# A Programmable Structured Peer-to-Peer Overlay

Marius Portmann[1], Sébastien Ardon[2], and Patrick Sénac[2]

[1] School of Information Technology and Electrical Engineering,
University of Queensland, Brisbane QLD 4064, Australia
marius@itee.uq.edu.au
[2] DMI / ENSICA
1 place Emile Blouin, 31000 Toulouse, France
{sardon,senac}@ensica.fr

**Abstract.** Structured peer-to-peer (P2P) overlay are scalable, robust and self-organizing in nature, and provide a promising platform for a range of large-scale distributed applications. Applications proposed to date utilize a similar key-based routing service but "re-invent the wheel" by deploying their own dedicated structured P2P overlay network. This is highly inefficient and results in a significant duplication of work in terms of development, deployment and maintenance of the overlays. To address this problem, we propose a PROgrammable STructured P2P infrastructure (PROST), which allows the dynamic and incremental deployment of multiple applications over a single structured P2P overlay. In this paper, we outline the PROST architecture and discuss the implementation of our prototype.

**Keywords:** Programmable networks, peer-to-peer.

## 1 Introduction

In recent times, structured peer-to-peer (P2P) overlay networks have attracted a lot of attention in the research community. These systems are also referred to as Distributed Hash Tables (DHT), since Hash Tables are a common service abstraction implemented by structured P2P overlays. All the proposed structured P2P systems (e.g. [1], [2], [3]) share the features of an efficient lookup mechanism, fault-tolerance, scalability and a self-organizing nature. These characteristics make structured P2P systems an ideal building block for a wide range of large-scale distributed applications.

All currently proposed applications using structured P2P overlay are tightly integrated with their own implementation of a dedicated structured P2P overlay network. Obviously, this results in an undesirable duplication of effort, and more importantly, this result in a higher cost of operation, in particular in terms of network traffic as every application with its own P2P overlay separately incurs the cost involved in deploying and maintaining it.

To address this problem, we present a PROgrammable STructured P2P infrastructure (PROST), an overlay architecture, which allows multiple applications to share a common overlay network. This is achieved by allowing the dynamic and incremental deployment of applications and services onto overlay nodes. Our proposed infrastructure allows P2P applications to be run much more efficiently since the cost for

D. Hutchison et al. (Eds.): IWAN 2005, LNCS 4388, pp. 145–155, 2009.

deployment and maintenance of the overlay network is amortized over multiple applications. Finally, the development of new applications is also greatly facilitated by providing developers with a simple API to access the basic services provided by structured P2P systems.

The remaining of this paper is organized as follows: Section 2 gives a brief background on structured P2P systems. In Section 3 we introduce PROST, our programmable P2P platform and outline its architecture. Section 4 discusses the dynamic deployment of applications in PROST. In Section 5 we present our proof-of-concept implementation of PROST. Section 6 and 7 discuss remaining challenges and related work, and Section 8 concludes the paper.

## 2 Structured P2P Overlays – Background

Structured P2P overlays have recently gained popularity due to their ability to locate objects in a network very efficiently, with a cost of typically *O(log N)* messages exchanged, where N is the number of nodes in the overlay. This is in contrast to unstructured P2P systems, where lookup operations are based on flooding messages in the network, resulting in a cost that scales linearly in the number of nodes [20].

At the heart of every structured P2P system is a key-based routing (KBR) mechanism [8]. Every node in the overlay is assigned an identifier or *nodeID* from a large identifier space, typically 128 or 160 bits. Application-specific objects (files, database records, etc.) are assigned unique identifiers called keys, chosen from the same identifier space. Keys are typically assigned to an object by hashing the name of the object. Based on its key, each object is mapped to a node that is responsible for it, called its root node. The details of how this mapping is done vary for different P2P systems. In Pastry [2] for example, keys are mapped to the node with the closest *nodeID*. This key-to-node mapping is implemented by the KBR mechanism, through routing of lookup messages to their destination nodes, i.e. the root nodes.

To achieve this efficiently, the overlay topology is tightly controlled. Each node has a small routing table with a number of carefully chosen links to peer nodes. Lookup messages are routed to their destination along these overlay links, in typically *O(Log N)* hops. The key-to-node mapping of the KBR mechanism forms the basis of all structured P2P systems, upon which a range of higher layer services and applications can be implemented.

Figure 1 shows a layered model of structured P2P overlays as it was proposed in [8]. The Key-based routing (KBR) layer represents the greatest common denominator for all structured P2P overlay systems. As explained previously, this layer is responsible for routing messages to key's root nodes. Built on top of it are higher layer communication abstractions, such as Distributed Object Location and Routing (DOLR), Distributed Hash Tables (DHT), and group anycast/multicast (CAST) which essentially provide different communication primitives on which to build applications. Finally, applications reside on the third layer. The applications shown in Figure 1 are distributed storage systems (CFS [7], PAST [12], OceanStore [4]), group communication/multicast systems (Scribe [9], Bayeux [10]), content distribution (Split Stream) [5] and a generic Indirection Infrastructure (I3) [6].

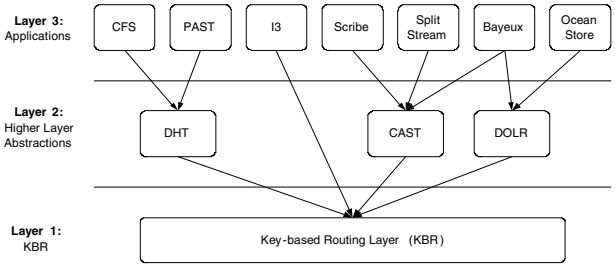


**Fig. 1.** A Layered Model of Structured P2P Systems

The higher layer abstractions shown at layer 2 of Figure 1 are quite diverse. Future applications will require an even wider range of functionality and service abstractions. This makes the implementation of layer 2 as a static service layer in our view difficult, if not impossible.

With PROST, we propose to implement the basic KBR layer as a shared static infrastructure. Layers 2 and 3 are implemented via a programmable layer that allows dynamic and on-demand deployment of applications and services. The following sections describe the PROST architecture of our programmable structured P2P system in more detail.

## 3 PROST – A Programmable Structured P2P Overlay

Even though all current structured P2P systems provide the same basic service of key-to-node mapping via the Key-based routing mechanism, they all export different APIs with slightly different semantics. In order to implement a generic structured P2P infrastructure that is not tied to one particular system, we also need a generic API to access the basic services of structured P2P overlays. Such an API has been proposed by Dabek et al.in [8]. The authors show that the API can easily be implemented by all current structured P2P systems and that it is rich enough to allow the implementation of all current higher layer service abstractions and applications. This makes the API an ideal building block for our proposed infrastructure.

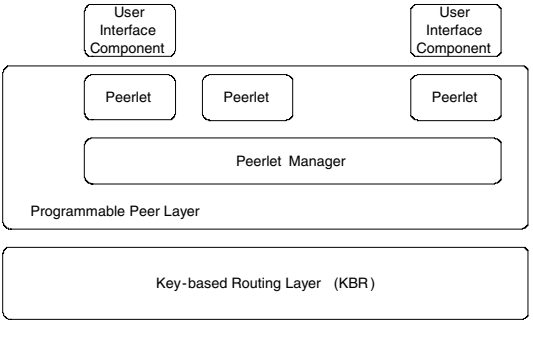


**Fig. 2.** PROST Node Architecture

Figure 2 illustrates the two-tiered architecture of a node in PROST. The KBR layer forms the basis of the infrastructure and is used to map keys onto node. Its API is similar to that proposed in [8]. On top of this KBR layer resides the Programmable Peer Layer, which comprises the functionality of Layers 2 and 3 of Figure 1. Applications and services are deployed in this layer dynamically loaded *peerlets*. Each of the components is discussed in more detail in the following sections.

### 3.1 Key-Based Routing Layer

The Key-based routing layer forms the base of the PROST infrastructure, and is used to map keys onto nodes. This is implemented via the API's *route()* method, which delivers a message to the node responsible for a given key (root node). To enable multiple applications to share a common KBR layer, every message sent over the overlay network needs to include an *application identifier* (app id), allowing the demultiplexing of messages and their delivery to the appropriate application, i.e. peerlet, similarly to the port number in transport protocols. In addition, the *route()* API primitive can take a first node ID as a parameter. This mechanism allows the application to bypass the default KBR routing process and specify the first-hop to use when sending a message (useful for some application-level multicast applications).

The *application identifier* and *application type* parameters are necessary for the correct handling of messages and are therefore included in the header of each message routed by the KBR layer, as shown in Figure 3. The message further contains the destination key of the message as well as a *Peerlet Code Locator* parameter. This parameter informs the nodes about the mechanism and location for downloading the peerlet code.

Applications running over structured peer-to-peer systems can be broadly classified in two categories. The first type of application, which we call *end-node* applications, only requires the destination or end nodes of the routing path to perform application-specific operations. A typical example of end-node application is any application based on the DHT concept, i.e. the storage of {key, values} couples. On the other hand, the second type of application, which we call per-hop applications, involves intermediary nodes in the routing path to perform per-hop operations on "their" messages before they are forwarded. This may include changing the next hop ID (i.e. changing the routing of a message).

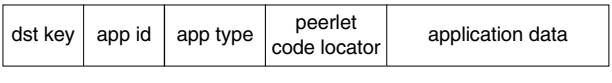


**Fig. 3.** PROST message format

To this end, the API defines a *forward()* callback method that is invoked at each node that forwards a message. This upcall informs the application that the message M with a key k is about to be forwarded to the node with ID *nextHopNode*. Examples of applications requiring per-hop treatment are multicast routing [9], [10] application-level multicast infrastructure [9] or result aggregation [11]. For example, in the SCRIBE event notification service [9], nodes requires the underlying key-based routing layer to invoke

its forward upcall in order to build the multicast event dissemination tree on the way from the subscriber to the rendezvous point. PROST uses the binary message parameter application type to differentiate between these two classes of applications, as shown on Figure 3. Note that intercepted messages are only sent to applications with ID matching that of the message.

The KBR layer also provides mechanisms that deal with the transient and unreliable nature of individual nodes. The deal with node failure, the KBR layer defines set of replication nodes, called a replica set, for each key. In case a root node is unavailable, the KBR layer is responsible to route a message to one of the available replica nodes. Furthermore, the API's *replicaSet()* method gives the application access to the current set of replication nodes, in order to keep the replication nodes synchronized. Finally, the routing tables of peer nodes need to be maintained and updated appropriately in case of node failure, or in case of nodes joining and leaving the overlay. Applications can be informed of such events via the API's *update()* callback method. We refer to [8] for a detailed discussion of the API.

### 3.2 Programmable Peer Layer

Above the KBR layer, the programmable peer layer performs all the operations concerning peerlets deployment, execution and termination. As mentioned previously, peerlets are mobile code modules, which are dynamically loaded and installed, similarly to any plug-in architecture. Peerlets implement the actual P2P applications.

The Peerlet Manager, shown on Figure 2, is responsible for the loading and instantiation of peerlets. It also enforces the node's local security and access control policy by controlling and limiting the peerlets' access to local resources such as CPU, storage and the network. It further provides security by isolating peerlets from the host system to limit the impact of malicious or faulty peerlets. Currently, resource management in PROST is kept to a basic sandbox model, as explained in section 5. We are looking at applying existing research results, e.g. from the active network community to further improve on this.

Applications on peer nodes in PROST consist of two components: a peerlet and a user interface component. The peerlets implements the server-component of the application. Again, this functionality can be as simple as a DHT, but it can be arbitrarily complex, depending on the application. The user interface component of an application allows end-users to access the services provided by the peerlets. This is typically a Graphical User Interface, but this is determined by the application and not restricted by our infrastructure. While peerlets can be deployed without a user interface component on a node, user interface components require a peerlet to be deployed on the same node.

## 4 Dynamic Application Deployment

In this section, we discuss how multiple applications can co-exist in PROST and how new applications can be dynamically deployed.

### 4.1 End-Node Applications

As mentioned previously, end-node applications only need to invoke application-specific functionality at the destination node of a message in the KBR layer. This is

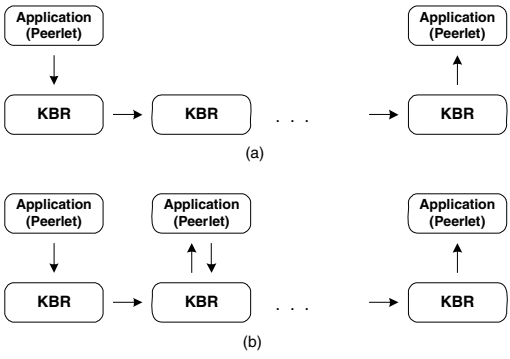


**Fig. 4.** Message flow in End-node (a) and Per-hop Applications (b)

illustrated in Figure 4a. A simple DHT is an example for such an application. The intermediary nodes in the routing path are not involved in the application and simply forward messages to their destination.

In a typical scenario, a peer application initiates an operation on another node designated by a key, by invoking the *route()* method (c.f figure 3). For example a *put(key,data)* operation in a DHT. This results in the KBR layer to route a message to the root node in charge of the key. In this case, the application type field tells intermediary nodes in the routing path that no application-specific code needs to be invoked and the message is therefore simply forwarded according to the default routing rules of the KBR layer Upon receiving the message, the root node lookup the message application id, and checks if the corresponding peerlet is installed. If this is the case, the message is delivered to the peerlet; otherwise, the node automatically downloads the necessary code from the source specified by the Peerlet Code Locator field in the message. Once the peerlet is loaded and instantiated it is passed the message and it can perform the necessary tasks.

### 4.2 Per-Hop Applications

As mentioned in section 3.1, the second category of applications considered in PROST is per-hop applications, which requires per-hop operations to be performed at the intermediary nodes along the routing path. This is illustrated in Figure 4b. Per-hop applications, in contrast to End-node applications, therefore require the corresponding peerlets to be installed at the intermediary nodes of a message's routing path. The process of dynamic application deployment is therefore slightly different from the previous case: at every node in the path the application's peerlet is invoked via the *forward()* upcall of the KBR layer's standard interface (c.f. section 3.1). This call passes the message up to the application and allows it to override the standard routing behavior or to perform the application-specific tasks.

As in the case of End-node applications, an operation begins with an application invoking the *route()* method to send a message to the node responsible for a key k. The message is forwarded to the first node in the path to its destination. In this case, the application type field specifies that application-specific functionality needs to be invoked at intermediary nodes. Therefore, the node calls the *forward()* method of the

peerlet with the specified application identifier. If the corresponding peerlet is not installed, the node's peerlet manager automatically downloads the code and installs it, in the same way as described in the previous section. Then, the peerlet's *forward()* method is called which gives the application the opportunity to perform any per-hop operations. After the *forward()* method returns, the message is sent to the next node on its path and the process is repeated until it reaches its final destination.

### 4.3 Application Deployment Policing

In a general use case, most nodes will want to control the type and which particular peerlet they run. This can be due to security or legal reason. It is therefore possible that some nodes are not able or not willing to install and run certain peerlets. Note that this problem is irrelevant if we consider the use of PROST in a corporate context, or more generally when a trust relationship exists between nodes and peerlet providers.

This problem of non-cooperating nodes is different for per-hop-applications than for end-node applications. PROST requires per-hop applications, to tolerate a certain amount of non-cooperating intermediary nodes and implement a graceful degradation of service. The impact of this limitation on the performance of several existing per-hop applications is currently under investigation. On the other hand, the situation where a message's destination node refuses to install a peerlet is more severe. This problem is treated in PROST in the same way as general node failures are treated in most P2P systems, by means of replication. As briefly discussed in Section 3.1, the KBR layer provides applications with the necessary mechanisms for implementing replication.

## 5 Implementation

We have implemented a proof-of-concept prototype of PROST in Java. The KBR layer of our prototype is based on the implementation of the CHORD [1] protocol from the XML-Store project [21], with a few minor modifications. We replaced the UDP-based RPC mechanism with Java's Remote Method Invocation (RMI). This increased the code stability and allowed us to easily detect node failures in the CHORD overlay, which was not implemented in the original XML-Store code. The downside of using RMI for inter-node communication in a peer-to-peer overlay is the relatively high overhead. However, the focus of our first proof-of-concept implementation was simplicity and ease of development rather than performance.

We also implemented the standard API for structured P2P systems. In addition to the methods defined in [8], we implemented a *lookupPeerlet()* method. This recursive method takes a message (see Figure 3) and the corresponding parameters and delivers it to the root node responsible for the given key k, in the same way the API's *route()* method does it, including the invocation of per-hop-operation if required. In contrast to *route()*, the *lookupPeerlet()* method is blocking returns a RMI reference of the peerlet the message was delivered to. With such a reference, peerlets can invoke any application-specific operations on each other, using Remote Method Invocation. For example, in the case of a DHT, the application would simply call the *put()* and *get()* methods of the remote peerlet.

Java's support for mobile code greatly facilitated the implementation of the Programmable Peer Layer. Peerlets are implemented as Java classes implementing the Peerlet interface. The corresponding class files are dynamically loaded by the Peerlet Manager from remote sources via Java's URLClassLoader. In our implementation, we used a standard web server as a peerlet code server, which can be authenticated and secured using standard methods. We are currently working on implementing the peerlet code server functionality in a distributed fashion as an integral part of PROST.

We currently employ two basic methods to protect a node from malicious (or faulty) third party code. Firstly, peerlets executing on a PROST node are contained to a restricted environment, also referred to as a sandbox. The sandbox provides a separate name and address space for each peerlet and limits its access to functionality and resources on the host node. Secondly, PROST supports the concept of code signing, where each peerlet is digitally signed by its producer. When dynamically loading peerlet code, a node can verify its authenticity and the identity of its producer. This allows restricting the loading of peerlets from trusted sources only, depending on a node's local security policy.

To evaluate the design and usability of our infrastructure, we implemented two sample applications: an instant messaging (IM) application and a yellow pages-style service directory that maps service categories to a list of service providers. We successfully deployed and tested these applications simultaneously on a small overlay of 20 nodes on 5 physical machines. Deploying new P2P applications, even on small overlays, is typically a very tedious and costly operation. Application deployment in PROST is relatively easy and involves the following steps: first, a 128-bit application identifier is assigned pseudo-randomly, with a negligibly small probability of a collision. Then, the peerlet code needs to be made available on a code server. An initial peerlet and the corresponding user interface component must be installed manually on one of the overlay nodes. Further deployment of peerlets is done automatically, as described in Section 4. For example, the creation of new service categories in our service directory application results in the automatic deployment of peerlets on the nodes responsible for these categories i.e. the corresponding key. Reliability through replication has not been implemented yet for our two sample applications. This is one of the issues that we will address in our future work.

## 6 Future Work

There are a large number of issues remaining to be addressed in PROST. The first challenge that comes to mind in PROST is the security aspect of the KBR layer, as identified in [16]. These security problems are shared by all structured P2P systems. Some ideas proposed in [17] can be applied in the context of PROST and they provide partial solutions to some of the problems. However, we believe the basis of any security mechanism in P2P overlays are strong and verifiable node identities. We are currently exploring the use of crypto-based IDs [19] for our infrastructure. We are also investigating the use of Threshold Cryptography [18] to implement a distributed admission control mechanism.

Secondly, resource management and resource policing aspects in PROST also needs to be further developed. We believe some existing results from the active

network (AN) research can be re-used in this context. Thirdly, there are a number of performance aspects to be investigated, such as quantifying the impact of the larger overlay network on delays and on probability of failure/cache miss. Finally, we plan to develop a range of applications to further evaluate our infrastructure and perform performance measurements on a medium scale deployment of PROST.

## 7 Related Work

The lack of a generic shared infrastructure for structured P2P system has also been identified as a being a problem in [13]. As a solution, the authors propose OpenHash (subsequently re-named OpenDHT), an open, publicly accessible DHT service that runs on a set of infrastructure hosts. It provides applications with the simple put/get interface of a hash table. Applications for which this interface is sufficient are called Lite Applications, and can be directly implemented on top of OpenHash. For the majority of applications for which a simple DHT interface is not adequate, OpenHash serves as a distributed rendezvous mechanism that allows discovery and coordination between nodes implementing the same applications. The actual P2P applications are deployed on arbitrary nodes outside the OpenHash infrastructure. This represents a departure from the pure P2P paradigm that stipulates symmetric roles of all nodes, and is one of the main points in which OpenHash differs from our approach. Finally, OpenHash does not provide a mechanism for the dynamic deployment of application and services.

In [8], a standard API for the KBR layer, which is common to all structured P2P systems, is proposed. The authors go further and state their intention to define static APIs for a range of higher layer abstractions (Layer 2 in Figure 1) such as DHTs, DOLR, CAST etc. This is in contrast to our approach, which only defines a static API for the KBR layer, but provides maximum flexibility and ease of deployment for higher layer functionality via a programmable platform.

In [3], the authors mention that their structured P2P system has an extensible API and it can be shared by multiple applications. However, the paper does not address the details of this and neither does it provide a mechanism to dynamically deploy new functionality and applications.

JXTA [22] is a project that defines a set of protocols and concepts for P2P computing, such as peer and resource discovery, peer communication and peer group management. One of the main point in which JXTA implementations differ from our proposed infrastructure is the way in which messages routing is implemented. JXTA uses an adaptive source-based routing, where routes are initially computed by the sender. This is in contrast to our proposal that specifically focuses on the Key-based routing mechanism of structured P2P systems. Furthermore, JXTA does not provide a mechanism for the dynamic deployment of applications.

Finally, the idea of programmable overlay architecture is not new and has been proposed previously by a number of authors [14], [15]. However, to the best of our knowledge, our proposed architecture is the first to apply the idea of programmability in the context of structured P2P overlays to alleviate the overlay applications from maintaining their own dedicated overlay network

## 8 Conclusions

In this paper, we outlined the idea of PROST, a programmable structured P2P platform that allows dynamic deployment of new distributed applications and services. It is built on top of a key-based routing layer that provides a scalable and efficient lookup service. The programmability of our proposed infrastructure is a new concept in the context of structured P2P networks. It allows the basic P2P routing infrastructure to be shared by multiple applications, thereby also sharing the cost of deployment and maintenance, while providing a maximum degree of flexibility to accommodate the requirements of wide range of current and future applications.

We believe that the availability of a shared and programmable infrastructure, as proposed in this paper, would greatly encourage and facilitate the innovation and development of new large-scale distributed applications using structured P2P functionality. Our work presented here is in its early stages, and with many unresolved issues. However, we believe our novel approach provides a promising solution that can serve as a versatile platform for a wide range of distributed applications.

## References

1. Stoica, I., Morris, R., Lien-Nowell, D., Karger, D.R., Kaashoek, M., Dabek, F., Balakrishnan, H.: Chord: A Scalable P2P Lookup Protocol for Internet Applications. In: ACM SIGCOMM 2001, San Diego, CA (August 2001)
2. Rowstron, A., Druschel, P.: Pastry: Scalable, distributed object location and routing for large-scale peer-to-peer systems. In: Guerraoui, R. (ed.) Middleware 2001. LNCS, vol. 2218, p. 329. Springer, Heidelberg (2001)
3. Zhao, B.Y., Huang, L., Stribling, J., Rhea, S.C., Joseph, A.D., Kubiatowicz, J.D.: Tapestry: A Resilient Global-scale Overlay for Service Deployment. IEEE Journal on Selected Areas in Communications (January 2004)
4. Kubiatowicz, J., Bindel, D., Chen, Y., Eaton, P., Geels, D., Gummadi, R., Rhea, S., Weatherspoon, H., Weimer, W., Wells, C., Zhao, B.: OCEANSTORE: An Architecture for Global-Scale Persistent Storage. In: Proceedings of the 9th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS), Cambridge, MA. ACM, New York (2000)
5. Castro, M., Druschel, P., Kermarrec, A., Nandi, A., Rowstron, A., Singh, A.: Splitstream: High-bandwidth multicast in cooperative environments. In: 19th ACM Symposium on Operating Systems Principles (2003)
6. Stoica, I., Adkins, D., Zhuang, S., Shenker, S., Surana, S.: Internet indirection infrastructure. In: Proceedings of ACM SIGCOMM (August 2002)
7. Dabek, F., et al.: Wide-area cooperative storage with CFS. In: Proceedings of the 18th ACM Symposium on Operating Systems Principles (SOSP 2001), Banff, Canada (October 2001)
8. Dabek, F., et al.: Towards a common API for structured P2P overlays. In: Proceedings of IPTPS 2003, Berkeley, CA (February 2003)
9. Rowstron, A., Kermarrec, A.-M., Castro, M., Druschel, P.: SCRIBE: The design of a large-scale event notification infrastructure. In: Crowcroft, J., Hofmann, M. (eds.) NGC 2001. LNCS, vol. 2233, p. 30. Springer, Heidelberg (2001)
10. Zhuang, S.Q., Zhao, B.Y., Joseph, A.D.: Bayeux: An architecture for scalable and fault-tolerant wide-area data dissemination. In: The Proceedings of the 11th ACM/IEEE NOSSDAV, New York (June 2001)

11. Huebsch, R., Hellerstein, J.M., Lanham, N., Thau Loo, B., Shenker, S., Stoica, I.: Querying the Internet with PIER. In: Proceedings of the 9th International Conference on Very Large Data Bases (VLDB 2003), Berlin, Germany, September 9-12 (2003)
12. Druschel, P., Rowstron, A.: PAST: Persistent and anonymous storage in a peer-to-peer networking environment. In: Proceedings of the 8th IEEE Workshop on Hot Topics in Operating Systems (HotOS-VIII) (2001)
13. Karp, B., Ratnasamy, S., Rhea, S., Shenker, S.: Spurring adoption of DHTs with openHash, a public DHT service. In: Voelker, G.M., Shenker, S. (eds.) IPTPS 2004. LNCS, vol. 3279, pp. 195–205. Springer, Heidelberg (2005)
14. Fry, M., Ghosh, A.: Application Level Active Networking. Computer Networks 31(7), 655–667 (1999)
15. Ardon, S., Gunningberg, P., Ismailov, Y., Landfeldt, B., Portmann, M., Seneviratne, A., Thai, B.: Mobile Aware Server Architecture: A distributed proxy architecture for content adaptation. In: Proceedings of The 11th annual Internet Society Conference (INET 2001), Stockholm, Sweden (June 2001)
16. Sit, E., Morris, R.: Security considerations for peer-to-peer distributed hash tables. In: Druschel, P., Kaashoek, M.F., Rowstron, A. (eds.) IPTPS 2002. LNCS, vol. 2429, p. 261. Springer, Heidelberg (2002)
17. Castro, M., Druschel, P., Ganesh, A.J., Rowstron, A., Wallach, D.S.: Secure Routing for Structured Peer-to-Peer Overlay Networks. In: Proceedings of the 5th Symposium on Operating System Design and Implementation (OSDI 2002), Boston, Massachusetts, USA, December 9-11, 2002, USENIX Association (2002)
18. Saxena, N., Tsudik, G., Hyun Yi, J.: Admission Control in Peer-to-peer: Design and Performance Evaluation. In: Proceedings of the 1st ACM Workshop on Security of Ad Hoc and Sensor Networks, Fairfax, Virginia (2003)
19. Montenegro, G., Castelluccia, C.: Crypto-based identifiers (CBIDs): Concepts and applications. ACM Transactions on Information and System Security (TISSEC) 7 (2004)
20. Lv, Q., Cao, P., Cohen, E., Li, K., Shenker, S.: Search and Replication in Unstructured Peer-to-Peer Networks. In: Proceedings of the 16th annual ACM International Conference on Supercomputing (ICS 2002), New York, USA (2002)
21. Thorn, T., Fennestad, M., Baumann, A.: A Distributed, Value-Oriented XML Store, Master's Thesis IT, University of Copenhagen (2002)
22. Traversat, B., Abdelaziz, M., Duigou, M., Hugly, J., Pouyoul, E., Yeager, B.: Project JXTA Virtual Network Sun Microsystems Inc. (October 28, 2002), `http://www.jxta.org`