

A generic P2P architecture for anonymous services.

Vincent Naessens

Bart De Decker

Bart De Win

Report CW 370, September 2004



Katholieke Universiteit Leuven
Department of Computer Science

Celestijnenlaan 200A – B-3001 Heverlee (Belgium)

A generic P2P architecture for anonymous services.

Vincent Naessens

Bart De Decker

Bart De Win

Report CW 370, September 2004

Department of Computer Science, K.U.Leuven

Abstract

Peer-to-peer systems exhibit, due to their decentralized structure, interesting anonymity properties and are therefore often used as a supporting service for anonymous applications. Unfortunately, there is currently no overall framework that captures the various peer-to-peer models and their specific features to guide the development of anonymous applications. This paper presents a generic and flexible architecture for peer-to-peer systems that separates applications and supporting services. The architecture is structured in different layers and each layer comprises a set of generic and reusable building blocks. To demonstrate the power of the architecture, we have implemented an anonymous chat application on top of the architecture.

Keywords : P2P, Anonymity.

1 Introduction

A peer-to-peer (P2P) system is a distributed architecture where, in order to achieve a common goal, many identical participants (peers) cooperate using a decentralized paradigm. The symmetric relationship between the participating computers distinguishes this architecture from other, typically asymmetric systems, like client-server systems. In the context of anonymous services, one of the important advantages of P2P systems –and probably the most important reason for their existence– is their inherent potential to provide anonymity. The symmetric organization of the peers and the lack of fixed dependencies between them make it possible to distribute trust and accountability in the system. Different anonymity characteristics can be achieved in this way, among others: sender or receiver anonymity, content anonymity, query anonymity, and so forth.

Unfortunately, developing applications that are based on P2P services is hard because of several reasons. First, current P2P systems are often developed ad-hoc and their implementation typically depends on the particular application at hand. As a result, it is unclear how P2P services should be structured and used in new applications. Second, P2P services and applications are typically built as one monolithic system, which clearly hinders their reuse for other applications. Third, many P2P systems are currently available and it is difficult to choose for a particular application the most appropriate one (e.g., the one with the best anonymity properties). Finally, when considering the combination of different P2P services, the cooperation between them is unclear because of the lack of a common foundation. Our research tackles the first two problems.

In this paper, we present a generic P2P architecture for anonymous services. The primary contribution of this architecture is a common foundation for the development of P2P systems. One of the core ideas of the architecture is the separation of applications and P2P services. For the construction of the architecture, we explicitly focused on two characteristics: generality and flexibility. The former ensures that different applications and P2P services fit into the architecture, while the latter enables the replacement of (parts of) applications and services independently. For this purpose, we have defined several building block abstractions that must be re-implemented for different P2P services and that can be reused easily.

The rest of this paper is structured as follows. The next section gives an overview of the most important application domains and routing models. In section 3, we present the P2P architecture. The architecture is independent of the application domain and routing model. The fundamental building blocks at each layer in the architecture are discussed in section 4. A chat application is built on top of the P2P architecture. The chat application is discussed in section 5, after which the advantages/disadvantages of the approach will be discussed. The paper ends with a conclusion.

2 Classification of P2P systems

The main goal of this section is to give an overview of existing P2P systems. First, current P2P systems are classified into different application domains (horizontal classification). Second, three established P2P models are described (vertical classification). For detailed information, we refer to [6].

2.1 Taxonomy of P2P applications

Distributed computing. Distributed computing achieves processing scalability by aggregating the resources of a large number of individual Internet PCs. Typically, distributed computing requires applications that are run in a proprietary way by a central controller. One of the first widely visible distributed computing events occurred in January 1999, where distributed.net, with the help of several tens of thousands of Internet computers, broke the RSA challenge in less than 24 hours using a distributed computing approach. This made people realize how much processing power can be available from idle Internet PCs. Examples of such systems are Beowulf [7], MOSIX [5] and Condor [17].

File sharing. Content storage and exchange is one of the areas where P2P technology has been very successful. Multimedia content, for instance, inherently requires large files. Napster [18] and Gnutella [15] have been used by Internet users to circumvent bandwidth limitations that make large file transfers unacceptable with classical mechanisms. Other examples are Oceanstore [14], Chord [13] and Publius [20].

Collaboration. Collaborative P2P applications aim at application level collaboration between users. The inherent ad-hoc nature of P2P technology makes it a good fit for user-level collaborative applications. These applications range from instant messaging and chat, to online games, and shared applications that can be used in business, educational and home environments. Examples are Kazaa [3] and Groove [11].

Platforms. Operating systems are becoming decreasingly relevant as environments for applications. Middleware solutions [4, 2], such as Java Virtual Machines, or Web browsers and servers are the dominant environment that is of interest to users as well as to developers of applications. In that regard, it is likely that future systems will increasingly depend on some other sort of platform that will be a common denominator for users and services connected to the Web or in an ad-hoc network.

2.2 Taxonomy of P2P models

Centralized directory model. This model was made popular by Napster [18]. The peers of the community connect to a central directory where they publish information about the content they offer for sharing. Upon request from a peer, the central index will match the request with the best peer in its directory. The best peer could be the one that is cheapest, fastest, or the most available, depending on the user needs. Then a file exchange will occur directly between the two peers. This model requires some managed infrastructure (the directory server), which hosts information about all participants in the community.

Flooded requests model. The flooding model is a decentralized P2P model which requires no advertisement of shared resources. Instead, each request of a peer is flooded (broadcast) to directly connected peers, which themselves flood their peers etc., until the request is answered or until a maximum number of flooding steps is reached. This model, for instance used by Gnutella [15], requires a lot of network bandwidth, and hence does not prove to be very scalable, but it is efficient in limited communities such as a company network.

Document routing model. The document routing model, used by FreeNet [12], is the most recent approach. Each peer of the network is assigned a random identifier and each peer also knows a given number of peers. When a document is published (shared) on such a system, an identifier is assigned to the document based on a hash of the document's contents and its name. Each peer will then route the document towards the peer with an identifier that is most similar to the document's identifier. This process is repeated until the nearest peer's identifier is identical to the current peer's identifier. A document lookup follows a similar procedure.

3 Overview of the architecture

Our generic architecture consists of three layers: the application layer, the peer layer and the connection layer. Between each layer, a fixed API ensures low coupling between the layers. Besides the layered design, an anonymity library contains reusable basic blocks that implement algorithms needed to provide anonymity services. Peer level blocks and connection level blocks are composed of these basic blocks in order to meet the anonymity requirements.

The *peer layer* contains building blocks that can be identified in most P2P systems. These blocks are defined in such a way that they can be used to implement these P2P platforms.

The *connection layer* contains building blocks that implement different types

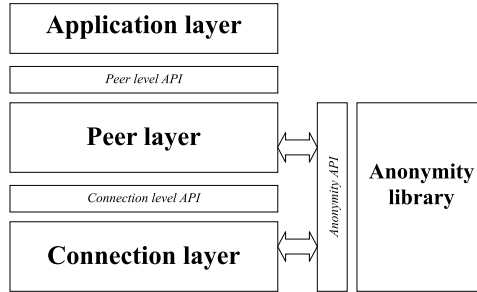


Figure 1: Overview of the P2P Architecture.

of connections such as (anonymous) synchronous connections and (anonymous) asynchronous connections. They provide an interface to the peer layer. By separating the peer level services and connection level services, we try to achieve a flexible and modular design.

The *anonymity library* contains basic blocks needed to achieve anonymity services such as encryption, reordering of messages, splitting and padding of data, etc. The API provided by this basic blocks can be called by the peer layer blocks and the connection layer blocks. By providing an API, the peer layer and the connection layer consider the basic blocks as black boxes. If the peer layer wants to reorder data, it uses a reordering basic block by calling the reorder method from the API without worrying about the implementation of the reordering algorithm.

To achieve a certain type and degree of anonymity, a proper composition of basic building blocks should be made by peer layer blocks and connection layer blocks. If the connection layer support anonymous connections, many anonymity properties are realised in the connection layer and less should be provided at peer layer. On the other hand, if the connection layer does not support anonymous connections, the components at peer layer are fully responsible for hiding the identity of peers towards other peers.

4 Implementation

In this section, we elaborate on the contents of each layer. First, we discuss the basic building blocks that are part of the anonymity library. Second, the identifier and locator concepts are explained. These concepts are used in the P2P system throughout the different layers. Peer level blocks are discussed in the third section. Finally, connection level building blocks are discussed.

4.1 Anonymity library

Many techniques are reused in systems that provide an anonymous service. For instance, a wide variety of mixes exists currently. Many of them use a reordering algorithm to prevent message tracing. The anonymity library contains these basic building blocks for anonymity. The building blocks are split in two categories: building blocks for changing appearance and building blocks for changing the message flow. We give some examples of blocks that belong to each of these categories. They are discussed in detail in [1].

- Changing appearance. The intention of these blocks is to change the appearance of messages. Encryption, padding, information substitution and compression belong to these category of basic blocks.
- Changing flow. These blocks change the message flow. Examples of these blocks are reordering, delaying, inserting dummy traffic, filtering, ...

When providing a service, a good composition of building blocks is essential to meet the anonymity requirements. It is hard to determine whether a composition is good or not. There are some rules of thumb, among others: appearance changing blocks will necessarily be used in combination with blocks that change the message flow. They must be composed in the right order. Basic blocks are composed both at peer layer and at connection layer. If the connection layer already supports reordering of messages, it is superfluous for the peer layer to also reorder the messages. If the connection layer provides anonymous connections (by making a good connection level composition of basic blocks or by using an existing connection level system such as an onion routing system), less blocks are needed at peer layer. However, building blocks can appear at both layers. Encryption may be necessary at connection as well as peer level. Information can be substituted at peer layer and at connection layer.

4.2 Identifiers - locators

Applications must be developed independently of the particular P2P system and vice versa. To support this requirement, we introduced nodes and units. We outline the difference between these two concepts and associate identifiers and locators with each of these concepts.

4.2.1 Node

Node - NodeIdentifier. A node is the fundamental component of a P2P system. Each node is associated with a communication address (e.g., IP address and port) where incoming calls can be sent to. Each node in a P2P system also has a unique nodeIdentifier which represents the node. In

pure P2P systems, every node is a peer and vice versa. In hybrid systems, additional (assisting) components can also be nodes. For instance, central servers or directory servers are also nodes.

NodeLocator. A nodeLocator contains information to contact a node in the P2P system. In P2P systems without anonymity, a locator can be an IP address and port. For anonymous nodes, a locator can be an anonymous path to the node. An (reply) onion is an example of a nodeLocator. Remark that a nodeLocator refers to one single node in the system. It is also important to note that one node can have different nodeLocators. They represent different ways to contact the node.

4.2.2 Unit

Unit - UnitIdentifier. A unit represents a resource in a P2P application. A unitIdentifier is a unique identifier of a unit. The concept of a unit depends on the application domain: in file storage systems, each file is a unit, while a chat session is a unit in chat applications, a process is a unit in distributed computing, and so forth. We distinguish between single units and composed units.

- **Single unit.** A single unit is a unit that is atomic and can not be split any further. For instance, a file that is stored at one node is a single unit. A single unit is located at a particular place within a node.
- **Composed unit.** A unit can also be a composition of other units. If a file is split in different parts that are stored at different nodes, these parts are also units. A chat session is also composed of different units that participate in the session. A process created for distributed computing is composed of threads where the computation takes place. Hence, each thread of the process also has a unique unitIdentifier.

UnitLocator. A unitLocator is an information structure to retrieve/contact a single unit in the P2P system. The location of a unit consists of a node and a place of the unit within that particular node. In systems without anonymity, a nodeLocator could consist of the address of the node and an offset that points to the place of the unit within that node. If the location of a unit must be anonymous, the nodeLocator can be an encrypted path to the node and the place of the unit within that node. Files stored at an anonymous node can be retrieved through their unitLocator that does not reveal information about the physical location of the unit. Another example is an encrypted url (used by TAZ servers [9]). TAZ servers keep encrypted urls (unitLocators) that are used to retrieve anonymous webpages (units).

4.3 Peer level building blocks

In this section, we give an overview of building blocks that are present in most peer platforms and outline the functionality of each block. The building blocks are application independent. They can be used for file sharing, distributed computing, collaboration, ...

We divide the presented blocks in three main categories: a registration block, management blocks (storage manager, locator manager and reputation manager) and communication blocks (push, accept, pull, reply and forward). Only the communication blocks call the connection level API.

4.3.1 Registration

A registration block registers peers and units in the P2P system. When a peer initializes, a new register block is created. The implementation of this block depends on the P2P model. The peer can be registered at a central server or at a neighbouring peer. It looks up the correct `nodeIdentifier` (central server or neighbouring node) and uses a push block (see further) to send the registration request to the right node. After registration, the application can use this block to register units.

4.3.2 Information Management

Storage manager. This block manages units that are stored at a node. A typical example of a unit is a file or part of a file that is stored. A storage manager is created when a node starts up. Each node has one storage manager. There are methods to add units, remove units and retrieve units. Remark that a stored unit is always associated with a `unitIdentifier`.

Locator manager. A locator manager has two main functionalities. First, this block creates locators for nodes and units. Second, it keeps a mapping table. `NodeIdentifiers` and `unitIdentifiers` for single units are mapped to locators. `UnitIdentifiers` for composed units are mapped to a list of other `unitIdentifiers` that represent the subunits. Just like storage managers, a locator manager is created when a node initializes. New locators are added to the mapping table if other nodes share locators with that node or if the locator manager creates new locators. Remark that only identifiers are made visible towards the application. The application needs these identifiers to contact/retrieve a unit.

Reputation manager. While the advantages of a P2P system are obvious, the distributed nature and the independence of the different peers allow for misbehaving entities in the system. A typical example of such misbehavior is the phenomenon of freeloading in a file sharing system, where certain peers communicate predominantly in the incoming direction and as such

profit from all available units without providing new units. To overcome this kind of problems, different types of reputation models are being used in P2P implementations.

In a very decentralized model, every node keeps records for every peer it has had contact with in the past and its behavior (what actions does it do, does it act as promised, ...). Gossiping or other techniques may be used to distribute this information through the system. In a more centralized model, this data is kept in a centralized repository and can be consulted and updated by the peers.

4.3.3 Communication

The push block and the accept block are the two basic blocks for communication. The other blocks (pull, reply, forward) are actually a combination of these two basic blocks.

Push. This block sends data to other nodes in the system. A push block is created the first time data is pushed. Before data is actually sent over a connection, it can be reordered, delayed, encrypted, ... Thus, a push block consists of a composition of basic blocks that are in the anonymity library. A push block is associated with the `nodeIdentifier` of the destination node. If a connection exists to the node, the data are sent over that connection. Otherwise, a new connection is created. A push block can be used by the application or other building blocks.

Sometimes, not all data should be pushed the same way. For instance, a file can be split in several subunits that are sent to several locations. If a chat message is pushed to another node in the system, the message should not be split. A different push block is created when data is handled in a different way before it is actually sent.

Accept. An accept block receives data that arrives at a peer. Data can be received from different connections. The retrieved information is passed to a receiver object. For instance, data can be passed to an application object, a reputation manager, a storage manager, etc... Remark that each receiver object must implement a common receiver interface.

Pull. This building block is required when a peer requests information. For instance, a peer can pull a file that is stored remotely. A peer can also pull information from a message board or pull a locator from a central server. Before the request is sent, it can be reordered, encrypted, etc... So, a pull block also consists of basic blocks from the anonymity library. Just like a push block, a pull block is created the first time data is pulled. Different entities can pull data. For instance, the application can pull a file. A reputation block can also pull reputation information from other nodes.

A pull block is actually a combination of a push block and an accept block. The push block sends the request and the accept block retrieves the answer to that request. Different implementations are possible. We suggest two alternatives:

- Synchronous pull. The request and the answer are sent over the same connection. The connection must be open until the the pull block receives an answer.
- Asynchronous pull. The request and the answer are sent over different connections. In this case, the pull block must know which answer corresponds to which request.

Reply. A reply block is composed of an accept block and a push block. The request is received by an accept block, handled by the reply block and the answer is returned by a push block. The reply block can call methods on other blocks to handle the request. For instance, a reply block can retrieve a file from the storage block. The reply block can also retrieve information from the reputation manager.

Replies can be sent over the same connection (synchronous reply) or over another connection (asynchronous reply).

Forward. A forward block just forwards data (both requests and answers). An instance of this block is created the first time data should be forwarded by a node. Data is retrieved, handled by a forward block (encrypted, reordered ...) and forwarded towards another node.

We distinguish between a simple forward block and a bidirectional forward block. A simple forward block consists of an accept block and a push block. The information that the block receives is just forwarded. A bidirectional forward is a composition of a reply block and a pull block. A node sends a request to a forward block. That block must forward the request, retrieve the answer and send the answer to the original node.

4.4 Connection level building blocks

We divide connections in two caterogies: anonymous connections and non-anonymous connections. A connection is anonymous if at least sender or recipient is anonymous towards the other party . A connection is non-anonymous if the endpoints reveal information about their identities.

Existing implementations of anonymous connections can be plugged in at this level such as onion routing [10], Crowds [19], Hordes [16] ... However, the framework also provides sufficient support for the designer to make a particular implementation of such an anonymous connection system. A static method is provided to set up connections. This method needs a `nodeLocator`

(for instance, an onion) to set up a connection to another node in the peer system.

4.5 Comparison with other systems

In this section, we compare the architecture with three other systems. We show that the architecture is flexible enough to extend or implement existing systems.

Onion routing. Onion routing [10] is an example of an anonymous connection system. Proxies already exist for electronic mail and web browsing. If a proxy is written for P2P systems, onion routing can be inserted in the architecture at connection level. The locator manager keeps a table that maps identifiers to reply onions (locators). If an application wants to retrieve a unit, it passes the right unitIdentifier to a pull block. The pull block retrieves the corresponding reply onion from the locator manager and passes this information to the connection layer. The request is then sent over an anonymous connection.

CROWDS. Crowds [19] is also an anonymous connection system. Crowds is in fact only designed for web browsing. However, it can be used for anonymous connections. Messages are forwarded to other crowd members. The system can be inserted at peer level or at connection level. In the latter case, participants of the P2P system are not the only members of the crowd. Other members can join the crowd (for providing the anonymous connection system).

TARZAN. TARZAN [8] is an anonymous connection system that uses peers as mixes. The set of mixes varies dynamically as mixes enter or leave the system. Every user can start up a mix, but users who don't start up a node can nevertheless send messages through other mixes in the system.

5 A chat application

5.1 Overview of the application

To demonstrate the power of the architecture described in the previous section, a chat application was built on top of the P2P architecture. We first give an overview of the core functionality of the application and its anonymity requirements. Then, we discuss five use cases. The use cases illustrate the interaction of the building blocks in this particular implementation. We discuss what happens when a node registers to the system, when a chat session is set up, how a node can subscribe to a session, how messages are exchanged between nodes and how files are stored in the peer system.

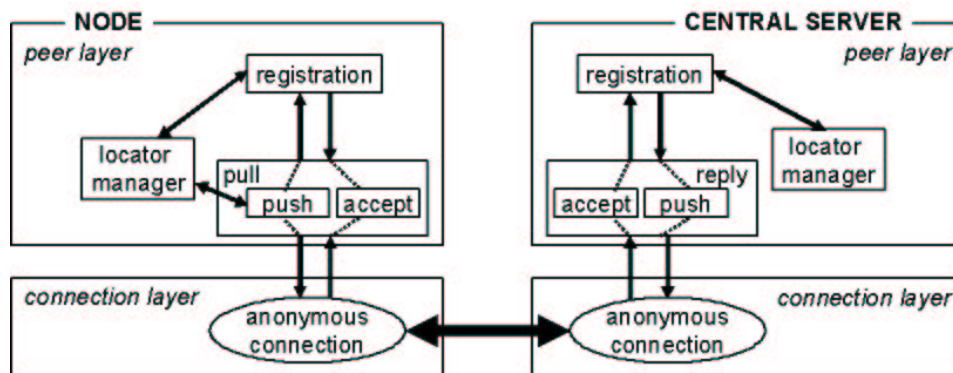


Figure 2: Node registration in the peer system.

Core functionality. The chat application comprises three central concepts: chat sessions, participants of a chat session and files that are related to a chat session. We discuss these units:

- **Chat session.** A chat session is a composed unit administered at a node. A chat session unit consists of a set of participants and files that are associated with the chat session. The location of the chat session must be anonymous.
- **Participant.** A participant is a member of a chat session. Participants are also anonymous.
- **File.** Participants can post files and make them part of the chat session. For instance, if a chat session is set up to discuss a medical problem, a participant can post an interesting paper about that topic. The files are posted anonymously and can only be retrieved by participants of that chat session.

Registration of a node. Every client that wants to participate in the P2P system launches a node. We use the central server model. During the registration phase, the node sends a registration request to the central server over an anonymous connection. The new node gives its nodeLocator to the central server and the server returns a unique identifier for that node. The central server stores the nodeLocator and the nodeIdentifier in its locator manager and informs the new user about the other nodes. It also informs the other nodes about the new node over an anonymous connection (based on the nodeLocators).

Setting up a chat session. If a node sets up a new chat session, it pushes a unitLocator to the central server over an anonymous connection.

The central server returns a `unitIdentifier`. It also sends the `unitIdentifier` and the `unitLocator` to each node over an anonymous connection. The nodes store this information in their locator manager. This way, the other nodes are informed about the new unit. They can use this locator to subscribe to the chat session.

Subscribing to a session. If a new user subscribes to the session, it creates a participant unit and a corresponding `unitLocator`. It pushes a participation request (that includes the `unitIdentifier` and `unitLocator`) to the chat session unit by using the session's `unitLocator`. The chat session unit informs the new user about the other participants and files in the session. It also informs the other participants about the new participant by sending the new `unitLocator` to them. This way, the new participant can be contacted directly by the others.

Exchanging messages between nodes. Participants that have subscribed to a chat session, can exchange messages. At application level, the node composes a message and adds an identifier. This identifier is mapped to a locator at peer level. If there is some open connection to that node, the message is pushed over that connection. Otherwise, a new connection is created.

Storage of information in the peer system. If a node wants to post a file into the peer system, the push block asks the locator manager for a locator. Then, the file is sent to a different node. The accept block of the receiving node passes the unit to its storage manager. The storage manager creates a new entry in its mapping table and stores the file. The locator for the file is sent to the corresponding chat session. This unit pushes the locator for the file to the participants of the chat session. Each node that wants to retrieve the file uses the `unitlocator`.

6 Discussion

This approach is a work in progress and the architecture presented in this paper is a first stable iteration in this process. We are convinced that different P2P models and applications can be implemented using the architecture. Although the framework is not fully mature, yet it already gives a number of advantages:

- The *implementor of a P2P system* should be encouraged to use the architecture for several reasons. First, it provides placeholders for many P2P models. Second, the number of concepts and building blocks is minimal. Third, the layered structure of the architecture separates peer level services and other supporting services.

- By separating the application from the peer platform, an *application programmer* can build many applications on top of the same peer system.

Although splitting services is a noble goal, separation of services is often very difficult. For instance, if a node pushes data to the same node at different times, an open connection to that node can be reused or another connection can be set up for each data transfer. However, the latter case has less anonymity strength. Longstanding anonymous connections provide better anonymity properties. We refer to the discussion about static versus dynamic paths in Crowds [19].

The implementation of the building blocks also depends on the functional requirements and the anonymity requirements of the application. Therefore, the application needs control over the other layers (peer layer and connection layer) in the architecture. To achieve this goal, the definition of a meta interface is appropriate. We illustrate the effect of application requirements on the composition of building blocks at peer layer and connection layer:

- For anonymous communication, random delaying of data is often used to mix traffic on a communication. However, if an application requires real-time communication (e.g., a chat application), delaying is not a good option. Hence, it can not be inserted as a building block for anonymous communication. Reordering data and inserting dummy traffic are good alternatives and can be combined. On the other hand, if the P2P system is used for anonymous file storage, delaying file transfer at an intermediate node is a good option.
- The behaviour of a P2P system also depends on the application. For instance, if the application requires a lot of anonymous communication and mixes are implemented at peer level, reordering messages at this level is a good mechanism. However, dummy traffic is more efficient when the amount of traffic is low.
- The lifetime of a peer can also have an impact on the implementation of locators. If a unitLocator routes through a fixed path of peers, the unit can not be retrieved when a peer on the path leaves the system.

7 Conclusion

This paper presents a generic and flexible P2P architecture. In order to achieve these goals, we suggest to abstract common P2P concepts (such as node, unit and locator) and define building blocks that are present in many P2P systems. Another important advantage of this approach is the separation of the application and the rest of the P2P system. This considerably simplifies the design and implementation of the P2P application. By means

of the example of a chat application, we demonstrate the feasibility of this approach.

Still, we think that a complete separation of the application from the rest of the system is not appropriate. The implementation of building blocks at peer layer also depends on the functional and anonymity requirements of the application. We would like to focus our research in the future on this issue, for example by defining a meta interface that passes the application requirements to the underlying layers.

References

- [1] Anonymity and privacy in electronic services: Technologies. <https://www.cosic.esat.kuleuven.ac.be/apes/>.
- [2] The jxta homepage. <http://www.jxta.org/>.
- [3] The kazaa homepage. <http://www.kazaa.com/>.
- [4] .net passport technical overview. Technical report, MICROSOFT, 2001.
- [5] A. Barak and A. Litman. Mos: a multicomputer distributed operating system. *Software - Practice and Experience*, 15(8):725–737, 1985.
- [6] Rajan Lukose Kiran Nagaraja1 Jim Pruyne Bruno Richard Sami Rollins Zhichen Xu Dejan S. Milojevic, Vana Kalogeraki. Peer-to-peer computing. Technical report, HP Laboratories Palo Alto, 2002.
- [7] D. Savarese J.E. Dorband U.A. Ranawak D.J. Becker, T. Sterling. Beowulf: A parallel workstation for scientific computation. In *Proceedings of the International Conference on Parallel Processing*, 1995.
- [8] Michael J. Freedman, Emil Sit, Josh Cates, and Robert Morris. Introducing tarzan, a peer-to-peer anonymizing network layer. In *International Workshop on Peer to Peer Systems*, LNCS, March 2002.
- [9] I. Goldberg and D. Wagner. Taz servers and the rewebber network: Enabling anonymous publishing on the world wide web. *First Monday*, 1998.
- [10] Michael G.Reed, Paul F.Syverson, and David M. Goldschlag. Anonymous connections and onion routing. *IEEE Journal on Selected Areas in Communications*, 16(4):482–494, May 1998. Special issue on Copyright and Privacy Protection.
- [11] Groove.net. <http://www.groove.net/>.
- [12] B. Wiley I. Clarke, O. Sandberg and T.W. Hong. Freenet: A distributed anonymous information storage and retrieval system. In *Designing Privacy Enhancing Technologies: International Workshop on Design Issues in Anonymity and Unobservability*. Springer, 1999.
- [13] D. Karger F. Kaashoek I. Stoica, R. Morris and H. Balakrishnan. Chord: A scalable peer-to-peer lookup service for internet applications. In *Proceedings of the SIGCOMM*, 2001.
- [14] Y. Chen S. Czerwinski P. Eaton D. Geels R. Gummadi R. Rhea H. Weatherspoon W. Weimer C. Wells J. Kubiawics, D. Bindel and

- B. Zhao. Oceanstore: An architecture for global-scale persistent storage. In *Proceedings of the Ninth international Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS 2000)*, 2000.
- [15] Gene Kan. *Peer-to-Peer: Harnessing the Power of Disruptive Technologies*, chapter 8, pages 94–122. O’Reilly, March 2001.
- [16] Brian Neil Levine. Hordes: A multicast based protocol for anonymity.
- [17] M. Litzkow and M. Solomon. Supporting checkpointing and process migration outside the unix kernel. In *Proceedings of the USENIX Winter Conference*, 1992.
- [18] Napster. <http://www.napster.com/>.
- [19] Michael K. Reiter and Aviel D. Rubin. Crowds: anonymity for Web transactions. *ACM Transactions on Information and System Security*, 1(1):66–92, 1998.
- [20] Marc Waldman, Aviel D. Rubin, and Lorrie Faith Cranor. Publius: A robust, tamper-evident, censorship-resistant, web publishing system. In *Proc. 9th USENIX Security Symposium*, August 2000.