Secure Acknowledgment of Multicast Messages in Open Peer-to-Peer Networks

Antonio Nicolosi and David Mazières
NYU Department of Computer Science
{nicolosi,dm}@cs.nyu.edu

Abstract

We propose a new cryptographic technique, *Acknowledgment Compression*, permitting senders of multicast data to verify that all interested parties have either received the data or lost network connectivity. Joining the system and acknowledging messages both require bandwidth and computation logarithmic in the size of a multicast group. Thus, the technique is well-suited to large-scale, peer-to-peer multicast groups in which neither the source nor any single peer wishes to download a complete list of participants. In the event that a high percentage of nodes are malicious, a message may fail to verify. However, in such cases the source learns the real network address of a number of malicious nodes.

1 Introduction

Peer-to-peer (P2P) multicast systems are a promising technology for inexpensively distributing information to large numbers of nodes. Traditional, centralized distribution schemes consume prohibitive amounts bandwidth, transmiting data repeatedly from a handful of sources. In contrast, P2P schemes can exploit the high aggregate bandwidth of all peers to disseminate data without requiring any particularly high-bandwidth nodes. Indeed, several P2P multicast systems have already been built, based on trees [4], forests [5], meshes [8], and other topologies.

Unfortunately, current P2P multicast schemes provide no way to confirm the reliable delivery of information. Yet for many applications, a data source might like to know if all interested parties have received messages. For example, a publisher might wish to confirm the receipt of a multicast invalidation for widely-cached popular data before considering a modification operation complete. Alternatively, the maintainers of an operating system distribution might like to know that everyone who wants security-critical updates has received the latest one.

In the P2P setting, multicast message acknowledgment is complicated by the open nature of the systems. Not only might many of the nodes in a multicast group be malicious, but the source of a message has no way of even knowing precisely who wants to receive the data. In large-scale P2P systems, even just transmitting a complete list of all nodes in a multicast group might consume undesirable amounts of bandwidth. Fortunately, it is reasonable to assume malicious nodes cannot subvert the routing structure of the Internet itself. Thus, particularly given the number and diversity of participants that P2P systems are designed for, a good node is likely to be able to communicate with almost any other honest node given that node's network address.

In this paper, we propose a new cryptographic mechanism, *Acknowledgment Compression*, that enables confirmation of messages delivered to large numbers of participants in an open network. Using this new primitive, a node that joins a multicast group receives a signed receipt from the data source acknowledging its entry into the system. For efficiency, arbitrarily many such join operations can be aggregated; no matter how many people join, the maximum computation and bandwidth required at any node is logarithmic in the total size of the multicast group. In particular, a join receipt does not explicitly name each new node, as no single node may even wish to consume the bandwidth necessary to download such a list.

Once a node has joined the system, it must acknowledge any messages sent to the multicast group. When a node fails to produce an acknowledgment, the multicast source can likely uncover its network address and contact that node directly, or else delegate the task—for instance to a probabilistically chosen subset of another part of a multicast tree. In an unlucky case, it may be impossible to recover the address of an unreachable participant, but then the multicast source will learn the network address of malicious or faulty nodes who are blocking proper operation of the system.

2 Our Scheme

In this section, we define a model for Acknowledgement Compression, and present an efficient instantiation based on the cryptographic properties of special algebraic groups. Our scheme provides cryptographic proof of message deliver at the cost of consuming a small (logarithmic in the number of users in the system) amount of storage, computing time, and upstream/downstream bandwidth at each node.

2.1 The Model

An Acknowledgment Compression scheme comprises four protocols—Init, Join, Leave and Collect—with the following operational semantics. The content provider (or *source*) uses the Init protocol to carry out an initialization phase, at the end of which a public key for a secure digital signature scheme is produced and made widely available as a public parameter of the just-created multicast group. After the initial setup, the system grows by allowing batches of multiple user additions at discrete time instants, handled by the Join protocol. Normally, users would leave gracefully by invoking the Leave protocol, but some robustness mechanism is provided to recover from unanticipated node departures (or *crashes*). The fundamental operation of our scheme is the Collect protocol, which allows the source to specify a payload message m (e.g., an acknowledgment for the last multicast message) and to obtain a compact proof that every unfailed user in the group saw and endorsed the message m. As a side-effect, Collect also purges the system of failed users.

2.2 A Cryptographic Construction

This Section presents a specific construction within the framework of Section 2.1, based on cryptographic groups known as *Gap Diffie-Hellman* (GDH) groups. We address the related cryptographic literature in Section 4. Briefly, a GDH group $G = \langle g \rangle$ is an algebraic group of prime order q for which no efficient algorithm can compute g^{ab} for random $g^a, g^b \in G$, but such that there exists an efficient algorithm $D(g^a, g^b, h)$ to decide whether $h = g^{ab}$.

Digital signatures can be obtained from such groups as follows. A user secret key is a random value $x \in \mathbb{Z}_q$; the corresponding public key is $y \leftarrow g^x$. The signature on a message m is computed as $\sigma \leftarrow H(m)^x$, where H is some cryptographic hash function (e.g., SHA-1). The validity of a putative signature σ on a message m under the public key y is tested by checking that $D(y, H(m), \sigma) = 1$. The key property of this signature scheme that we will exploit in our construction is that the product of two signatures of the same message m under two different public keys y_1, y_2 yields a signature of m under the combined public key $y \leftarrow y_1 y_2 = g^{(x_1+x_2)}$, since $H(m)^{x_1} H(m)^{x_2} = H(m)^{(x_1+x_2)}$.

We now describe an Acknowledgment Compression scheme for the case that nodes don't crash abruptly; in Section 2.3 we discuss how to make the construction robust against node failures.

For the sake of clarity, we detail the case in which the communication happens via an (almost) balanced multicast tree with bounded maximum branching factor k (which most P2P multicast schemes strive to achieve anyway); however, our techniques can easily be adapted to the case of multiple multicast trees [5, 9], or other application-level multicast schemes and information dispersal algorithms.

Notation. In describing the protocols, we will make use of the following notation. 1_G denotes the identity element in G. For a node i, let y_i denote its public key, Loc_i denote its location (e.g, its network address), and T_i denote the subtree rooted at i. We will refer to the quantity:

$$Y_i \overset{def}{=} \prod_{j \in T_i} y_j$$

as the *combined public key* for the subtree T_i. Furthermore, we will denote i's children with $i_1, \ldots, i_k$, and i's parent with i^1; more generally, let i^d denote node i's d^{th} ancestor—i.e., the node sitting d levels above i in the tree. Many of the quantities associated with the system (most notably, the combined public key Y_i associated with each subtree T_i) *evolve* during the lifetime of the system as a consequence of members joining and leaving the multicast tree. To distinguish between old and new values, we will use a prime notation for old values e.g., Y_i' refers to the value of the combined public key for T_i *before* the user additions of the current execution of Join.

The Init protocol. Init starts by choosing a cryptographic hash function H and a GDH group $G = \langle g \rangle$, along with a secret key/public key pair $(x_s, y_s \leftarrow g^{x_s})$ for the source. It then initializes a variable $\mathsf{Y}_s \leftarrow y_s$, which will always hold the value of the combined public key Y_s for the whole multicast tree, T_s. This will be guaranteed by the Join and Leave protocols, which will update Y_s in such a way that the source, as well as every user of the system, can verify that the new value of Y_s is indeed the product of the public keys of all the users in the multicast tree.

The Collect protocol. Before detailing how the Join and Leave protocols each manage to fulfill their pledge, we show how, assuming that Y_s actually embodies the public keys of all users in the system, the Collect protocol can produce a compressed acknowledgment for a payload message m specified by the source.[1] First, the source prepares and signs a *collect request* message:

$$\mathit{CollectReq} \stackrel{def}{=} \{\text{“CollectReq”}, m\}$$

and sends this signature to all its children. Internal nodes forward such signature to their children; when the signed $\mathit{CollectReq}$ message gets to a leaf j, node j verifies the source's signature, signs the payload message m as $\sigma_j \leftarrow H(m)^{x_j}$, and replies to its parent with a *collect* message:

$$\mathit{CollectMsg}_j \stackrel{def}{=} \{\text{“CollectMsg”}, m, \sigma_j\}.$$

Upon receiving a reply from each of its k children $i_1, \ldots, i_k$, an internal node i combines its signature $\sigma_i \leftarrow H(m)^{x_i}$ with the multi-signatures $\Sigma_{i_1}, \ldots, \Sigma_{i_k}$ contributed by its children as:

$$\Sigma_i \leftarrow \sigma_i \cdot \prod_{l=1}^{k} \Sigma_{i_l}$$

and then it replies to its parent with the message:

$$\mathit{CollectMsg}_i \stackrel{def}{=} \{\text{“CollectMsg”}, m, \Sigma_i\}.$$

Eventually, the source will get a collect message from each of its children. After combining their signatures with its own, the source obtains what should be a multi-signature Σ_s on the message m, bearing the endorsement of each user in the system. To check this, the source tests whether Σ_s verifies correctly as a signature of m under the current combined public key Y_s, at which point the Collect operation terminates. Since, under standard cryptographic assumptions [1], no adversary can forge a multi-signature for a fresh message m, the source can infer that all the users whose public keys have been included in Y_s must have signed m. Finally, by the correctness of the Join protocol, this guarantees that each user in the system endorses the message m, as required of the Collect protocol.

The Join **protocol**. We now describe the details of the Join protocol in terms of a distributed computation carried out by the source together with all the (unfailed) nodes in the multicast tree. Figure 1 defines the format of the messages that will be exchanged during the protocol, as well as the state information maintained at each node, which requires $O(k + \log_k n)$ space for a system with n users, assuming a (roughly) balanced multicast tree with fan-out factor at most k at each node.

The protocol starts with the source executing the distributed algorithm REPORTJOINS described in Figure 2. REPORTJOINS initiates a post-order traversal of the multicast tree that gathers information about user additions since the last execution of Join.

[1]For security, the payload message m should be different across all executions of Collect; it is up to the application to provide such a guarantee, for instance by including a sequence number in each message.

Messages exchanged in the Join protocol	
$\mathit{CertReq}_i \stackrel{def}{=} \{\text{“CertReq”}, Y_i', Y_i, \mathit{Loc}_i, \mathit{Loc}_{i^1}\}$	
$\mathit{Cert}_i \stackrel{def}{=} \{\mathit{CertReq}_i\}_{(Y_i)^{-1}}$	
$C_{i,j} \stackrel{def}{=} \{\mathit{CertReq}_i\}_{(Y_j)^{-1}}$	// *j descendent of* i
$c_{i,j} \stackrel{def}{=} \{\mathit{CertReq}_i\}_{(y_j)^{-1}}$	// *j descendent of* i

State information stored at node i		
$\mathtt{Cert}_i$	::	current self-certificate for T_i
$\mathtt{Cert}_i'$	::	previous self-certificate for T_i
for each ancestor i^d*:*		
$\mathtt{y}_{i^d}$	::	public key y_{i^d} of node i^d
$\mathtt{Y}_{i^d}$	::	current value of combined public key Y_{i^d}
for each child i_l*:*		
$\mathtt{Cert}_{i_l}$	::	current self-certificate for T_{i_l}
$\mathtt{Cert}_{i_l}'$	::	previous self-certificate for T_{i_l}

Figure 1: Join protocol—Messages exchanged during the protocol and state information for node i.

The actual computation begins once the recursion has reached the leaves of the multicast tree. Pre-existing leaves simply report that they don't have any change, whereas a newcomer j finalizes its ingress into the system by returning to its parent a signed message containing its public key y_j and its location information Loc_j.

An internal node i whose children have all reported no changes reports to its parent that it has no changes. Otherwise, if at least one child i_l has reported changes (meaning that some user(s) just joined the subtree T_{i_l}), then i must update the combined public key Y_i to reflect the presence of the newcomers in the subtree T_i, as nodes in T_{i_l} belong to T_i, too. In order to report such a change to its parent, node i must construct a *self-certificate* Cert_i, cryptographically justifying the evolution of Y_i.

To obtain a self-certificate, node i has to get the signature on a *certificate request* $\mathit{CertReq}_i$ (cfr. Figure 1) from all its descendents—a self-certificate for a subtree is only valid if all nodes in that subtree endorse it. To achieve this goal, i puts its signature on $\mathit{CertReq}_i$, and invokes the SIGNCERT distributed algorithm (described in Figure 2), which recursively pushes the request down to all descendents.

Node i's descendents, however, do not blindly sign whichever message they get to see—they need to be convinced of the legitimacy of the certificate request $\mathit{CertReq}_i$. For this reason, node i attaches two self-certificates for each child i_l who has reported changes: a cached version Cert_{i_l}' from the previous execution of

```
$i$.REPORTJOINS()
 1. if $i$ is a pre-exisiting leaf return $\bot$
 2. else if $i$ is a new leaf then
 3.     $\texttt{Cert}'_i \leftarrow 1_G$
 4.     $\texttt{Cert}_i \leftarrow c_{i,i}$
 5.     return $\texttt{Cert}_i$
 6. else
 7.     for each child $i_l$ of $i$ $(l = 1, \ldots, k)$
 8.         $\texttt{Cert}'_{i_l} \leftarrow \texttt{Cert}_{i_l}$
 9.         $\texttt{Cert}_{i_l} \leftarrow i_l$.REPORTJOINS()
10.         $\mathrm{A}_{i_l} \leftarrow \langle \texttt{Cert'}_{i_l}, \texttt{Cert}_{i_l} \rangle$
11.     if $(\forall l = 1, \ldots, k)[\texttt{Cert}_{i_l} = \bot]$ return $\bot$
12.     $\mathrm{CheckCerts}(\mathrm{A}_{i_1}, \ldots, \mathrm{A}_{i_k})$
13.     $\texttt{Y}'_i \leftarrow \texttt{Y}_i$
14.     update $\texttt{Y}_i$ according to $\mathrm{A}_{i_1}, \ldots, \mathrm{A}_{i_k}$
15.     $\texttt{Cert}'_i \leftarrow \texttt{Cert}_i$
16.     for each child $i_l$ of $i$ $(l = 1, \ldots, k)$
17.         $C_{i,i_l} \leftarrow i_l$.SIGNCERT$(i, c_{i,i}, \mathrm{A}_{i_1}, \ldots, \mathrm{A}_{i_k})$
18.     $\texttt{Cert}_i \leftarrow c_{i,i} \cdot C_{i,i_1} \cdot \ldots \cdot C_{i,i_k}$
19.     return $\texttt{Cert}_i$
```

```
$j$.SIGNCERT$(i, c_{i,i}, \mathrm{A}_{i_1}, \ldots, \mathrm{A}_{i_k})$
 1. $\mathrm{CheckCerts}(\mathrm{A}_{i_1}, \ldots, \mathrm{A}_{i_k})$
 2. look up $i$ among $j$'s ancestors; let $i = j^d$
 3. verify the signature on $c_{i,i}$
 4. check that $c_{i,i}$ is consistent with $\mathrm{A}_{i_1}, \ldots, \mathrm{A}_{i_k}$
 5. update $\texttt{Y}_{j^d}$ $(= Y_i)$ according to $\mathrm{A}_{i_1}, \ldots, \mathrm{A}_{i_k}$
 6. if $i$ is a leaf then
 7.     $C_{i,j} \leftarrow c_{i,j}$
 8. else
 9.     for each child $j_l$ of $j$ $(l = 1, \ldots, k)$
10.         $C_{i,j_l} \leftarrow j_l$.SIGNCERT$(i, c_{i,i}, \mathrm{A}_{i_1}, \ldots, \mathrm{A}_{i_k})$
11.     $C_{i,j} \leftarrow c_{i,j} \cdot C_{i,j_1} \cdot \ldots \cdot C_{i,j_k}$
12. return $C_{i,j}$
```

```
$\mathrm{CheckCerts}(\mathrm{A}_{i_1}, \ldots, \mathrm{A}_{i_k})$
1. for $l = 1, \ldots, k$
2.     parse $\mathrm{A}_{i_l}$ as $\langle \texttt{Cert'}_{i_l}, \texttt{Cert}_{i_l} \rangle$
3.     if $\texttt{Cert}_{i_l} \neq \bot$ then
4.         verify the signatures on $\texttt{Cert}'_{i_l}$ and $\texttt{Cert}_{i_l}$
5.         if new PK in $\texttt{Cert}'_{i_l} \neq$ old PK in $\texttt{Cert}_{i_l}$ then
6.             throw InvalidCertsException
```

Figure 2: Join protocol—Pseudo-code for the distributed algorithms REPORTJOINS (as run by node i) and SIGNCERT (as run by node j, a descendent of i). Exception handlers for consistency check failures are not shown.

Join, and the current version $Cert_{i_l}$ that i_l just provided.

Given such accompanying documentation, each descendent j of node i can test the legitimacy of the certificate request sent by node i (cfr. the CheckCerts() procedure, Figure 2); after this check, j affixes its signature on $CertReq_i$ and forwards the execution of the SIGNCERT invocation down to its own children $j_1, \ldots, j_k$.

Upon hearing back from its children, j will obtain the partial signatures $C_{i,j_1}, \ldots, C_{i,j_k}$ relative to their subtrees $T_{j_1}, \ldots, T_{j_k}$. Then, j will piece all the parts together as:

$$C_{i,j} = c_{i,j} \cdot \prod_{l=1}^{k} C_{i,j_l}$$

and will reply to its parent with the partial signature $C_{i,j}$ on $CertReq_i$ relative to subtree T_j.

Eventually, node i will obtain partial signatures from all its children, which will enable i to compute the actual self-certificate $Cert_i$. At this point, node i will have enough information to justify the evolution of the combined public key for subtree T_i from its old value Y'_i to the new value Y_i, thus being able to finally complete its part in the REPORTJOINS by sending $Cert_i$ to its parent.

As the recursion of REPORTJOINS climbs up the tree, it will eventually reach the source. Then, as part of the SIGNCERT call invoked by the source, every node in the tree will get the source's signature on the message:

$$CertReq_s \overset{def}{=} \{\text{“CertReq”}, Y'_s, Y_s, Loc_s, Loc_s\}.$$

This, together with all the consistency checks performed during each step of the protocol, guarantees each new user that its public key has been included into the new value Y_s of the system's combined public key, and each old user that its public key has not been factored out.

The Leave **protocol**. The Leave protocol develops similarly to the Join protocol. In particular, a distributed algorithm REPORTLEAVES (akin to the distributed algorithm REPORTJOINS from Figure 2; not shown) traverses the multicast tree in post-order, collecting signed *leave messages* of the form:

$$\{\text{“LeaveMsg”}, y_j, 1_G, Loc_j, Loc_{j^1}\}_{(y_j)^{-1}}$$

for each departing user j.

Such messages are then forwarded up the tree, and in the process, the combined public key of each subtree that contained a departing node is updated, and the necessary self-certificates are constructed to provide evidence supporting the evolution of the affected combined public keys. We omit the details.

2.3 Dealing with Unanticipated Failures

During normal operation, the users of the system are often required to sign messages. Due to the cryptographic consistency checks performed by all the other users, a malicious user j cannot deviate from the protocol except by refusing to cooperate, thus preventing its ancestors from successfully completing subsequent Collects or Joins. However, a user j that crashes without executing the Leave protocol would cause its parent, node j^1, to experience a similar refusal, which creates a troublesome ambiguity.

Simply allowing node j^1 to drop its non-cooperating child, as is often done in multicast systems, would undermine the main goal of Acknowledgement Compression. Such a policy, in fact, would expose the system to a possible abuse, based on the difficulty of distinguishing the case that node j didn't want to release its signature from the case that j's parent, node j^1, pretended to have been unable to obtain a signature from j to make j look bad.

Instead, we deal with this problem by requiring j^1 to produce j's self-certificate, which contains j's network address. Thus, either j^1 can fail to cooperate and be ejected from the system, or else other nodes can obtain j's address and attempt to deliver the message to j. In this way, a list of unreachable network addresses can be compiled and forwarded up the tree. If none of the ancestors of these unreachable nodes can reach them to obtain the missing component of the acknowledgment, the source will obtain the list of network addresses.

3 Discussion

One limitation of Acknowledgment Compression is the bandwidth required to deal with crashes. As described in the previous section, the source node receives a list of network addresses of all nodes that have failed to acknowledge a message, so as either to deliver the message directly or to confirm each node's unreachability. Though the list could conceivably be partitioned between a small set of well-known, trusted nodes, if the churn is too high, it may simply require too much bandwidth for each trusted node to receive its portion of the list. Thus, the technique may be ill-suited to networks in which many nodes could exit ungracefully, unless the lifetime of a multicast group is considerably less than the half-life of nodes. For infrastructure nodes, however, such as news servers and routers, it seems reasonable to expect long uptimes and graceful exits.

There are two possible approaches to handling large numbers of crashed nodes, though the problem is future work—we do not have a complete solution along the lines of either approach. First, there may be ways of obtaining large numbers of effectively trusted nodes, so as to partition the work of downloading crashed nodes' locations and probing them. This might, for instance, be achieved through some PGP-like web-of-trust overlay. Second, verification could be probabilistically farmed out to large-enough subgroups of multicast peers that at least one of them is likely to be honest. Subgroups can confirm the results of their probes by supplying a product of missing acknowledgment components (for nodes they managed to deliver the message to), plus a compressed acknowledgment on the product of public keys of unreachable nodes.

A final, related issue is that while Acknowledgment Compression confirms reliable multicast, it is not in itself a multicast protocol. Much work exists on fault-resilient P2P infrastructures, but of course malicious nodes may still succeed in partitioning or otherwise subverting the P2P system, rendering honest nodes that are routable at the network layer unreachable in the overlay network. Acknowledgment Compression cannot prevent such attacks, but at least it exposes the network address of malicious nodes who can then potentially be excluded from the system. The reason is that any node unable to produce some necessary component of an acknowledgment signature must instead be able to show a signed $CertReq$ containing the $Location$ (e.g., network address) of the missing contact that is its child node. To avoid reporting the network address of an isolated honest node, all nodes through which the missing contact is supposed to be routable must fail to return the $CertReq$ certificate, thereby exposing themselves as malicious or unreliable.

4 Related Work

Recently, there has been much promising work on P2P multicast and data distribution, which can be combined with Acknowledgment Compression to obtain confirmed message delivery. Space limitations preclude an enumeration of P2P multicast projects.

GDH groups have been the focus of much cryptographic research [6, 7, 2]. The basic GDH signature scheme we use for a single node signature was proposed by Boneh et al. [3]. The idea of combining such signatures and public keys by multiplication was used by Boldyreva [1] to construct multi-signatures—a variation of digital signatures in which a group of users, each holding a unique signing key, produce signed documents that can later be verified to bear the endorsement of every signer in the group.

5 Summary

Acknowledgment Compression is a new cryptographic primitive useful for verifying message delivery in open, P2P multicast systems. Acknowledgment Compression allows senders to learn the network address of nodes that fail to acknowledge messages, yet without ever having to download a complete list of all participants. We propose an instantiation based on Gap Diffie-Hellman groups, and briefly discuss some usage issues for the technique.

Acknowledgments

We are grateful to Victor Shoup for his help with this work, and to Nelly Fazio, Mike Freedman and the anonymous reviewers for their comments on earlier drafts. This research was conducted as part of project IRIS (`http://project-iris.net/`), supported by the NSF under cooperative agreement #ANI-0225660. David Mazières is supported by an Alfred P. Sloan Research Fellowship.

References

[1] Alexandra Boldyreva. Efficient threshold signature, multisignature and blind signature schemes based on the gap-diffie-hellman-group signature scheme. Full length version of extended abstract in PKC'03, available at `http://eprint.iacr.org/2002/118/`, 2003.

[2] D. Boneh and M. Franklin. Identity-based encryption from the Weil pairing. In *Advances in Cryptology - Crypto '01*, volume 2139 of *LNCS*, pages 213–229. Springer-Verlag, 2001.

[3] D. Boneh, B. Lynn, and H. Shacham. Short signatures from the Weil pairing. In *Advances in Cryptology—AsiaCrypt'01*, volume 2248 of *LNCS*, pages 514–532, Berlin, 2001. Springer-Verlag.

[4] M. Castro, P. Druschel, A. Kermarrec, and A. Rowstron. SCRIBE: A large-scale and decentralized application-level multicast infrastructure. *IEEE Journal on Selected Areas in communications (JSAC)*, 20(8):1489–1499, October 2002.

[5] M. Castro, P. Druschel, A.-M. Kermarrec, A. Nandi, A. Rowstron, and A. Singh. Splitstream: High-bandwidth content distribution in a cooperative environment. In *SOSP '03*, Bolton Landing, NY, USA, 2003.

[6] A. Joux. A one round protocol for tripartite diffie-hellman. In *Proceedings of ANTS IV*, volume 1838 of *LNCS*, pages 385–394. Springer-Verlag, 2000.

[7] A. Joux and K. Nguyen. Separating decision diffie-hellman from diffie-hellman in cryptographic groups. Available at `http://eprint.iacr.org/2001/060/`, 2001.

[8] Dejan Kostić, Adolfo Rodriguez, Jeannie Albrecht, and Amin Vahdat. Bullet: High bandwidth data dissemination using an overlay mesh. In *SOSP '03*, Bolton Landing, NY, USA, 2003.

[9] A. Nicolosi and S. Annapureddy. P2PCAST: A peer-to-peer multicast scheme for streaming data. 1st IRIS Student Workshop (ISW'03). Available at: `http://www.cs.nyu.edu/~nicolosi/P2PCast.ps`, 2003.