the participating nodes. SplitStream can also accommodate nodes with different network capacities and asymmetric bandwidth on the inbound and outbound network paths. Subject to these constraints, it balances the forwarding load across all the nodes.

The key idea is to *split* the multicast content into $k$ stripes, and multicast each stripe in a separate multicast tree. Participants join as many trees as there are stripes they wish to receive. The aim is to construct this *forest* of multicast trees such that an interior node in one tree is a leaf node in all the remaining trees. In this way, the forwarding load can be spread across all participating nodes. We show that it is possible, for instance, to efficiently construct a forest in which the inbound and outbound bandwidth requirements of each node are the same, while maintaining low delay and link stress across the system.

The SplitStream approach also offers improved robustness to node failure and sudden node departures. Since ideally, any given node is an interior node in only one tree, its failure can cause the temporary loss of at most one of the stripes. With appropriate data encodings such as erasure coding [3] of bulk data or multiple description coding (MDC) [13, 15] of streaming media, applications can thus mask or mitigate the effects of node failures even while the affected tree is being repaired.

SplitStream assumes that the available network bandwidth among nodes is typically limited by the hop connecting the nodes to the wide-area network (WAN), rather than the WAN backbone. This scenario is increasingly common as private and business subscribers move to dedicated Internet connections with DSL-level or better bandwidth, and the capacity of the Internet and corporate Intranet backbones is rapidly increasing.

The key challenge in the design of SplitStream is to efficiently construct a forest of multicast trees that distributes the forwarding load, subject to the bandwidth constraints of the participating nodes in a decentralized, scalable, and

*This research was supported in part by Texas ATP (003604-0079-2001) and by NSF (ANI-0225660), `http://project-iris.net`.

self-organizing manner. SplitStream relies on a structured peer-to-peer overlay network called Pastry [19], and on Scribe [6], an application-level multicast system built upon this overlay to construct and maintain these trees.

The rest of this paper is organized as follows. Section 2 outlines the SplitStream approach in more detail. A brief description of Pastry and Scribe is given in Section 3. We sketch the design of SplitStream in Section 4. Section 5 describes related work and Section 6 concludes.

# 2 The SplitStream approach

In this section, we give a more detailed overview of SplitStream's approach to cooperative, high-bandwidth content distribution.

**Tree-based multicast** In all multicast systems based on a single tree, participating nodes are either interior nodes or leaf nodes. The interior nodes carry all the burden of forwarding multicast messages. In a $k$-level balanced tree with arity $f$, the number of interior nodes is $\frac{f^{k+1}-1}{f-1}$ and the number of leaf nodes is $f^k$. Thus, the fraction of leaf nodes increases with $f$. For example, more than half of the nodes are leaves in a binary tree, and over 90% of nodes are leaves in a tree with arity 16. In the latter case, the forwarding load is carried by less than 10% of the nodes; whilst all nodes have equal inbound bandwidth, the internal nodes have an outbound bandwidth requirement of 16 times the inbound bandwidth. Even in a binary tree, which would be impractically deep in most circumstances, the outbound bandwidth required by the interior nodes is twice that of their inbound bandwidth.

**SplitStream** SplitStream is designed to overcome the inherently unbalanced forwarding load in conventional tree-based multicast systems. SplitStream strives to distribute the forwarding load over all participating nodes, and respects different capacity limits of individual participating nodes. SplitStream achieves this by splitting the multicast content into multiple stripes, and using separate multicast trees to distribute each stripe.

Figure 1 illustrates how SplitStream balances the forwarding load among the participating nodes. In this simple example, the original content is split into two stripes and multicast in separate trees. For simplicity, let us assume that the original content has a bandwidth requirement of $B$, and that each stripe has half the bandwidth requirement of the original content. Each node other than the source subscribes to both stripes, inducing an inbound bandwidth requirement of $B$. As shown in Figure 1, each node is an internal node in only one tree and forwards the stripe to two children, yielding an outbound bandwidth requirement of no more than $B$.

In general, the content is split into $k$ stripes. Participating nodes may subscribe to a subset of the stripes, thus controlling their inbound bandwidth requirement in increments of $B/k$. Similarly, participating nodes may control their outbound bandwidth requirement in increments of $B/k$ by limiting the number of children they adopt. Thus, SplitStream can accommodate nodes with different bandwidths, and nodes with unequal inbound and outbound network capacities.

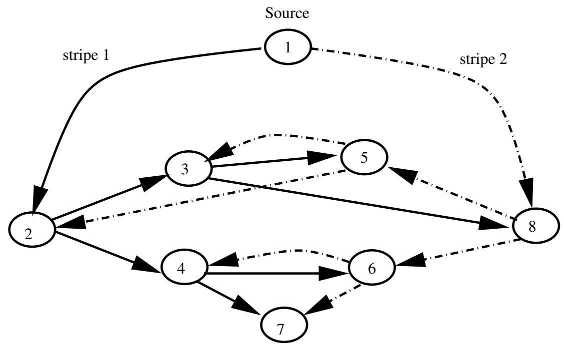


Figure 1: A simple example illustrating the basic approach of SplitStream. Original content is split into two stripes. An independent multicast tree is constructed for each stripe such that a node is an interior node in one multicast tree and a leaf in the other.

**Applications** SplitStream provides a generic infrastructure for high-bandwidth content distribution. Any application that uses SplitStream controls how the content it distributes is encoded and divided into stripes. SplitStream constructs the multicast trees for the stripes while adhering to the inbound and outbound bandwidth constraints of the nodes. Applications need to *(i)* encode the content such that each stripe requires approximately the same bandwidth; *(ii)* ensure that each stripe contains approximately the same amount of information and there is no hierarchy among stripes; and *(iii)* provide mechanisms to tolerate the intermittent loss of a subset of the stripes.

In order to tolerate the intermittent loss of a subset of stripes, some applications may provide explicit mechanisms to fetch content from other peers in the system, or applications may choose to use redundancy in encoding content, requiring more than $B/k$ per stripe in return for the ability to reconstitute the content from less than $k$ stripes. For example, a media stream could be encoded using MDC so that the video can be reconstituted from any subset of the $k$ stripes, with video quality proportional to the number of stripes received. If an interior node in the multicast tree for the stripe should fail, then clients deprived of the stripe are able to continue displaying the media stream at reduced quality until the multicast tree is repaired. Such an encoding also allows low-bandwidth clients to receive the video at lower quality by explicitly requesting fewer stripes.

Another example is the multicasting of file data, where each data block can be encoded using erasure codes to generate $k$ blocks, such that only a subset of the $k$ blocks are required to reconstitute the original block. Each stripe is then used to multicast a different one of the $k$ blocks. Participants subscribe to all stripes and once a sufficient subset of the blocks is received, the clients are able to reconstitute the original data block. If a client misses a number of blocks from a particular stripe for a period of time (while the stripe multicast tree is being repaired after an internal node has failed), the client can still reconstitute the original data blocks due to the redundancy. An example where multicasting of file data could be useful is the distribution of software patches and upgrades to institutions or end-users.

In general, while the contributed nodes could be the computers belonging to individual Internet subscribers or the desktop machines in a corporation, they could also be dedicated servers. For example, in Enterprise Content Delivery Networks (eCDNs), dedicated servers are placed through out a corporate network to facilitate access to company data and streaming media. Such eCDNs could utilize SplitStream to distribute content to the servers.

# 3 Background: Pastry and Scribe

In this section, we briefly sketch Pastry, a scalable, self-organizing, structured p2p overlay network, and Scribe, a scalable application-level multicast system based on Pastry. Both systems are key building blocks in the design of SplitStream.

**Pastry** In Pastry, nodes and objects are assigned random identifiers (called *nodeIds* and *keys*, respectively) from a large sparse id space. Keys and nodeIds are 128 bits in length and can be thought of as a sequence of digits in base $2^b$ (b is a configuration parameter with a typical value of 3 or 4). Given a message and a key, Pastry routes the message to the node with the nodeId that is numerically closest to the key, which is called the key's *root*.

In order to route messages, each node maintains a routing table and a leaf set. A node's routing table has about $log_{2^b}N$ rows and $2^b$ columns. The entries in row $n$ of the routing table refer to nodes whose nodeIds share the first $n$ digits with the local node's nodeId; the $(n+1)$th nodeId digit of a node in column $m$ of row $n$ equals $m$. The column in row $n$ corresponding to the value of the $(n+1)$th digits of the local node's nodeId remains empty. Routing in Pastry requires that at each routing step, a node normally forwards the message to a node whose nodeId shares with the key a prefix that is at least one digit longer than the prefix that the key shares with the present node's id. If no such node is known, the message is forwarded to a node whose nodeId shares a prefix with the key as long as the current node, but is numerically closer to the key than the present node's id.

Each Pastry node maintains a leaf set of neighboring nodes in the nodeId space, both to ensure reliable message delivery, and to store replicas of objects for fault tolerance.

The expected number of routing hops is less than $log_{2^b}N$. The Pastry overlay construction observes proximity in the underlying Internet. Each routing table entry is chosen to refer to a node with low network delay, among all nodes with an appropriate nodeId prefix. As a result, one can show that Pastry routes have a *low delay penalty*: the average delay of Pastry messages is usually less than twice the IP delay between source and destination [5]. Similarly, one can show the *local route convergence* of Pastry routes: the routes of messages route to the same key from nearby nodes tend to converge at a nearby intermediate node. Both of these properties are important for the construction of efficient multicast trees, described below. A full description of Pastry can be found in [19].

**Scribe** Scribe is an application-level multicast system built upon Pastry. A pseudo-random Pastry key, known as the *groupId*, is chosen for each multicast group. A multicast tree associated with the group is formed by the union of the Pastry routes from each group member to the groupId's root (which is also the root of the multicast tree). Messages are multicast from the root to the members using reverse path forwarding [9].

The properties of the Pastry overlay ensure that the multicast trees are efficient. The delay to forward a message from the root to each group member is low due to the low delay penalty of Pastry routes. Pastry's local route convergence ensures that the load imposed on the physical network is small because most message replication occurs at intermediate nodes that are close in the network to the leaf nodes in the tree.

Group membership management in Scribe is decentralized and highly efficient, because it leverages the existing, proximity-aware Pastry overlay. Adding a member to a group merely involves routing towards the groupId until the message reaches a member of the tree, followed by adding the route traversed by the message to the group multicast tree. As a result, Scribe can efficiently support large numbers of groups, arbitrary numbers of group members, and groups with highly dynamic membership.

The latter property, combined with an anycast [7] primitive recently added to Scribe, can be used to perform distributed resource discovery. As we will show in the next section, SplitStream uses this mechanism to discover nodes with spare forwarding capacity. A full description and evaluation of Scribe multicast can be found in [6]. Scribe anycast is described in [7].

# 4 SplitStream design

In this section, we sketch the design of SplitStream.

**Building interior-node-disjoint trees** SplitStream uses a separate Scribe multicast tree for each of the $k$ stripes. SplitStream exploits the properties of Pastry routing to construct trees with disjoint sets of interior nodes (called *interior-node-disjoint* trees). Recall that Pastry normally forwards a message towards nodes whose nodeIds share progressively longer prefixes with the message's key. Since a Scribe tree is formed by the routes from all members to the groupId, the nodeIds of *all* interior nodes have a common prefix of at least one digit with the tree's groupId. Therefore, we can ensure that $k$ Scribe trees have a disjoint set of interior nodes simply by choosing groupIds for the trees that all differ in the most significant digit.

Setting $k = 2^b$ ensures that each participating node has an equal chance of becoming an interior node in some tree. If $k$ is chosen such that $k = 2^i$ and $i \leq b$, then it is still possible to ensure this fairness by exploiting certain properties of the Pastry routing table, but we omit the details to conserve space. Without loss of generality, we assume that $k = 2^b$ in the rest of this paper.

**Limiting node degree** The resulting forest of Scribe trees is interior-node-disjoint and satisfies the nodes' constraints on the inbound bandwidth, but it does not necessarily satisfy the individual nodes' outgoing bandwidth constraints. Let us first consider the inbound bandwidth. A node's inbound bandwidth is proportional to the number of stripes to which the node subscribes. Note that every node has to subscribe to at least one stripe, the stripe whose stripeId shares a prefix with its nodeId, because the node may have to serve as an interior node for that stripe.

The number of children that may attempt to attach to a node is bounded by its indegree in the Pastry overlay, which is influenced by the physical network topology. In general, this number may exceed the number of children a node is able to support. For a SplitStream node to limit its outbound network bandwidth, it must limit its outdegree in the SplitStream forest, i.e., the total number of children it takes on.

Scribe has a built-in mechanism to limit a node's outdegree. When a node that has reached its maximal outdegree receives a request from a prospective child, it provides the prospective child with a list of its current children. The prospective child then seeks to be adopted by the child with lowest delay. This procedure continues recursively down the tree until a node is found that can take another child. In Scribe, this procedure is guaranteed to terminate because a leaf node is required to take on at least one child.

However, this procedure is not guaranteed to work in SplitStream. The reason is that a leaf node in one tree may be an interior node in another stripe tree, and may have already reached its outdegree limit with respect to that stripe tree. Next, we describe how SplitStream resolves this problem.

**Locating parents** The following algorithm is used to resolve the case where a node that has reached its outdegree limit receives a join request from a prospective child. First, the node adopts the prospective child regardless of the outdegree limit. Then, it evaluates its new set of children to select a child to reject. This selection is made in an attempt to maximize path independence and to minimize delay and link stress in the SplitStream forest.

First, the node looks for children that are subscribed to stripes whose stripeIds do not share a prefix with the local node's nodeId. (How the node could have acquired such a child in the first place will become clear in a moment). If the prospective child is among them, it is selected; else, one is chosen randomly from the set. If no such child exists, then the current node is an interior node for only one stripe tree, and it selects the child whose nodeId has the shortest prefix match with that stripeId. If multiple such nodes exist and the prospective child is among them, it is selected; else, one is chosen randomly from the set. The chosen child is then notified that it has been orphaned for a particular stripeId.

The orphaned child then seeks to locate a new parent in up to three steps. In the first step, the orphaned child attempts to attach to a former sibling that shares a prefix match with the stripeId for which it seeks a parent. The former sibling either adopts or rejects the orphan, using the same criteria as described above. This process continues recursively down the tree until the orphan either finds a new parent or no children share a prefix match with the stripeId.

**Spare capacity group** If the orphan has not found a parent, it sends an anycast message to a special Scribe group called the *spare capacity group*. All SplitStream nodes whose total number of stripe children is below their forwarding capacity limit are members of this group. Scribe delivers this anycast message to a node in the spare capacity group tree that is near the orphan in the physical network. This node forwards the message to a child, starting a depth-first search (DFS) of the spare capacity group tree. If the node has no children or they have all been checked, the node checks whether it receives the stripe to which the orphaned child seeks to subscribe. If so, it verifies that the orphan is not an ancestor in the corresponding stripe tree, which would create a cycle. To enable this test, each node maintains its path to the root of each stripe tree of which it is a member.

If both tests succeed, then the node takes on the orphan as a child; if as a result, the node has now reached its outdegree limit, it leaves the spare capacity group. If one of the tests fails, the node forwards the message to its parent, continuing the DFS of the spare capacity group tree until an appropriate member is found.

Anycasting to the spare capacity group may fail to locate an appropriate parent for the orphan, even after an appropriate number of retries with sufficient timeouts. There are two circumstances in which this can happen. If the spare capacity group is empty, then the SplitStream forest construction is infeasible, since an orphan remains after all forwarding capacity has been exhausted. In this case, the application on the orphaned node is notified that there is no forwarding capacity left in the system.

**Deadlocks** Otherwise, each member of the spare capacity group either does not provide the desired stripe, or it is a successor of the orphan in the stripe tree. If follows that none of the nodes in the desired stripe tree has unused forwarding capacity, although forwarding capacity exists in other stripes. This is a type of deadlock and can be resolved as follows. The orphan sends an anycast message to the desired stripe tree, which performs a randomized search of the stripe tree until it reaches a leaf node. The forwarding capacity of this leaf node must either be zero, or it must be consumed by children in different stripes (else, it would have been a member in the spare capacity group). In the former case, we ask the leaf's parent to drop the leaf and attach the orphan instead. Otherwise, the leaf node adopts the orphan and drops one of its current children randomly.

One can show that the above procedure is guaranteed to locate an appropriate parent for the orphan if one exists. Moreover, the properties of Scribe trees and the DFS of the spare capacity tree ensure that the parent is near the orphan in the physical network, among all prospective parents. This provides low delay and low link stress in the physical network. However, the algorithm as described may sacrifice interior-node-disjointedness, because the new parent may be already an interior node in another stripe tree. Thus, should the node fail, it may cause the temporary loss of more than one stripe for some nodes. Simulation results show that only a small number of nodes and stripes are typically affected.

**Maintaining path independence** It is possible to minimize this partial loss of path independence at the expense of higher delay, link stress, and cost of the forest construction. Note that completely path independent forest construction may be impractically expensive if the problem is highly constrained. However, one can bias the construction towards path independence at moderate cost.

One approach to preserving path independence is to add a third test during the DFS in the spare capacity group tree, which verifies that the prospective parent is not a predecessor to the orphan in any of the stripes to which the orphan subscribes. This ensures path independence, but may require a more extensive exploration of the spare capacity group tree, may yield a parent that is more distant in the physical network, and may not always locate a parent in the absence of sufficient excess forwarding capacity. One may balance these concerns by limiting the scope of the DFS, and relax the third test if no parent was found within that scope.

SplitStream can allow applications to control this tradeoff between independence, delay, link stress, total required forwarding capacity and overhead of forest construction according to its needs. A full evaluation of heuristics to maximize path independence is the subject of ongoing work.

**Preliminary results** We have performed a preliminary performance evaluation of SplitStream, by running 40,000 SplitStream nodes over an emulated network with 5050 core routers based on the Georgia Tech network topology generator. We constructed a SplitStream forest with 16 stripes, and assigned per-node inbound and outbound bandwidth limits that follow a distribution measured among Gnutella clients in May 2001 [20].

The result are very encouraging. During the SplitStream forest construction, the mean and median number of control messages handled by each node were 56 and 47, respectively. When multicasting a message in each stripe, the medians of the relative average delay penalty (RAD) and the relative maximum delay penalty (RMD), compared to IP multicast, where 2.17 and 2.88, respectively. These value are about 1.35 and 1.8 times higher, respectively, than the values measured in a single Scribe tree on the same topology. This increase reflects the principal cost of balancing the forwarding load across all participants in SplitStream.

We also considered the degree of independence in the SplitStream forest. Without any of the independence-preserving techniques described above, and with a highly constrained bandwidth allocation (outbound bandwidth not to exceed inbound bandwidth at any node), we found that over 95% of the nodes had independent (i.e., node disjoint) paths to the source in 12 or more of the 16 stripes to which they subscribed. Thus, even in pessimal cases, the loss of independence is modest. A more comprehensive evaluation of SplitStream will be presented in a forthcoming full paper.

# 5 Related work

Many application-level multicast systems have been proposed recently, e.g. [8, 14, 18, 21, 6, 1]. All are based on a single multicast tree.

Several systems use application-level multicast for streaming media [14, 10, 17]. SpreadIt [10] utilizes the participants, as SplitStream does, but creates a single multicast tree. However, unlike SpreadIt, SplitStream distributes the forwarding load over all participants using multiple multicast trees, thereby reducing the bandwidth demands on individual peers and increasing robustness.

Overcast [14] organizes dedicated servers into a source-rooted multicast tree using bandwidth estimation measurements to optimize bandwidth usage across the tree. The main differences between Overcast and SplitStream are *(i)* that Overcast uses dedicated servers whilst SplitStream utilizes the participants; *(ii)* Overcast creates a single bandwidth optimized multicast tree whereas SplitStream creates a forest of multicast trees assuming that the available network bandwidth among peers is typically limited by bandwidth of the links connecting nodes to the network rather than the network backbone. This scenario is increasingly common as the capacity of the Internet and corporate Internet backbones rapidly increase.

CoopNet [17] implements a hybrid system for streaming media, which utilizes multiple application-level trees with striping and Multiple Description Encoding (MDC) [13, 15]. The idea of using MDCs and exploiting path diversity for robustness was originally proposed by Apostolopoulos [**?**, **?**] to increase robustness to packet loss when streaming media. In CoopNet a centralized server is used to stream media. Clients contact the server requesting the media stream. If the server is not overloaded, it supplies the client with the stream. If the server becomes overloaded, then it redirects clients to already participating nodes. The stream is striped and several application-level multicast trees rooted at the server are created. There are two fundamental differences between CoopNet and SplitStream: *(i)* CoopNet uses a centralized algorithm (running on the server) to build the trees whilst SplitStream is completely decentralized and more scalable; and *(ii)* CoopNet does not explicitly attempt to manage the bandwidth contribution of individual nodes; however, it is possible to add this capability to CoopNet.

Nguyen and Zakhor [16] propose streaming video from multiple sources concurrently, thereby exploiting path diversity and increasing tolerance to packet loss. They subsequently extend the work in [16] to use Forward Error Correction [3] encodings. The work assumes that the client is aware of the set of servers from which to receive the video. SplitStream constructs multiple end-system based multicast trees in a decentralized fashion and is therefore more scalable.

In [4], algorithms and content encodings are described that enable parallel downloads and increase packet loss resilience in richly connected, collaborative overlay networks by exploiting downloads from multiple peers. SplitStream provides a complete system for content distribution in collaborative overlay networks. It explicitly stripes content and creates a multicast tree for each stripe. Also, SplitStream's primary goal is to spread the forwarding load across all participants.

FCast [12] is a reliable file transfer protocol based on IP multicast. It combines a Forward Error Correction [3] encoding and a data carousel mechanism. Instead of relying on IP multicast, FCast could be easily built upon SplitStream, for example, to provide software updates cooperatively.

# 6 Conclusions

We have sketched the design of SplitStream, a high-bandwidth content distribution system based on end-system multicast in cooperative environments. Preliminary performance results are very encouraging. The system is able to distribute the forwarding load among the participating nodes, subject to individual node bandwidth limits. When combined with redundant content encoding, SplitStream yields resilience to node failures and unannounced departures, even while the affected multicast tree is repaired. The overhead of the forest construction is modest and well balanced, and the resulting increase in delay penalty and link stress is modest, when compared to a conventional tree-based application-level multicast system. A forthcoming paper will present comprehensive results, including results of experiments using the PlanetLab Internet testbed.

# References


[1] J. G. Apostolopoulos. Reliable video communication over lossy packet networks using multiple state encoding and path diversity. In *Visual Communications and Image Processing*, Jan. 2001.

[2] J. G. Apostolopoulos and S. J. Wee. Unbalanced multiple description video communication using path diversity. In *IEEE International Conference on Image Processing*, Oct. 2001.

[3] S. Banerjee, B. Bhattacharjee, and C. Kommareddy. Scalable application layer multicast. In *ACM SIGCOMM*, Aug. 2002.

[4] K. Birman, M. Hayden, O. Ozkasap, Z. Xiao, M. Budiu, and Y. Minsky. Bimodal multicast. *ACM TOCS*, 17(2):41–88, May 1999.

[5] R. Blahut. *Theory and Practice of Error Control Codes*. Addison Wesley, MA, 1994.

[6] J. Byers, J. Considine, M. Mitzenmacher, and S. Rost. Informed content delivery across adaptive overlay networks. In *SIGCOMM'2002*, Pittsburgh, PA, USA, Aug. 2002.

[7] M. Castro, P. Druschel, Y. C. Hu, and A. Rowstron. Exploiting network proximity in peer-to-peer overlay networks, 2002. Technical report MSR-TR-2002-82.

[8] M. Castro, P. Druschel, A.-M. Kermarrec, and A. Rowstron. SCRIBE: A large-scale and decentralized application-level multicast infrastructure. *IEEE JSAC*, 20(8), Oct. 2002.

[9] M. Castro, P. Druschel, A.-M. Kermarrec, and A. Rowstron. Scalable peer-to-peer anycast for distributed resource management, 2003. Submitted.

[10] Y. Chu, S. Rao, and H. Zhang. A case for end system multicast. In *ACM Sigmetrics*, pages 1–12, June 2000.

[11] Y. K. Dalal and R. Metcalfe. Reverse path forwarding of broadcast packets. *CACM*, 21(12):1040–1048, 1978.

[12] H. Deshpande, M. Bawa, and H. Garcia-Molina. Streaming live media over a peer-to-peer network, Apr. 2001. Stanford University, CA, USA.

[13] P. Eugster, S. Handurukande, R. Guerraoui, A.-M. Kermarrec, and P. Kouznetsov. Lightweight probabilistic broadcast. In *DSN*, July 2001.

[14] J. Gemmell, E. Schooler, and J. Gray. Fcast multicast file distribution. *IEEE Network*, 14(1):58–68, Jan 2000.

[15] V. K. Goyal. Multiple description coding: Compression meet the network. *IEEE Signal Processing Magazine*, 18(5):74–93, Sept. 2001.

[16] J. Jannotti, D. Gifford, K. Johnson, M. Kaashoek, and J. O'Toole. Overcast: Reliable multicasting with an overlay network. In *OSDI 2000*, San Diego, CA, 2000.

[17] A. Mohr, E. Riskin, and R. Ladner. Unequal loss protection: Graceful degredation of image quality over packet erasure channels through forward error correction. *IEEE JSAC*, 18(6):819–828, June 2000.

[18] T. Nguyen and A. Zakhor. Distributed video streaming with forward error correction. In *Packet Video Workshop*, Pittsburgh, USA., 2002.

[19] V. Padmanabhan, H. Wang, P. Chou, and K. Sripanidkulchai. Distributing streaming media content using cooperative networking. In *NOSSDAV*, Miami Beach, FL, USA, May 2002.

[20] S. Ratnasamy, M. Handley, R. Karp, and S. Shenker. Application-level multicast using content-addressable networks. In *NGC*, Nov. 2001.

[21] A. Rowstron and P. Druschel. Pastry: Scalable, distributed object location and routing for large-scale peer-to-peer systems. In *IFIP/ACM Middleware 2001*, Heidelberg, Germany, Nov. 2001.

[22] S. Saroiu, P. K. Gummadi, and S. D. Gribble. A measurement study of peer-to-peer file sharing systems. In *MMCN*, San Jose, CA, Jan. 2002.

[23] S. Zhuang, B. Zhao, A. Joseph, R. Katz, and J. Kubiatowicz. Bayeux: An architecture for scalable and fault-tolerant wide-area data dissemination. In *NOSSDAV*, June 2001.