# Accessing and Distributing Streaming Events on DHT-based Systems

M. Milanesio, G. Ruffo
Dip. di Informatica - Università di Torino
{milane, ruffo}@di.unito.it

F. Ricchiuti, D. Albertin
CSP - Torino
{ricchiuti, albertin}@csp.it

## Abstract

*In this paper we present a time sensitive content distribution system under implementation with the following innovative features: (a) events can be accessed and distributed using a fully decentralized DHT-based platform, (b) the bootstrap problem is solved by means of a permanent core network made of always alive nodes, (c) the streams are served through a forest of collaborating nodes interested in the events, by means of a topology independent from the one of the underlying DHT structure.*

## 1 Introduction

There are two main categories in which the Content Distribution studies can be divided: we have infrastructure-based content distribution (e.g., the Content Delivery Networks) and peer-to-peer content distribution (e.g., Gnutella, Splitstream and CoolStreaming).
Peer-to-peer systems such as Gnutella depend on little or no dedicated infrastructure: in our specific case, talking about peer-to-peer based systems, in live or on-demand streaming content distribution, cannot be separated from talking about Application Level Multicast and structured overlay networks (i.e., DHT-based p2p systems). This well-formed structure is the base layer of our framework.

### 1.1 Roadmap

In Section 2 we present the state of the art in peer-to-peer service oriented frameworks and applications. Our proposal is entirely based on a DHT layered architecture, and it is presented using a general framework (formalized in Section 3). Our content distribution schema is designed in Section 4, while implementation details are showed in Section 5. The conclusions are given in Section 6.

## 2 Related Works

Structured peer-to-peer overlay networks (e.g., Chord [16], Kademlia [10] or Pastry [14]) are an efficient solution for querying and retrieving resources spread between the peers, as they provide a good level of scalability and robustness to frequent attacks and common problems in unstructured peer-to-peer systems (e.g., [5][1][6]).
Distributed hash tables (DHTs), which represent the routing infrastructure of the structured peer-to-peer networks, are a class of decentralized distributed systems that partition the ownership of a set of keys among participating nodes, and can efficiently route messages to the unique owner of any given key. Usually, the owner of the key (or, the peer who is responsible for that key) is a specific peer (or a set of peers, due to replication policies) whose unique identifier is "closer" to the key, according to some metric depending on the algorithm: this association is possible since resource and peers are indexed in the same identifier space. This approach to routing (namely, the key-based routing) leads to a cost for the message forwarding that is logarithmic [16] in the size of the network (i.e., the number of nodes).
Unfortunately, key based routing of Distributed Hash Tables is simple when keys are known in advance, but this cannot be always assumed at the service level. In current working systems, many solutions have been adopted. In [8], authors point out what are the main difficulties in querying the system using only the exact match lookup facility, and in [7] an XML name space schema is proposed for querying resources indexed in the DHT system. Emule's[1] latest versions support a Kademlia-like DHT, known as KAD. For querying resources on this DHT, the Emule client parses the query string: through a hash function a different key is calculated for each substring. On the other side, as a resource is inserted in the system, a set of meta-information is calculated and inserted with the specific key.

Peers involved in the structured peer-to-peer system join a muliticast group on which the service (identified by a $stream$, live or on-demand) is propagated.

[1] http://www.emule-mods.de

We briefly describe our domain as an abstraction of a generic Distributed Hash Table (DHT), where application-specific objects are mapped to a subset of active nodes. The number of replica reflects the application's desired degree of replication for that object. Objects can be inserted as a pair `<key, value>`, where the `key` is the identifier of the object (i.e., `value`). Thus, we need to map the items of three different sets (i.e., *nodes*, *resources* (or, even, values) and *index keys*) into the same *identifier space*, in order to build up our framework and to use the routing mechanism of the DHT. For doing this, we can define two (hash) functions $H_1$ and $H_2$ that map nodes and keys in the same identifier space[2]. We have also to define the subset of active nodes in order to build the graph defining the *overlay topology*. Each node in the active node subset is *responsible* for a set of key identifiers, so we need also a function that maps the identifier of a key to the set of active nodes responsible for it. Resources can be *inserted* as well as *accessed* by routing a DHT specific message, using the object identifier as the key.

## 2.1 A DHT example: Pastry

Here we briefly discuss about Pastry [14], as we used this overlay network for implementing a prototype for our framework.

Each Pastry node has a unique, 128-bit nodeId. The set of existing nodeIds is uniformly distributed; this can be achieved, for instance, by calculating the nodeId on a secure hash of the nodes IP address. Given a message and a key, Pastry reliably routes the message to the Pastry node with a nodeId that is numerically closest to the key, among all live Pastry nodes. In a Pastry network with *N* nodes, Pastry can route a message to any node in less than $\lceil log_{2^b} N \rceil$ steps (with *b* being a configuration parameter).

The identifiers of nodes and resources (i.e., *nodeIds* and *keys*) are represented through sequences of digits with base $2^b$. Pastry routes messages to the node whose nodeId is numerically closest to the given key: in 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 (or b bits) longer than the prefix that the key shares with the present nodes 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 nodeId.

Each node maintains a **routing table** of $\lceil log_{2^b} N \rceil$ rows with $2^b - 1$ entries each row. With concurrent node failures, eventual delivery is guaranteed unless *l / 2* or more nodes with adjacent nodeIds fail simultaneously ($l$ is an even integer parameter).

[2]Note that many times we have that $H_1 = H_2 = SHA_1$.

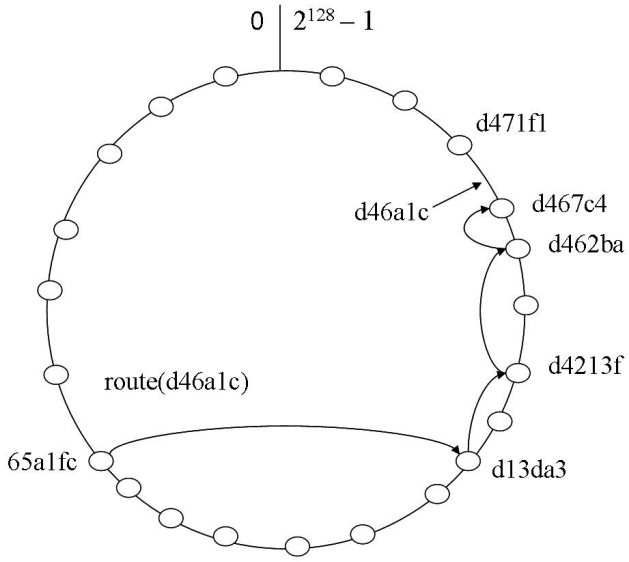


**Figure 1. Routing in Pastry from node 65a1fc of a message with id=d46a1c**

Given a row *n*, each entry in the routing table refers to a node whose nodeId matches the present nodes nodeId in the first *n* digits. The uniform distribution of nodeIds ensures an even population of the nodeId space, making only the first $\lceil log_{2^b} N \rceil$ rows to be populated.

In addition to the routing table, each node maintains IP addresses for the nodes in its ***leaf set***, i.e., the set of nodes with the *l / 2* numerically closest larger nodeIds, and the *l / 2* nodes with numerically closest smaller nodeIds. In each routing step (as in Figure 1), the current node normally forwards the message to a node whose nodeId shares with the key a prefix that is at least one digit (or *b* bits) longer than the prefix that the key shares with the current nodeId. If no such node is found in the routing table, 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 current nodeId. Such a node must exist in the leaf set unless the nodeId of the current node or its immediate neighbor is numerically closest to the key, or *l / 2* adjacent nodes in the leaf set have failed concurrently.

Pastry's routing infrastructure maintains several locality properties, i.e., the proximity metric is taken into consideration when a message is routed. The proximity metric is a scalar value that reflects the *distance* between any pair of nodes (e.g., RTT). Two of most important Pastry's locality properties are the so-called *short routes property*, which is concerned with the total distance that messages travel along Pastry routes, and the *route convergence property*, which is concerned with the distance traveled by two messages sent

to the same key before their routes converge.

Several applications and distributed systems are built on top of Pastry. Examples are Scribe [4], a distributed event notification infrastructure; Past [15], a distributed storage system and Splitstream [3], a streaming application that uses Scribe as a application level multicast for multimedia content delivering.
The Pastry's API provides all the functions for managing the different layers on top of it: from the *routemsg(Key k)* that perform the key based routing, to Past's level *insert(PastObject obj)* and *lookup(PastObject obj)* for storing and retrieving the inserted objects (e.g., files).

### 2.2 Applications on top of DHTs

Work on Application Level Multicast (e.g., Scribe [4]) is directly relevant to the live streaming aspect of our framework. There are two general approaches in building the Application Level Multicast: tree-building (e.g., Scribe) and flooding (e.g., CAN-Multicast [13]). In the first case, a single overlay network is built and each multicast group is defined via a spanning tree on it, while in the second each group is defined by a different overlay network, and all the routing information of the overlay network are used to broadcast multicast messages.
Coolstreaming [17], a totally distributed streaming platform for live events with many interesting features like the buffer map representation, and SplitStream [3], based on the Application Level Multicast proposed in Scribe, are just two of the possible approaches to the distribution of streaming media content. A key distinction of our work, for example w.r.t. SplitStream, is between *forwarder nodes* and *users*: the first join the group but they are not interested in the service, so they do not forward the payload but only service messages, while the latter are group members organized in a tree structure that forward the payload each other.

In 2002 Microsoft Research presented CoopNet [12], combining both aspects of content distribution mentioned before. The Cooperative Network of Coopnet addresses the problem of overloaded servers, having its client cooperating with each other to distribute content via caching schemas.
The framework presented here has been introduced in a previous work [11] consisting in an application in the e-learning domain. The implemented application is fully integrated and interoperable, and it is currently under testing on PlanetLab.

## 3 DHT-Service: inserting and searching structured information

We used *FreePastry* API for implementing the DHT (Pastry) and the multicast (Scribe) layer, while, for storing the XML document (see 3.1), we used PAST.
This multiple layer framework is depicted in Figure 2. Each peer takes part in the framework through different levels. The base layer, of course, is the real network topology, since it joins a distributed system. The DHT layer (performed by the FreePastry API) offers primitives for routing messages, assigning identifiers, retrieving values represented by keys (i.e., Pastry); for managing groups of peers in a multicast-like way and routing messages to groups instead of to peers (i.e., Scribe); for storing the `<key, value>` mappings and the XML documents (i.e., Past).
Peers from the multicast group organize themselves in a graph (i.e., a forest) through which the stream is propagated (as explained in Section 5).

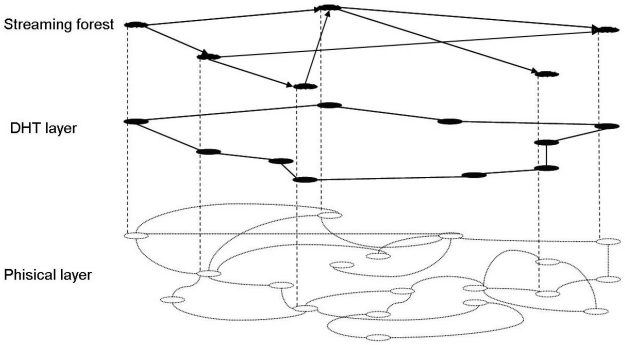


**Figure 2. Relationship between different layers.**

This layered framework allows to store informations in way such that the `value` associated to a key will be the `key` for another layer (see section 3.2).

### 3.1 Framework description

Each node using this framework (namely, DHT-Service[3]) has to store locally an XML name space defining the tags used in our message description language. Messages are simply resources that can be published in (and retrieved from) the overlay network. Each message is split in three tags, i.e., an *header*, a *body* and a *signature* (see Fig. 3).

[3]The name is given in contrast to the *Web Services* framework, where addressing and discovery of services are made by means of a central register server (e.g. UDDI registries).

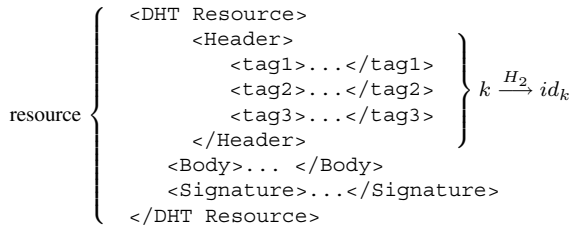


**Figure 3. The Resource XML template**

The main idea is that the header contains information knowable *a priori*, and that every user can build them by her own. The header is the key for the message, and it is passed to the hash function to obtain a key identifier. With such a key, it is possible to insert and to retrieve the associated resource, i.e., the body of the message, that contains further information.

### 3.2 Iterative key-based search/route

One of the most important problem to solve when using a Distributed Hash Table is the exact-match query method needed in the *key based routing*. This is easy when keys are known in advance, but it cannot be always assumed at the service level. It is important to find a way to get the exact key a resource is indexed with, in order to be able to send the appropriate query to the DHT-layer.
Using an XML name space, our key (i.e., the identifier used to index a resource) is the *header* of the XML document with the "ID card" of the indexed service (see Figure 3). What it must be done is a key structuring process in order to make peers able to calculate the exact key in terms of subsequent results.
Let's see in details:
**Insertion**: A *source peer* wanting to advertise his stream events sends a insertion message to the network. First, he has to access the XML template (see Figure 3) for publishing the content information, in order to insert the correct data using a well-formed set of tags.
Included in the *body* of the XML document there must be, among other information, the identifier of the multicast group (i.e., the *groupID*), on which the content is distributed. Information in the *header* can contain generic information (e.g., *Streaming Events*, as in Figure 4), as well as specific information (e.g., a given *GroupID*, as in Figure 5) that users accessed by way of previous queries in the iterative search process. If a message with the same header has already been inserted, the nodes responsible for replicas will check the credentials of the publisher and then update the resource (e.g., inserting other *Stream* tags in the *Streaming Events* message).
**Discovery and Join**: A peer who wants to join a specific multicast group, has to know the given *groupID*. To obtain this, it has to get the XML document indexed by the source peer. The peers should start then with a well-formed query, requesting the list of streaming events available at a certain moment. Suppose this query to be the string `Streaming Events`. Given the string to a secure hash function, the function will return the unique identifier associated, namely $id_s$. The peer can hence access the searched message with key $id_s$, containing all the available information on the stream events (see Figure 4). This document provides the list of services, given with the specified Ids. In general, after a low constant number of lookups, the peer is able to build up the complete header that, once lookup'd, will lead the peer to a document similar to the one in Fig. 5.

```
<DHT Streaming Events>
   <Header>
      <title>Streaming Events</title>
   </Header>
   <Body>
    <Stream>
    <Name>Movie One</Name>
    <Id>Adsof1023</Id>
    </Stream>
    <Stream>...</Stream>
 ...
   </Body>
   <Signature>...</Signature>
</DHT Streaming Events>
```

**Figure 4. Streaming Events**

```
<DHT Resource>
   <Header>
      <Name>Movie One</Name>
      <Type>mpeg4</Type>
      <Provider>MyProvider</Provider>
   </Header>
   <Body>
    <GroupId>1234wer093</GroupId>
    <Time>2/5/2006 20.00 C.E.T.</Time>
 <Other>...</Other>
   </Body>
   <Signature>...</Signature>
</DHT Resource>
```

**Figure 5. Resource**

### 3.3 Bootstrap node

A second key problem in DHT based system is the *bootstrap node*. A peer, in order to join a structured overlay, has to know at least one node already of the network [2]: the problem arises from the fact that no central server is available for granting connectivity (since the system is totally distributed), and there are no flooding mechanisms as the one used in the unstructured p2p systems (e.g., Gnutella). As far as current p2p systems do not provide a good solution to this problem, in the mentioned article authors presented a joining method based on the presence of a Certification Authority giving a trusted *nodeID* for joining a *universal ring*, and, from there, looking up the specified service. Our

framework distinguish the peers involved into two encapsulated networks (see Figure 6): a Core Network and a Edge Network. Nodes in the core network run a lighter version of the system stack, that only forward and store application messages. Secondly, they provide the access point to the DHT for the peers implementing the whole application. On the contrary, users and service providers run nodes in the edge network.

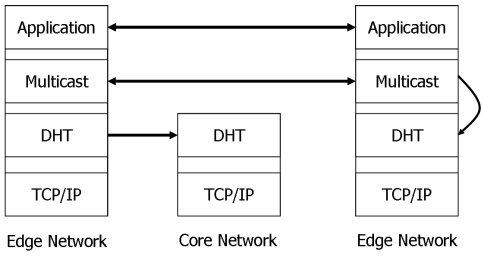


**Figure 6. Layered architecture**

## 4 Streaming through a collaborating forest of nodes

We can define a bijection between the set of different segments in which a stream is divided $(s_1, ..., s_n)$ and the elements of an array $(Arr[1], ..., Arr[n])$. Having this on each peer in the multicast group, then if ( $Arr[i] = 1$ ), the segment $s_i$ was already received by the node. This approach is similar to the buffer map representation of Coolstreaming [17]. Each element of the array contains the segments needed by the node, and all group members exchange this array with their partner periodically, granting the possibility of setting a scheduling in terms of peers and segments.

Moreover, a chocking procedure like BitTorrent's one [9] is necessary to make the peers behave better with a better QoS on the system since, for example, TCP congestion control behaves very poorly when sending over many connections at once. On the other side, a choking procedure makes each peer able to use a tit-for-tat algorithm to ensure a consistent download rate to the peer. There are several criteria a good choking algorithm should meet: it should cap the number of simultaneous uploads for good TCP performance and it should avoid the so-called *fibrillation* (i.e., a quick choking and "unchoking" of the bandwidth capacity). Moreover, it should be granted a periodic test on alternative paths for the messages, in order to find out if they might be better than the currently used ones.

In our implementation, we have a source peer which has some content (live, or on-demand) to distribute, and that represents the *root* of the forest. Peers connect each other and find the Multicast Group via the DHT's procedures. A subset of group members join together a tree structure and start receiving/forwarding the stream. The reader should notice that this tree is similar, in some ways, to the SplitStream one (and also, but less than the former) to the CoopNet tree.
We can summarize our approach in terms of differences with the previous cited works. Compared with SplitStream, forwarders peers are only interested in message routing, not in transferring the payload, as it happens in CoopNet. Compared with CoopNet, where peers are associated in a tree structure, we have a forest, i.e., a strongly connected graph in which group partners will exchange their buffer maps and messages in a BitTorrent-like manner (as CoolStreaming does). Finally, compared with CoolStreaming, we put an intermediate application level multicast layer granting the possibility of having, among other features, a restricted access to some specified streams.

All the affinities and the differences within our framework wrt other applications can be summarized as in table 1.

## 5 Implementation details

The content distribution tool is currently under implementation. The language we chose is Java, due to its portability and to the fact that the FreePastry API is coded in Java. The information given in this section are subject to change, even if the forest topology depicted in the previous section is the target solution.

In our implementation, the group joining and the retrieving of the XML document are done at the same time. The connection parameters (e.g., IP/Port of the *root* of the tree) are packed in a "Content" object, sent to the peer searching for the associated key.

A **join** request arrives to the root that, depending on the available bandwidth, allows (**accept**) or denies (**refuse**) the incoming connections. In case a call is refused, it will be forwarded to one randomly chosen son: by doing this, a balanced growth of the tree is achieved. As soon as the tree starts to grow, the root start streaming the content. However, this joining procedure will became outdated when the *scribe* layer will be fully integrated to our system: the root will not be involved to every join request, because *forwarders* will help new nodes to discover other peers participating the same event.
We suppose that a node wanting to receive the stream will not leave the forest until the transmission ends. Anyway, peer failures are possible. When a node that is receiv-

| | Node Forwarding | Forest Structure | Buffer Map Representation | Time Sensitive Streaming |
|---|---|---|---|---|
| BitTorrent | X | | X | |
| SplitStream | X | X | | X |
| CoopNet | X | | | X |
| CoolStreaming | X | | X | X |
| Our Appl. | X | X | X | X |

**Table 1. Our framework wrt other Applications.**

```
n.join(Stream s, Node root)

if (accept)
  //start receiving from s from root
  n.startReceive (Stream s, Node root)
  if (refuse) {
     Node n2 = n.chooseRandomSon();
     n.join(Stream s, Node n2);
}
...
catch(EndOfReceiving Exception) {
  for each c in S //set of node sons
    /*stop sending stream to each
     *node c the node n was serving
     */
      n.StopTransmission(Node c);
 n.join(Stream s, Node root);
}
```

**Figure 7. Pseudo-Code for the Join Operation and the Restore procedure**

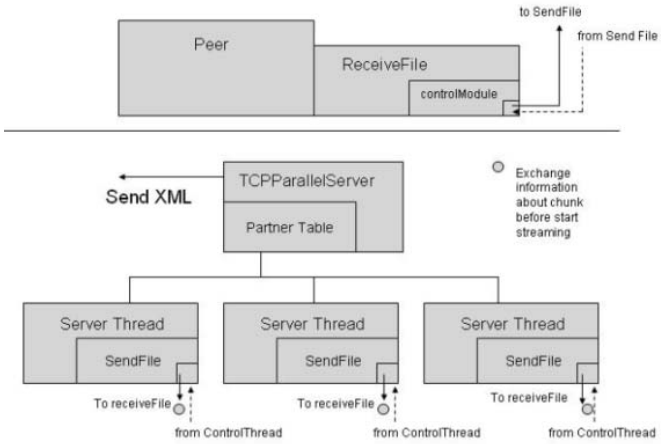


**Figure 8. Application Modules.**

ing/transmitting to a given peer fails, the **leave** procedure is called to re-establish the correct behaving of the tree: peers that were streaming to the failed node just stop the transmission to that node; on the contrary, peers that were receiving from the failed node recognize the anomalous end of transmission after a time-out interval. If that node was the only transmitting peer, the *orphans* start with a new **join** operation, in order to find out their new position in the tree. If other nodes are still transmitting, then the receiving peers re-negotiate bandwidth and missing segments with the group.

In case of massive failures, it is strictly necessary to avoid too many *join messages* to the source: to do that, each node will wait for a random time before sending a new join request.

In figure 8 a simple scheme of our application is given.

A Java module, namely the **Data Transfer Module** 7, manages the structure of the distribution topology and the data streaming. After joining the forest, a peer creates its own *Partner Table*, i.e., a dynamic table containing a list of nodes that the peer is transmitting to (and receiving from), with their attributes (e.g., IP address and listen port). Partner Table manages the insertions and the removals of partner nodes. Data transfer is made through a buffered stream. Initially the peers exchange some information and later they exchange segments of the audio/video stream.

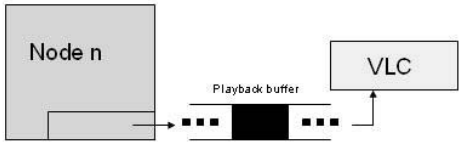


**Figure 9. Buffered Stream.**

Finally, the **Control Module** is split into two different threads, working together: the **ReceiveFile** and the **ControlThread**. The first allows to receive the data in predefined buffer and then it appends the buffer to the temporary output file 9. This file, during this procedure, is sent as input to an external player which allows the user to see the stream. The second thread listens to the control port on which, at the end of transmission, the receiving peer sends an *End-Of-File* string that determines the effective end of that specific segment.

In order to playback the received stream, we use an ex-

ternal player. Our choose has been VLC MediaPlayer[4], because it is completely open source, and there is a lot of technical support available on the web. VideoLan is also multiplatform and it is able to playback quite all multimedia format. VideoLan is also completely controllable from an external application, thanks to its complete set of command-line options. There is no limitation in our implementation about the use of VLC so any other multimedia player can be used, the only requirement is that the player should be able to accept the file from the command line interface.

## 6 Conclusion and Ongoing Work

In this article we introduced a framework for publishing and retrieving multimedia content distribution services, built on a totally distributed overlay network. Using a DHT to address the peers and the resources, and using a Multicast layer to build our groups, we can separate the specific functions of each layer. This paper characterizes basically our approach to the problem of building a Streaming Application on a DHT.

## 7 Acknowledgements

This work, jointly developed by CSP Sca.r.l. (Turin, Italy) and the Department of Computer Science at the University of Turin, has been financially supported by the Italian FIRB 2001 project number RBNE01WEJT "Web MiNDS".

## References

[1] E. Adar and B. A. Huberman. Free riding on gnutella. *First Monday*, Sept. 2000.

[2] M. Castro, P. Druschel, A. Kermarrec, and A. Rowstron. One ring to rule them all: Service discovery and binding in structured peer-topeer overlay networks, 2002.

[3] M. Castro, P. Druschel, A.-M. Kermarrec, A. Nandi, A. Rowstron, and A. Singh. Splitstream: High-bandwidth content distribution in a cooperative environment. In *Proc. of IPTPS'03*, February 2003.

[4] M. Castro, P. Druschel, A.-M. Kermarrec, and A. Rowstron. Scribe: A large-scale and decentralised application-level multicast infrastructure. *IEEE Journal on Selected Areas in Communication*, October 2002.

[5] N. Daswani and H. Garcia-Molina. Query-flood dos attacks in gnutella. In *Proc. of the 9th ACM conference on Computer and communications security*, pages 181–192. ACM Press, 2002.

[6] N. Daswani, H. Garcia-Molina, and B. Yang. Open problems in data-sharing peer-to-peer systems. In *ICDT 2003*, 2003.

[4] http://www.videolan.org

[7] P. Felber, E. Biersack, L. Garces-Erice, K. Ross, and G. Urvoy-Keller. Data indexing and querying in dht peerto -peer networks, 2004.

[8] M. Harren, J. Hellerstein, R. Huebsch, B. Loo, S. Shenker, and I. Stoica. Complex queries in dht-based peer-to-peer networks, 2002.

[9] I. B. R. in BitTorrent. http://www.bittorrent.com/.

[10] P. Maymounkov and D. Mazieres. Kademlia: A peer-to-peer information system based on the xor metric. In *Proc. of IPTPS02*, March 2002.

[11] M. Milanesio and G. Ruffo. Publishing, retrieving and streaming lectures via application level multicast. In *Proc. of the IEEE ICIW'06*, 2006.

[12] V. N. Padmanabhan, H. J. Wang, P. A. Chou, and K. Sripanidkulchai. Distributing streaming media content using cooperative networking. Technical Report MSR-TR-2002-37, Microsoft Research, Redmond, WA, 2002.

[13] S. Ratnasamy, M. Handley, R. Karp, and S. Shenker. Application-level multicast using content-addressable networks. *LNCS*, 2233, 2001.

[14] A. Rowstron and P. Druschel. Pastry: Scalable, distributed object location and routing for large-scale peer-to-peer systems. In *Proc. of IFIP/ACM International Conference on Distributed Systems Platforms (Middleware)*, pages 329–350, November 2001.

[15] A. I. T. Rowstron and P. Druschel. Storage management and caching in past, a large-scale, persistent peer-to-peer storage utility. In *Proc. of Symposium on Operating Systems Principles*, pages 188–201, 2001.

[16] I. Stoica, R. Morris, D. Karger, M. F. Kaashoek, and H. Balakrishnan. Chord: A scalable peer-to-peer lookup service for internet applications. In *Proc. of the 2001 Conf. on Applications, technologies, architectures, and protocols for computer communications*, pages 149–160. ACM Press, 2001.

[17] X. Zhang, J. Liu, B. Li, and T.-S. P. Yum. Donet/coolstreaming: A data-driven overlay network for live media streaming. In *Proc. of IEEE INFOCOM'05*, March 2005.