Accordion

Anonymous Peer-to-Peer File Storage

Morten Franck

IT University of Copenhagen
skyfer@itu.dk

Peter Gath Hansen

IT University of Copenhagen
gath@itu.dk

Abstract

This thesis presents different techniques for achieving different kinds of anonymity in computer networks. In the first part of the thesis, the techniques are described and their anonymity properties are analyzed. Also, in the first part a survey of various existing peer-to-peer, anonymizing systems is presented.

In the second part of the thesis, a design and a prototype implementation of an anonymous, low-latency, peer-to-peer file storage system, Accordion, built on practical and efficient DC-networks is presented and its anonymity properties analyzed. The system offers sender anonymity for publishers of files and for the nodes in the network that store the files. The prototype is tested in different ways to increase confidence in the design.

Supervisor:
Henning Niss
IT University of Copenhagen
hniss@itu.dk

June 1, 2005 (Semester F2005)

Contents

Chapter 1

Introduction

This project is about anonymous communication in computer networks, e.g. the Internet. The need for anonymity is becoming more and more apparent as the different uses of and the number of users on the Internet grows rapidly. We believe the main motivation for anonymous communication is free speech. The political and ethical implications of anonymity in computer networks have been discussed in e.g. Free Haven [Din], The Eternity Service [And96] and YÅPS [Boe03], but only the technical issues surrounding anonymity in the Internet will be discussed in this report.

Most work on anonymity is based on one paper written by David Chaum in 1981 [Cha81]. In this paper, Chaum presents a technique called *mix-networks* that allow for a user A to anonymously send a messsage through the mix-network to another user B (and for B to respond anonymously to A). This technique has been widely used for different applications, such as web browsing and email, publication/file storage systems, and frameworks for providing generic, anonymous communication over TCP/IP.

Another technique, the Dining Cryptographers-network, *DC-network*, presented by Chaum in 1988 [Cha88] has receieved less attention although the technique can be *unconditionally* secure (as described in section 2.3). The reason for the low interest in DC-networks is most likely due to the higher network cost. Chaum himself, however, suggests methods for more practical DC-networks which weakens the anonymity properties but achieves stronger anonymity properties than mix-networks.

In this thesis we present *Accordion*, a low-latency, anonymous, peer-to-peer file storage system based on practical and efficient DC-networks. Accordion thus achieves strong anonymity properties and low-latency retrievals of files. The system works in a true peer-to-peer setting and is based on an existing routing layer. The focus when designing Accordion has been more on anonymity and less on e.g. functionality aspects.

In the same area as anonymous file storage systems exists anonymous publication systems. There is no well-defined distinction of the two but we see anonymous publication focusing more on only documents and resisting censorship on these and file storage as a generic service to store files of all types but without necessarily providing censorship-resistance.

Thus the focus of this work will be on anonymity.

The rest of this chapter presents some terminology that will be used in the rest of the report. A common terminology types eases reading and enables direct comparison of techniques.

Chapter 2 first presents some basic techniques for achieving different types of anonymity under different assumptions. Second, a brief survey of anonymity in peer-to-peer systems is presented. Finally, the anonymity properties of the techniques are summarized.

Based on the description of techniques and the survey of systems, Chapter 3 presents the design of Accordion, including its anonymity properties and a comparison with related work.

Chapter 4 gives an overview of the implementation of a proof-of-concept application of Accordion and presents tests to convince the reader of the correctness of the design.

Chapter 5 concludes and presents some challenges in anonymous communication in computer networks.

1.1 Terminology

Pfitzmann and Köhntopp[KP01] define *anonymity* as:

Anonymity is the state of being not identifiable within a set of subjects, the anonymity set.

The anonymity set is the set of all possible subjects who might cause an action.

This work does not attempt to give a formal model of anonymity properties of different techniques or systems. Serjantov [Ser04] presents a model based on entropy for measuring anonymity in certain types of systems (mix-networks). Incorporating such a model into this work is, however, out of scope.

1.1.1 Anonymity Types

Most existing literature agrees on at least two types of anonymity (or "actions"): sender and recipient anonymity. *Sender* anonymity is defined as:

Sender anonymity is the property that a particular message is not linkable to any sender and that to a particular sender, no message is linkable.

Recipient anonymity is defined equivalently. The sender is the only entity that knows the identity of the recipient.

In the case that a node's sender anonymity has been broken and another node's recipient anonymity has been broken, *unlinkability* may still be achieved for the message sent by the sender and received by the recipient:

If a node A sends a message M and a node B receives M but it cannot be determined whether the

message sent by A is the same as the message received by B, then A and B achieve unlinkability for M.

A type of anonymity used in bi-directional communication is *responder* anonymity:

The recipient of a message M does not know the identity of the sender of M to which it replies.

A stronger type of anonymity is *unobservability*:

Unobservability is the state of sending or receiving a particular message being indistinguishable from sending or receiving any message at all.

This means that "real" messages are not discernible from other messages, e.g. "dummy messages" (also known as "white noise"). In this context a "message" refers to an application level message and not e.g an IP-packet.

By this definition, it follows that if unobservability is achieved then so is anonymity because "a particular message" cannot be distinguished from a dummy message. Unlinkability follows since although it can be determined that the packets sent by A are the same as the packets received by B, it cannot be determined what constitutes a "message" and thereby that A sent a message. Hence, there is no message to link.

1.1.2 Adversary Types

Adversaries are entities that in some way try to break the anonymity of a communication, ie. identify the endpoints of the communication, establish a link between a message sent by one node and received by another (ie. break unlinkability) or determine that a message is being sent (ie. break unobservability). Adversaries can take on many different forms ranging from governments or major corporations with almost unlimited resources to small companies or individuals. An adversary can thus either be considered computationally *limited* or *unlimited*. An anonymizing system is *unconditionally* (or information-theoretic) secure if not even an computationally unlimited adversary can break the anonymity of the system and only *computationally* secure if either a limited or an unlimited adversary is able to break the anonymity of the system.

In this work two computationally limited adversary types will be considered (see below). It is assumed that none of these are able to break standard encryption algorithms such as 3-DES, AES or RSA or secret-key exchange algorithms such as Diffie-Hellmann by performing e.g. cryptanalytic or man-in-the-middle (MiM) attacks on these.

Passive A passive adversary has the ability to observe traffic on both the incoming and outgoing links of some nodes in the total set of nodes.

Active An active adversary has the ability to block or insert traffic on some links and to compromise some nodes.

Each time an adversary is mentioned in the rest of this report, the abilities of the specific

adversary will be specified, i.e., which links or nodes the adversary is able to observe traffic on or compromise, respectively.

1.1.3 Types of Attacks

Just as adversaries can take on many forms so can the types of attacks that an anonymizing system must be able to withstand. Attacks on such systems can be divided into the following categories:

Technical. Performing attacks by e.g. observing or modifying network traffic or compromising nodes.

Social. As mentioned in previous literature on the subject of anonymity and censorship-resistance, e.g., FreeHaven [Din] people could claim that by taking part in such a network participants are helping terrorists communicate or people in downloading questionable or illegal material and apply social pressure on the people that run system nodes.

Political/legal. Governments or courts could decide to make it illegal to use or participate in the anonymizing system.

Physical. Nodes could be shut down or physically destroyed.

Some technical attacks where a passive adversary observes network traffic can be avoided by using *payload encryption*. Payload encryption refers to encryption of the payload of a TCP packet with a previously exchanged symmetric key exchanged using a secret-key exchange algorithm. Therefore, a passive adversary will be able to determine the destination application by observing the destination port number (because it appears in unencrypted form), but the same adversary will be unable to determine the contents of the packet.

It is noted, however, that merely hiding the *contents* of a message is *not* the same as protecting the identity of the sender or recipient of a message, but can protect against the adversary learning *what* anonymity is broken for (the adversary is thus only able to break anonymity for *some* message).

Only technical attacks will be considered in the following. Technical attacks on an anonymizing system can be divided into:

Attacks on functionality. In some way not following the intended protocol by e.g. not forwarding traffic, deleting data, etc.

Attacks on anonymity. Reducing the anonymity set of either the sender or recipient of a message by performing various attacks.

Functionality in an anonymizing system can be broken in many different ways and is of course dependent on the purpose of the system in question.

Pfitzmann and Köhntopp's anonymity set metric will not be used to measure the degree of anonymity an anonymizing system can achieve. Rather, specific attacks and their impacts on different anonymity types are described.

Chapter 2

Anonymity

A basic *anonymizing system* works as illustrated in Figure 2.1.

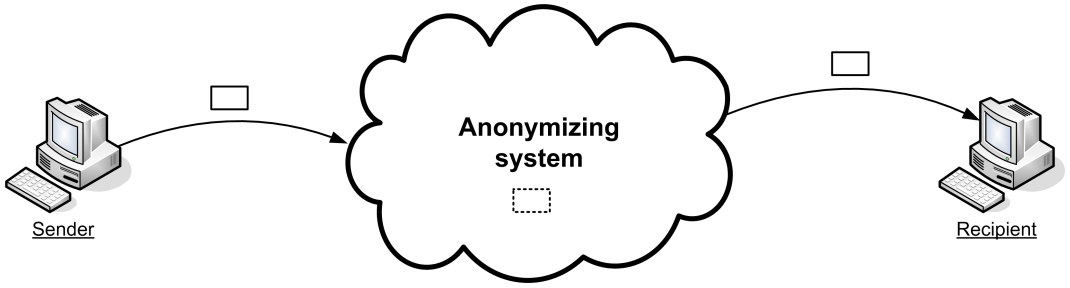

Figure 2.1: A basic anonymizing system where a user A sends a message to another user B through the system.

In this basic setting, A number of users – not part of the anonymizing system – send messages to and receive messages from the anonymizing system containing one or more system nodes responsible for the anonymization. In some settings users are part of the anonymizing system, ie. users are capable of taking part in anonymizing a message. This includes two techniques described in section 2.3 and 2.4, respectively, and peer-to-peer networks. A brief description of the effect of applying peer-to-peer technology to an anonymizing system is described in section 2.5.

The overall purpose of this chapter is to describe the ways in which a message can be sent anonymously by A (ie. sender anonymity), received anonymously by B (ie. recipient anonymity), how the sending of a message by A and the receival of the same message by B can remain unlinkable, how B can respond anonymously to A and finally how the sending or receival of a message can be unobservable. This chapter also serves as motivation for the design of an anonymous, peer-to-peer file storage system, Accordion, presented in Chapter 3.

Anonymizing systems can be designed using one or more of four techniques:

- Proxy (Single, forwarding node)
- Mix-networks
- DC-networks
- Broadcasting

In anonymizing systems built on the proxy or mix-network techniques, a basic attack is possible:

Wiretapper Attack A passive adversary observing messages on, *wiretapping*, both the incoming and outgoing link of a node and observing one or more messages on the outgoing link but no messages on the incoming link, will be able to *prove* that that node was the sender of the message because no other node could have sent it and thus break the node's sender anonymity. The attack can be performed even if (payload) encryption is used, but in that case the adversary will not be able to tell *what* the node is sending and will thus only break its sender anonymity for *some* message.

In anonymizing systems where users are not part of the system (proxy, mix-networks and non-peer-to-peer) another basic attack is possible:

Edge Attack a passive adversary observing traffic on the incoming link of the first node in the system or the outgoing link of the last node in the system or alternatively an active adversary controlling the first or the last node in the system (by e.g. having compromised it) will be able to break the sender or recipient anonymity of the node it receives a message from or sends a message to, merely by looking at the source or destination field in the message, respectively, and knowledge that this node is not a system node. As in the wiretapper attack, if encryption is used, the adversary will only know that it broke the node's anonymity for some message.

In general, a passive adversary performing either a wiretapper or an edge attack in an anonymizing system built on either the proxy or mix-network technique or a system where users are not part of the system, sender- and/or recipient anonymity is broken. The wiretapper attack, however, is hard to perform on arbitrary nodes because it requires that the adversary can observe messages on the local network the node in question is a part of.

2.1 Proxy (Single, forwarding node)

A simple anonymizing system is one in which a user A sends a message Msg through a single, forwarding node known as a *proxy*, or a *trusted third party*. This is illustrated in figure 2.2.

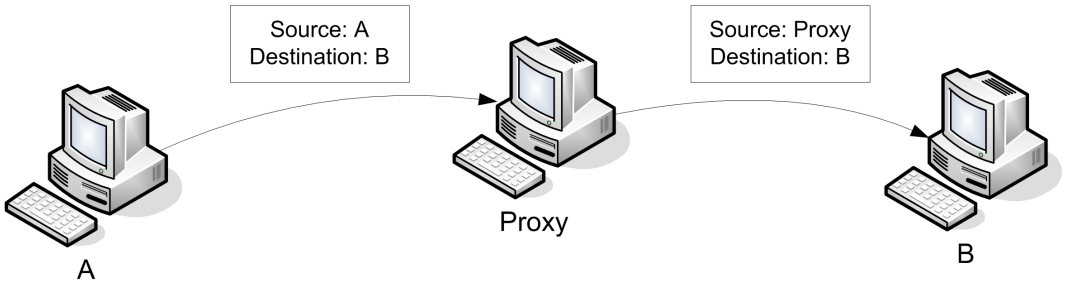

Figure 2.2: User A sends a message to B through a proxy.

The proxy replaces (or rewrites) the source address of *Msg* with its own address and *forwards* this message to *B*.

The technique described here can only be used for anonymous one-way communication, ie. *B* cannot send an anonymous reply to *A* without knowing the identity of *A* (ie. responder anonymity cannot be achieved). Mix-networks, described in section 2.2, aim to achieve this goal.

In figure 2.2 there is only a single, forwarding node. The technique can be extended to multiple, forwarding nodes in a number of ways, e.g. as described in section 2.6.1. The basic anonymity properties of a system built on a single or on multiple forwarding nodes are the same with the only difference being the number of connections to perform attacks on or nodes to compromise.

2.1.1 Anonymity Properties

In general, the proxy technique is vulnerable to both wiretapper and edge attacks.

Sender Anonymity *A* achieves sender anonymity because *B* will see the proxy's (and not *A*'s) address as the source address and hence cannot determine who the actual sender of the message was. This will be known as the *proxy argument*.

Recipient Anonymity *B* cannot achieve recipient anonymity because the proxy must know the identity of *B* to be able to send the message to *B*.

Unlinkability *A* and *B* cannot achieve unlinkability for *Msg* with respect to the proxy because it can see both *A*'s and *B*'s address in *Msg*. If *Msg* is not payload encrypted, a passive adversary can break unlinkability by performing an edge attack on the proxy.

If payload encryption is used, a passive adversary will still be able to break unlinkability because the bit pattern of incoming messages to the proxy will match the bit pattern of the outgoing message, thus making it possible to *correlated* incoming and outgoing traffic.

In figure 2.3, a passive adversary can use traffic characteristica to correlate incoming and outgoing traffic and will be able to gather *statistical evidence* to conclude—or at least strongly suspect—that A was the sender and B the recipient of a given message Msg and thus break their unlinkability for the message Msg. A goal is thus to hide traffic characteristica in such a way that an adversary will not be able to correlate incoming and outgoing messages. Mix-networks described in section 2.2 aim to achieve this goal.

Responder Anonymity A cannot achieve responder anonymity because the proxy must know the identity of A to be able to send a reply from B to A.

Unobservability Unobservability can be achieved using white noise on the link from A to the proxy and on the link from the proxy to B. However, a number of attacks on this approach is possible. These will be described in section 2.2.3.

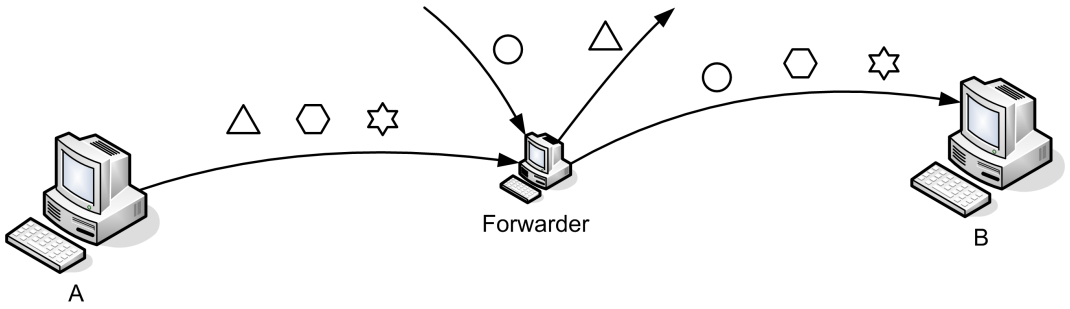

Figure 2.3: A passive adversary observing traffic on the forwarder's incoming and outgoing link will be able to suspect that A is communicating with B from message surface characteristics alone, because a disproportionately large number of messages (2 out of 3 in this example) are forwarded from A to B. Payload encryption doesn't prevent this attack.

2.2 Mix-networks

Mix-networks were designed in 1981 by David Chaum for "Untraceable Electronic Mail, Return Addresses, and Digital Pseudonyms" [Cha81]. A mix-network consists of a number of clients and one or more *mix-servers* each of which can be seen as a forwarding node with the added capabilities of batching and mixing incoming messages before forwarding them. To provide unlinkability against active adversaries that are able to compromise some (but not all) of mix-servers on a path and to achieve bi-directional communication, *layered* (nested) encryption is used.

A mix-system can either be arranged as a *mix-cascade* or as a *mix-network*. In mix-cascades all messages sent by users always travel through the same path of mix-servers. In mix-networks users choose the mix-server paths themselves. In the following we will only focus on mix-networks. For a comparison of mix-cascades and mix-networks see [Ser04].

An example mix-network is illustrated in Figure 2.4.

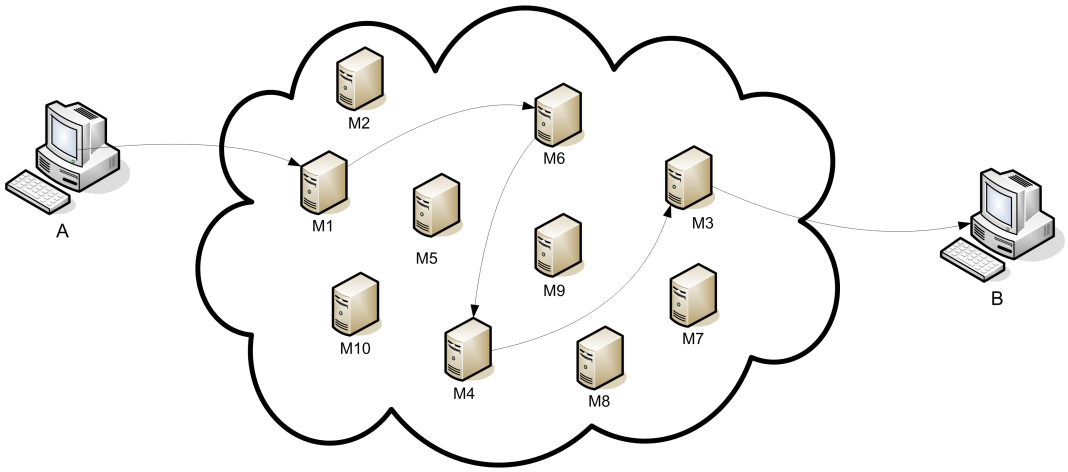

Figure 2.4: An example of a mix-network where A sends a message to B through mix-servers M_1, M_6, M_4, and M_3.

For a node A to anonymously send a message Msg through a mix-network to a node B, A must posses the public keys of the mix-servers through which it wants to send its message. For this purpose, each mix-server in the network possesses an asymmetric key pair from which all users can request the public key using some out-of-bands mechanism (ie. not using the mix-network itself to retrieve the keys). After choosing one or more mix-servers to form a path through the network, A prepares its message for routing thus:

1. Encrypt the message, Msg, with the public key of the last mix-server, M_n, on the path.
2. Encrypt the resulting message with the public key of mix-server $M_n - 1$, $M_n - 2$, etc., until finally encrypting the message with the public key of the first mix-server M_1 on the path.

In order for each mix-server on the path to know where to forward the decrypted message, in each step above, A also appends the address of the next hop on the path (a mix-server or, finally, the recipient, B) after each encryption.

As an example, in Figure 2.4, A might choose mix-servers M_1, M_6, M_4, M_3 to be on the path. In that case a message to a recipient, B, would look thus:

$$\mathrm{PUK}_{M_1}(\mathrm{PUK}_{M_6}(\mathrm{PUK}_{M_4}(\mathrm{PUK}_{M_3}(\mathrm{PUK}_B(Msg), \mathrm{ADR}_B), \mathrm{ADR}_{M_3}), \mathrm{ADR}_{M_4}), \mathrm{ADR}_{M_6})$$

where PUK_B is the public key of B, PUK_{M_x} and ADR_{M_x} is the public key and address of mix-server x, respectively, and Msg is the message that A wants to send to B.

Because of the way that layers of encryption are added to a message, a message is also sometimes known as an "onion". The encryption of Msg with the public key of B is to ensure that the last mix-server on the path will be able to read it.

When a mix-server receives a message, it first decrypts it with its private key ("peels off" a layer of encryption) and then performs some mixing algorithm. The study of mixing algorithms is an area of continous research, see for example [Ser04]. We focus on Chaum's original proposals. The key elements in Chaum's original mixing algorithm is to *delay* messages for a certain amount of time or until a certain number of messages have arrived and then to *reorder* the messages, e.g., lexicographically before passing them on to the next mix server on the path. Also, all outgoing messages from the mix-server are *padded* to the same size in order to prevent volume attacks by passive adversaries, as described later. The last mix-server on the path forwards the message to the intended recipient, in this case B, which is able to decrypt the message using its private key.

Batching and reordering messages (mixing) at mix-severs prevents passive adversaries from correlating the incoming and outgoing messages of a mix-server. If outgoing messages were not batched and reordered before being forwarded, the fact that the mix-server removes one layer from the onion (thus making it impossible to correlate the remaining onion with the original onion) would not prevent the adversary from correlating ingoing and outgoing messages simply by sequence, i.e., the first incoming message is also the first outgoing message, etc.

Before explaining the anonymity properties of mix-networks, the next section describes how a node A anonymously can receive a reply to a previously sent message. The message type used for this purpose are sometimes referred to as *reply onions* and it provides A with *responder anonymity*.

2.2.1 Anonymous Replies

This section describes how mix-networks allow a recipient to respond anonymously to a sender through the use of pseudonym keys.

For the sender A to send a message to which B can later reply without knowing the identity of A, A first generates an asymmetric key pair, PUK_{pseudo}, which does not reveal A's true identity but only provides a pseudonym to uniquely identify A. A then creates a reply onion for the recipient, B, from which it wishes to receive the anonymous reply.

Inside the reply onion is the real address of A, encrypted with the public key of the last mix-server on the path, i.e., the last mix-server between A and B. In the example below, only one mix-server, M, is used, but A may easily create a multi-layered reply onion as described previously. Also included in the reply onion is PUK_{pseudo} (so B can encrypt its reply to A) and the message itself:

$$\text{PUK}_M(\text{ADR}_A), \text{PUK}_{pseudo}, \text{ADR}_B \rightarrow M \rightarrow B$$

When B has received the message from A, it may reply by encrypting its reply message with PUK_{pseudo} and then choosing a path of one or more mix-servers such that the first mix is the one that can decrypt the message that contains the address of A, in this case mix-server M. This message is then sent back to A through M:

$$\text{PUK}_M(\text{ADR}_A), \text{PUK}_{pseudo}(\text{MSG}), \text{ADR}_A \rightarrow M \rightarrow A$$

When receiving the reply from B, mix-server M decrypts A's address with its own private key and is then able to forward the message directly to A, whose identity is thus kept secret from B.

2.2.2 Anonymity Properties

In general, the mix-network technique is vulnerable to both wiretapper and edge attacks and also a number of attacks (described next) that can break unlinkability.

Sender Anonymity A achieves sender anonymity due to the proxy argument.

Recipient Anonymity B achieves recipient anonymity because its address is hidden in the layered encryption.

Unlinkability A and B achieve unlinkability for a message because a passive adversary that intercepts the message from A to B on a link in between two mix-servers on the path because that message does not contain the address of A nor B in plain text.

Unlinkability can be achieved because the bit pattern of outgoing messages from a mix-server cannot be correlated with the bit pattern of incoming messages to the same server, because each mix-server peels off a layer of encryption and thus changes the bit pattern of the outgoing message (see section 2.2.3). Therefore, a message intercepted on the outgoing link of one mix-server cannot directly be correlated to any other message coming out of another mix-server anywhere in the network.

An active adversary may compromise some or all of the mix-servers on a path and these mix-servers may *collaborate* to break unlinkability. In general, as long as just one mix-server is honest (not collaborating) on the path of a given message, unlinkability can be preserved because incoming and outgoing traffic cannot be correlated across the entire path. However, if collaborating mix-servers are consecutive on the path of a given message they will be able to reveal part of the path for the message. If all mix-servers on the path collaborate they will be able to break the unlinkability of a sender and recipient for the message.

In general, the size of individual messages between mix-servers cannot be used to correlate messages belong to a particular session either, because all messages between mix-servers are padded to the same size.

Responder Anonymity A achieves responder anonymity through the use of a pseudonym key, assuming a passive or active adversary cannot observe traffic on the connection of or

compromise the first mix-server on the reply path from B to A, respectively, because only that mix-server will be able to see the address of A as the recipient.

Unobservability Unobservability in mix-networks can be achieved by the use of white noise. Attacks on unobservability through the use of white noise is described in the next section.

2.2.3 Attacks

The basic scenario is the one where the adversary knows that there is only one sender A and one recipient B. Unlinkability is broken automatically if the adversary observes a message going through the system: only A could have sent the message and only B could have received it. The following attacks assume that there is at least both two possible senders and two possible recipients.

If a passive adversary is able to intercept messages on the link from A to the first mix-server on the path or on the link from the last mix-server to B, A's sender anonymity or B's recipient anonymity will be broken, respectively. If the passive adversary is able to intercept messages on both of these links, he may try to carry out the passive end-to-end attack described below and thus break the unlinkability between A and B.

The attacks on unlinkability described below represent some common attacks and have previously been studied in, e.g., [RP03] and [Ser04]. Other attacks are possible (see [Ser04]) but these will not be described here.

Volume Attack In a volume attack, an adversary observes traffic on the incoming and outgoing links at a mix-server in an attempt to correlate the number of ingoing messages from each sender (another mix-server or a client) with the number of outgoing messages from the mix-server. This is illustrated in figure 2.5.

If A_1 sends m messages to this mix-server, and A_2 sends n messages to this mix-server, then even though the mix-server reorders the messages before forwarding them, the passive adversary will know that m outgoing messages from the mix-server are forwarded to M_3 (and ultimately B_1) and that n messages are forwarded to M_7 (and ultimately B_2), even if the mix-server padded all $m + n$ messages to the same size before forwarding them. To protect against this attack, white noise may be employed between mix-servers. This way, a passive adversary cannot determine the number of incoming and outgoing messages from the mix-server anymore and is thus unable to correlate messages by volume.

End-to-end Attack This attack is a special case of the volume attack described above. If the passive adversary can observe traffic on the links between A and the first mix-server and on the link between the last mix-server and B, it may be able to observe a correlation between the number of messages sent by A and the number of messages received by B and thus deduce with some probability that A and B are communicating. This is illustrated in figure 2.6. Protection against this attack is to insert white noise on

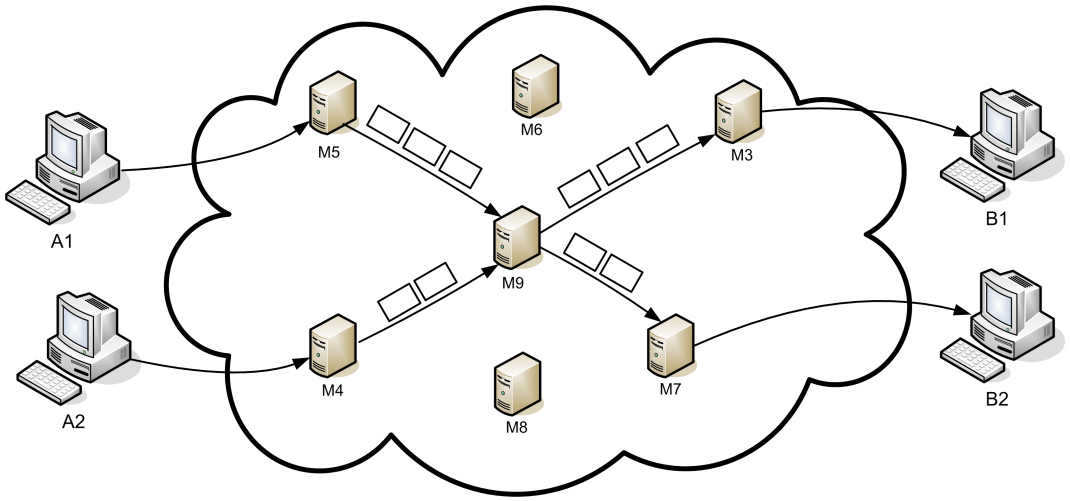

Figure 2.5: Traffic analysis at a single mix (M9). To a passive attacker observing traffic on M9's links, the fact that three messages from M5 are forwarded to M3 and two messages from M4 are forwarded to M7 is a good indication that A1 is communicating with with B1 and A2 communicating with B2.

the link between A and the first mix-server on the path. In this setting a volume attack is not possible since the adversary cannot tell how many messages A sends (if any) and correlate that number with the number of messages received by B.

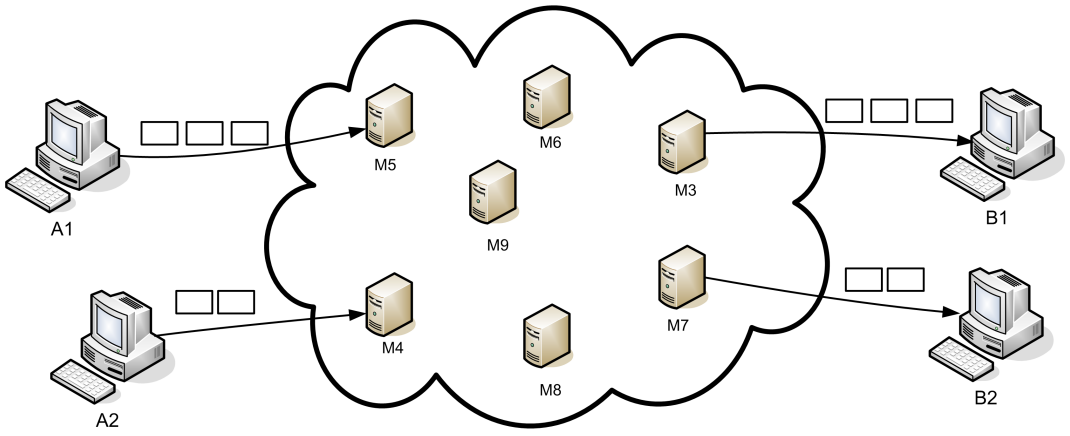

Figure 2.6: A passive attacker observing traffic on A1's and A2's outgoing links and B1's and B2's incoming links may be able to correlate the number of incoming and outgoing messages and from that deduce with a certain probability that A1 is communicating with B1 and A2 with B2.

There are also a number of attacks that an active adversary can carry out. The following describes two such attacks for which there is currently no protection:

Blocking Attack If an active adversary has compromised the first mix-server on the path, white noise between A and this first mix-server is useless, because the compromised

mix-server will be able to tell white noise from real messages, and therefore an end-to-end attack is again possible if the adversary is also able to intercept incoming traffic to B or compromise the last mix-server on the path to B. Protection against this attack is possible by using end-to-end white noise instead of link-to-link white noise. That is, A must send dummy messages encrypted with B's public key through all the mixes on the path to B instead of just to the first mix-server such that only B is able to distinguish between dummy messages and real messages. None of the mix-servers will be able to tell "real" messages from the white noise and A and B's unlinkability is preserved. However, even this bandwidth-intensive protection does not protect against active attackers controlling mixes at each end of the path. The first mix-server may block traffic from A a number of times and if the last mix-server on the path observes "holes" in the traffic to B corresponding to the blockings performed by the first mix-server, the adversary may conclude with some probability that A communicates with B.

The $n-1$ attack If a mix-server batches exactly n incoming messages before forwarding them, an adversary may send $n-1$ messages to the mix-server and then wait for another message to be sent to the mix-server. Because the adversary is able to correlate $n-1$ of the outgoing messages from the mix-server with the $n-1$ messages sent to the mix-server (because it sent them itself), the adversary will be able to determine which mix-server the last message is forwarded to, and thus reveal part of the route for that message.

2.3 DC-networks

DC-networks ("Dining Cryptographer-nets", also known as "Superposed Sending") is a technique invented by David Chaum in 1988 [Cha88] to achieve "Unconditional sender and recipient untraceability" among a number of communicating entities. Using the definitions from section 1.1, "untraceability" as used in [Cha88] is synonymous with "anonymity". Chaum explains his technique through the following scenario [Cha88, p. 1]:

> Three cryptographers are sitting down to dinner at their favorite three-star restaurant. Their waiter informs them that arrangements have been made with the maitre d'hotel for the bill to be paid anonymously. One of the cryptographers might be paying for the dinner, or it might have been NSA (U.S. National Security Agency). The three cryptographers respect each other's right to make an anonymous payment, but they wonder if NSA is paying. They resolve their uncertainty fairly by carrying out the following protocol:
>
> Each cryptographer flips an unbiased coin behind his menu, between him and the cryptographer on his right, so that only the two of them can see the outcome. Each cryptographer then states aloud whether the two coins he can see–the one he flipped and the one his left-hand neighbor flipped–fell on the same side or on different sides. If one of the cryptographers is the payer, he states the opposite of what he sees. An odd number of differences uttered at the table indicates that a cryptographer is paying; an even number indicates that NSA is paying (assuming that the dinner was paid for only once). Yet if a cryptographer is

paying, neither of the other two learns anything from the utterances about which cryptographer it is.

The scenario described above is illustrated in Figure 2.7. The paying cryptographer achieves *sender anonymity* because none of the two non-paying cryptographers knows which of the two other cryptographers actually paid, ie. who lied about the outcome of the coinflips. The sender anonymity is *unconditional* because it is impossible for any one of the cryptographers to know who lied.

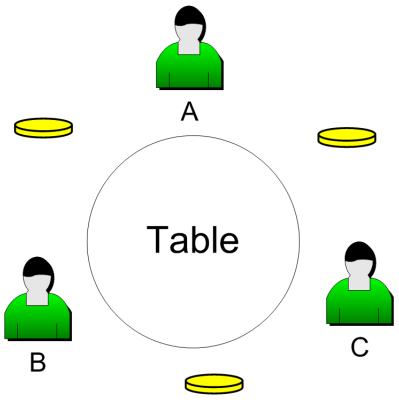

Figure 2.7: Three dining cryptographers around a table. Each cryptographer can only see the coin to his immediate left and right.

It can be seen from this example that if a cryptographer knew the outcome of both coin flips for another cryptographer, he would be able to tell if that cryptographer was lying, so no cryptographer must know the outcome of both coin flips for any other cryptographer, or the anonymity of that cryptographer will be broken.

2.3.1 Generalized DC-networks

A generalisation of DC-networks depicts participants as vertices in a graph and shared coins as edges. To make a functional DC-network, the graph must contain at least three vertices if any of them are to stay anonymous to the other vertices in the graph. With two vertices, only non-participant vertices are unable to distinguish between the two potential senders in the graph. All vertices must be connected such that each vertex is connected to at least two other vertices. In the case of three vertices, this could form a ring as illustrated in figure 2.8. In a graph with k vertices, each vertex may be connected with up to $k-1$ other vertices, to form a *fully connected* graph, as illustrated in figure 2.9.

In Chaum's dinner scenario, each cryptographer "reads aloud" the outcome of the coin flips he can see. In the graph terminology adopted here, this corresponds to each vertex *broadcasting* the outcome of the coin flips to every other vertex in the graph. If the number of vertices is k, each vertex sends the coin flip outcome to $k-1$ other vertices, resulting in $k \times (k-1) = k^2 - k$ messages, regardless of how the vertices are connected.

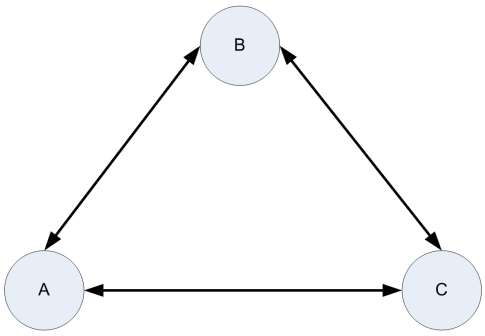

Figure 2.8: Three vertices forming a ring-shaped DC-network. Each vertex shares a secret bit with its two neighbours.

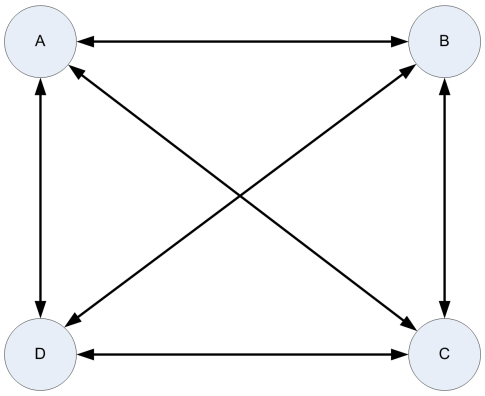

Figure 2.9: Four vertices forming a fully connected DC-network. Each vertex shares a secret bit with every other vertex in the network.

The scenario above described sender anonymity but recipient anonymity is also achievable in a DC-network if every vertex in the graph has anonymously (via e.g. the DC-network itself) published a pseudonym public key, i.e., a public key that does not reveal the identity of the node in possession of the corresponding private key. A participant node may now encrypt its message using one of these public keys and send the encrypted message using the DC-network protocol. The identity of the sending node (its sender anonymity) is protected as described previously, and since the only node capable of decrypting the message is the intended recipient, this node is anonymous to all other participant nodes because no other vertex can know which vertex is able to decrypt the message and hence achieves recipient anonymity.

This method of achieving recipient anonymity can be used by all techniques that send messages encrypted with pseudonym keys to multiple recipients, e.g., also by broadcasting as described in section 2.4.

An alternative to using public-key cryptography is to use a one-time pad. The message is then encrypted with this one-time pad instead of the recipient's public key, and only the intended recipient is able to decrypt the message. The problem with this solution is that the one-time pad must be as long as the message it encrypts, and it must be communicated to the recipient securely and anonymously. However, if this is possible with the one-time pad, the sender might as well send the message itself through the same channels.

2.3.2 Practical DC-networks

The DC-network protocol can be adapted to computer networks by representing all messages as 0's and 1's. The binary exclusive-or operation (XOR, denoted by $\oplus$) can be used for implementing the cryptographers' utterances from Chaum's description, "same" or "different" as 0 or 1, respectively, because it has the property that an even number of "same" or "different" as input will cancel each other out, while an odd number of "same" or "differ-

ent" as input will rsult in "same" or "different" as output, respectively. Vertices are replaced by computers (or nodes), and each edge in the graph, representing a shared coin, is replaced by a link between two nodes in a network through which a single bit (for "heads" or "tails") may be exchanged.

In a network with a ring topology where each node is connected to two other nodes, the outcome of computing $e_1 \oplus e_2$ for a node (denoted p) is either 0 (in the case of $p = 1 \oplus 1 = 0 \oplus 0 = 0$) or 1 (in the case of $p = 1 \oplus 0 = 0 \oplus 1 = 1$). In the case of a fully connected network, each node simply extends this calculation to include the bits exchanged on all its links, $p = e_0 \oplus e_1 \ldots \oplus e_{k-1}$, instead.

The bit communicated on each link is thus XOR'ed exactly twice in the case of a ring topology and $k - 1$ times in the case of a fully connected network, once by each node connected by a link. Therefore, the final result for each node will be 0 if any even number of nodes invert their result, and 1 if any odd number do so.

The node wishing to communicate an anonymous message (in this case a single bit) broadcasts $p \oplus 1$ in the case where its message is 1 and $p \oplus 0$ (or, equivalently, just p) in the case where its message is 0. In both cases, all the other nodes not wishing to communicate an anonymous message send $p \oplus 0$ or equivalently, just p.

All nodes may obtain the anonymous message by calculating $p_0 \oplus p_1 \oplus \cdots \oplus p_{k-1}$, where p_0 is the message the node broadcasted to all other node, and $p_1 \ldots p_{k-1}$ are the messages the node received from all other nodes.

For messages longer than a single bit, new secret bits are exchanged between nodes for each bit to be communicated, and the protocol repeated.

2.3.3 Efficient DC-networks

To build more efficient DC-networks, the frequent exchanges of secret bits described above may be minimized by having nodes perform a secure, one-time exchange of a seed to a pseudo-random number generator (PRNG).

When two nodes have exchanged this seed, they may use it to initialise each their own PRNG and then generate an infinite amount of identical, pseudo-random streams of bits fast without any further communication. This solution means only a few bytes need to be exchanged *once* to set up the DC-network in contrast to the naïve method where nodes would have to exchange up to $k - 1$ bits for each bit to be communicated.

2.3.4 DC-networks With External Recipients

DC-networks as described so far require that each time a node wishes to send one bit of information, it broadcasts this bit to all other nodes in the DC-network. However, if recipient anonymity is not needed, the recipient of a message may be *external* to the DC-network, as illustrated by figure 2.10.

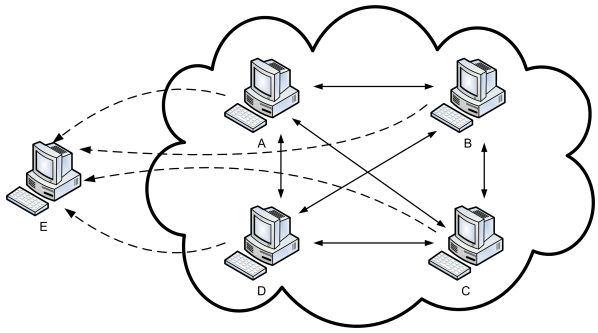

Figure 2.10: A DC-network where the recipient is not part of the DC-network and thus all nodes know the identity of the recipient.

Instead of broadcasting to every other node in the DC-network, each node now sends its output to an external node. Upon receiving data from all nodes in the DC-network, the external node may construct the message sent by the DC-network by calculating $p_1 \oplus p_2 \ldots p_k$ where p is the outcome sent by each node in the DC-network, and k is the number of nodes in the DC-network. The external node is now in possession of the message sent by the DC-network but is unable to know which node in the DC-network actually sent it, ie. sender anonymity is achieved. Since each node in the DC-network now needs only to send to a single node, the external node, this lowers the total number of messages send from $k^2 - k$ to k, one from each node in the DC-network.

2.3.5 DC-network Example

In the following example, one "round" in the DC-network corresponds to broadcast of one bit, so for example, eight rounds of broadcasting are required to send one byte.

Assume three nodes, A, B, and C are connected in a ring topology (identical to example in figure 2.8). The actual eight bits shared between A, B, and C for each round in this example are shown in Table 2.1.

	#1	#2	#3	#4	#5	#6	#7	#8
A↔B	1	0	0	1	1	0	0	1
B↔C	0	1	0	1	0	0	1	1
C↔A	1	0	0	1	1	1	0	0

Table 2.1: Bits shared between nodes for the eight rounds. A↔B signifies bits shared between A and B, and #1 means round 1, #2 means round 2, etc.

Now B wants to broadcast the eight bits $(11001001)_2$ anonymously. In order to get the result for each round, the three nodes each calculate $s_1 \oplus s_2$ where s_1 and s_2 are the node's shared bits. A node wishing to send a 0 and nodes not wishing to send at all simply broadcast $s_1 \oplus s_2$, while a node wishing to send a 1 broadcassts $s_1 \oplus s_2 \oplus 1$. The result for the first round of this example is:

$$\begin{array}{rccccccc} A: & 1 & \oplus & 1 & & & = 0 \\ B: & 0 & \oplus & 1 & \oplus & 1 & = 0 \\ C: & 0 & \oplus & 1 & & & = 1 \end{array}$$

In the above, node B wants to send a 1, so it inverts the XOR of its shared bits by XOR'ing it with 1. All nodes may calculate the final result for any round by XOR'ing all results for that round, in this case $0 \oplus 0 \oplus 1 = 1$. Thus, someone sent a 1 in this round, but the identity of the sender is unknown to all nodes but the sender itself. The results for all eight rounds can be seen in Table 2.2.

Bit to send:	1	1	0	0	1	0	1	0
A	0	0	0	0	0	1	0	1
B	0	0	0	0	0	1	0	1
C	1	1	0	0	1	0	1	0
Result:	1	1	0	0	1	0	1	0

Table 2.2: An example of eight rounds of the DC-net protocol. Results are calculated by XOR'ing the numbers output by A, B, and C in a column. Node B is the sender.

2.3.6 Disruption in DC-networks

One of the biggest problems with DC-networks is the problem of disruption: one or more nodes in a DC-network can disrupt the network by accidentally or maliciously lying about their "coin flips" even if they were not supposed to according to the protocol thereby destroying the property that all bits are XOR'ed in the result exactly k times. To make matters worse, they can do so anonymously due to the nature of the DC-network.

Chaum proposes a disruption protocol, i.e., a protocol to prevent disruption (and ban disrupters) in [Cha88], but according to Waidner and Pfitzmann [WP90], the protocol is based on an "unrealistic assumption" of a reliable broadcast medium, i.e., a medium that guarantees that each message sent by one participant is received unaltered by all other participants in the network. Chaum's protocol is also susceptible to a number of attacks described in [WP90]. Waidner and Pfitzmann suggest a number of enhanced protocols resulting in a protocol based on the idea of "digital signatures whose forgery by an unexpectedly powerful attacker is provable".

Disruption protocols, however, are out of scope for this work.

2.3.7 Anonymity Properties

DC-networks achieve *unconditional* sender and recipient anonymity because it is impossible for any node that knows at most $k - 1$ (where k is the number of participants) shared bits (or seeds) to tell which node is lying (sending) or receiving a message, respectively. Practical

DC-networks that achieve unconditional anonymity can be built by exchanging bits or seeds using one-time pads.

In practice, however, one-time pads are not used. Instead, a bit can be shared by the use of public key cryptography which removes the "unconditional" part because an adversary *could* break the encryption and hence learn the value of the bit, reducing the anonymity to *computationally* secure. Also, instead of sharing single bits more efficient DC-networks can be built by exchanging seeds for pseudo-random number generators (PRNGs) using a secure key-exchange algorithm. One might argue that the latter is more secure because a passive adversary only has one attempt to intercept the exchanged seeds but many attempts if just one bit is exchanged (and many bits must be sent. In that case, the node's anonymity will be broken for only one bit, of course.). *If* all $k-1$ seeds for a node are intercepted that node cannot achieve sender anonymity in that particular DC-network anymore. Once a node's seeds are exchanged without interception, however, and assuming that PRNGs cannot be broken (ie. their outcomes predicted), the node's sender anonymity cannot be broken [1].

Sender Anonymity In general, DC-networks provide *sender anonymity* against an adversary that knows at most $k-2$ of a node's shared bits. If the adversary knows all $k-1$ bits a node shares with other nodes, it will be able to tell whether the given node is sending (lying). Assuming that it is increasingly difficult to gain knowledge of an increasing number of bits, this implies using fully-connected DC-networks to maximize the number of shared bits that an adversary must be able to learn to expose the identity of the sender.

Recipient Anonymity *Recipient anonymity* can be achieved by encrypting a message with a pseudonym key that is only known to the intended recipient and hence everyone will receive the message but only the intended recipient will be able to read the contents of the message.

Unlinkability If being able to read the message is defined as "receiving the message" then using pseudonym keys also provides *unlinkability* because no one knows who holds the key and hence no one will be able to determine which node could read the message which is a part of the requirements to break unlinkability.

Responder Anonymity Responder anonymity in DC-networks with either internal or external recipients can be achieved, in both cases by broadcasting the reply to all participants in the DC-network.

Unobservability is not possible in DC-networks because even though it cannot be known who is the sender (and perhaps recipient) of a message, all the nodes in the network still know when a message is being sent.

[1] This is not the same as unconditional anonymity, however, because an adversary *could* have intercepted the exchanged bits or seeds and thus be able to break sender anonymity

2.4 Broadcasting

In broadcast protocols, all participants send messages to all other participants at a constant rate through a global broadcast channel. Obviously, this is an extremely expensive technique, even compared to "original" DC-networks where messages are only broadcasted when a node has an actual message to send.

When participants have an actual message to communicate, they send *signal* messages to this global broadcast channel, and at all other times they send *noise* messages. Figure 2.11 shows how four nodes may be connected in a broadcast network.

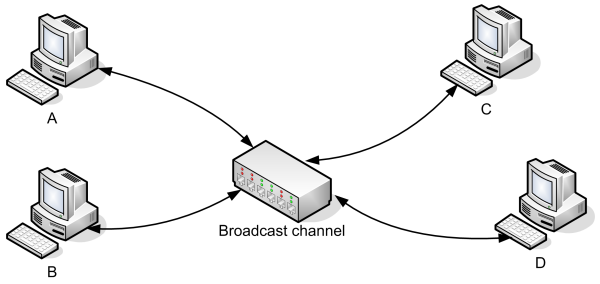

Figure 2.11: Four nodes set up as a broadcast system with a common global communication channel. All nodes send and receive messages to and from the broadcast channel at a fixed rate, and only the intended recipient is able to distinguish white noise from real messages.

2.4.1 Anonymity Properties

Broadcast protocols provide users with unobservability and hence also sender- and recipient anonymity, and unlinkability. Unobservability is achieved by a broadcast protocol against both passive and active adversaries because all nodes send messages to the global communication channel at a constant rate, and all messages are encrypted such that the adversary and all non-recipients cannot distinguish real messages from noise (as described in section 2.3.1).

Thus, if an adversary intercepts messages on the outgoing link of one of the participants in the network, it is impossible to tell whether the messages are noise or real messages and therefore impossible to tell if the participant being wiretapped is communicating with someone. The same argument applies to the case where the adversary intercepts *incoming* traffic to a participant.

2.5 The Peer-to-Peer Factor

The techniques presented in the previous sections can be used to build anonymizing systems in both a classic centralized, client/server approach or in a peer-to-peer context. This section gives an overview of advantages and disadvantages of the two, both in the context of generic applications and anonymizing systems.

The motivation for peer-to-peer networks in general stems from the problems and/or limitations with the classic client-server approach, e.g.:

- Lack of scalability to many users due to bottlenecks.
- Single-point-of-failure [2].
- A small, static set of servers makes the system more vulnerable to e.g. denial-of-service (DoS) attacks because the adversary always knows the identity of the servers to attack.
- Geographically closely placed servers makes the system vulnerable to e.g. physical attacks or accidents.

Peer-to-peer networks attempt to overcome these problems and/or limitations. In "real" peer-to-peer networks there is no central server and thus no single-point-of-failure. State information is distributed throughout the network and thus no single node (or small set of nodes) has an overview of the entire network. If the network is *structured* as e.g. Chord [SMLN+03] or Pastry [RD01a] it will scale to millions of users with guarantees about the number of routing hops (see section 3.1.3). The set of system nodes is as large as the set of nodes in the network since every node joining becomes a system node. Also, because nodes may join and leave at arbitrary times, specific system nodes become very difficult to attack. Finally, because nodes from potentially all over the world are part of the network, nothing short of a global catastrophy will make the entire network unusable.

However, a number of problems with peer-to-peer networks in general exist, e.g.:

- The dynamics of the network can make it hard to build systems that e.g. must keep data available at all times or provide long-lived communication channels.
- If any node can join the network, node heterogenity in terms of CPU and bandwidth capacity can have performance implications.
- It is easier for an adversary to insert malicious system nodes than in a centralized system where authentication of a relatively small and static set of nodes can easily be done.
- Nodes accidentally or maliciously not following protocol, by e.g. not forwarding messages or not saving data.

Anonymizing systems can as mentioned be built using a client-server or a peer-to-peer approach. Specific problems for client-server anonymizing systems include (but are not limited to):

- Centralized trust. The user must trust the individuals or organization running the anonymizing system to keep the user's identity secret.

[2] Or alternatively, few points of failure.

- Users are not part of the system which makes *edge attacks* possible.
- Administratively closely placed servers (e.g. within the same organization) makes the system vulnerable to e.g. legal attacks.

Peer-to-peer anonymizing systems attempt to overcome these problems and/or limitations. Trust is distributed and no fixed real subset of nodes knows the identities of all users. Because of the dispersal and dynamism of nodes it is extremely difficult for an adversary to be global and to observe traffic on specific connections or to compromise specific nodes that make up the anonymous channel for a given communication. Also, because users are part of the anonymizing system edge-attacks are not possible. Because nodes are usually both geographically and adminstratively dispersed legal attacks become very hard.

Problems with peer-to-peer anonymizing systems include (but are not limited to):

- How the number of adversaries joining the system can be kept under a certain limit.
- Authentication of nodes.
-
- Adversaries preventing nodes from learning about the rest of the network such that affected nodes always will send messages through the nodes controlled by the adversary.
- Motivation for users to provide anonymity for other users.

We feel that the benefits of peer-to-peer networks outweigh the downsides, and thus the following survey will only describe systems built on the peer-to-peer approach. Also, systems built on the classic centralized approach have been studied extensively previously.

2.6 Survey of Peer-to-peer Anonymizing Systems

Most anonymizing systems so far have been built on a centralized client-server approach, these include (but are not limited to): Anonymizer [ano], The Eternity Service [And96], Onion Routing [GRS99] (Tor [DMS04]), Mixminion [DDM03], Free Haven [Din], Publius [MWC00], Dagster [SW] and Tangler [WM01]. They differ widely in their purpose (ranging from generic anonymous TCP/IP communication (e.g. Onion Routing), to anonymous file storage system with algorithms for content migration and reputation (Free Haven)) but they all aim to offer some form of anonymity for the sender (or initiator/publisher/storer, respectively).

Recently, systems based on a peer-to-peer network have been designed, analyzed and implemented. The following briefly describes a few selected, concrete peer-to-peer anonymizing systems focusing on the unique contribution of each system (and not on the specific systems's anonymity properties) to enlighten some of the challenges that exist within the area

of building anonymizing systems in a peer-to-peer context. In general, each system has the same anonymity properties as the technique on which it is built. However, due to the peer-to-peer factor both passive and active attacks become harder to perform than in traditional systems built on the same techniques and thus peer-to-peer anonymizing systems are intuitively more secure.

A comparison of the anonymity properties of specific systems is out of scope for this work. GAP [BG03] takes the first steps towards such a comparison (for some of the systems [3]), but as the authors state "The systems differ widely in their respective costs and benefits and it is thus difficult if not impossible to make a fair comparison".

The following presents a survey of some existing anonymizing systems built on peer-to-peer technology that achieve different kinds of anonymity (at least sender anonymity) for their users while overcoming the limitations and/or problems of centralized anonymizing systems.

2.6.1 Crowds

Crowds is a peer-to-peer system described by Reiter and Rubin [RR97] in 1998. The system extends the proxy technique by adding multiple forwarding nodes as described in this section. The system works on the application level and provides a means for users to issue anonymous web requests, ie. achieve sender anonymity.

Because users are part of the anonymizing system itself, a node receiving a message from another node cannot tell whether that node was the original sender or merely forwarding the message (unless a wiretapper attack can be performed). This is similar to the *proxy argument* but here nodes cannot tell the difference between the system and the senders.

Users are grouped into a (preferably) geographically dispersed collection, called a crowd, which retrieves the result of a web request issued by the user. The web servers are not part of the crowd and hence recipient anonymity is not possible.

When users join the crowd, they are informed of the identity of the other crowd members, and these nodes are also informed that the new user has joined the crowd.

Once a user has joined the crowd, he may issue an anonymous request to a web server through the crowd. The user simply forwards his request to a randomly chosen member of the crowd instead of directly to the web server. This member chooses uniformly at random whether to *submit* the request to the web server specified in the request, or to *forward* the request to another randomly chosen member of the crowd. The probability of this choice, p_f, is biased in favour of forwarding, i.e., $p_f > 1/2$. In either case, the member records the id the of the member from which it received the request, the id of the node to which it forwarded the request or the id of the server to which it submitted the request, and a *session id* identifying the connection. It then saves this information, typically for 30 minutes, such that when a reply to the request is received, the node knows which node to forward the reply

[3] Although the authors don't differentiate between systems and the techniques they are built on.

to.

The process is illustrated in Figure 2.12.

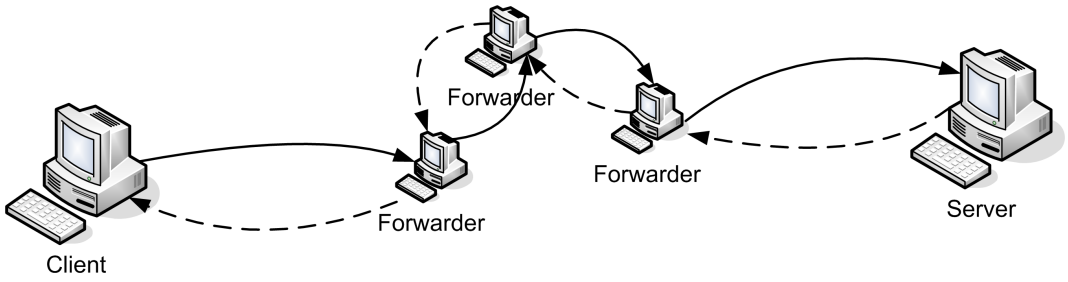

Figure 2.12: A client sends a request (solid lines) through three random crowd members, the last of which chooses to forward the request to the web server. The web server replies to the random crowd member that sent the request (dashed lines), which in turn forwards the reply from the web server back through the chain of random crowd members to the original client.

The set of crowd members through which a message passes constitute a *path* for that message, and any crowd member may be part of a number of paths simultaneously. Paths are static for a pre-defined period of time such that requests to different web servers from the same crowd member will be routed along the same path until paths are reset and new members join the network. This is known as a *path reformation* and can be used to perform the *predecessor attack* described next.

Ultimately, after the request has passed through a number of crowd members, the web server receives the request, processes it, and returns the answer to the crowd member from which it received it. Since this member has a record of having forwarded one or more messages with the associated session id, it simply forwards the reply from the web server to the member from which it originally received the request. This process continues until the original initiator of the request receives the answer from the first randomly chosen node to which it chose to forward the request.

Predecessor Attack

Crowds is susceptible to a statistical attack called the *predecessor attack*, first described in [RR97] and later extended in [WALS01]. A number of attackers may simply join the crowd and wait for paths to be reformed. After each reformation, each attacker logs its immediate predecessor when a path is formed through the attacker. The attack is based on the intuition that given a large number of path reformations, the path initiator I is more likely than any other node to appear as the immediate predecessor of one of the attackers. The more path reformations the attackers are able to observe, the more clear it will be that I is the path initiator. Furthermore, the higher the ratio of attackers to the total number of nodes in the crowd, the less path reformations need be observed to identify I with increasing probability and thus break I's sender anonymity. The following example illustrates this.

Assume a network with 15 crowd members with node ids 1–15. Nodes 11–15 are malicious

Node Id	0	1	2	**3**	4	5	6	7	8	9
Times logged	1	2	3	**15**	5	6	4	7	4	3
Frequency in %	2	4	6	**30**	10	12	8	14	8	6

Table 2.3: The number of times each node id has been observed as predecessor to one of the attackers and the corresponding frequency in percent.

attackers trying to identify one or more path initiators among the remaining 10 crowd members.

The malicious nodes each log their immediate predecessor for a given session id over, e.g., 10 path reformations in a list and collectively end up with the following lists:

11. {1,3,3,2,7,8,3,5,7,4}
12. {6,3,5,4,4,8,9,3,4,3}
13. {7,4,5,3,2,3,6,3,9,3}
14. {7,6,8,3,5,7,5,3,1,3}
15. {5,3,7,7,9,0,6,8,3,2}

Each list consists of 10 node ids of predecessors to nodes 11–15, with 50 ids in total. Each individual id is observed a number of times as summarised by Table 2.3:

This table shows that the node with id 3 occurs far more often than any other observed node id. Because the path initiator is more likely to appear in more paths than a random crowd member, node 3 in the example above is more likely to be the path initiator than any of the other observed nodes.

2.6.2 Hordes

Hordes [LS02] is a system for sender and responder anonymity on the transport layer using Crowds for sender anonymity and multicast for responder anonymity[4].

Because Hordes is built on Crowds which in turn is based on the proxy technique it is susceptible to the same attacks as Crowds and the proxy technique, e.g. the wiretapper and predecessor attack.

In Hordes, when a user joins the network it joins a group, a *horde*, that has a multicast address specified by a server. From this horde an initiator can anonymously communicate with a (non-anonymous) responder by sending a message through a Crowds path within the horde specifying the multicast address as the source. The responder replies to the multicast address and the initiator receives the reply remaining anonymous within the horde.

[4]The authors claim that "the forwarding mechanisms from mix-servers could easily be adapted for use on the Hordes forward path".

For Hordes to be efficient it is required that the network infrastructure (routers) supports multicast communication. It is worth noting that in current multicast protocols no single entity can determine the membership of a multicast group; this requires the coordination of all routers that make up a given group.

Because of the asymmetry of the initiator sending a message via unicast and the responder replying via multicast, protocols for reliability and congestion control are needed.

2.6.3 Freenet

Freenet [CMH+02] is an anonymous, peer-to-peer file storage system built on a Crowds-like technique. Nodes that publish files in Freenet, *publishers*, achieve sender anonymity and nodes that store files, *storers*, achieve recipient anonymity during publication of a file unless an adversary is able to perform a wiretapper attack. During retrieval, nodes that retrieve files, *clients*, and storers achieve recipient and sender anonymity, respectively, because of the proxy argument. The anonymity properties of Freenet during publication are similar to the anonymity properties of Accordion (presented in the next chapter) during publication. Accordion, however, achieves stronger anonymity properties than Freenet during retrieval due to the use of DC-networks.

The main contributions of Freenet are its use of different types of file identifiers, keys, and its caching/replication strategy.

Keys

The most basic key in Freenet is the keyword-signed key (KSK) which is basically just a hash of the filename.

$$KSK : H(filename)$$

However, two other key types which are used for avoiding "key squatting" and for updateable files, respectively, are also used:

SSK To prevent accidental or malicious "key squatting", ie. choosing popular filenames as keys for junk files thereby making it harder to locate and retrieve the "real" file, a *signed subspace key* (SSK) is used. A user I randomly generates an asymmetric key pair (PRK_I,PUK_I) from which the public key, PUK_I, serves as I's namespace and the private key, PRK_I, is used to sign the file. The SSK for a file is then defined as:

$$SSK : H(H(\text{filename}) \oplus H(PUK_I))$$

To allow for retrieval of the file, PUK_I is published along with the filename. In order to insert

a file in I's namespace PRK_I is needed so no other users can prevent I from inserting a file with a particular filename.

CHK To update files a *content hash key* (CHK) is used. First, to insert an updateable file the CHK is calculated:

$$CHK : H(file)$$

Then (CHK,file) is inserted. Also, an SSK is calculated and $(SSK, S_{PRK}(CHK))$ is inserted to create a "pointer" to the file, where S denotes is a signing function. The file can then be retrieved through the SSK.

To update a file, a user first inserts the modified file under its new CHK. Then, to update the pointer, the original SSK is inserted along with the new CHK. When the insert reaches a node that possesses the SSK, a key collision will occur: the node on which the collision occurs checks the signature on the new CHK and verifies that it is both valid and more recent and overwrites the old CHK value with the new one. The old version of the file will now only be available through the old CHK value.

Caching

Requests for files in Freenet are forwarded through a number of nodes until a node that can fulfill the request is hit or a hops-to-live value in the request reaches 0 (in which case a "Request failed" message is sent back through the path). In the case of a succesful request, the file is passed back to all nodes on the path and each node associates the file with the corresponding key. A subsequent requests for the same file by the same client will thus be handled locally. A request for a file with a similar key will be forwarded to the previously succesful data source. This means that nodes over time will improve both their local routing tables and data stores by storing more and more pointers to nodes that store files with similar ids and storing the files with similar ids themselves.

The caching strategy both improves efficiency because files potentially get nearer to clients that request them but also serves as a replication mechanism that ensures availability in the case of node failures.

2.6.4 MorphMix

MorphMix is an anonymizing system built on the mix-network technique that provides anonymity on the transport layer enabling applications to be built on top of it. The main contributions of the MorphMix system are the tunnel setup process and its collusion-detection mechanism.

Tunnel setup As opposed to traditional mix-networks the initiator selects only the first hop in tunnel. Subsequent nodes are added by letting the last recently added node choose the next hop. The choice of the next hop is done by using a *witness*: the role of the witness is to ensure that the last recently added node doesn't either simulate or single-handedly choose the remaining nodes of the tunnel. The witness is chosen randomly from the set of nodes that the initiator knows and the basic assumption is that it is unlikely that an adversary is colluding with this randomly chosen witness.

When the tunnel is set up, the initiator encrypts a message as described in section 2.2 but using symmetric keys shared with each hop in the tunnel as opposed to the public keys of the hops. These keys are deleted when a tunnel is torn down. In classic mix-networks an adversary could perform an attack where it recorded traffic and later gained access to the mix-servers to obtain their (permanent) private keys thereby enabling the adversary to decrypt the recorded data. This attack is not possible in MorphMix because once the tunnel is torn down and the keys deleted no adversary will be able to decrypt old traffic on that were encrypted with these keys. This is also known as *perfect forward secrecy*.

Tunnels are changed frequently. This is done both to strengthen the collusion-detection mechanism (described next) and to minimize the amount of traffic that an adversary controling a tunnel will be able to see.

Collusion-detection mechanism To avoid tunnels composed of malicious nodes that can collude to reveal the identity of the initiator, all MorphMix nodes keep track of the tunnels they have setup previously. The basic assumption is that if a malicious node is hit when setting up a tunnel this node will suggest other malicious nodes to complete the tunnel.

For each hop added to a tunnel the following is registered: the witness used to add the hop and a list of *candidates* that were presented to the initiator as the next possible node in the tunnel. Based on the variation of these nodes a node can calculate a *correlation* value that can later be used to determine (or rather, make a qualified guess as to) whether a given tunnel contains malicious nodes.

It should be noted that the mechanism described above could be defeated if the adversary more cleverly not only suggests other malicious nodes as candidates but also benign nodes. This could lower the correlation value to a non-suspicious level. In MorphMix [RP02] a more sophisticated method is suggested. This is, however, out of scope for this project.

2.6.5 TAP

TAP (Tunneling approach for Anonymity in P2P systems) is an anonymizing system built on the mix-network technique that aims to solve the functionality problem (in contrast to an anonymity problem) of a node in a tunnel that disconnects from the network and thereby breaks the tunnel. It does this by basing hops in a tunnel on ids in a structured network (e.g. a distributed hash table (DHT)) instead of basing them on the IP-addresses of specific nodes. TAP makes sure that the keys used for the nested mix-encryption are always replicated on the numerical neighbours of the node *responsible* for a given hopId (numerically closest to

the hopId), the replica set. To do this replication TAP relies on the DHT to notify it of nodes joining and leaving. Assuming that the replica set is always up-to-date a hop in a tunnel in TAP will continue to work unless all nodes in the replica set are unavailable.

Access to (ie. use or deletion of) a tunnel hop is protected by a password known only to the initiator of the tunnel and the replica set. To protect against collisions in hopIds these can be computed as a hash of e.g. the nodeId or the node's PUK or PRK and some additional information. To create the hops for the first tunnel a node must use a bootstrap tunnel. This could be done by using an existing system such as Onion Routing [GRS99] or an existing TAP-tunnel.

2.6.6 GAP

GAP (GNU Anonymous Protocol) is an anonymizing system built on the mix-network technique used by GNUnet to provide an anonymous file-sharing system. Sender and recipient anonymity in GAP work by not only mixing queries and replies (coming from the GNUnet application) but forwarding these to a number of other nodes based on e.g. local CPU and network load. Because of this, the layered encryption and because nodes can forward messages to multiple nodes an adversary will not be able to correlate the incoming and outgoing messages.

For example, if an adversary sees that a node A receives one message from a node B and one message from a node C and that A sends out three messages, the adversary cannot know whether the three outgoing messages are B's message being forwarded to three other nodes, B's message being forwarded once and C's message twice, A's message twice and C's message once etc.

A node forwarding queries for other nodes records the source id of the query and associates it with the id of the query. This way, when a node receives a reply it checks whether it needs to forward the reply to another node. A node that receives a reply might also choose to copy the content to its local data store thereby providing content migration similar to the caching used by Freenet.

2.6.7 Herbivore

Herbivore [GRPS] is an anonymizing system built on DC-networks with external recipients. It provides users with sender and responder anonymity against passive adversaries.

Users are divided into *cliques*, each forming a DC-network. Cliques are of size at least k, where k is a predetermined constant, and a global topology algorithm ensures this invariant is maintained throughout the lifetime of the system. If a clique grows too large to communicate efficiently (more than $3k$ members), new smaller cliques are formed from the members of this clique. Similarly, if the number of members in a clique falls below k, the remaining members are moved to other cliques. Two protocols control the behaviour of Herbivore.

The entry protocol in Herbivore serves three purposes:

1. It prevents nodes from choosing which clique to join. If nodes could choose which clique to join, an adversary could place itself in a clique with only one other node and hence break the anonymity of that node.
2. It maintains the invariant that cliques are of approximately equal size by randomizing which cliques to put newly joined nodes in.
3. It limits the rate at which new nodes may join the network. This deters DoS-attacks by malicious nodes joining the network in rapid succession.

The round protocol defines the way members of a clique send data anonymously and detects tampering of messages. It consists of three phases:

Reservation Phase Members wishing to send during the next round reserve a random slot in a bit vector and anonymously broadcast this vector to all other members. All others anonymously broadcast the bit vector with no slots reserved to all other members.

Transmission Phase Each node that has reserved a slot in the reservation phase anonymously broadcasts its message in the appropriate slot. Since in a DC-network, all participants (including the sender) also receives the broadcast message, collisions and malicious tampering is easily detected (not prevented) by the sender by comparing the data it sent with the data it receives. If collision or tampering occurs, the sender simply tries to re-send during the next around. Alternatively, the sender may choose to join another clique.

Exit Phase This phase serves to protect long-running network transactions against traffic analysis. Nodes may anonymously request other nodes to delay their departure if it is engaged in a long-running transaction. However, it cannot force nodes to stay in the clique, since these may simply leave abruptly by crashing or ignore the request altogether.

By virtue of the properties of DC-networks and the three phases in the round protocol, Herbivore is resilient to a number of different attacks. Collusion attacks where malicious nodes gather in a clique to single out communications by the remaining node are made less probable by the random entry protocol. The threat of DoS-attacks through a large number of malicious nodes joining simultaneously is mitigated by rate-limiting joining. Herbivore does not provide a disruption protocol but instead suggests that nodes join another clique if disruption occurs.

Finally, Herbivore is not resistant to statistical analysis against long-lived transactions. If a particular node, A, is disproportionately often a member of a clique which contacts a particular server, the attacker has statistical reason to believe that A is the node communicating with that server. According to [GRPS], this is a fundamental limitation of any system that provides anonymity in a clique.

2.6.8 $\mathcal{P}^5$ (Peer-to-Peer Personal Privacy Protocol)

$\mathcal{P}^5$ is an anonymizing system built on the broadcast technique to provide users with unobservability, but mitigates the negative effects of scaling such a system by creating a hierarchy of broadcast channels, constructed as a binary tree. Participants choose themselves at which level in the tree they wish to be placed in; lower levels provide stronger anonymity because there are more groups a participant *could* be in, but it comes at the expense of efficiency, because messages have to be broadcasted to more nodes to reach their destination. Since nodes, according to the protocol, are allowed to drop any message they see themselves unable to process for any reason, placing oneself at a low level in the tree may result in lost messages.

All messages are encrypted such that only the recipient may decrypt them. This can be achieved by, e.g., encrypting messages with the recipient's pseudonym key. All messages are also hop-by-hop encrypted.

Even a system like $\mathcal{P}^5$, however, does not scale well due to its excessive traffic overhead.

2.7 Summary of Properties

This section summarizes the properties of the four techniques presented in this chapter on which anonymizing systems can be build. First, table 2.4 summarizes the anonymity properties of each technique.

	SA	RA	UL	RespA	UO
Proxy	✓(1)	÷	÷	÷	÷
Mix-networks	✓(1)	✓(1)	✓	✓(2)	÷
DC-networks	✓	✓(3)	✓	✓	÷
Broadcasting	✓	✓(3)	✓	✓	✓(4)

Table 2.4: Summary of anonymity properties for different anonymizing techniques. SA: sender anonymity, RA: recipient anonymity, UL: unlinkability, RespA: responder anonymity, UO: unobservability.

(1) This only holds if an adversary is not able to perform either a wiretapper or an edge attack (as described in the beginning of this chapter).

(2) Responder anonymity is achieved by the sender encrypting the message with a pseudonym key that does not identify the sender but only holds if the adversary is not able to compromise the last node on the reply path.

(3) Recipient anonymity is achieved by encryption of the message with the recipient's pseudonym key.

(4) In general, unobservability can only be provided by systems built on the broadcasting technique where all messages are sent at a constant rate by all participants *to* all participants and are indistinguishable from white noise.

The proxy (single, forwarding node) or a similar Crowds-like technique with multiple, forwarding nodes provides an efficient way to achieve sender anonymity. The number of messages depend on the number of forwarding nodes that they pass through. The biggest advantage of the proxy technique is the simple design that allows for efficient implementations. The biggest disadvantage of the proxy technique is the single point of failure and trust that makes the technique vulnerable to both passive as well as active adversaries.

Mix-networks provide reasonably efficient, anonymous two-way communication (responder anonymity) and a method of achieving unlinkability against primarily passive adversaries using public key cryptography. As in the proxy technique, the number of messages is dependent on the number of mix-servers that a message has to pass through. Mix-networks introduces delays when batching (and reordering) messages and hence some systems based on mix-networks might not be suitable for real-time applications such as web browsing or instant messaging.

DC-networks (can) provide unconditional anonymity at the expense of an increase in the number of messages transferred. In the case of "original" DC-networks (where recipients are part of the network), the total number of messages is $k^2 - k$. In the case of external recipients, the number of messages is k. Other than the high number of messages compared to the proxy or mix-network technique, a disadvantage of DC-networks is the hard problem of stopping disruption. More practical solutions that only achieve computationally secure anonymity is possible through the use of public-key cryptography and/or pseudo-random number generators.

The broadcasting technique achieves unobservability at the expense of all users sending messages at a constant rate to all other users.

Specific systems have the same anonymity properties as the technique on which they are built but due to the peer-to-peer factor both passive and active attacks become harder to perform than in traditional systems built on the same techniques and thus peer-to-peer anonymizing systems are intuitively more secure.

Chapter 3

Accordion

In this section we present the design of a low-latency, anonymous, peer-to-peer file storage application, Accordion (the same overall design goals as Freenet) built on an existing routing layer. The overall purpose of Accordion is for a user to be able to anonymously store a file in a peer-to-peer network and for another user later to be able to (anonymously) retrieve the file with only a small delay.

The following list comprises the design goals for Accordion:

- Sender anonymity for publishers of data.
- Sender anonymity for storers of data when sending data to a client during retrieval.
- Strong deniability for storers of data, ie. they can provably deny having any knowledge of the data they are storing.
- Public, scaleable and fully decentralized (real peer-to-peer) network.
- Lookup guarantees for requests, ie. if a requested piece of data exists in the network, the system must guarantee returning it to the requestor.
- Low latency transfers of small to medium sized files.
- High degree of availability of published files in the face of multiple node failures.

We will use the term "publication" (and hence the terms "to publish" and "a publisher") to describe the process of storing a file in the network.

3.1 Design Goals

Before presenting Accordion itself, the details of the design goals are described.

3.1.1 Anonymity

In a file-storage system a user publishes a file to a number of nodes in the network and these nodes will store the file for a certain time. If perfect anonymity is assumed, no one will, when publication is complete, be able to link the file with its original owner [1]. The fact that users will thus issue requests to the nodes in the network that store the file and not to the publisher (as in a file sharing system) suggests that the storers of a file need a stronger type of anonymity than the publisher because an adversary has more time to perform attacks on the storer of a file than on the publisher of the file.

The anonymity of senders is considered more important than that of clients receiving files and thus recipient anonymity for clients is not part of Accordion. The reason why sender anonymity is considered more important is because nodes in a file storage system that store files for other nodes should not be held accountable for the contents of these files.

Unlinkability is not an obvious demand in an anonymous file-storage system because there are no long-lived communication channels on which to perform attacks to break unlinkability for a given message between two endpoints. Who communicates with whom changes constantly due to the nature of the application.

Unobservability is considered an overly strong type of anonymity and is thus not a design goal. Also, the technique to achieve it is too expensive if the system is going to be used in practice.

3.1.2 Low Latency

To meet the goal of low latency transfers of files, the system cannot be built on a technique that introduces long delays as some mix-networks designs do [Din]. Delays in the order of seconds due to extra routing hops are acceptable, however.

3.1.3 Real Peer-to-peer

Anyone should be able to run an Accordion node and there should be no central authority to control who joins the network and who can shut down the system by e.g. performing legal attacks. Also, the system must be able to scale to thousands or even millions of users. The potential distribution of nodes across the entire planet increases latency because of more (and perhaps longer) routing hops compared to direct TCP connections but it also makes it harder for an adversary to e.g. break a certain node's anonymity or perform censorship-resistance of a given file.

These goals suggest designing the system as a real peer-to-peer network. To avoid having to design a peer-to-peer network from scratch, Accordion should be built on top of an ex-

[1] Of course, the contents of the file might expose the publisher but this is the responsibility of the users and not the system.

isting routing layer whose job it is to route messages in the peer-to-peer network. To route messages efficiently and to provide lookup guarantees, the routing layer must be *structured*.

Several different structured routing and location schemes exist, e.g., Tapestry [ZKJ01], Pastry [RD01a], Kademlia [MM02], and Chord [SMLN+03].

The design of Accordion is independent of a specific routing layer, but Pastry is chosen because a freely available implementation [PDZ04] that has been used in several real applications, e.g., PAST [RD01b], Scribe [CDKR02], and SplitStream [CDK+03], exists. The following describes how Pastry works.

Pastry

Pastry is the specification of a *peer-to-peer, self-organizing overlay network* which enables routing and location of messages between nodes. It is used as the underlying routing mechanism for Accordion, and the details of terminology, routing, and location shall therefore be described here.

In Pastry [RD01a], each node is assigned a unique id, usually computed by calculating a hash of the node's IP address. Using a secure hash function such as SHA-1, all node ids will—with high probability—be uniformly distributed over the set of all possible values of node ids.

Routing a message from source to destination requires a key. Pastry routes a message to the (live) node with the node id numerically closest to the key (called the *root node* for that key). Each node through which a message passes forwards the message to a node whose node id shares with the key a prefix that is at least one digit longer than the prefix that the key shares with the current node (for this reason this technique is known as *prefix routing*). In case no such node exists in the current node's routing table, the message is forwarded to a node whose node id shares a prefix with the key as long as the current node, but is numerically closer to the key than the current node id. According to [RD01a], this is always possible.

The average number of routing steps in Pastry is less than $\log_{2^b}(N)$, where N is the number of nodes in the network and b is a configuration parameter with a typical value of 4, so that the average number of routing steps is less than $\log_{16}(N)$.

Part of the routing information kept by each node in the network is called the *leaf set*. It contains the ids of the $l/2$ numerically closest smaller and $l/2$ numerically closest larger nodes present in the network, for the same configuration parameter l.

According to the original Pastry paper [RD01a], Pastry exhibits a *short routes property*. Experiments show that the average distance traveled by a Pastry message is between 1.59 and 2.2 times the underlying Internet route, where "distance" typically is measured in terms of ping latency.

3.1.4 Availability

It is an important task of a file storage system such as Accordion to increase the availability of files stored in the system. Three commonly used methods are:

1. Replication of files. All data (files) is stored n times, amount of disk space used to store data increases by factor n.
2. Erasure coding of files, where data is split into n parts, of which m arbitrary parts is needed to reconstruct data ($m \leq n$). Needed amount of disk storage and bandwidth increases by $1/r, r = m/n$.
3. Both of the above, so that data is split into n parts and each fragment is replicated n times. Needed amount of storage and bandwidth increases by factor n/r.

According to [WK02], using replication increases disk and bandwidth usage by a factor 11 compared to using erasure coding. However, erasure coding results in a higher number of lookups to contact the nodes that store each of the parts, whereas contacting the node storing a replicated file only requires one lookup.

3.1.5 Deniability

Anonymity protects the identity of the nodes that store data. Deniability is a property that is useful if a node's anonymity has been broken. Deniability comes in two forms:

Plausible deniability is the ability for a node to plausibly deny knowledge of content stored on the node.

Strong deniability is the ability for a node to prove that it could not have knowledge of content stored on the node.

Erasure coding combined with a secret sharing scheme both increases availability and makes strong deniability possible at the best disk usage/bandwidth ratio. The following section describes the details of erasure coding and secret sharing and how to combine them.

Erasure Coding and Secret Sharing

A secret sharing scheme has the same property as an erasure coding scheme that data is split into n parts, of which m arbitrary parts are needed to reconstruct the data. However, a secret sharing scheme enables partitioning data, here known as the secret S, into n parts in such a way that *any* subset of m parts, $m < n$, make S easily reconstructable, but even complete knowledge of any $m - 1$ parts reveals no information about S. This is also called an m-threshold scheme.

It can be shown [Kra94] that for a secret sharing scheme to achieve perfect (information theoretic) secrecy, each of the n parts must be of length at least as S itself, which is impractical for the purposes of, e.g., file storage.

Krawczyk [Kra94] has devised a strong, space-efficient and computationally secure secret sharing scheme in which each part is of size $|S|/m + |k|$ where $|k|$ is the size of a symmetric encryption key, typically 128 or 256 bits. The size of this key does not depend on the size of S. The scheme uses an erasure coding algorithm in addition to a secret sharing scheme to achieve a smaller part size. Using erasure coding, each part is of size $|S|/m$ (as opposed to size $|S|$ in a secret sharing scheme), but erasure coding schemes alone do not have the strong property that each share will provably not reveal any information about S.

In Krawczyk's scheme, the secret S is first encrypted using a symmetric encryption algorithm and a key k. Then, the encrypted secret, C, is partitioned into n parts using erasure coding. The key is also partitioned into n parts, but using perfect secret sharing instead of erasure coding. Since it is reasonable to assume that $|k| \ll |C|$, the partitioning of k using secret sharing is not expected to increase the overall resource usage in any significant way.

Distributing S in this scheme is done by distributing all n pairs of the form (C_i, k_i), $1 < i < n$ to the participants. Any m participants may now reconstruct S easily by first reconstructing the key k, then reconstructing the encrypted secret C, and finally use k to decrypt C. Any $m - 1$ or less participants will not be able to gain any information about k due to the properties of secret sharing, nor will they be able learn anything about S because it was encrypted before erasure coding.

A general point to note regarding erasure coding and secret sharing is that a participant wishing to reconstruct the original data must contact m data providers (nodes) instead of just a single provider as is the case without erasure coding or secret sharing. This may result in a higher, total latency before the data can be reconstructed, but in exchange for this loss efficiency comes a higher assurance for the client that the data is retrievable, because more than $n - m + 1$ nodes must be unavailable before it is no longer possible to reconstruct the original data.

3.2 System Overview

Accordion is a peer-to-peer, file storage system. The design is based on the idea of storing (parts of) files in small DC-networks (with external recipients) that are part of the larger network similar to the Herbivore design. In Accordion, however, nodes are only members of cliques, ie. are part of a DC-network, if they help store data. Because cliques in Accordion are built using PRNGs, only minimal communication between clique members is necessary when the DC-network is set up, and no round protocol is needed. Also, because cliques only exist when data is retrieved by a client, ie. nodes do not join a clique when they join the network, no global topology algorithm is needed.

As in Herbivore, the size of the DC-network involves a tradeoff between the "degree" of anonymity a node can achieve in that DC-network and the efficiency with which data can

be sent from that DC-network.

Accordion has two basic primitives: *publish* and *retrieve*. These primitives will be explained in the two following sections.

Storage of a file in Accordion is the process of a *publisher* sending the file into the network anonymously to a number of nodes that can later supply the file to a *client* requesting it. The actual file data is split into several *parts* using erasure coding, and each of these parts are stored in the network separately by running the publish process once for each part. Retrieval of a file by a client is the process of requesting a sufficiently large number of parts to reconstruct the file requested.

Different types of nodes store information that is necessary for a client to retrieve a file, ie. a node in the network can play different *roles* that serve different purposes during either publication or retrieval. Data that is stored in the network is either actual data (the parts of the file) or some kind of meta data describing either the file or a part, including pointers to nodes where the actual data is stored. The overall process of storing and retrieving a file in Accordion is illustrated in figure 3.1.

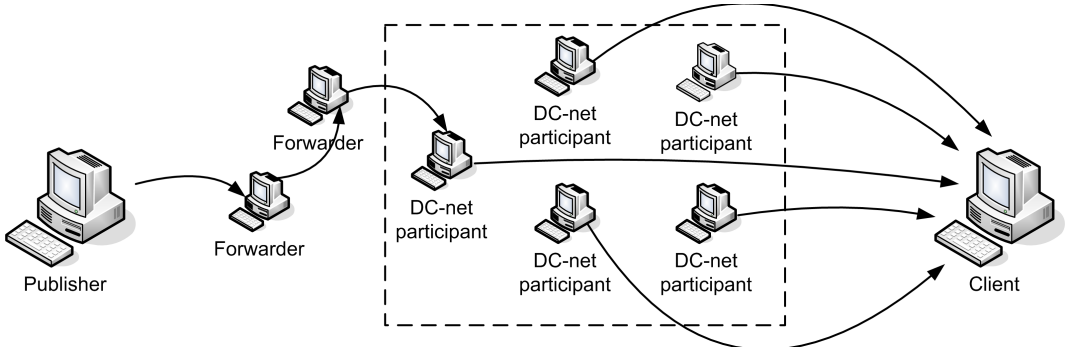

Figure 3.1: The overall process of storing and retrieving a file in Accordion. A publisher stores a file through a number of forwarding nodes, and at some point, five nodes set up a fully connected DC-network between them. Each participant in this DC-network sends a data stream calculated the same way as in section 2.3.5 to the client, which is then able to XOR the streams together to retrieve the requested file.

In the following, an overview of the storage and retrieval processes in Accordion is given. Technical details are deliberately omitted at first to make it easier to understand the overall processes. In sections 3.3.1 and 3.3.2, this overview is followed by a more precise, technical description of the same two processes intended to "fill in the gaps" from the overviews. Finally, in section 3.8, an analysis of the anonymity properties of and some attacks on Accordion, is presented.

3.2.1 Storage Process Overview

The storing of a file consists of three overall steps:

1. Storing file meta data (data that tells clients which nodes to contact to request the particular file, ie. pointers), including a file id that is calculated from the filename.
2. Splitting the file and the encryption key into several parts and storing each part separately.
3. Setting up a DC-network per part that will provide that part of the file on request.

In step 1, the publisher stores meta data for the file he wishes to store. This meta data is used later by clients to locate and request the file. In step 2, the publisher stores parts of the file he wishes to store on random nodes. For each part, one of these nodes will choose to save the part locally. The node that does this is called the *part storer* and this node, along with a number of randomly chosen nodes are called the *part provider list*. The part storer, together with the other nodes in the part provider list will later form a DC-network (step 3) to anonymously provide a client with the particular part of the file stored by the part storer.

This marks the end of the publication process which provides sender anonymity to publishers and storers of data. The next section describes how a client may retrieve a complete file stored in this way knowing only the filename (file id).

3.2.2 Retrieval Process Overview

Retrieval is the process of sending a request containing a file id into the Accordion network and getting back a sufficient number of the parts of the requested file, if available, to reconstruct the file. The retrieval process proceeds in the following steps:

1. Retrieve ids of file parts.
2. Retrieve part provider lists for these parts.
3. Retrieve each file (and encryption key) part from nodes forming a DC-network for that part.

In step 1 of the retrieval process, the client obtains the ids for the parts of the file he wishes to retrieve. These ids will enable it to find the ids of the nodes forming DC-networks for each part (step 2), and from each of these DC-networks he may now request one part of the file and the corresponding part of the encryption key. Upon having retrieved a sufficient number of file and encryption key parts, the client may reconstruct the original file locally.

The following sections present a more precise, technical description of the storage and retrieval processes in Accordion, including details of what data is included in the messages described in the overview. It also gives some of the reasoning behind the design.

3.3 System Details

This section describes the details of publication and retrieval in Accordion.

3.3.1 Storage Process Details

The details of the storage process is divided into a number of distinct steps, performed by several different nodes. These steps will be described in the following. The whole process is illustrated in figures 3.2 and 3.3. In the following, the *publisher* is the node wanting to store a file in the Accordion network.

In the following, the symbol $\rightsquigarrow$ is synonymous with using a Crowds-like technique, so, e.g., $A \rightsquigarrow B$ means A sends a message to B through a number of randomly chosen forwarding nodes.

Step 1: H(filename) = fileId Calculate the hash of the file name using secure hash function H, resulting in the file id. File identifiers in Accordion are chosen as the hash of some descriptive string (in this case the file name) because it provides a convenient way to create unique ids suitable for routing through Pastry, but in principle, any unique id suffices. To protect against "key squatting" a signed-subspace key as in Freenet (as described in section 2.6.3) could be used.

Step 2: $\mathbf{E}_{enckey}$(file) = C Encrypt the file using symmetric encryption algorithm E with encryption key *enckey*, resulting in the encrypted file, C. The reason why the file is encrypted *before* erasure coding is that it ensures that the nodes that will store data (as explained later) can strongly deny having any knowledge of whatever parts they are storing for other users of the network. Erasure coding alone is not sufficient for strong deniability, since it is not designed to make it impossible to gain any information about the whole file from any one part. As described in section 3.1.5, a secret sharing scheme provides the property that any subset of m, $m \leq n$ parts will reconstruct the original data, but no $m-1$ parts will reveal *any* information about the original data. However, to ensure this property, each of the n parts must be at least the size of the original data which makes it unusable in a system like Accordion because files can be big. Instead, the scheme described in section 3.1.5 is used, and that is the reason why the file is encrypted and erasure-coded and the encryption key split using a secret sharing-scheme before publication.

Step 3: EC(C) = $\mathbf{part}_1, \mathbf{part}_2, \ldots, \mathbf{part}_n$ Split the encrypted file C into n parts using any suitable erasure coding algorithm, EC, resulting in parts $1 \ldots n$.

Step 4: H($\mathbf{part}_1$) ... H($\mathbf{part}_n$) = $\mathbf{part\ id}_1$, $\mathbf{part\ id}_2$ ... $\mathbf{part\ id}_n$ Calculate the content hash of all created parts individually, resulting in the *part id list*, and create a list of tuples of the form (part id, part size, part data), one for each part.

Step 5: SS(enckey) = $\mathbf{enckey}_1, \mathbf{enckey}_2, \ldots, \mathbf{enckey}_n$ Split the symmetric encryption key *enckey* into n parts using any suitable perfect secret sharing scheme.

Step 6: Publisher $\rightsquigarrow$ Meta root node : (fileId, part id list) Through a number of randomly chosen forwarding nodes, send the *file meta data* to the meta root node for the file id.

Step 7: Publisher $\rightsquigarrow$ PS$_i$: (part$_i$ id, part$_i$ size, part$_i$, enckey$_i$), $1 < i \leq n$ Also through a number of random forwarding nodes, for all n parts, send a tuple consisting of a part of the file, its id and size and a part of the encryption key to a random node that will act as part storer.

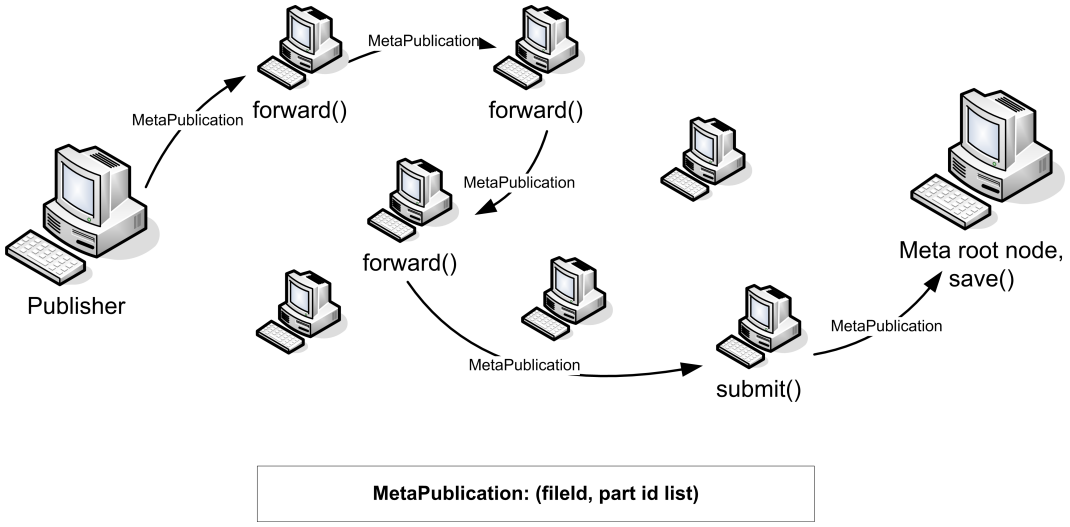

Figure 3.2: Step 6 of the publication process: publication of file meta data (file id, part id list) from the publisher to the meta root node through a number of random forwarders.

When one of the n part storers receive the message from step 7, it executes the following:

Step 1: GeneratePPL() = part provider list The *part provider list* or *ppl* is a list of k unique randomly chosen node ids, including the part storer's own node id, that will provide a client with a part during retrieval by being set up as a fully-connected DC-network among the k nodes. As described earlier, the value of k involves a tradeoff between the number of nodes that the part storer can remain anonymous among and the overhead in the amount of data transferred. After the list has been generated, the part storer shuffles the list such that no one will be able to determine the part storer's identity merely by its position in the list.

Step 2: GenerateSeedList() = seed list The *seed list* is a list of (own id,(neighbour id,seed)) pairs specifying seeds for all edges in a fully-connected DC-network using PRNGs as described in section 2.3.3. The seed list thus specifies, for each node in the ppl, a list of its neighbours and the value of the seed which it shares with that neighbour. A seed list for one node will be known as a *local seed list*.

Step 3: Part storer $\rightsquigarrow$ Part publisher : (part id, part size, enckey size, ppl, seed list) The *part meta data* contains the sizes of that part of the file and encryption key such that the other part providers will know how many bytes to XOR when sending data according to the DC-network protocol during retrieval.

The part meta data gets sent through a number of forwarding nodes chosen randomly *only* from the ppl itself to the *part publisher*, ie. the node that will finalize the part as described next. This node is necessary, because if the protocol stated that the part storer was responsible for submitting part provider list to the part root node, then the part root node would know the identity of the part storer.

When forwarding, the part storer and the part providers *must* choose a node from the part provider list. In the case where the part storer could choose to forward the message to *any* node in the network, such a node receiving the message would be able to suspect that the sender of the message was the part storer because it would be unlikely that the part storer had chosen another *part provider* node first and that node then had forwarded the message to the "outside" node.

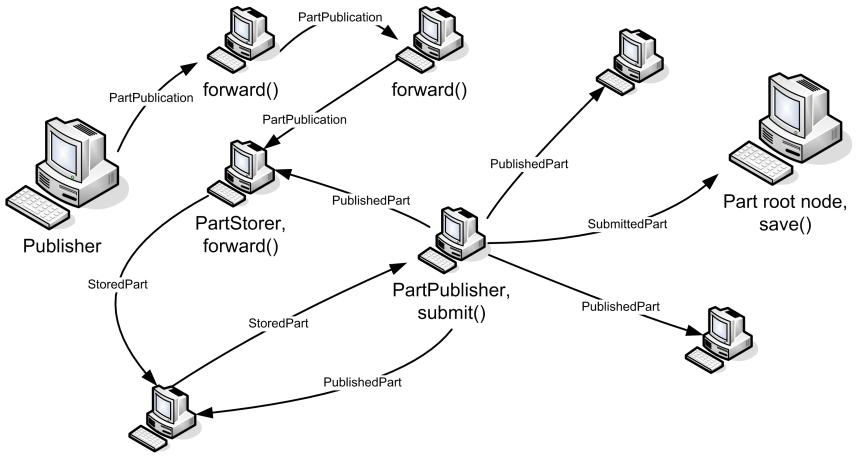

Figure 3.3: An illustration of the part publication process. The Publisher forwards a message containing (part id, part, enckey) through a number of forwarding nodes, one of which chooses to be Part Storer. The part storer then forwards a message containing (part id, part size, enckey size, ppl, seed list) through a number of forwarding nodes (all members of the part provider list) and one of these chooses to be Part Publisher and forwards (part id, part size, enckey size, ppl, local seed list) message to each of the nodes in the part provider list and (part id, ppl) to the part root node.

Upon receiving the message originally from the part storer from one of the other part providers, the part publisher executes the following algorithm to complete the publication of the part.

PartPublisher → PartRN : (part id, ppl) The part publisher sends the part id and the part provider list to the part root node for that part.

PartPublisher → all other nodes in ppl: (part id, size, ppl, local seed list) For each of the $k-1$ other nodes in the ppl, the part publisher sends the part id, the part size, the part provider list and the local seed list. This is done to ensure that all nodes in the part provider list know that they are part of the DC-network for that particular part. A

node in the part provider list receiving a message of this form checks to see if the list contains exactly k elements and that its own node id exists in the list and saves or discards the message accordingly.

Root nodes (meta and part) do not overwrite entries in case of identical ids (file or part). Instead, they save multiple entries and return these to the client during retrieval (see next section).

To ensure strong deniability there must be separate root nodes for the file id and the part ids such that part storers and part providers cannot ascertain—knowingly or accidentally—which file the part they are storing (or providing) is a part of.

Publication of a file is considered successful iff:

- One meta root node saves the tuple (fileId, part id list).
- At least m of n tuples on the form (part id, part provider list) are saved by one part root node per part.
- At least m of n part storers save the tuple (part id, part, enckey).
- At least $k \times m$ part provider nodes (including the part storer) receive and save the part meta data (part id, part size, enckey size, ppl, local seed list) generated by the part storer.

Since replies are not used during publication the only way to discover whether a file was successfully published is to try to retrieve it at a later point.

3.3.2 Retrieval Process Details

Retrieval of a file is the process of supplying a file id to some nodes in the network and getting back either the requested file or an error code. The process is illustrated in figure 3.4 and consists of several steps as detailed below.

Step 1: H(filename) = fileId The client calculates the hash of the file name to get the file id.

Step 2: Client → MetaRN: (fileId) The client sends the file id to the meta root node for that file id.

Step 3: MetaRN → Client: (fileId, part id list) The meta root node responds with the part id list for the given file id (or a list of part id lists if the meta root node is storing multiple, identical file ids) or an error code in case the meta root node didn't have entry for the file id.

Step 4: Client → all n PartRN: (part id) If a positive reply was received from the meta root node, the client sends, in parallel, a part id to each of the n part root nodes.

Step 5: All *n* PartRN → Client: (pid, ppl) Each of the *n* part root nodes respond to the client with a part provider list for the requested part id (or a list of part provider lists if the part root node is storing multiple, identical part ids) or an error code in case the part root node didn't have entry for the requested part id.

Step 6: Client → all *k* nodes in ppl: (part id) If a positive reply was received from a part root node, the client sends, in parallel, the part id to each of the *k* part provider nodes in the corresponding part provider list to request that part.

Step 7: All *k* nodes in ppl → Client: (part id, DC-outcome) Each of the *k* nodes in the part provider list reply with the part id and the outcome of executing the DC-net protocol as described in section 2.3.5. Included in the DC-outcome is the part data and the encryption key data. Together, the nodes in the part provider list provide the client with one part of the file and one part of the encryption key.

If the client successfully receives *m* parts of a file and *m* parts of the corresponding encryption key, *enckey*, it can reconstruct *enckey* and the file—and finally decrypt the file using *enckey* resulting in a succesfully retrieved file.

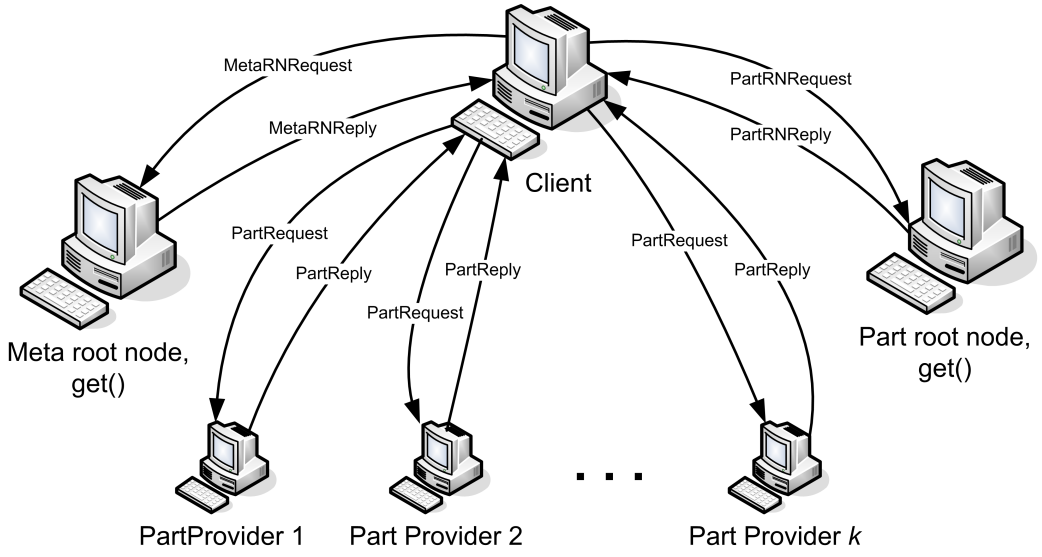

Figure 3.4: Retrieval of the file meta data and one part of the file and encryption key.

3.4 Node Join And Leave

Accordion is public system and hence any node can freely join the network by supplying the IP address of a *bootstrap* node, ie. a node that knows a number of other nodes in the network. When a node joins the network it will perform checks to see whether it will be responsible for any ids (play the role of either meta or part root node). This is described in the next section. This check is not performed by part providers because messages for part providers are routed for the exact node id of the part provider and thus a node joining the network cannot end up getting a message originally intended for an existing part provider. This is also a security measure to ensure that an active adversary cannot insert itself in "the middle of" a part provider list and thus try to single out the part storer by controlling $k - 1$ in the DC-network (see section 3.8.4).

Nodes can leave the network at any time and there is no exit protocol. This is because nodes can disconnect (voluntarily or by failure) at arbitrary times and hence a exit protocol might not be executed. Only part provider nodes locally saves (persists) any data (meta or actual) that it is responsible for when they disconnect from the network (or periodically while connected), such that if/when the node later connects, its state can be restored (of course, this state might not be useful anymore because of changes in the network, e.g. one or more other nodes in ppls that the node is a part of might have disconnected).

3.5 Availability

As described in section 3.1.5, to increase the availability of files stored in a file storage system both replication and erasure coding can be used. Increasing the availability of a file not only means increasing the availability of the file itself but also of the meta data describing the file.

As described in section 3.1.5, the availability of the actual files is ensured through erasure coding producing a number of parts that constitute a file. The same technique could be used for ensuring availability of the meta data describing the file but since the size of meta data is usually small (compared to files themselves) this would result in an unnecessary high number of lookups each returning only very little information ($1/n$ of the meta data). Therefore, replication is chosen for ensuring availability of meta data in Accordion.

In Accordion, two types of meta data exists:

- Meta data describing the file (stored on meta root nodes)
- Meta data describing the parts (stored on part root nodes)

Each of these types of meta data are replicated as follows: when a node becomes root node (meta or part) for an id (file or part) it sends the corresponding row (key,value) to the j nodes in the leaf set with id numerically closest to the id (and itself). It is then the job of Pastry to notify Accordion about changes in the leaf set, ie. if a node joins or leaves, such that the

row can always be found on the j nodes with node ids numerically closest to the id. This is identical to how PAST [RD01b] maintains j replicas of files and how TAP maintains tunnels, see section 2.6.5.

3.5.1 Attacks

Attacks on the availability of files will be carried out by attacking one of the three previously mentioned entities that constitute a file. Generally, three types of attacks are possible:

- Disconnecting nodes (either root nodes or part storers) making certain parts or even files unavailable (until the file is possibly republished, see next section).
- Causing disruption in the DC-networks.
- Acting maliciously in some other way by e.g. not forwarding messages or not saving data.

Direct protection against these attacks is not offered by Accordion. Instead, two factors either implicitly protect against the attacks or minimize the effects of the attacks:

- The peer-to-peer factor (see section 2.5), e.g. the geographic or administrative dispersal of nodes makes it hard for a limited adversary to control or attack specific nodes.
- Erasure-coding of files and replication of root node data. Even if an adversary is able to disconnect specific nodes or act maliciously as described above, it is unlikely that it will be able to make more than $n - m + 1$ parts unavailable.

Other systems such as Free Haven [Din] attempt to identify malicious behavior by the use of a reputation system. Research is, however, still needed in this area and is out of scope for this project.

Attacks on the anonymity of Accordion will be described in section 3.8.

3.6 Data Management Strategy

Only a finite amount of disk space exists and thus the storage of a file in Accordion (or any other file storage system) will fail if there is no free disk space. Several solutions to this problem are possible:

- Introducing a timeout value when publishing files. This way a publisher would set a timeout value in e.g. hours or days on the file level (or -1 for no expiration) and both root nodes and part providers would periodically check the status of their tables and delete entries that have timed out. Of course, it could not be guaranteed that nodes would actually neither save nor delete entries according to the protocol.

- Adding the ability to delete files (and meta data). Authentication of users could be done in a way similar to Freenet's signed-subspace key but using a pseudonym key as not to identify the publisher. Since users can (should) not be forced to delete their files this does not guarantee available disk space.
- A distributed garbage collection process could detect "stale" parts by detecting stale part provider lists, ie. part provider lists where one or more nodes have disconnected. Note that a signal could not be sent to the publisher of a file because its identity is not known.

It is not a design goal of Accordion to offer permanent file storage. Rather, publishers of files are expected to *republish* their files periodically. An alternative to this would be for part storers to automatically and periodically republish the part(s) they store. This could be done in two ways:

- If part storers republish parts and then delete them, availability of those parts wouldn't necessarily be increased since the new nodes that would become part storers could disconnect from the network moments after becoming part storers, making the parts unavailable.
- If part storers republish parts without deleting them locally then parts are effectively replicated which increases disk and bandwidth by a factor of 11 (see section 3.1.5) which is unacceptable.

Therefore, manual republication by publishers is chosen.

3.7 Miscellaneous Design Issues

This section briefly describes some design issues that are relevant in a file-storage system.

3.7.1 Censorship-resistance

A censorship-resistant system is a system with the property that it is hard for any adversary to exercise censorship on a given file in such a way that the file becomes unavailable. Anonymity is often used as a tool in censorship-resistant systems. This is because users do not only want their content to remain available but also to be anonymous when publishing or retrieving it.

Although relevant for a file-storage system, censorship-resistance is not a direct goal in Accordion and therefore, only the minimal censorship-resistance that erasure coding of files and replication of meta data offers, is provided. Systems such as Publius [MWC00], Tangler [WM01] and Free Haven [Din] focus on censorship-resistance.

3.7.2 Load-balancing

Root nodes that point to popular parts might experience a massive amount of requests and in those cases load balancing would be useful to reduce the CPU and bandwidth consumption of these nodes. One way to do this, is to build a multicast group (using e.g. Scribe [CDKR02]) such that all nodes in the group hold the root node information and then using anycast upon retrieval contacting an arbitrary node in the group. The node that is contacted in the group is dependent on the client's position in the network. This balances the load over the entire group.

There is no obvious way to load balance part provider lists that provide parts that are part of popular files but the fact that clients can retrieve *arbitrary m* parts of a file provides some implicit load balancing.

3.7.3 Searching for Keys

Accordion does not provide a direct mechanism for searching for file identifiers, or keys: the client has to know the precise filename that the file was published under to be able to calculate the file id and retrieve it.

One way to improve upon this (as suggested in Freenet [CMH+02]) is to extend meta data by adding additional descriptive keywords when publishing a file. This way several root nodes will hold pointers to the file and the client can find the file without knowing the precise filename of the file.

3.7.4 Updateable Files

It is not possible for a user that has stored a file in Accordion, to update that file later. An altered version of a file stored later can be seen as an updated file and thus Accordion supports so-called *archival storage* as in e.g. OceanStore [KBC+00].

An update function could be implemented using a content hash key as described in section 2.6.3.

3.8 Anonymity Properties

In this section we present the anonymity properties of Accordion. The roles that seek anonymity in Accordion are the publisher and the part storer. The publisher obviously seeks sender anonymity during both the publication of meta data and parts. The part storer seeks sender anonymity during publication and during retrieval when sending a file to a client.

Unlinkability is not an explicit demand for Accordion and hence only protection against passive adversaries via payload encryption and padding is offered. Neither unobservability is a

goal for Accordion and it is not possible because white noise is not used during publication and the client is not part of the DC-network during retrieval, so all participating nodes in the DC-network will know the identity of the client.

In general, the publisher achieves the anonymity properties of the basic proxy technique described in section 2.1.1 because of the Crowds-like technique used and the part storer achives the anonymity properties of the DC-network technique described in section 2.3 during retrieval including any attacks described on these techniques. The part storer also achieves recipient anonymity wrt. the publisher because the publisher cannot know which node chooses to save the part.

Accordion uses payload encryption on all Pastry links, ie. the link between the source of a Pastry message and the message's final destination, such that a passive adversary can only tell that any observed message is a Pastry message (the TCP payload is encrypted but the port number for the Pastry application is in cleartext). If all Pastry traffic to and from the node whose connection is being monitored (publisher or part storer) is encrypted, a passive adversary will not be able to break that node's sender anonymity, because it cannot determine whether the message sent was a message sent by Accordion or another application built on top of Pastry. If the adversary knows that Accordion is the only application built on Pastry that encrypts messages, the adversary can break the node's sender anonymity but cannot know for what.

During both meta- and part-publication an active adversary controlling the first randomly chosen node after the publisher, cannot break neither the publisher's sender anonymity because eventhough this node can see the message in cleartext it will not know whether the source address it sees is the original source address, cf. the *proxy argument*. Also, during part publication, the message sent from the part storer to the first node in the part provider list is padded to the size of the message that contained the part. Due to this and the proxy argument neither a passive nor an active adversary will be be able to identify the part storer.

Below is a non-exhaustive, but representative list of attacks and how they affect Accordion's anonymity properties.

3.8.1 Passive And Active Attack

From the properties above it follows that if a passive and an active adversary collaborate they will be able to break the publisher's sender anonymity for a given file or part (meta- or part-publication, respectively) because the passive adversary can perform the *wiretapper attack* and know *that* the publisher is publishing something and R_1 will know *what* was published.

3.8.2 Volume Attack

A passive adversary can also use the volume of messages going out of a node to perform the wiretapper attack. If this adversary observes a suspiciously large number of messages going out of a node, all of the same size and equal to that of a message containing a part, it can

with high probability conclude that this node was publishing something thus breaking the node's sender anonymity because it is unlikely that any other non-publishing node would send out the same number of equally sized messages in such a short time frame. Again, if the attack is combined with an active adversary controlling (some of) the destinations of the outgoing messages, the attack can reveal (some of) the parts that were published.

This attack can of course be avoided if a delay between each published part is introduced.

3.8.3 Statistical Attacks

At least two types of attack where a distributed adversary gathers statistical information to break (sender) anonymity of nodes are possible in Accordion. The first is similar to the predecessor attack described in 2.6.1 and the second is similar to the statistical attack described in 2.6.7.

Predecessor Attack A predecessor-type kind of attack can be carried out in Accordion by an active adversary controlling a number of nodes in the network, ie. collaborating nodes. Each path set up during publication of either meta or part data corresponds to a path reformation because each message takes a different route through the network. This means that the publisher will be the node that most often appears as the predecessor to one of the collaborating nodes and these nodes can gather statistical evidence to break the publisher's sender anonymity.

Herbivore Statistical Attack An adversary observing a node in Herbivore disproportionately often being a member of a clique that contacts a particular server, has statistical reason to believe that the node is communicating with that server and thus break its sender anonymity. The same attack in Accordion would be carried out by an adversary retrieving a specific part multiple times over a longer period of time and observing that a particular node was always a member of the part provider lists for that part. This is not the case, however, as part storers stay in the same part provider list. Also, if a file is republished, it is very unlikely that the same nodes (in a big network) would be chosen as part storers again.

3.8.4 k-1 Attack

An adversary that has compromised the $k-1$ other nodes than the part storer in a part provider list will be able to break the part storer's sender anonymity because it knows that only that node could store the part. Equivalently, if the part storer chooses $k-1$ nodes controlled by the adversary when creating the part provider list, the adversary will be able to break the part storer's sender anonymity.

All attacks so far has been on the publication process. If no attacks have taken place during publication, the sender anonymity of the part storer during retrieval cannot be broken due

to the nature of DC-networks.

The attacks on Accordion described above are hard to perform because nodes are chosen randomly throughout the entire network and it is hence unlikely that an adversary is able to compromise or listen on the connections of the "right" nodes. An adversary that is able to perform one of the attacks on the part storer described above will only break anonymity for the given part. In that case, the part storer has *strong deniability*. Attacks on the publisher's sender anonymity will also with high probability only break the anonymity for one or few parts because it is unlikely that the adversary is lucky enough to control all the nodes that are chosen as the first nodes on the publication paths of seperate parts.

3.9 Comparison of Systems

The only existing system that matches Accordion's overall design goals (anonymous, low-latency, peer-to-peer file storage) is Freenet. It is therefore relevant to compare the properties of Accordion with those of Freenet.

Freenet uses a Crowds-like technique during both publication and retrieval and thus achieves the anonymity properties of this technique. However, unlike the basic proxy technique, Freenet, like Accordion, achieves *recipient anonymity* during publication wrt. the publisher and a passive adversary that cannot perform a wiretapper attack on the node that will store the file because these entities cannot know which node will ultimately store the file (the key-node link cannot be established). In Accordion, though, recipient anonymity is only achieved among the k nodes in the part provider list.

Freenet is susceptible to some of the same attacks as Accordion, e.g. the Predecessor attack (described in the previous section).

When a file is retrieved in Freenet it is cached on the nodes on the reply path. Accordion has no caching strategy and Freenet is thus likely to provide more efficient retrievals than Accordion. Also, since files are erasure coded in Accordion, latency is likely to be higher since more nodes must be contacted to retrieve a file. Finally, the amount of data transferred during retrieval in Accordion is k doubled due to the DC-networks.

On the other hand, Accordion provides stronger anonymity properties for storers (and thereby senders) of data due to DC-networks. Accordion also provides strong deniability whereas Freenet only provides plausible deniability.

Other systems have similar design goals, such as Free Haven, GNUnet and Publius. However, GNUnet is peer-to-peer but is file sharing (not file storage), Free Haven and Publius are both centralized systems, Free Haven does not provide low-latency transfers and none of the systems provide strong deniability.

Chapter 4

Implementation and Test

This chapter describes the implementation and test of a prototype of Accordion as presented in the design in the previous chapter. The prototype should be seen as proof-of-concept application. The Java code can be found in appendix A.

4.1 Implementation Overview

A prototype has been developed in Java and uses one of the Java implementations of Pastry, FreePastry [PDZ04], as its routing layer. The prototype uses FEC [Cha01] for erasure coding of files and the javax.crypto package for encryption routines.

The implementation has the following limitations compared to the design:

- Payload encryption and padding of messages. An adversary will thus be able to perform attacks as described in section 3.8.
- Strong deniability. The current prototype only supports plausible deniability since decryption keys for published files are sent in one piece to meta root nodes.
- Backup of root nodes. If a root node (meta or part) leaves the network the corresponding file / part ids will be unavailable unless published again.
- Random choice of part providers. The k nodes chosen for part provider lists are chosen locally from the part storer's routing table. This means that the prototype only supports DC-networks as large as the maximum number of unique nodes in the FreePastry routing table.

Also, since the prototype is supposed to function as a proof-of-concept application, no real user interface has been developed.

Libraries that implement encryption, padding and secret sharing exist and thus payload encryption, padding and strong deniability could be implemented within a reasonable timeframe. Also, to implement backup of root nodes, FreePastry offers events that signal changes in e.g. the route and leaf set (as described in section 3.5).

4.2 Implementation Details

Figure 4.1 shows a class diagram of the central classes of Accordion (excluding third-party libraries).

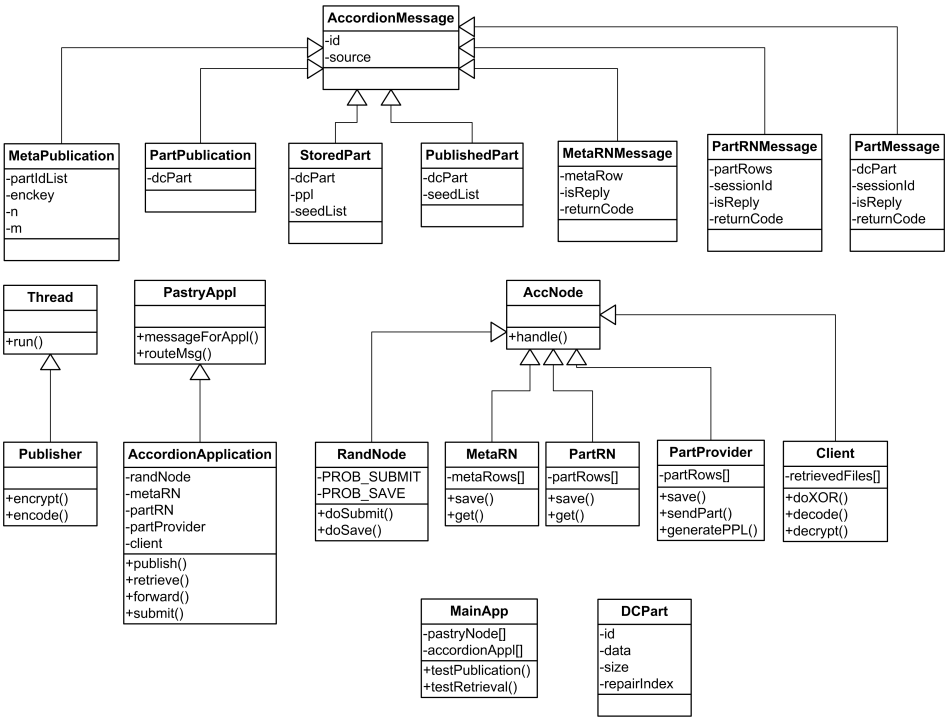

Figure 4.1: A class diagram of the Accordion source code, excluding third-party libraries.

The MainApp class contains two methods to test publication and retrieval of a file, respectively. The class also contains a list of PastryNode objects that each contain a reference to an AccordionApplication. When a node publishes or retrieves a file from the MainApp, the appropriate AccordionApplication object will take over. This object extends the PastryAppl class from the FreePastry library and will in turn call methods in FreePastry for sending messages. An AccordionApplication can send an AccordionMessage in one of the following ways:

forward(AccordionMessage am) Send am to a randomly chosen node.

submit(MetaPublication mp) Send mp to the meta root node for mp's id.

submit(Id rn, AccordionMessage am) Send am to the (part) root node for am's id.

Each method is implemented using the routeMsg method from PastryAppl that send the message to the live node in the network with node id numerically closest to key.

An AccordionApplication contains references to the six different roles a node in the network can play: Publisher, RandNode, MetaRN, PartProvider, PartRN and Client. Each of these classes implements the AccNode interface that merely specifies that the role in some way must be able to *handle* a message of type AccordionMessage.

In the following, each time a node saves a message, it does this by saving an object that represents a row containing the same information as the message and adding this row to a local table of these rows.

4.2.1 Publisher

The purpose of a Publisher object is – given a file and a corresponding filename – to publish these as described in section 3.3.1. The publisher generates one MetaPublication message and n PartPublication messages and sends each of these to a randomly chosen node. Each of these nodes handles the message as described next.

4.2.2 RandNode

A RandNode object plays the role of a node that gets randomly selected by the Publisher during either publication of meta data, publication of parts or by the part providers when forwarding the message containing the part meta data within the part provider list. A RandNode object handles an AccordionMessage according to the following list:

MetaPublication or StoredPart Forward or submit. When submitting, let a MetaRN or PartProvider object handle the message, respectively.

PartPublication Forward or save. When saving, let a PartProvider object handle the message.

PublishedPart Let a PartProvider object handle the message.

The probability of a RandNode object either forwarding/submitting or forwarding/saving is controlled by two parameters:

PROB_SAVE The probability that a randomly chosen node will act as part storer and save the part contained in the PartPublication message.

PROB_SUBMIT The probability that a part provider will submit the StoredPart message to the rest of the part provider list and the part root node.

The effect of changing the value of these parameters can be seen in the tests in section 4.6.

4.2.3 MetaRN

A MetaRN object handles a MetaPublication message by saving it locally (during publication) and a MetaRNMessage by returning any rows matching the file id contained in the message to the source of the message, the client (during retrieval).

4.2.4 PartProvider

A PartProvider object either plays the role as the part storer or as one of the other nodes in the part provider list. A PartProvider object handles an AccordionMessage according to the following list:

PartPublication Save PartPublication message and hence become part storer. This includes generating the part provider list, generating seed lists that specify the fully connected DC-network, and sending a StoredPart message to a randomly chosen node within the part provider list.

StoredPart This node will become the part publisher. This includes saving the StoredPart message (if part storer != part publisher, otherwise the information will already have been saved by the part storer) and sending a PublishedPart message to the remaning nodes in the part provider list and a SubmittedPart message to the part root node.

PublishedPart Save the PublishedPart message (the part storer does not save it again, though).

PartMessage According to the DC-network protocol, send a message containing a stream of bytes to the source of the original message. This means, if the node can find an entry in its table corresponding to the id in the PartMessage, it will pull out the $k-1$ PRNG-seeds for that id and create $k-1$ "coins" (byte streams) to be XOR'ed together. The part storer will also XOR the bytes of the file with the $k-1$ byte streams.

4.2.5 PartRN

A PartRN object handles a SubmittedPart message by saving it locally (during publication) and a PartRNMessage by returning any rows matching the part id contained in the message to the source of the message, the client (during retrieval).

4.2.6 Client

A Client object takes as input a file id and is responsible for contacting the relevant nodes to retrieve the corresponding file, if available at that time. A Client object handles an AccordionMessage according to the following list:

MetaRNMessage If the meta root node returned a MetaRNMessage containing one or more rows, pick one of these, save it (using the session id as described below) and contact the part root nodes corresponding to the part ids contained in the message.

PartRNMessage If a part root node returned a PartRNMessage containing one or more rows, pick one of these, save it and contact the part providers in that row.

PartMessage If a part provider returned a PartMessage with a return code equal to 0, do the following:

1. Find the bucket corresponding to the session id.
2. Find the subbucket corresponding to the part id.
3. If that subbucket does not exist, create it and in it save the byte stream sent by that part provider.
4. If the subbucket does exist and there are fewer than k byte streams in it, save the byte stream.
5. If k byte streams have been retrieved but not XOR'ed yet, XOR the k byte streams to form a part. Also, increment a counter that keeps tracks of how many parts have been succesfully retrieved.
6. If the value of that counter has reached m and the file has not yet been decoded, decode the file and finally decrypt the file using the key saved when receiving MetaRNMessage from the meta root node.

Since a part storer learning the corresponding file id for the part it is storing, would break its plausible deniability, an integer session id is used to keep track of the incoming PartMessages from part providers such that potential PartMessages from simultaenous file transfers can be separated. A session id thus corresponds to a file id but without revealing any information about the file.

4.2.7 Running the Application

The following briefly describes how the prototype application can be run either as a simulated network or in a real network.

Simulated Network

Using FreePastry's built-in network simulator an entire network can be simulated on one physical node. The application takes the number of nodes as an argument and FreePastry then creates a Pastry network according to [RD01a], including setting up routing tables, leaf sets etc. Each of the nodes created will be a Pastry node with an Accordion application on top. Once the network has been set up, a test publication followed by a test retrieval of a junk file with a pseudo-random filename can be performed.

Real Network

Due to the way FreePastry has been developed switching from a simulated to a real network, is basically setting a flag. Then a Java runtime enviroment (JRE), FreePastry, FEC and of course the Accordion application need to be deployed on the nodes that are going to make up the network. When this is done, one node is started as a bootstrap node and the remaining nodes are then started with a reference to the bootstrap node. From there on, the program is run as in the simulated network.

4.3 Test Strategy

The purpose of the test is to verify the functionality of the Accordion. Test of third-party libraries such as FreePastry and FEC have not been performed because (1) the correctness of these libraries have already been tested by their authors and FreePastry has been used in real-world applications such as PAST, Scribe and SplitStream, and (2) such tests are out of scope for this work.

Full correctness tests have not been performed, ie. the application has not been tested for all values of all possible parameters. To do this, a construction of a formal model of the system is needed. Instead, the tests aim to convince the reader that the prototype implements the functionality of the design correctly.

Also, because the tests have primarily been performed in a simulated network, performance tests have not been performed. We can only verify that a file can be successfully retrieved seconds after it has been published and that there are no delays when retrieving a file.

The following types of tests have been performed:

- Role test. Verification that all roles such as root nodes, part storers, and part providers are handled by the correct nodes in the network using fixed parameters (see below). This includes verification of the steps that make up publication and retrieval. A succesful role test is a prerequisite for the functional tests.
- Anonymity test. Verify that the Crowds-like and DC-network techniques that Accordion uses for publication and retrieval, respectively, function properly.
- Functional tests. View the system as a black box and verify that a file can be successfully published and retrieved, adjusting different parameters.

Different parameters affect the test of a peer-to-peer application, e.g. network size and the dynamics (ie. node join and leave) of the network. The following parameters have been taken into account when performing tests on Accordion:

- The size of the network, N.

- The size of the part provider list, ie. the DC-network, that provides a file to a client, k.

All tests have been performed in a simulated network, ie. using FreePastry's built-in network simulator. The application has also been tested successfully in a small LAN with 5-10 nodes.

4.4 Role Test

The following section presents an example of the publication and retrieval processes as they work in the Accordion application, complete with output so as to illustrate and explain the individual steps with a real example in a simulated network.

The parameters for the example are as follows:

Network size (N): 1000 nodes

Part provider list (DC-network) size (k): 5 nodes

Erasure coding: $m = 4$ and $n = 16$ (only the details of one part publication will be shown below)

File name: ''`The meaning of Life.txt`''

File contents: ''`In the beginning God created the heaven and the earth. And the earth was without form, and void; and darkness was upon the face of the deep. And the Spirit of God moved upon the face of the waters. And God said, Let there be light: and there was light. And God saw the light, that it was good: and God divided the light from the darkness. And God called the light Day, and the darkness he called Night. And the evening and the morning were the first day.`''

PROB_SAVE: 0.1, to make it likely that the publication message will pass through many nodes before being saved.

PROB_SUBMIT: 0.4, to make it unlikely that the message is passed around from one part provider node to another many times.

File Name Hash (fid): `0xCA843F9B9BC8809CE76DAB3570F339402F330578`

Encryption key: `0x59C05AF4EA8D4FD1`

To enable direct comparison with the retrieved data later, here is a hexadecimal representation of the string from the excerpt from Genesis above:

```
Hexadecimal representation of contents in plaintext:
496E20746865206265676967696E6E696E6720476F64206372656174652064207468652
```

```
068656176656E20616E64207468652065617274682E20416E6420746865206561
727468207761732076974686F757420666F726D2C20616E6420766F69643B206
16E64206461726B6E6573732077617320757 06F6E207468652066616365206F66
207468652064656 5702E20416E64207468652053706972697 4206F6620476F642
06D6F7665642075706F6E207468652066616365206F662074686520776174657 2
732E20416E6420476F6420736169642C204C65742074686572652062652065206C696
768743A20616E64207468657265207761732 06C696768742E20416E6420476F64
207361772074686520 6C696768742C207468617420697420776173206 76F6F643
A20616E6420476F64206469766964656420746865206C696768742066726F6D20
746865206461726B6E6573732E20416E6420476F642063616C6C6564207468652
06C6967687420446 1792C20616E64207468652064617 26B6E6573732068652063
616C6C6564204E696768742E20416E64207468652065766 56E696E6720616E642
0746865206D6F726E696E67207765726520746852066697273742064617 92E
(length = 454 bytes)
```

Before the file is published, its contents are encrypted. The encrypted contents are (in hexadecimal because the contents are no longer pure ASCII):

```
Hexadecimal representation of encrypted data:
0xD3D599E2FD5C18FE754468220830F7372B5CC6A683E3121384E61F196F8D679A
589967AA864F27E1EFB6E0ED7F18BCB3DF5D01DDAD8AEF083E7EC26278458E441E
E72DA0DC6C841D63EEB2221493FA4CAE44D6E6D5133862E56992C7488C9BC769CD
BC11AC8BBEE038F1D6BAA22EBCBC4B6CF400160C224CF447F6F405373FB2BE9E19
EF2E0F9244A3CE81CC7CA83E1FB173CE59C96E810B38C8F0FCAE8D6E121C46112C
865F2A796A25844659A4778628EF055C8741AF83017228CBCCEE6CB2E96EC06B0A
976751C45DEDE95E5581321E78A4BEC2D55FD69B480545872473F23CF9531B852E
7F9D489152AD77E28340C05E0F1D5CB4790776C3EC29F29B46ADE6C69B6A0BD0B3
3F7398D3ED4CE8D739B2675F4E4EDE2FBE9D7A502915B870FA8C6CABEA30915762
0277098F807007E5A13D7DF13A2141694FE68F258F62E3C2E912A8DA76FDC0B5DE
497C7BE09AC7F9979ED2BA7414107FEC85C747D27D44A9EFB79EA0844732A1DD2D
3A3F3E9C981829D606ED60252D39DE497C7BE09AC7F985355035EE4D239A1C6374
6BB5070C8777459F24B22D036A9AD8F08C62A031672BB5ADEA9E11F328B8C7DFA9
92104434EE17754E6A55B4DD2E107E350BC472D54B73EB614118BF10
```

The encrypted file is 456 bytes long (as opposed to the original string of 454 bytes), because the encryption algorithm works on blocks of 8 bytes, and thus the length of the cipher text is padded to the nearest multiple of 8. Note that if the same file is published twice, the encrypted contents will differ because the encryption algorithm uses a new, random initialization vector each time.

Parts of the contents of the 16 erasure coded parts and parts of their correspondig part ids (hash of contents) are seen from the following program output:

```
Data for part 1: <0xD3D599...>, part id: <0xA6C65C..>
Data for part 2: <0xF40016...>, part id: <0x0F2EEF..>
```

```
Data for part 3: <0x852E7F...>, part id: <0x0779B4..>
Data for part 4: <0x107FEC...>, part id: <0x2DDDA1..>
Data for part 5: <0x91C261...>, part id: <0xB80C71..>
Data for part 6: <0xF39C95...>, part id: <0xF99059..>
Data for part 7: <0x52D603...>, part id: <0x2C718D..>
Data for part 8: <0x902D97...>, part id: <0x3EE311..>
Data for part 9: <0xE793B1...>, part id: <0xC70488..>
Data for part 10: <0xFF4830...>, part id: <0xC463AD..>
Data for part 11: <0x5FFA40...>, part id: <0xC44C81..>
Data for part 12: <0x27F1C6...>, part id: <0x20E916..>
Data for part 13: <0x818224...>, part id: <0xF18F54..>
Data for part 14: <0xCB0481...>, part id: <0xAF047E..>
Data for part 15: <0xF993F7...>, part id: <0x6B6237..>
Data for part 16: <0x215AB1...>, part id: <0x95405D..>
```

4.4.1 Publication Test

The first step in publication is for the publisher to publish meta data for the file he wishes to store. In this example, node `<0xADD471..>` is publisher:

```
Publisher <0xADD471..> publishing meta data for file id <0xCA843F..>

Random node <0x2E1E56..> forwarding MetaPublication for
       file id <0xCA843F..> to another random node
Random node <0xA89BD6..> forwarding MetaPublication for
       file id <0xCA843F..> to another random node
Random node <0x2EAA25..> forwarding MetaPublication for
       file id <0xCA843F..> to another random node
Random node <0x507BA4..> submitting MetaPublication for
       file id <0xCA843F..> to the meta root node

Meta root node <0xCAB00A..> saved meta data for file id <0xCA843F..>.
       Meta data includes the following part ids: <0xA6C65C..>,
       <0x0F2EEF..>, <0x0779B4..>, <0x2DDDA1..>, <0xB80C71..>,
       <0xF99059..>, <0x2C718D..>, <0x3EE311..>, <0xC70488..>,
       <0xC463AD..>, <0xC44C81..>, <0x20E916..>, <0xF18F54..>,
       <0xAF047E..>, <0x6B6237..>, <0x95405D..>,
       and the encryption key is: 59C05AF4EA8D4FD1
```

As the program output above shows, the meta data is correctly forwarded through a number of randomly selected nodes before being submitted to the meta root node. As expected, the meta data above comprises the 16 correct part ids and the symmetric encryption key used by the publisher to encrypt the file.

The next step is for the publisher to publish each of the erasure coded parts of the file. In this test there are 16 parts in total, but only publication of part 8 will be illustrated and explained here, since the output for all 16 parts differs only in part ids and node ids for nodes that will handle the publication for the particular part.

Part publication is similar to meta publication in that data (in this case a PartPublication message) is sent through a number of random forwarding nodes before one node chooses to become part storer:

```
Publisher <0xADD471..> publishing part data for part id <0x3EE311..>

Random node <0x201F3E..> is forwarding a PartPublication message
Random node <0xA46283..> is forwarding a PartPublication message
Random node <0xB69331..> is forwarding a PartPublication message
Random node <0x7B801D..> is forwarding a PartPublication message
Random node <0x0B2E61..> is forwarding a PartPublication message
Random node <0x193353..> is forwarding a PartPublication message
Random node <0x5E92A8..> is forwarding a PartPublication message
Random node <0x5519DB..> is forwarding a PartPublication message
Random node <0xBFE8AA..> is forwarding a PartPublication message
Random node <0x1415C6..> is forwarding a PartPublication message
Random node <0x47A9A8..> is forwarding a PartPublication message
Random node <0xD53A06..> is forwarding a PartPublication message
Random node <0xAB893F..> is forwarding a PartPublication message
Random node <0xEB2D38..> is forwarding a PartPublication message
Random node <0x7F73E7..> is forwarding a PartPublication message
Random node <0xF9490B..> is forwarding a PartPublication message
Random node <0xD16D73..> is forwarding a PartPublication message
Random node <0xE32ED8..> is forwarding a PartPublication message
Random node <0x739916..> is forwarding a PartPublication message
Random node <0x8F67D9..> is forwarding a PartPublication message
Random node <0xB79A19..> is forwarding a PartPublication message
Random node <0x38C258..> is forwarding a PartPublication message
Random node <0xC62D34..> is forwarding a PartPublication message
Random node <0x3ACED1..> is forwarding a PartPublication message
Random node <0x3FB8EB..> is forwarding a PartPublication message
Random node <0xD8BB58..> is forwarding a PartPublication message
Random node <0xED8DB3..> is forwarding a PartPublication message
Random node <0x6BA435..> is forwarding a PartPublication message
Random node <0xF5924F..> is forwarding a PartPublication message
Random node <0x15ADF8..> is forwarding a PartPublication message

Node <0x86047A..> is PartStorer and saved all data for part id <0x3EE311..>
     and selected the following nodes to be included in the part provider
     list: <0x81E76B..>, <0x86EA09..>, <0x86047A..>, <0x8225EC..>, <0x861A00..>
```

Notice that the part provider list has been shuffled so that the part storer's own id is at a random index in the list.

The part storer creates a message containing the part meta data, including the part provider list, `StoredPart`. From this point on, the message is passed around the nodes in the part provider list only until one part provider chooses to be part publisher:

```
Random ppl node <0x86EA09..> is forwarding StoredPart message <0x3EE311..>
Random ppl node <0x861A00..> is forwarding StoredPart message <0x3EE311..>

PartPublisher <0x8225EC..> saved information about part id : <0x3EE311..>
```

In this example, node `<0x8225EC..>` chooses to become Part Publisher, but it could have been any of the nodes in the part provider list. Now the part publisher notifies all nodes in the part provider list that they are part providers for part id `<0x3EE311..>` and finally submits part information to the part root node:

```
PartProvider <0x81E76B..> saved information for part id <0x3EE311..>
PartProvider <0x861A00..> saved information for part id <0x3EE311..>
PartProvider <0x86EA09..> saved information for part id <0x3EE311..>

Node <0x3EBE92..> is part root node for part id <0x3EE311..>
```

This marks the end of a successful publication for one part. Illustrated next is the retrieval process for file with file id `<0xCA843F..>` which part `<0x3EE311..>` is part of.

4.4.2 Retrieval Test

The first step in retrieval is for the client to contact the meta root node for the file id he wishes to retrieve. In this example, node `<0x061E91..>` is client.

```
Client <0x061E91..> contacting meta root node requesting file id <0xCA843F..>
```

The meta root node `<0xCAB00A..>` receives this request and responds:

```
Meta root node <0xCAB00A..> received a request for file id <0xCA843F..>
     from <0x061E91..>, sending part id list
```

The part id list contains the 16 part ids stored by the meta root node during publication, as well as the symmetric encryption key used by the publisher to encrypt the file. (As noted earlier, only plausibe deniability is implemented, and thus the meta root node knows the

full encryption key.) When the client receives this list, it may start contacting part root nodes to acquire part provider lists for the 16 parts, here only illustrated for part 8 (the rest of the requests differ only in the ids of the nodes contacted). Both part root nodes and part providers are contacted in parallel:

```
Client <0x061E91..> contacting part root node for part id <0x3EE311..>
```

The part root node <0x3EBE92..> receives this request and responds:

```
Part root node <0x3EBE92..> received request for part id <0x3EE311..>,
     sending part provider list.
```

The client receives the part provider list for part id <0x3EE311..> from the part root node and may now contact each of the five part providers individually and request the part from them:

```
Client <0x061E91..> contacting part provider node <0x86047A..>
       for part id <0x3EE311..>
Client <0x061E91..> contacting part provider node <0x81E76B..>
       for part id <0x3EE311..>
Client <0x061E91..> contacting part provider node <0x861A00..>
       for part id <0x3EE311..>
Client <0x061E91..> contacting part provider node <0x86EA09..>
       for part id <0x3EE311..>
Client <0x061E91..> contacting part provider node <0x8225EC..>
       for part id <0x3EE311..>
```

The part providers each receive the request and respond with a stream of bytes according to the DC-network protocol:

```
Client <0x061E91..> first DCPart, data: 4989ED38B1C1D1815E7259C09F6CA
9422D81C5F23463B38CD43B756BB5323B996F1FA8BA826C4E8CF04DE2867067780214
3D1224B00CCFF1F0152C66B9753031D60CA9D1C18DE7704686FF532294DB14766C04A
F4EB10870AB505993D2C9F656D58A73FDD76DE8D9DBD52C8066899DD0240F
for pid: <0x3EE311..>, from node: <0x86047A..>

Client <0x061E91..> adding a DCPart, data: 958B4E2343D12C9C63351CFB79
F2CA6766063F3B3A62D387B92988AC31F7EB5FA5A0F8DC6A46892AFCDF1A4DEF15CC7
6068EC98263A0731E55B5CFA8402BA1E3A07E07A7478F12AAB4CDBDA0657D52362DEE
6ACC67576E2AFC5FBB79698C93B87ED0D2601FFD77F8204D6E75BCED43BB0884
for pid: <0x3EE311..>, from node: <0x81E76B..>

Client <0x061E91..> adding a DCPart, data: 68DDFC86A58E0EF38C39018A58
```

```
68C483AAD5919F32F82EE9493914640B01FE862253D4F9A019FD606D65B922505565B
16E72099FF2C64632FCC76F9D6CDC085DE7B7643E71BA881E9D15478A59A9D195A983
36D4BF835F49275649A37E9E4868D331D51D4B3468C77F31BFA5A06D82F79EF6
for pid: <0x3EE311..>, from node: <0x861A00..>

Client <0x061E91..> adding a DCPart, data: C34F0235E82FB9509171AF7D01
7E36F2E13F0D11F27A556F44080BB8DCDA92707433769873 83A1895F4F28F71F95945
ECECE97FEB9291F4A10032C5CC91D9E45418BD09CF2EFD45BBE456DA8F4FE1A118C20
FF61C12CAA74F85CD618F0A49237323D07242A12C6CD67370E71225BD5507BDC
for pid: <0x3EE311..>, from node: <0x86EA09..>

Client <0x061E91..> adding a DCPart, data: E7BDCA6641C649DC1D0D5AAA92
5A260AB17C85797D8F81E6915FDDF101590138E3677B144BA7205457300E2A42263FB
C13DA774CF17B1867D9A2999546DAFB615BAB3A01EAFCD8BD3D8F43B67D00F137EA25
D38176B711E06E0177B34C31D3630A7028B9A74BCD3B81CED11230C86E35DABB
for pid: <0x3EE311..>, from node: <0x8225EC..>
```

The client then XORs these five streams together to get the original part number 8:

```
XOR'ed data: 902D97CEFE7703623D02B1662DD2B75EB111E33EB30C9A6BF17C3FEA
5247BD087FB889137017BB1B6988673492947A27A1D5328B6938FDF090C6399A1A45F
CAB8BE520D5EFAB7122EC942B6797BEB3919404745721FE8287E6040AE2794E6CD240
265B1D0EFDFC1062502233689AE7F9131A for part id <0x3EE311..>
```

This XOR'ed data matches the data for part number 8, and the client has thus retrieved that part successfully. Three more parts are required to reconstruct the file since $m = 4$, so the process is repeated (in parallel) three more times:

```
XOR'ed data: E793B101FDC5B442550D5112DF8E2178148804C76EF2B12637494B79
D773CE5036D9E65E6E0670ABB4FD0CB9B74C0063930260EE0538E2F3F35D5D6EA3099
0979189B5E418539F2625DAF877C93E690A6D8EC6630A1993577FC5F4CDA02570C7C0
71C1423E18917775042D2794EAB87797AE for part id <0xC70488..>

XOR'ed data: FF48308896DE1F3A87890E1F8D3FC8C12FAD63C4F0CD13F3804D3182
8E13A643650779DB4DB5932DA392FB419AF5E0B3C9A06612AEDBEC5A32319F8BAA515
F5423FD87DA515C72F069710A14C35311942F58BF3FA8499E3885747128DE55F30C95
DFA16F2218BBCA9DB8D2A606EDE9BE8D93 for part id <0xC463AD..>

XOR'ed data: 5FFA40D72512837E04E5637D20A09D56DB814CC436B0EB7D5D349F2B
CBD4FED9AE70BFBC222A0C3A8E5BA18E8F36FC0D761BC0EB378D0C294CB96CE41006A
FF9EF53831C8247D06859F82E7238EFB8476A7B9A5EE750A49A04B205AC7ACBAB789D
F132B1C258E3D920DCB31C80D96E5275B4 for part id <0xC44C81..>
```

These results concur with the part contents for parts 8, 9, 10, and 11 published the publisher, so the client has successfully retrieved four erasure coded parts of the file and may now reconstruct the encrypted file from these four parts and then decrypt it:

```
4 out of 16 parts received, decoding... success!

Decoded data: D3D599E2FD5C18FE754468220830F7372B5CC6A683E3121384E61F1
96F8D679A589967AA864F27E1EFB6E0ED7F18BCB3DF5D01DDAD8AEF083E7EC2627845
8E441EE72DA0DC6C841D63EEB2221493FA4CAE44D6E6D5133862E56992C7488C9BC76
9CDBC11AC8BBEE038F1D6BAA22EBCBC4B6CF400160C224CF447F6F405373FB2BE9E19
EF2E0F9244A3CE81CC7CA83E1FB173CE59C96E810B38C8F0FCAE8D6E121C46112C865
F2A796A25844659A4778628EF055C8741AF83017228CBCCEE6CB2E96EC06B0A976751
C45DEDE95E5581321E78A4BEC2D55FD69B480545872473F23CF9531B852E7F9D48915
2AD77E28340C05E0F1D5CB4790776C3EC29F29B46ADE6C69B6A0BD0B33F7398D3ED4C
E8D739B2675F4E4EDE2FBE9D7A502915B870FA8C6CABEA309157620277098F807007E
5A13D7DF13A2141694FE68F258F62E3C2E912A8DA76FDC0B5DE497C7BE09AC7F9979E
D2BA7414107FEC85C747D27D44A9EFB79EA0844732A1DD2D3A3F3E9C981829D606ED6
0252D39DE497C7BE09AC7F985355035EE4D239A1C63746BB5070C8777459F24B22D03
6A9AD8F08C62A031672BB5ADEA9E11F328B8C7DFA992104434EE17754E6A55B4DD2E1
07E350BC472D54B73EB614118BF10 (456 bytes)
```

This output matches the contents of the encrypted file published previously. Now the client may decrypt this file with the encryption key retrieved from the meta root node to finish the retrieval process:

```
Decoding completed successfully. Decrypting... success!
Encryption key: 59C05AF4EA8D4FD1

Decrypted data:
496E20746865520626567696E6E696E6720476F64206372656174656564207468652
068656176656E20616E642074686520656174682E20416E6420746865206561
72746820776173207769746F757420666F726D2C20616E6420766F69643B206
16E64206461726B6E6573732077617320757061F6E20746865206661636520F66
207468652064656570E20416E642074686520537069726974206F6620476F642
06D6F7665642075706F6E207468652066616365206F662074686520776174657
732E20416E6420476F6420736169642C204C65742074686572652065206C696
768743A20616E642074686572652077617320C696768742E20416E6420476F64
207361772074686520C696768742C207468617420697420776173206F6F643
A20616E6420476F64206469766964656420746865206C696768742066726F6D20
746865206461726B6E6573732E20416E6420476F642063616C6C6564207468652
06C6967687420461792C20616E642074686520646172B6E6573732068652063
616C6C6564204E696768742E20416E642074686520576656E696E6720616E642
0746865206D6F726E696E67207765726520746852066697273742064617 92E
(length = 454 bytes)
```

This output matches the contents of the original published part (encrypted and plaintext), and thus the client has successfully retrieved the file with file id <0xae5ca9...>. One final verification lies in the following output, the original string from Genesis which was published:

```
String representation of the decrypted data:
In the beginning God created the heaven and the earth. And the
earth was without form, and void; and darkness was upon the face
of the deep. And the Spirit of God moved upon the face of the
waters. And God said, Let there be light: and there was light.
And God saw the light, that it was good: and God divided the
light from the darkness. And God called the light Day, and the
darkness he called Night. And the evening and the morning were
the first day.
```

4.5 Anonymity Test

As described in section 3.2, Accordion uses a Crowds-like technique for achieving sender anonymity during publication and DC-networks for sender anonymity during retrieval. During publication the publisher and the part storer seek sender anonymity and during retrieval the part storer seeks sender anonymity..

As seen in section 4.4.1, the publisher sends the meta data through a number of forwarding nodes to the meta root node. A forwarding node that receives the message cannot know whether the node that from which it received the message was the sender or a forwarder and thus the publisher achieves sender anonymity for the message containing the meta data wrt. an adversary that is not able to perform the wiretapper attack.

Section 4.4.1 also shows that the same argument holds when the publisher sends the message containing the part meta data and the actual data through a number of forwarding nodes to the part storer (figure 3.3) and when the part storer sends the part meta data (part data minus actual data) to the part publisher (also illustrated in figure 3.3).

Adjusting the probabilities for submitting or saving affects the average number of nodes that a message will be forwarded through. This is illustrated in figure 4.2 where 10 publications have been performed for each of the probabilities $10\%, 20\%, \ldots 100\%$. For probability 0%, a run of the program results in a Java stack overflow error after the message has been forwarded a number of times. For probability 100%, the message is never forwarded (as expected). As expected, the average number of nodes through which a message is forwarded decreases almost linearly with the increase in the probability of saving that message.

During a retrieval operation each of the part provider nodes sends data to the client. According to the DC-network protocol, all part provider nodes except the part storer send random junk data using the previously set up seeds. The part storer also sends data that seems to be random junk data to a passive adversary, but which in fact has the real data xor'ed into it.

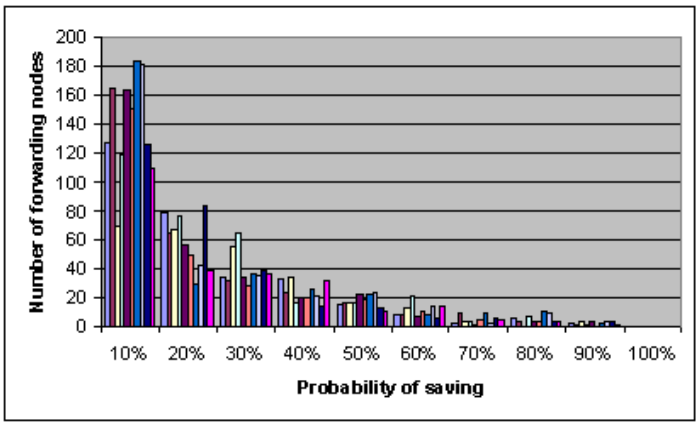

Figure 4.2: Number of forwarding nodes relative to the probability of saving.

The part storer in this example thus achieves sender anonymity among the four other nodes in the part provider list wrt. any passive adversary.

If the same node publishes the same file twice, both the meta data and part meta data messages are likely to be sent through different nodes for each published file (depending on network size) which verifies that each forwarding node chooses the next node at random. Note that both the meta and part root nodes remain the same for both publications because the network is static. Root nodes, however, do not seek anonymity and their identity can (must) thus be able to be found determenistically (via Pastry).

4.6 Functionality Test

This test aims to convince the reader that a file can be succesfully retrieved in Accordion, adjusting two different parameters: the network size N and the size of the DC-network (part provider list) k.

$N = 0$ does not make sense because nothing can be stored in such a network. $N = 1$ and $N = 2$ do not make sense because no anonymity can be achieved in such networks. $N = 3$ is thus considered the minimal network. As described in section 3.2, the choice of value for k involves a tradeoff between the degree of anonymity that the part storer can achieve and the efficiency of the network due to the k-fold increase in transferred data. However, $k \leq N$ must hold.

In theory, k may be as large as $N - 1$, but in the current implementation, $2 \leqq k \approxeq 10$ because the k nodes the part storer chooses for the part providers are selected from nodes in the part storer's local routing table. In a simulated network with $N = 1000$ there are typically no more than 10 nodes in a node's routing table, and therefore this value is a practical limitation to the current implementation.

One way to overcome this limitation would be for the part storer to route "ping" messages

to randomly generated ids and receive replies from the actual nodes responsible for the generated ids. However, this solution would break the part storer's anonymity because the nodes receiving "ping" messages would know that the source of such a message would be a part storer. To solve this problem, the ping messages could be routed through a Crowds-like path such it would take a wiretapper attack to break the part storer's anonymity.

However, with respect to both efficiency and degree of anonymity, a value of k of around 10 seems acceptable.

4.6.1 Network Configurations

Two interesting network configurations have been identified: a minimal network and DC-network and a "typical" network and DC-network. In the first case, $N = 3$ and $k = 2$, and in the second case we choose $N = 1000$ and $k = 5$.

The minimal configuration is interesting in that it tests whether a node is able to handle multiple roles (and multiple roles for multiple parts), e.g., part storer and part root node (this is bound to happen when e.g. 16 parts are published in a network with only 3 nodes). The typical configuration is interesting in that it tests how often, or rather rarely, plays more than one role.

Minimal Configuration

To test the capability of nodes handling multiple roles for multiple parts, the same file as in section 4.4 is published in a minimal-configuration network. Part of the output observed is:

```
Meta root node <0xD69455..> saved meta data for file id CA843F9B9BC88
09CE76DAB3570F339402F330578. Meta data includes the following part ids:
<0x32F9EF..>, <0xCA3E06..>, <0x00F398..>, <0x33CA9C..>, <0x54EC45..>,
<0x48299B..>, <0xE1CE73..>, <0x739773..>, <0x68B593..>, <0x79EF22..>,
<0xD624E6..>, <0x137615..>, <0x924C82..>, <0x68DC07..>, <0xB30F2A..>,
<0xACD916..>, and the encryption key is: d1f623133d3747f3

(...)

Part storer <0xD69455..> saved data for part id <0x32F9EF..> and
selected the following nodes to be included in the part provider list:
<0xD69455..>, <0x40E845..>

(...)

Random node <0xD69455..> is forwarding a PartPublication message

(...)
```

```
Node <0xD69455..> is part root node for part id <0xCA3E06..>
```

The output above shows that node <0xD69455..> is both meta root node, part storer, random forwarder, and part root node for the same part, and thus that a single node is capable of correctly handling several roles for the same part.

The output below shows that node <0xD69455..> is capable of simultaneously handling multiple roles for multiple parts.

```
Part storer <0xD69455..> saved data for part id <0x00F398..> and
selected the following nodes to be included in the part provider list:
<0xD69455..>, <0x40E845..>

Node <0xD69455..> is part root node for part id <0x00F398..>

Incidentally, the PartPublisher is the same node as the PartStorer,
namely <0xD69455..>
```

Typical Configuration

To test how often a node plays multiple roles, the same file as in section 4.4 is published in a typical-configuration network. Part of the output observed is:

```
Meta root node <0xCA71F0..> saved meta data for file id CA843F9B9BC88-
09CE76DAB3570F339402F330578. Meta data includes the following part ids:
<0xBF18C1..>, <0xD22DE6..>, <0x05B039..>, <0x79D105..>, <0x258CE6..>,
<0xA6D4CD..>, <0xAD8225..>, <0x48E3D8..>, <0x85853C..>, <0xB132B3..>,
<0xB9764F..>, <0xF05C55..>, <0x756D87..>, <0x697074..>, <0xFF4841..>,
<0x287BBF..>, and the encryption key is: 7593436eeba589f2

(...)

Part storer <0x1A8E4E..> saved data for part id <0xBF18C1..> and
selected the following nodes to be included in the part provider list:
<0x108C97..>, <0x1A8E4E..>, <0x1215B1..>, <0x13983E..>, <0x148B1F..>

(...)

Node <0xBF3796..> is part root node for part id <0xBF18C1..>
```

The output above shows that node <0xCA71F0..> is meta root node, node <0x1A8E4E..> is part storer, and node <0xBF3796..> is part root node. Thus, in this single publication, no single node has several roles in the publication of a single part.

The graph in figure 4.3 shows that the average number of roles for each node in a network of 1000 thousand nodes increases with the number of parts published, as expected. For each number of parts published, 10 publications were performed and the number of roles played by each node (excluding the role of random forwarder) was plotted. All publications and retrievals were successful.

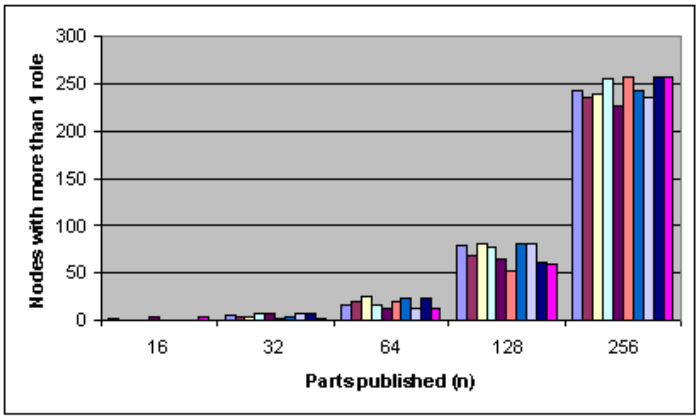

Figure 4.3: The number of roles played by each node increases with the number of published parts when keeping network size fixed.

4.6.2 Dynamic Network

So far all tests have been performed in a static network. The following tests show Accordion's resistance to leaving nodes and implicitly tests if our use of the FEC library [Cha01] works correctly. Tests where nodes join the network have not been performed.

For the typical network configuration with a network with $N = 1000$, $k = 5$, and EC parameters $m = 4$, $n = 16$, a successively larger number of random nodes are killed after a file has been published and before it is retrieved, and it is then tested if it is still possible to retrieve the file using the parts still available from the nodes still alive.

In some runs of the publication and retrieval tests where random nodes are killed before retrieval, one of the nodes killed may include the meta root node. Because no backup of root nodes is performed, this is fatal to the retrieval process, but it says nothing about whether our use of the FEC library is correct, nor does it verify the positive effect of erasure coding on resilience.

Therefore, two different tests have been made. The first test is shown in figure 4.4. It illustrates 10 runs of the publication and retrieval processes for an increasing number of dead nodes where the meta root node was *not* among the nodes killed, i.e., runs where the meta root node was killed were not counted toward the result. The chart shows that for the 10 test runs, even when as many as 30% of the nodes were randomly killed, the file was successfully retrieved every single time, and when 40% of the nodes were killed, the file was still retrieved successfully 6 out of 10 times.

The second test attempts to assess the effect on resilience of backing up meta root node data (or rather, the effect of *not* doing it). The result of this test is shown in figure 4.5 shows 50 runs of publication and retrieval where the runs in which the meta root node was killed counted towards the result. Compared to figure 4.4, the number of successful retrievals drops significantly when 30% of the nodes are killed, and it therefore seems prudent to implement backup of meta root node data.

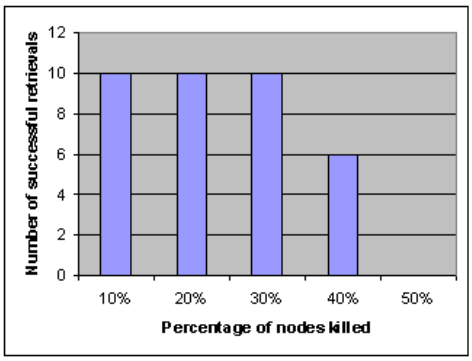

Figure 4.4: Number of successful retrievals when a number of random nodes are killed after publication (ignoring runs where the meta root was one of the nodes killed). 10 publications were performed for each of the number of killed nodes.

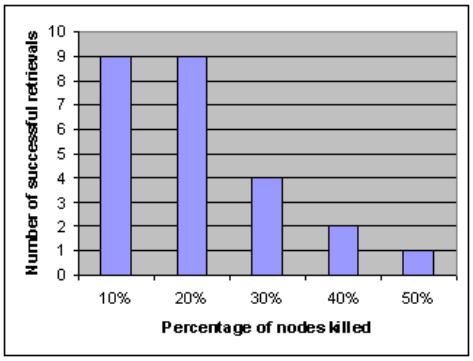

Figure 4.5: The same test as in figure 4.4, except that runs where the meta root was one of the nodes killed count towards the result.

One final output from a successful retrieval in one of the above test runs increases confidence in our use of the FEC library:

```
Decoding part with repairIndex: 0
Decoding part with repairIndex: 2
Decoding part with repairIndex: 6
Decoding part with repairIndex: 10
```

The above means that the retrieved file has been reconstructed from erasure-coded parts #0, #2, #6, and #10, i.e., four random parts out of the 16 parts originally published. Several other combinations of parts have been observed in successful retrievals but are omitted here for brevity.

Chapter 5

Conclusion

In this report we have described a number of different anonymizing techniques and analyzed their corresponding anonymity properties, showing different ways how these properties may be degraded by various attacks. None of the techniques are completely secure in practice, but practical DC-networks seem to be the most secure choice (excluding the overly inefficient broadcast technique), because if the network has been set up securely (ie. without attacks), sender anonymity cannot be broken (assuming the outcome of the pseudo-random number generators cannot be predicted) [1]. A brief survey of existing peer-to-peer anonymizing systems has also been presented to give the reader an overview of the status anonymizing systems in peer-to-peer networks.

We have designed, implemented, and tested a novel design for an anonymous, low-latency, peer-to-peer file storage application, Accordion. Publication in Accordion is based on a Crowds-like technique and retrieval on DC-networks which have only been used in practice a very limited number of times, e.g. in [Mar99].

The simulated tests that have been performed increase confidence in the both anonymity and functional aspects of the design. The current implementation does not support backup of root nodes but still maintains a high degree of availability of files in the face of node failures. Also, minimal, successful testing has been performed in a small, real network.

Accordion provides a unique combination of the following features:

- Efficiently implemented DC-networks using pseudo-random number generators.
- Erasure coding of files ensuring high availability even in spite of attacks.
- Strong deniability through a combination of erasure coding, symmetric encryption and secret sharing. This makes it impossible to reconstruct the encryption key with less than m parts of it but it is of course still possible to perform attacks on the encrypted file itself. Assuming that encryption cannot be broken, no user (or adversary) is able

[1] This is not the same as unconditional anonymity, however, because an adversary *could* have intercepted the exchanged bits or seeds and thus be able to break sender anonymity

to tell what parts of other users' files they are storing, thus providing them with strong deniability of such knowledge.

Compared to related work such as Freenet and GNUnet, Accordion provides strong anonymity properties through DC-networks, lookup guarantees through the use of structured routing and low latency retrievals because no extra routing hops and no delays are introduced.

A number of attacks on Accordion are possible but these are hard to perform due to the nature of peer-to-peer networks and will possibly only break anonymity for part of a file. If the anonymity of nodes that store parts is broken these nodes will still have strong deniability as defense.

This project has described different techniques to achieve anonymity in computer networks and presented the design and implementation of an anonymous, low-latency, peer-to-peer file storage system. However, in the area of anonymity both theoretical and practical challenges exist:

- Formal models for measuring the anonymity properties of an anonymizing system. [Ser04] takes the first step in this direction but only analyses mix-networks.
- Comparing the anonymity of specific systems. GAP [BG03] presents an informal comparison of some systems.
- More robust implementations of the many interesting ideas that has come forward in the area of anonymizing systems recently.
- More tests of implementations that focus more on e.g. performance and design of intuitive user interfaces to get a larger user base, potentially providing both stronger anonymity and e.g. more content in systems such as file storage systems.

We believe that the area of anonymity in computer networks is an area in continuous growth that will see practical implementations in use by the public within a few years.

Bibliography

[And96] Ross Anderson. The eternity service, 1996.

[ano] Anonymizer.

[BG03] Krista Bennett and Christian Grothoff. gap – Practical Anonymous Networking. In *Designing Privacy Enhancing Technologies*. Springer-Verlag, 2003.

[Boe03] Christian Boesgaard. Yaaps: Yet another anonymous publication system (report number 03-03-28). Technical report, 2003.

[CDK+03] M. Castro, P. Druschel, A. Kermarrec, A. Nandi, A. Rowstron, and A. Singh. Splitstream: High-bandwidth multicast in cooperative environments, 2003.

[CDKR02] M. Castro, P. Druschel, A. Kermarrec, and A. Rowstron. SCRIBE: A large-scale and decentralized application-level multicast infrastructure. *IEEE Journal on Selected Areas in communications (JSAC)*, 2002. To appear.

[Cha81] David L. Chaum. Untraceable electronic mail, return addresses, and digital pseudonyms. *Commun. ACM*, 24(2):84–90, 1981.

[Cha88] David L. Chaum. The dining cryptographers problem: Unconditional sender and recipient untraceability. *Journal of Cryptography*, 1:65–75, 1988.

[Cha01] Justin Chapweske. Java fec library 1.0.3, 2001.

[CMH+02] Ian Clarke, Scott G. Miller, Theodore W. Hong, Oskar Sandberg, and Brandon Wiley. Protecting free expression online with freenet. *IEEE Internet Computing*, 6(1):40–49, 2002.

[DDM03] George Danezis, Roger Dingledine, and Nick Mathewson. Mixminion: Design of a type iii anonymous remailer protocol. In *SP '03: Proceedings of the 2003 IEEE Symposium on Security and Privacy*, page 2, Washington, DC, USA, 2003. IEEE Computer Society.

[Din] Roger Dingledine. The free haven project: Design and deployment of an anonymous secure data haven. *MIT Master's Thesis*, `http://www.freehaven.net/doc/freehaven.pdf`.

[DMS04] R. Dingledine, N. Mathewson, and P. Syverson. Tor: The second-generation onion router, 2004.

[GRPS] Sharad Goel, Mark Robson, Milo Polte, and Emin Gün Sirer. Herbivore: A scalable and efficient protocol for anonymous communication. *Cornell University Computing and Information Science Technical Report, TR2003-1890*, `http://www.cam.cornell.edu/~sharad/herbivore/herbivore.pdf`.

[GRS99] David Goldschlag, Michael Reed, and Paul Syverson. Onion routing. *Commun. ACM*, 42(2):39–41, 1999.

[KBC+00] John Kubiatowicz, David Bindel, Yan Chen, Steven Czerwinski, Patrick Eaton, Dennis Geels, Ramakrishna Gummadi, Sean Rhea, Hakim Weatherspoon, Chris Wells, and Ben Zhao. Oceanstore: an architecture for global-scale persistent storage. In *ASPLOS-IX: Proceedings of the ninth international conference on Architectural support for programming languages and operating systems*, pages 190–201, New York, NY, USA, 2000. ACM Press.

[KP01] Marit Köhntopp and Andreas Pfitzmann. Anonymity, unobservability, and pseudonymity — a proposal for terminology. 2001.

[Kra94] Hugo Krawczyk. Secret sharing made short. In *CRYPTO '93: Proceedings of the 13th Annual International Cryptology Conference on Advances in Cryptology*, pages 136–146, London, UK, 1994. Springer-Verlag.

[LS02] Brian Neil Levine and Clay Shields. Hordes: a multicast based protocol for anonymity. *J. Comput. Secur.*, 10(3):213–240, 2002.

[Mar99] David Martin. Local anonymity in the internet. Technical report, 1999.

[MM02] P. Maymounkov and D. Mazieres. Kademlia: A peer-to-peer information system based on the xor metric, 2002.

[MWC00] Aviel D. Rubin Marc Waldman and Lorrie Faith Cranor. Publius: A robust, tamper-evident, censorship-resistant, web publishing system. In *Proc. 9th USENIX Security Symposium*, pages 59–72, August 2000.

[PDZ04] Romer Gil Jeff Hoye Y. Charlie Hu Sitaram Iyer Andrew Ladd Alan Mislove Animesh Nandi Ansley Post Charlie Reis Atul Singh Peter Druschel, Eric Engineer and Rong Mei Zhang. Freepastry 1.3.2, 2004.

[RD01a] Antony Rowstron and Peter Druschel. Pastry: Scalable, distributed object location and routing for large-scale peer-to-peer systems. *IFIP/ACM International Conference on Distributed Systems Platforms (Middleware), Heidelberg, Germany*, `http://research.microsoft.com/~antr/PAST/pastry.pdf`, 2001.

[RD01b] Antony Rowstron and Peter Druschel. Storage management and caching in past, a large-scale, persistent peer-to-peer storage utility. In *SOSP '01: Proceedings of the eighteenth ACM symposium on Operating systems principles*, pages 188–201, New York, NY, USA, 2001. ACM Press.

[RP02] Marc Rennhard and Bernhard Plattner. Introducing morphmix: peer-to-peer based anonymous internet usage with collusion detection. In *WPES '02: Proceedings of the 2002 ACM workshop on Privacy in the Electronic Society*, pages 91–102, New York, NY, USA, 2002. ACM Press.

[RP03] Marc Rennhard and Bernhard Plattner. Practical anonymity for the masses with mix-networks. In *WETICE '03: Proceedings of the Twelfth International Workshop on Enabling Technologies*, page 255, Washington, DC, USA, 2003. IEEE Computer Society.

[RR97] Michael K. Reiter and Aviel D. Rubin. Crowds: Anonymity for web transactions. Technical report, 1997.

[Ser04] Andrei Serjantov. On the anonymity of anonymity systems. *University of Cambridge Technical Report No. 604*, `http://www.cl.cam.ac.uk/TechReports/UCAM-CL-TR-604.html`, 2004.

[SMLN+03] Ion Stoica, Robert Morris, David Liben-Nowell, David R. Karger, M. Frans Kaashoek, Frank Dabek, and Hari Balakrishnan. Chord: a scalable peer-to-peer lookup protocol for internet applications. *IEEE/ACM Trans. Netw.*, 11(1):17–32, 2003.

[SW] Adam Stubblefield and Dan S. Wallach. Dagster: Censorship-resistant publishing without replication. Technical report, Rice University, U.S.A.

[WALS01] M. Wright, M. Adler, B. Levine, and C. Shields. An analysis of the degradation of anonymous protocols. 2001.

[WK02] Hakim Weatherspoon and John Kubiatowicz. Erasure coding vs. replication: A quantitative comparison. In *IPTPS '01: Revised Papers from the First International Workshop on Peer-to-Peer Systems*, pages 328–338, London, UK, 2002. Springer-Verlag.

[WM01] Marc Waldman and David Mazires. Tangler: a censorship-resistant publishing system based on document entanglements. In *CCS '01: Proceedings of the 8th ACM conference on Computer and Communications Security*, pages 126–135, New York, NY, USA, 2001. ACM Press.

[WP90] Michael Waidner and Birgit Pfitzmann. The dining cryptographers in the disco: unconditional sender and recipient untraceability with computationally secure serviceability. In *EUROCRYPT '89: Proceedings of the workshop on the theory and application of cryptographic techniques on Advances in cryptology*, page 690, New York, NY, USA, 1990. Springer-Verlag New York, Inc.

[ZKJ01] Ben Y. Zhao, John D. Kubiatowicz, and Anthony D. Joseph. Tapestry: An infrastructure for fault-tolerant wide-area location. Technical report, Berkeley, CA, USA, 2001.

Appendix A

Java Source Code for Accordion

A.1 AccNode.java

```
package accordion;

import accordion.messagetypes.AccordionMessage;

public interface  AccNode
{
  public void handle(AccordionMessage am);
}
```

A.2 AccordionApplication.java

```
package accordion;

import accordion.messagetypes.*;
import accordion.tabletypes.*;

import rice.pastry.Id;
import rice.pastry.NodeHandle;
import rice.pastry.NodeId;
import rice.pastry.PastryNode;

import rice.pastry.client.PastryAppl;
import rice.pastry.direct.NetworkSimulator;
import rice.pastry.messaging.Address;
import rice.pastry.messaging.Message;
import rice.pastry.leafset.LeafSet;
import rice.pastry.routing.RouteSet;
import rice.pastry.routing.RoutingTable;
import rice.pastry.routing.SendOptions;
import rice.pastry.security.Credentials;
import rice.pastry.security.PermissiveCredentials;

import javax.crypto.*;
import javax.crypto.interfaces.*;
```

```
import javax.crypto.spec.*;
import java.math.*;
import java.security.*;
import java.security.spec.*;

import java.util.ArrayList;
import java.util.Hashtable;
import java.util.HashSet;

import java.util.Iterator;
import java.util.Random;

import com.onionnetworks.fec.FECCode;
import com.onionnetworks.fec.FECCodeFactory;
import com.onionnetworks.util.Buffer;


class AccordionApplication extends PastryAppl
{
  //--------------------Pastry stuff--------------------

  public static Address appAddress = new AccordionAddress();
  private static Credentials cred = new PermissiveCredentials();

  private static class AccordionAddress implements Address
  {
    private int myCode = 0x1984abcd;
    public int hashCode() { return myCode; }
    public boolean equals(Object obj) { return (obj instanceof AccordionAddress); }
    public String toString() { return "[AccordionAddress]"; }
  }

  private NetworkSimulator simulator;
  private PastryNode pn;

  //----------------------------------------------------------

  private Random prng;

  /* Different Accordion node "roles". An Accordion node
   * (application) can potentially play all roles.
   */
  private RandNode rand;              //Only one per physical node (make static in non-simulation)
  private MetaRN metarn;              //Only one per physical node (make static in non-simulation)
  private PartRN partrn;              //Only one per physical node (make static in non-simulation)
  private PartProvider partprovider;  //Only one per physical node (make static in non-simulation)
  private Client client;              //Only one per physical node (make static in non-simulation)

  public static KeyPair keypair;      //Only one asymmetric keypair per node
                                      //Made static for performance reasons under simulation.


  /* k is the size of the ppl */
  public static int k = 5;

  public static boolean isSimulation = false;
```

```
public AccordionApplication(PastryNode pn) //Real network
{
    super(pn);
     initStuff (pn);
    //pn.registerApp(this);
}

public AccordionApplication(PastryNode pn, NetworkSimulator sim) //Simulated
{
    super(pn);
    simulator = sim;
    isSimulation = true;
     initStuff (pn);
}

private void initStuff (PastryNode pn)
{
  this.pn = pn;
  prng = new Random();
  if (keypair==null) keypair = generateAKP();

  metarn = new MetaRN(this);
  partrn = new PartRN(this);
  partprovider = new PartProvider(this, pn);
  rand = new RandNode(this, partprovider);
  client = new Client(this);

  //LeafSet ls = pn.getLeafSet();
  //System.out.println("Node " + getNodeId() + ": " + ls);

}

public void messageForAppl(Message message)
{
  //System.out.print("Node " + getNodeId().toStringFull() + " received a message ");
  AccordionMessage am = (AccordionMessage)message;

  //-----Publication-----

  if (am instanceof MetaPublication)
  {
    MetaPublication mp = (MetaPublication)am;
    NodeId nid = new NodeId(mp.id.toByteArray());

    if (isClosest(nid))
    {
      metarn.handle(mp);
      //System.out.println("Node " + getNodeId() + " playing MetaRN [ " + mp.toString() + " ]");
    }
    else
    {
      rand.handle(mp);
      //System.out.println("Node " + getNodeId() + " playing RandNode [ " + mp.toString() + " ]");
    }
  }

  else if (am instanceof PartPublication)
```

```
{
  PartPublication partpub = (PartPublication)am;
  rand.handle(partpub);
  //System.out.println("Node " + getNodeId() + " playing RandNode [ " + partpub.toString() + " ]");
}

else if (am instanceof StoredPart)
{
  StoredPart sp = (StoredPart)am;
  rand.handle(sp);
  //System.out.println("Node " + getNodeId() + " playing RandNode [ " + sp.toString() + " ]");
}

else if (am instanceof PublishedPart)
{
  PublishedPart pubpart = (PublishedPart)am;
  partprovider.handle(pubpart);
}

else if (am instanceof SubmittedPart)
{
  SubmittedPart subpart = (SubmittedPart)am;
  partrn.handle(subpart);
}

//------Retrieval------

// ----Client <-> MetaRN------

else if (am instanceof MetaRNMessage)
{
  MetaRNMessage mrm = (MetaRNMessage)am;

  if (! mrm.isReply)
  {
    metarn.handle(mrm);
    //System.out.println("Node " + getNodeId() + " playing MetaRN [ " + mrm.toString() + " ]");
  }
  else
  {
    client .handle(mrm);
    //System.out.println("Node " + getNodeId() + " playing Client [ " + mrm.toString() + " ]");
  }
}

//------Client <-> PartRN------

else if (am instanceof PartRNMessage)
{
  PartRNMessage prm = (PartRNMessage)am;

  if (! prm.isReply)
  {
    partrn.handle(prm);
    //System.out.println("Node " + getNodeId() + " playing PartRN [ " + prm.toString() + " ]");
  }
  else
```

```
    {
      client .handle(prm);
      //System.out.println("Node " + getNodeId() + " playing Client [ " + prm.toString() + " ]") ;
    }
  }

  //−−−−−PartProvider −> Client (send the part status)−−−−−

  else if (am instanceof PartMessage)
  {
    PartMessage pm = (PartMessage)am;

    if (! pm.isReply)
    {
      //System.out.println("Node " + getNodeId() + " is playing PartProvider and is receiving a request
          from the client .") ;
      partprovider.handle(pm);
    }
    else
      client .handle(pm);

    //System.out.println("Node " + getNodeId() + " playing Client [ " + pm.toString() + " ]") ;
  }
}

//−−−−−Implementation of abstract methods from PastryAppl−−−−−

public Address getAddress() { return appAddress; }

public Credentials getCredentials() { return cred ; }

public boolean enrouteMessage(Message msg, Id key, NodeId nextHop, SendOptions opt)
{
  //System.out.println("Enroute " + msg + " at " + getNodeId().toStringFull());
  //System.out.println("Message enroute at " + getNodeId().toStringFull());
  return true;
}

// Use these to implement backup of root node−functionality
public void leafSetChange(NodeHandle nh, boolean wasAdded)
{
  //System.out.print("In " + getNodeId() + "'s leaf set , " + " node " + nh.getNodeId() + " was ");
  //if (wasAdded) System.out.println("added"); else System.out.println("removed");
}

public void routeSetChange(NodeHandle nh, boolean wasAdded)
{
  //System.out.print("In " + getNodeId() + "'s route set , " + " node " + nh.getNodeId() + " was ");
  //if (wasAdded) System.out.println("added"); else System.out.println("removed");
}

public void notifyReady()
{
  //System.out.println("Node " + getNodeId() + " ready, waking up any clients");
  //sendRndMsg(new Random());
}
```

```
public void publish(String filename)
{
  Publisher publisher = new Publisher(this, filename, pn);
  publisher.start();
}

public void retrieve(String filename)
{
  Id fileId = new Id(hash(filename.getBytes()));
  client.go(fileId);
}

public void forward(AccordionMessage am)
{
  byte[] material = new byte[20];
  prng.nextBytes(material);
  Id rndid = new Id(material);

  forward(rndid, am);
}

public void forward(Id key, AccordionMessage am)
{
  routeMsg(key, am, cred, null);
  if (isSimulation) while(simulator.simulate());
}

public void submit(MetaPublication mp)
{
  Id metarn = mp.id;
  submit(metarn, mp);
  //System.out.println("Node " + getNodeId() + " is submitting MP to MetaRN");
}

public void submit(Id rn, AccordionMessage am)          //Used by PS when submitting to PartRNs
{
  forward(rn, am);
}

//-----Utilies methods-----

public byte[] hash(byte[] bytes)
{
  MessageDigest md = null;

  try
  {
    md = MessageDigest.getInstance("SHA-1");
    md.update(bytes);
  }
  catch(Exception e) { System.out.println(e); }

  return md.digest();
}

/*
 * Find the (asymetric) set difference between two sets.
```

```
 */
public ArrayList difference(ArrayList ppl, ArrayList sl)
{
  ArrayList list = new ArrayList();
  HashSet pplset = new HashSet(ppl);
  HashSet slset = new HashSet(sl);

  pplset.removeAll(slset); //{ppl} \ {sl}
  for (Iterator i = pplset.iterator(); i.hasNext();) list.add((Id)i.next()); //Convert HashSet to
      ArrayList
  return list;
}

public AccRow getRow(ArrayList table, Id key)
{
  AccRow row = null;

  for (Iterator i = table.iterator(); i.hasNext();)
  {
    row = (AccRow)i.next();
    if (row.id!=null && row.id.equals(key)) return row;
  }

  return null;
}

public void removeRow(ArrayList table, Id key)
{
  try
  {
    AccRow row = null;

    for (int i=0; i<table.size()-1; i++)
    {
      row = (AccRow)table.get(i);
      if (row.id!=null && row.id.equals(key))
      {
        table.remove(i);
        break;
      }
    }
  }
  catch (IndexOutOfBoundsException ie) { System.out.println(ie.toString()); }
}

public NodeId getRandomPPLNode(ArrayList ppl)
{
  int rand = prng.nextInt(ppl.size());
  //System.out.println("Rand: " + rand);
  return (NodeId)ppl.get(rand);
}

//-----Crypto stuff-----

private KeyPair generateAKP()
{
  KeyPairGenerator kpg = null;
```

```
    DHParameterSpec dhs = new DHParameterSpec(skip1024Modulus, skip1024Base);

    try { kpg = KeyPairGenerator.getInstance("DH"); }
    catch(NoSuchAlgorithmException ne) { System.out.println(ne.toString()); }

    try { kpg. initialize (dhs); }
    catch(InvalidAlgorithmParameterException ie) { System.out.println(ie.toString()) ; }

    return kpg.generateKeyPair();
  }

  public byte[] calculateKey(byte[] puk)
  {
    KeyFactory kf = null;
    PublicKey pk = null;
    KeyAgreement ka = null;
    byte[] sharedsecret = null;

    try
    {
      kf = KeyFactory.getInstance("DH");
      ka = KeyAgreement.getInstance("DH");
      ka. init (keypair.getPrivate());
      X509EncodedKeySpec xks = new X509EncodedKeySpec(puk);
      pk = kf.generatePublic(xks);
      DHParameterSpec dhs = ((DHPublicKey)pk).getParams();

      ka.doPhase(pk, true);
      sharedsecret = ka.generateSecret();
    }
    catch(Exception e) { System.out.println(e.toString()) ; }

    return sharedsecret;
  }

  // The 1024 bit Diffie-Hellman modulus values used by SKIP
  private static final byte skip1024ModulusBytes[] = {
    (byte)0xF4, (byte)0x88, (byte)0xFD, (byte)0x58,
    (byte)0x4E, (byte)0x49, (byte)0xDB, (byte)0xCD,
    (byte)0x20, (byte)0xB4, (byte)0x9D, (byte)0xE4,
    (byte)0x91, (byte)0x07, (byte)0x36, (byte)0x6B,
    (byte)0x33, (byte)0x6C, (byte)0x38, (byte)0x0D,
    (byte)0x45, (byte)0x1D, (byte)0x0F, (byte)0x7C,
    (byte)0x88, (byte)0xB3, (byte)0x1C, (byte)0x7C,
    (byte)0x5B, (byte)0x2D, (byte)0x8E, (byte)0xF6,
    (byte)0xF3, (byte)0xC9, (byte)0x23, (byte)0xC0,
    (byte)0x43, (byte)0xF0, (byte)0xA5, (byte)0x5B,
    (byte)0x18, (byte)0x8D, (byte)0x8E, (byte)0xBB,
    (byte)0x55, (byte)0x8C, (byte)0xB8, (byte)0x5D,
    (byte)0x38, (byte)0xD3, (byte)0x34, (byte)0xFD,
    (byte)0x7C, (byte)0x17, (byte)0x57, (byte)0x43,
    (byte)0xA3, (byte)0x1D, (byte)0x18, (byte)0x6C,
    (byte)0xDE, (byte)0x33, (byte)0x21, (byte)0x2C,
    (byte)0xB5, (byte)0x2A, (byte)0xFF, (byte)0x3C,
    (byte)0xE1, (byte)0xB1, (byte)0x29, (byte)0x40,
    (byte)0x18, (byte)0x11, (byte)0x8D, (byte)0x7C,
```

```
    (byte)0x84, (byte)0xA7, (byte)0x0A, (byte)0x72,
    (byte)0xD6, (byte)0x86, (byte)0xC4, (byte)0x03,
    (byte)0x19, (byte)0xC8, (byte)0x07, (byte)0x29,
    (byte)0x7A, (byte)0xCA, (byte)0x95, (byte)0x0C,
    (byte)0xD9, (byte)0x96, (byte)0x9F, (byte)0xAB,
    (byte)0xD0, (byte)0x0A, (byte)0x50, (byte)0x9B,
    (byte)0x02, (byte)0x46, (byte)0xD3, (byte)0x08,
    (byte)0x3D, (byte)0x66, (byte)0xA4, (byte)0x5D,
    (byte)0x41, (byte)0x9F, (byte)0x9C, (byte)0x7C,
    (byte)0xBD, (byte)0x89, (byte)0x4B, (byte)0x22,
    (byte)0x19, (byte)0x26, (byte)0xBA, (byte)0xAB,
    (byte)0xA2, (byte)0x5E, (byte)0xC3, (byte)0x55,
    (byte)0xE9, (byte)0x2F, (byte)0x78, (byte)0xC7
  };

  // The SKIP 1024 bit modulus
  private static final BigInteger skip1024Modulus = new BigInteger(1, skip1024ModulusBytes);

  // The base used with the SKIP 1024 bit modulus
  private static final BigInteger skip1024Base = BigInteger.valueOf(2);

}
```

A.3 AccPart.java

```
package accordion;

import java.io.Serializable;
import rice.pastry.Id;
import java.util.ArrayList;
import com.onionnetworks.util.Buffer;

public class AccPart implements Comparable, Serializable
{
  public ArrayList dcparts = new ArrayList();      //List of DCPart objects
  public Buffer data;
  public int repairIndex;
  public boolean xorDone = false;

  public AccPart() {}
  public String toString() { return "AccPart"; }

  public int compareTo(Object o)
  {
    AccPart ap = (AccPart) o;

    if (this.repairIndex < ap.repairIndex)
      return -1;
    else if (this.repairIndex == ap.repairIndex)
      return 0;
    else
      return 1;
  }
}
```

A.4 Client.java

```
package accordion;

import accordion.messagetypes.AccordionMessage;
import accordion.messagetypes.MetaRNMessage;
import accordion.messagetypes.PartRNMessage;
import accordion.messagetypes.PartMessage;

import accordion.tabletypes.MetaRNRow;
import accordion.tabletypes.PartRNRow;

import rice.pastry.Id;
import rice.pastry.NodeId;

import java.util.Collections;
import java.util.ArrayList;
import java.util.Enumeration;
import java.util.Hashtable;
import java.util.Iterator;

import javax.crypto.spec.SecretKeySpec;
import java.security.AlgorithmParameters;
import javax.crypto.Cipher;

import com.onionnetworks.fec.FECCode;
import com.onionnetworks.fec.FECCodeFactory;
import com.onionnetworks.util.Buffer;

class Client implements AccNode
{
  private AccordionApplication app;
  private Hashtable files = new Hashtable();       //List of ECFile objects
  private Integer sessionId = new Integer(0);      //Incremented for each file transfer

  public Client(AccordionApplication aa)
  {
    app = aa;
  }

  public void go(Id fileId)
  {
    MetaRNMessage mrm = new MetaRNMessage(app.appAddress, app.getNodeId(), fileId);
    System.out.println("Client " + app.getNodeId().toString() + " contacting meta root node requesting
        file id " + fileId.toString());
    app.forward(fileId, mrm);

    int tmp = sessionId.intValue();
    tmp++;
    sessionId = new Integer(tmp);
  }

  public synchronized void handle(AccordionMessage am)
  {
    if (am instanceof MetaRNMessage)
    {
      MetaRNMessage mrm = (MetaRNMessage)am;
```

```
    if (mrm.metaRows.size()>0)
    {
        //System.out.println("MetaRN returned the following results: ");
        //for (Iterator i = mrm.metaRows.iterator(); i.hasNext();) System.out.println((MetaRNRow)i.next
            ());
        //System.out.print("Please choose an entry: ");
        MetaRNRow metarow = (MetaRNRow)mrm.metaRows.get(0);

        //System.out.println("M = " + metarow.M);

        ECFile currentfile = new ECFile(metarow);
        files.put(sessionId, currentfile);

        //Contact the root node for each part
        PartRNMessage prm = null;
        Id partId = null;
        //System.out.println("Contacting " + metarow.partIdList.size() + " PartRN(s).");

        for (Iterator j = metarow.partIdList.iterator(); j.hasNext();)
        {
            partId = (Id)j.next();
            prm = new PartRNMessage(app.appAddress, app.getNodeId(), partId, sessionId.intValue());
            app.forward(partId, prm);
            System.out.println("Client␣"+app.getNodeId()+"␣contacting␣part␣root␣node␣for␣part␣id␣" +
                partId.toString());
        }
    }
    else System.out.println("−␣ERROR:␣Reply␣from␣the␣meta␣root␣node␣"+mrm.source+"␣was␣empty
        .");
}

else if (am instanceof PartRNMessage)
{
    PartRNMessage prm = (PartRNMessage)am;

    if (prm.partRows.size()>0)
    {
        //System.out.println("One PartRN returned the following results: ");
        //for (Iterator i = prm.partRows.iterator(); i.hasNext();) System.out.println((PartRNRow)i.next());
        //System.out.print("Please choose an entry: ");
        PartRNRow partrow = (PartRNRow)prm.partRows.get(0);

        NodeId nid;
        PartMessage pm;

        for (Iterator i = partrow.partProviderList.iterator(); i.hasNext();)
        {
            nid = (NodeId)i.next();
            System.out.println("Client␣"+app.getNodeId()+"␣contacting␣part␣provider␣node␣" + nid + "␣for␣
                part␣id␣" + partrow.id);
            pm = new PartMessage(app.appAddress, app.getNodeId(), partrow.id, prm.sessionId);
            app.forward(nid, pm);
        }
    }
    else System.out.println("−␣ERROR:␣Reply␣from␣part␣root␣node"+prm.source+"␣for␣part␣id␣"+prm.
        id+"␣was␣empty.");
```

```
}

else if (am instanceof PartMessage)
{
  PartMessage pm = (PartMessage)am;

  if (pm.returnCode==0) //Only if a part provider knew about the part do we process it
  {
    ECFile currentfile = null;
    AccPart accpart = null;

    currentfile = (ECFile) files.get(new Integer(pm.sessionId));

    System.out.println("pm.id: " + pm.id + ", dcpart.id: " + pm.dcpart.id);

    accpart = (AccPart)currentfile.accparts.get(pm.dcpart.id);

    if (accpart==null)
    {
      System.out.println("Client "+app.getNodeId()+" first DCPart, data: " + HexDump.toHexShort(
          pm.dcpart.data) + " for pid: " + pm.dcpart.id + ", from node: " + pm.source);
      accpart = new AccPart();
      accpart.dcparts.add(pm.dcpart);
      accpart.repairIndex = pm.dcpart.repairIndex;
      currentfile.accparts.put(pm.dcpart.id, accpart);
    }
    else
    {
      if (accpart.dcparts.size()<app.k)
      {
        System.out.println("Client "+app.getNodeId()+" adding DCPart #" + (accpart.dcparts.size()+1)
            + ", data:" + HexDump.toHexShort(pm.dcpart.data) + " for pid:" + pm.dcpart.id + ", from
            node: " + pm.source);
        accpart.dcparts.add(pm.dcpart);
      }
    }

    if ((accpart.dcparts.size()==app.k) && accpart.xorDone==false)
    {
      accpart.data = doXOR(accpart.dcparts);
      accpart.xorDone = true;
      currentfile.current_m++;
      System.out.println("Current m: " + currentfile.current_m);
    }

    //At this point, we know that there are m AccParts ready for decoding, we just need to find out
        which
    if (currentfile.current_m==currentfile.m)
    {
      ArrayList xoredparts = new ArrayList();

      for (Enumeration e = currentfile.accparts.elements(); e.hasMoreElements();)
      {
        accpart = (AccPart)e.nextElement();
        if (accpart.xorDone) xoredparts.add(accpart);
      }
```

```
            if (! currentfile.notDecoded)
            {
              System.out.print("Client␣received␣"+currentfile.m+"␣parts,␣decoding...");
               currentfile.data = decode(xoredparts, currentfile.m, currentfile.n);
              System.out.print("\nSuccess!␣Now␣Decrypting...");
              decrypt(currentfile.data, currentfile.enckey);
               currentfile.notDecoded=true;
              System.out.println("␣success!");
            }
          }

        }
        else System.out.println("−␣ERROR:␣Node␣" + pm.source + "␣didn't␣know␣about␣part␣with␣id␣" +
             pm.dcpart.id);
    }
}

private Buffer doXOR(ArrayList dcparts)
{
    byte[] realdata = null;
    DCPart dcpart = null;

    if (dcparts.size()>0)
    {
      dcpart = (DCPart)dcparts.get(0);  //All parts must have the same length!
      realdata = new byte[dcpart.size];

      Iterator j = null;

      for (int i=0; i<realdata.length; i++)
      {
        for (j = dcparts.iterator(); j.hasNext();)
        {
          dcpart = (DCPart)j.next();
          realdata[i] ^= dcpart.data[i];
        }
      }
    }

    System.out.println("Client␣XOR'ing␣part␣provider␣byte␣streams␣for␣part␣id␣"+dcpart.id.toString()+"\
         nResult:␣" + HexDump.toHex(realdata));

    return new Buffer(realdata);
}

private byte[] decode(ArrayList xoredparts, int m, int n)
{

    Buffer[] receiverBuffer = new Buffer[m];
    int[] receiverIndex = new int[m];
    Enumeration e = null;
    AccPart onepart = (AccPart)xoredparts.get(0);
    int i = 0;

    int packetsize = ((DCPart)onepart.dcparts.get(0)).size; //All parts must have the same length and
         onepart.length = packetsize
    byte[] received = new byte[m∗packetsize];
```

```
    Collections.sort(xoredparts);

    for ( Iterator it = xoredparts.iterator() ; it.hasNext(); )
    {
      onepart = (AccPart)it.next();
      System.out.println("Decoding␣part␣with␣repairIndex:␣" + onepart.repairIndex);
      System.arraycopy( onepart.data.getBytes(), 0, received, i*packetsize, onepart.data.getBytes().length );
      receiverIndex[i] = onepart.repairIndex;
      i++;
    }

    //Create Buffers for the encoded data
    for( i = 0; i < m; i++ )
      receiverBuffer[i] = new Buffer( received, i*packetsize, packetsize );

    FECCodeFactory factory = FECCodeFactory.getDefault();
    FECCode fec = FECCodeFactory.getDefault().createFECCode(m,n);

    //Finally we can decode
    fec.decode(receiverBuffer, receiverIndex);
    System.out.println("Decoded␣data:␣" + HexDump.toHex(received) + "(" +received.length + "␣bytes)");

    return received;

  }

  private void decrypt(byte[] encryptedfile, SecretKeySpec enckey)
  {
    byte[] decryptedfile = null;

    try
    {
      System.out.println("Encryption␣key:␣" + HexDump.toHex(enckey.getEncoded()));
      Cipher bfc;
      bfc = Cipher.getInstance("DES/ECB/PKCS5Padding"); //Blowfish/CBC/PKCS5Padding
      bfc.init(Cipher.DECRYPT_MODE, enckey);

      decryptedfile = bfc.doFinal(encryptedfile);
      System.out.println("Decrypted␣data:␣" + HexDump.toHex(decryptedfile) + ",␣length:␣" + decryptedfile.
          length);
      System.out.println("String␣representation␣of␣the␣decrypted␣data:␣"+new String(decryptedfile));
    }
    catch(Exception e) { System.out.println(e); }
  }

}

class ECFile
{
  public SecretKeySpec enckey;
  public int n, m;
  public int current_m = 0;
  public boolean notDecoded = false;
  public Hashtable accparts = new Hashtable(); //List of AccPart objects

  public byte[] data;
```

```
  public ECFile(MetaRNRow mrr)
  {
    enckey = mrr.enckey;
    n = mrr.n;
    m = mrr.m;
  }
}
```

A.5 DCPart.java

```
package accordion;

import java.io.Serializable;
import rice.pastry.Id;

public class DCPart implements Serializable
{
  public Id id;
  public byte[] data;
  public int size;
  public int repairIndex;

  public DCPart(int size)
  {
    this.size = size;
    data = new byte[size];
  }

  public String toString() { return "DCPart"; }

  public String toStringFull()
  {
    String s = "[DCPart] id: " + id + ", size: " + size + " and data: ";

    if (data==null) s += "null";
    else s+= HexDump.toHex(data).substring(0,10) + "...";

    return s;
  }
}
```

A.6 MainApp.java

```
package accordion;

import rice.pastry.NodeHandle;
import rice.pastry.PastryNode;
import rice.pastry.PastrySeed;
import rice.pastry.PastryNodeFactory;
import rice.pastry.dist.DistPastryNodeFactory;
import rice.pastry.direct.DirectPastryNodeFactory;
import rice.pastry.direct.NetworkSimulator;
import rice.pastry.direct.EuclideanNetwork;
```

```
import rice.pastry.standard.IPNodeIdFactory;
import rice.pastry.standard.RandomNodeIdFactory;

import java.io.IOException;
import java.io.BufferedReader;
import java.io.InputStreamReader;

import java.net.InetSocketAddress;
import java.net.InetAddress;
import java.net.UnknownHostException;

import java.util.Date;
import java.util.Iterator;
import java.util.NoSuchElementException;
import java.util.Random;
import java.util.TreeMap;
import java.util.Vector;

public class MainApp
{
  private static boolean isSimulation = false;
  private static boolean isBootstrap = true;

  //Simulation
  private static NetworkSimulator simulator;

  //Real network
  private static int PROTOCOL = DistPastryNodeFactory.PROTOCOL_WIRE;
  private static String BSHOST = null;
  private static int BSPORT = 5009;

  //Common
  private static Vector pastryNodes;
  private static Vector accordionApps;
  private static TreeMap pastryNodesSorted;
  private static PastryNodeFactory factory;
  private static Random rng;

  public MainApp()
  {
    pastryNodes = new Vector();
    accordionApps = new Vector();
    pastryNodesSorted = new TreeMap();
    rng = new Random(PastrySeed.getSeed());

    if (isSimulation)
    {
      simulator = new EuclideanNetwork();
      factory = new DirectPastryNodeFactory(new RandomNodeIdFactory(), simulator);
    }
    else
    {
      factory = DistPastryNodeFactory.getFactory(new IPNodeIdFactory(BSPORT), PROTOCOL, BSPORT);
    }
  }

  //------Real network------
```

```
/**
 * Gets a handle to a bootstrap node. First we try localhost, to see
 * whether a previous virtual node has already bound itself there.
 * Then we try nattempts times on BSHOST:BSPORT. Then we fail.
 *
 * @param firstNode true of the first virtual node is being bootstrapped on this host
 * @return handle to bootstrap node, or null.
 */
protected NodeHandle getRealBootstrap()
{
  InetSocketAddress addr = null;

  if (!isBootstrap && BSHOST!=null && BSHOST!="\n")
  {
    System.out.println("BSHOST: " + BSHOST);
    addr = new InetSocketAddress(BSHOST, BSPORT);
  }
  else
  {
    try
    {
      addr = new InetSocketAddress(InetAddress.getLocalHost().getHostName(), BSPORT);
    }
    catch(UnknownHostException e)
    {
      System.out.println(e);
    }
  }

  NodeHandle bshandle = ((DistPastryNodeFactory)factory).getNodeHandle(addr);
  return bshandle;
}

/**
 * Create a Pastry node and add it to pastryNodes. Also create a client
 * application for this node, so that when this node comes up (when
 * pn.isReady() is true), this application's notifyReady() method
 * is called, and it can do any interesting stuff it wants.
 */
public PastryNode makeRealNode()
{
  NodeHandle bootstrap = getRealBootstrap();
  PastryNode pn = factory.newNode(bootstrap); // internally initiateJoins
  pastryNodes.addElement(pn);

  AccordionApplication app = new AccordionApplication(pn);
  accordionApps.addElement(app);
  return pn;
}

//-----Simulation-----

private NodeHandle getSimulationBootstrap(boolean firstNode)
{
  NodeHandle bootstrap = null;
  try
```

```
    {
      PastryNode lastnode = (PastryNode) pastryNodes.lastElement();
      bootstrap = lastnode.getLocalHandle();
    }
    catch (NoSuchElementException e) {}
    return bootstrap;
  }

  private void makeSimulatedNode()
  {
    NodeHandle bootstrap = getSimulationBootstrap(pastryNodes.size() == 0);
    PastryNode pn = factory.newNode(bootstrap);
    pastryNodes.addElement(pn);

    AccordionApplication app = new AccordionApplication(pn, simulator);
    accordionApps.addElement(app);
    if (bootstrap != null) while(simulate());
  }

  private boolean simulate()
  {
    boolean res = simulator.simulate();
    return res;
  }

  private void killNodes(int num)
  {
    PastryNode pn = null;

    for (int i=0; i<num; i++)
    {
      int n = rng.nextInt(pastryNodes.size());

      //Don't kill the client
      if (n!=1)
      {
          pn = (PastryNode)pastryNodes.get(n);
          pastryNodes.remove(n);
          accordionApps.remove(n);
          killNode(pn);
          System.out.println("Killed " + pn.getNodeId());
      }
    }
  }

  private void killNode(PastryNode pn)
  {
    NetworkSimulator enet = (NetworkSimulator)simulator;
    enet.setAlive(pn.getNodeId(), false);
  }

  //-----

  private void testPublication(int i)
  {
    AccordionApplication app = (AccordionApplication)accordionApps.elementAt(i);
    app.publish("The Meaning of Life.txt");
```

```
}

private void testRetrieval(int i)
{
  AccordionApplication app = (AccordionApplication)accordionApps.elementAt(i);
  app.retrieve("The␣Meaning␣of␣Life.txt");
}

//-----

private static void parse(String args[])
{
  for (int i = 0; i < args.length; i++)
  {
    if (args[i].equals("--help"))
    {
      System.out.println("Usage:␣MainApp␣[-simulation␣yes/no]␣[-bootstrap␣ip]");
      System.out.println("");
      System.exit(1);
    }
  }

  for (int i = 0; i < args.length; i++)
  {
    if (args[i].equals("-simulation") && i+1 < args.length)
    {
      if (args[i+1]!=null && args[i+1].equals("yes"))
      {
        isSimulation = true;
        break;
      }
    }
  }

  for (int i = 0; i < args.length; i++)
  {
    if (args[i].equals("-bootstrap") && i+1 < args.length)
    {
      BSHOST = args[i+1];
      if (BSHOST!=null && BSHOST!="") isBootstrap = false;
      break;
    }
  }

}

//-----

public static void main(String args[])
{
  parse(args);
  MainApp driver = new MainApp();

  BufferedReader br = new BufferedReader(new InputStreamReader(System.in));

  if (driver.isSimulation)
  {
```

```
        int nodes = 1000;
        double killfactor  = 0.1;

        String strnodes = null;

        System.err.print("Number␣of␣nodes␣to␣create␣(default␣" + nodes + "):␣");
        try { strnodes = br.readLine(); } catch(IOException ie) { System.out.println(ie.toString()); }
        if (strnodes!=null && !strnodes.equals("")) nodes = Integer.parseInt(strnodes);

        for (int i=0; i<nodes; i++) driver.makeSimulatedNode();

        System.err.println("\nCreated␣" + nodes + "␣nodes.␣Press␣ENTER␣to␣test␣publication␣and␣retrieval
            .\n");
        try { System.in.read(); } catch(IOException ie) { System.out.println(ie.toString()); }

        driver.testPublication(0);
        System.err.println("Press␣ENTER␣to␣kill␣nodes");
        try { System.in.read(); } catch(IOException ie) { System.out.println(ie.toString()); }

        driver.killNodes((int)(nodes*killfactor));
        System.err.println("Press␣ENTER␣to␣start␣a␣test␣retrieval");

        try { System.in.read(); } catch(IOException ie) { System.out.println(ie.toString()); }
        driver.testRetrieval(1);
        System.err.println("Press␣ENTER␣to␣exit␣program");

        try { System.in.read(); } catch(IOException ie) { System.out.println(ie.toString()); }
      }
      else
      {
        //PastryNode pn = driver.makeRealNode();
        driver.makeRealNode();

        if (!isBootstrap)
        {
          System.out.println("Press␣ENTER␣to␣start␣a␣test␣publication.");
          try { System.in.read(); } catch(IOException ie) { System.out.println(ie.toString()); }
          driver.testPublication(0);
          System.out.println("Press␣ENTER␣to␣start␣a␣test␣retrieval.");
          try { System.in.read(); } catch(IOException ie) { System.out.println(ie.toString()); }
          driver.testRetrieval(0);
        }

      }
    }

}
```

A.7 MetaRN.java

```
package accordion;

import accordion.messagetypes.AccordionMessage;
import accordion.messagetypes.MetaPublication;
import accordion.messagetypes.MetaRNMessage;
import accordion.tabletypes.MetaRNRow;
```

```
import rice.pastry.Id;
import rice.pastry.NodeId;

import java.util.ArrayList;
import java.util.Iterator;

import java.security.AlgorithmParameters;

class MetaRN implements AccNode
{
  private AccordionApplication app;
  private ArrayList publishMetaTable;

  public MetaRN(AccordionApplication aa)
  {
    app = aa;
    publishMetaTable = new ArrayList();
  }

  public void handle(AccordionMessage am)
  {
    if (am instanceof MetaPublication)
    {
      MetaPublication mp = (MetaPublication)am;
      save(mp);
    }
    else if (am instanceof MetaRNMessage)
    {
      MetaRNMessage mrm = (MetaRNMessage)am;
      System.out.println("Meta root node "+app.getNodeId()+" received a request for file id "+mrm.id
          +" from "+mrm.source.toString()+", sending part id list");
      mrm.metaRows = get(mrm.id);
      mrm.isReply = true;
      NodeId client = mrm.source;
      mrm.source = app.getNodeId();
      app.forward(client, mrm);
    }
  }

  private void save(MetaPublication pm)
  {
    MetaRNRow row = new MetaRNRow();
    row.id = pm.id;
    row.partIdList = pm.partIdList;
    row.enckey = pm.enckey;
    row.n = pm.n;
    row.m = pm.m;
    publishMetaTable.add(row);
    System.out.println("Meta root node " + app.getNodeId() + " saved meta data for file id " + row.id.
        toStringFull()+". "+pm.toStringFull());
  }

  private ArrayList get(Id fid)
  {
    ArrayList resultrows = new ArrayList();
    MetaRNRow row = null;
```

```
    //System.out.println("pmt size: " + publishMetaTable.size());
    //System.out.println(app.getNodeId() + " is looking for mp");

    for (Iterator i = publishMetaTable.iterator(); i.hasNext();)
    {
      row = (MetaRNRow)i.next();
      //System.out.println("MRR at MetaRN: " + row);
      if (row.id.equals(fid)) resultrows.add(row);

      //System.out.println("MetaRN, M: " + row.M);
    }

    return resultrows;
  }

}
```

A.8 PartProvider.java

```
package accordion;

import rice.pastry.NodeHandle;
import rice.pastry.Id;
import rice.pastry.NodeId;
import rice.pastry.PastryNode;
import rice.pastry.routing.RouteSet;
import rice.pastry.routing.RoutingTable;
import rice.pastry.standard.RandomNodeIdFactory;

import accordion.messagetypes.AccordionMessage;
import accordion.messagetypes.PartPublication;
import accordion.messagetypes.StoredPart;
import accordion.messagetypes.PublishedPart;
import accordion.messagetypes.SubmittedPart;
import accordion.messagetypes.PartMessage;

import accordion.tabletypes.PartProviderRow;
import accordion.tabletypes.SeedRow;

import java.util.ArrayList;
import java.util.HashSet;
import java.util.Iterator;
import java.util.Random;

public class PartProvider implements AccNode
{
  private AccordionApplication app;
  private PastryNode ownnode;

  private ArrayList partProviderTable;
  private Random prng = new Random();

  public PartProvider(AccordionApplication aa, PastryNode pn)
  {
    app = aa;
```

```
    ownnode = pn;
    partProviderTable = new ArrayList();
  }

  /*
   * "Handle" the incoming AccordionMessage.
   */
  public void handle(AccordionMessage am)
  {
    if (am instanceof PartPublication)
    {
      save(am);
    }
    else if (am instanceof StoredPart)
    {
      save(am);
    }
    else if (am instanceof PublishedPart)
    {
      save(am);
    }
    else if (am instanceof PartMessage)
    {
      PartMessage pm = (PartMessage)am;
      //System.out.println("Node " + app.getNodeId() + " is calling sendPart() for pid : " + pm.id);
      sendPart(pm);
    }
  }

  /*
   * Save the incoming message by performing the actions appropriate
   * for the role the current node is currently playing.
   */
  private void save(AccordionMessage am)
  {
    //Performed by PS
    if (am instanceof PartPublication)
    {
      PartPublication partpub = (PartPublication)am;

      //1 Save the part
      PartProviderRow rowtoadd = new PartProviderRow(partpub.dcpart.size);
      rowtoadd.id = partpub.dcpart.id;
      rowtoadd.dcpart.id = partpub.dcpart.id;
      rowtoadd.dcpart.size = partpub.dcpart.size;
      rowtoadd.dcpart.data = partpub.dcpart.data;
      rowtoadd.dcpart.repairIndex = partpub.dcpart.repairIndex;
      partProviderTable.add(rowtoadd);

      ArrayList partProviderList = generatePPL();

      System.out.println("Part␣storer␣" + app.getNodeId() + "␣saved␣data␣for␣part␣id␣" + rowtoadd.id + "␣
          and␣selected␣the␣following␣nodes␣to␣be␣included␣in␣the␣part␣provider␣list:␣");

      for ( Iterator ppli = partProviderList.iterator() ; ppli.hasNext();) System.out.print((NodeId)ppli.next
          () + ",␣");
```

```
    //2 Build a set of node pairs from the PPL
    HashSet uniquepairs = new HashSet();
    Seed seed = null;
    NodeId x = null, y = null;

    for (int i=0; i<partProviderList.size(); i++)
    {
      x = (NodeId)partProviderList.get(i);

      for (int j=0; j<partProviderList.size(); j++)
      {
        y = (NodeId)partProviderList.get(j);
        seed = new Seed(x,y);
        if (!x.equals(y) && !contains(uniquepairs, seed)) uniquepairs.add(seed);
      }
    }

    //3 Generate seeds by assigning a random seed to each pair
    ArrayList seedlist = new ArrayList();

    for (Iterator i = uniquepairs.iterator(); i.hasNext();)
    {
      seed = (Seed)i.next();
      seed.value = Math.abs(prng.nextLong());
      seedlist.add(seed);
      //System.out.println(seed.toStringFull());
    }


    //4 Generate the PS' local seed list
    rowtoadd.seedList = getSeedList(seedlist, app.getNodeId());

    //5 Send a message containing part meta data and seedlist to a random PPL node
    partpub.dcpart.data = null;
    StoredPart sp = new StoredPart(app.appAddress, null, partpub.dcpart, partProviderList, seedlist);

    NodeId pplnode = app.getRandomPPLNode(partProviderList);
    app.forward(pplnode, sp);
  }

  //***TO-DO: CHECK THAT THE LIST IS AT LEAST k LONG BEFORE SAVING!
  //***TO-DO: CHECK THAT THE CURRENT NODE IS IN THE PPL BEFORE SAVING!

  //Performed by PartPublisher
  else if (am instanceof StoredPart)
  {
    StoredPart sp = (StoredPart)am;
    PartProviderRow rowtoadd = (PartProviderRow)app.getRow(partProviderTable, sp.dcpart.id);

    ArrayList localSeedList = null;

    if (rowtoadd==null) //If PS itself becomes PartPublisher it should not save the part again
    {
      //1 Save the part meta data
      rowtoadd = new PartProviderRow(sp.dcpart.size);
      localSeedList = getSeedList(sp.seedList, app.getNodeId());
```

```
        rowtoadd.id = sp.dcpart.id;
        rowtoadd.dcpart.id = sp.dcpart.id;
        rowtoadd.dcpart.size = sp.dcpart.size;
        rowtoadd.dcpart.data = null;
        rowtoadd.dcpart.repairIndex = sp.dcpart.repairIndex;
        rowtoadd.seedList = localSeedList;
        partProviderTable.add(rowtoadd);
        //System.out.println("Node " + app.getNodeId() + " is PartPublisher. It saved " + rowtoadd.
            toStringFull());
        System.out.println("PartPublisher " + app.getNodeId() + " saved information about part id : " +
            rowtoadd.id.toString());
      }
      else
        System.out.println("Incidentally, the PartPublisher is the same node as the PartStorer, namely
            "+app.getNodeId()+"\n");

      //2 Send the part meta data to the rest of the nodes in the PPL
      PublishedPart pubpart;
      NodeId pplnode = null;

      for ( Iterator i = sp.partProviderList.iterator () ; i.hasNext();)
      {
        pplnode = (NodeId)i.next();
        if (! app.getNodeId().equals(pplnode))
        {
          localSeedList = getSeedList(sp.seedList , pplnode);
          pubpart = new PublishedPart(app.appAddress, null, sp.dcpart, localSeedList);
          app.forward(pplnode, pubpart);
          //System.out.println("PartPublisher \t -> " + app.getNodeId() + " sent a PublishedPart to node " +
              pplnode);
        }
      }

      //3 Submit the part meta data to the PartRN
      SubmittedPart subpart = new SubmittedPart(app.appAddress, null, sp.dcpart.id, sp.partProviderList);
      app.submit(sp.dcpart.id, subpart);
      //System.out.println("Node " + app.getNodeId() + " is submitting a part to PartRN, the node with id
          closest to : " + sp.dcpart.id);

    }

    //Performed by {PPL} \ {PartPublisher}
    else if (am instanceof PublishedPart)
    {
      PublishedPart pubpart = (PublishedPart)am;
      //System.out.println("PartProvider \t -> " + app.getNodeId() + " looking for pid : " + pubpart.dcpart.
          id);
      PartProviderRow rowtoadd = (PartProviderRow)app.getRow(partProviderTable, pubpart.dcpart.id);

      if (rowtoadd==null) //PS has already saved the part and does therefore not save the part again
      {
        rowtoadd = new PartProviderRow(pubpart.dcpart.size);

        rowtoadd.id = pubpart.dcpart.id;
        rowtoadd.dcpart.id = pubpart.dcpart.id;
        rowtoadd.dcpart.size = pubpart.dcpart.size;
        rowtoadd.dcpart.data = null;
```

```
        rowtoadd.dcpart.repairIndex = pubpart.dcpart.repairIndex;
        rowtoadd.seedList = pubpart.seedList;
        partProviderTable.add(rowtoadd);
        //System.out.println("PPL Node " + app.getNodeId() + " is a part provider. It saved " + rowtoadd.
            toStringFull());
        System.out.println("PartProvider␣"+app.getNodeId()+"␣saved␣information␣for␣part␣id␣" +
            rowtoadd.id.toString());
      }

    }
  }

  /*
   * Send the local data for the part with id = pid according to the DC-network protocol to node = client.
   */
  private void sendPart(PartMessage pm)
  {
    Id pid = pm.id;
    NodeId client = pm.source;

    //System.out.println("PS/PP \t\t -> " + app.getNodeId() + " looking for pid: " + pid);
    PartProviderRow row = (PartProviderRow)app.getRow(partProviderTable, pid);
    //System.out.println("*****pid: " + pid + " = row.dcpart.id : " + row.dcpart.id + "?") ;

    if (row==null)
      pm.returnCode = -1;
    else
    {
      Random prng = null;
      SeedRow sr = null;
      long seed;
      byte[] coin;
      ArrayList coins = new ArrayList();

      for ( Iterator i = row.seedList.iterator () ; i.hasNext();)
      {
        sr = (SeedRow)i.next();
        seed = sr.value;
        prng = new Random(seed);
        coin = new byte[row.dcpart.size];
        prng.nextBytes(coin);
        coins.add(coin);

        System.out.println("Seed␣element␣" + sr.id + "␣at␣node␣" + app.getNodeId());
      }

      pm.dcpart = new DCPart(row.dcpart.size);
      pm.dcpart.id = row.dcpart.id;
      pm.dcpart.repairIndex = row.dcpart.repairIndex;

      boolean done = false;

      Iterator j = null;
      for (int i=0; i<row.dcpart.size; i++)
      {
        for (j = coins.iterator () ; j.hasNext();)
        {
```

```
            coin = (byte[]) j.next();
            pm.dcpart.data[i] ^= (byte)coin[i];
          }

          if (row.dcpart.data!=null) //Done by PS
          {
            if (!done)
            {
              done = true;
              System.out.println("PS " + app.getNodeId() + " is storing " + HexDump.toHex(row.dcpart.data)
                  );
            }
            pm.dcpart.data[i] ^= row.dcpart.data[i];
          }
          //else
          //System.out.println("Part Provider " + app.getNodeId() + " is storing " + HexDump.toHex(row.
              dcpart.data));
        }

        pm.source = app.getNodeId();
        pm.isReply = true;
        app.forward(client, pm);
      }
    }

    /*
     * Generate a part provider list containing k elements. Currently
     * do this from random entries in the current node's routing table.
     */
    private ArrayList generatePPL()
    {
      HashSet hs = new HashSet();                    //Use a set to ensure unique nodes
      RoutingTable rt = null;
      RouteSet[] rtrow = null;
      RouteSet rs = null;
      NodeHandle nh = null;
      int addedNodes = 0;

      hs.add(app.getNodeId());

      rt = ownnode.getRoutingTable();
      //System.out.println(rt);

      while (addedNodes<app.k-1)
      {
        //System.out.println("Looking for PPL nodes...");

        for (int i=0; i<rt.numRows(); i++)
        {
          if (addedNodes>=app.k-1) break;
          rtrow = rt.getRow(i);

          for (int j=0; j<rtrow.length; j++)
          {
            if (rtrow[j]!=null)
            {
              if (Math.random() > 0.5 && addedNodes<app.k-1)
```

```
            {
              nh = rtrow[j].get(0);
              if (hs.add(nh.getNodeId())) addedNodes++;
            }
          }
        }
      }
    }

    NodeId node = null;
    ArrayList ppl = new ArrayList();

    for (Iterator i = hs.iterator(); i.hasNext();)
    {
      node = (NodeId)i.next();
      ppl.add(node);
      //System.out.println("PS -> " + app.getNodeId() + " added a ppl node with id " + node);
    }


    shuffleList(ppl);
    //System.out.println("Node " + app.getNodeId() + " generated a PPL with " + ppl.size() + " nodes");
    return ppl;
  }


  /*
   * Shuffles the PPL so PS' position in the list cannot be determined.
   */
  private void shuffleList(ArrayList ppl)
  {
    int count = (int)(Math.random()*50);
    int r1, r2;
    NodeId tmp = null;

    for (int i=0; i<count; i++)
    {
      r1 = (int)(Math.random()*(ppl.size()-1));
      r2 = (int)(Math.random()*(ppl.size()-1));

      tmp = (NodeId)ppl.get(r1);
      ppl.set(r1, (NodeId)ppl.get(r2));
      ppl.set(r2, tmp);
    }
  }

  /*
   * Gets a "local" seed list for a PPL node from the "global" seed
   * list generated by PS
   */
  private ArrayList getSeedList(ArrayList seedlist, NodeId own)
  {
    Seed seed = null;
    SeedRow sr = null;
    ArrayList localseedlist = new ArrayList();
    boolean add = false;
```

```
    for ( Iterator  i =  seedlist . iterator () ;  i .hasNext();)
    {
      add = false;
      seed = (Seed)i.next();

      if  (seed.x.equals(own))
      {
        sr  =  new SeedRow(seed.y);
        sr .value = seed.value;
        add = true;
      }

      if  (seed.y.equals(own))
      {
        sr  =  new SeedRow(seed.x);
        sr .value = seed.value;
        add = true;
      }

      if  (add)  localseedlist .add(sr);
    }

    return  localseedlist ;
  }

  private boolean contains(HashSet seedlist, Seed seed)
  {
    boolean result = false;
    Seed current = null;

    for ( Iterator  i =  seedlist . iterator () ;  i .hasNext();)
    {
      current = (Seed)i.next();
      result  = (current.x.equals(seed.x) && current.y.equals(seed.y)  ||  (current.x.equals(seed.y) && current.
          y.equals(seed.x)));
      if  ( result )  break;
    }

    return  result ;
  }

}
```

A.9 PartRN.java

```
package accordion;

import accordion.messagetypes.AccordionMessage;
import accordion.messagetypes.SubmittedPart;
import accordion.messagetypes.PartRNMessage;

import accordion.tabletypes.AccRow;
import accordion.tabletypes.PartRNRow;

import rice.pastry.Id;
import rice.pastry.NodeId;
```

```
import java.util.ArrayList;
import java.util.Iterator;

class PartRN implements AccNode
{
  private AccordionApplication app;
  private ArrayList publishPartTable;

  public PartRN(AccordionApplication aa)
  {
    app = aa;
    publishPartTable = new ArrayList();
  }

  public void handle(AccordionMessage am)
  {
    if (am instanceof SubmittedPart)
    {
      SubmittedPart subpart = (SubmittedPart)am;
      save(subpart);
    }
    else if (am instanceof PartRNMessage)
    {
      PartRNMessage prm = (PartRNMessage)am;
      System.out.println("Part root node "+app.getNodeId()+" received request for part id "+prm.id+",
          sending part provider list.");
      prm.partRows = get(prm.id);
      prm.isReply = true;

      NodeId client = prm.source;
      prm.source = app.getNodeId();

      app.forward(client, prm);
    }
  }

  private void save(SubmittedPart subpart)
  {
    PartRNRow row = new PartRNRow();
    row.id = subpart.id;
    row.partProviderList = subpart.partProviderList;
    publishPartTable.add(row);
    System.out.println("Node " + app.getNodeId() + " is part root node for part id " + row.id.toString());
  }

  private ArrayList get(Id partId)
  {
    ArrayList resultrows = new ArrayList();
    AccRow row = null;

    for (Iterator i = publishPartTable.iterator(); i.hasNext();)
    {
      row = (AccRow)i.next();
      if (row.id.equals(partId)) resultrows.add(row);
    }
```

```
    return resultrows;
  }


}
```

A.10 Publisher.java

```
package accordion;

import accordion.messagetypes.MetaPublication;
import accordion.messagetypes.PartPublication;

import com.onionnetworks.fec.FECCode;
import com.onionnetworks.fec.FECCodeFactory;
import com.onionnetworks.util.Buffer;

import rice.pastry.Id;
import rice.pastry.NodeHandle;
import rice.pastry.NodeId;
import rice.pastry.PastryNode;
import rice.pastry.messaging.Address;
import rice.pastry.leafset.LeafSet;
import rice.pastry.routing.RouteSet;
import rice.pastry.routing.RoutingTable;

import java.security.AlgorithmParameters;
import java.security.MessageDigest;
import java.util.ArrayList;
import java.util.Arrays;
import java.util.Iterator;
import java.util.Random;
import javax.crypto.Cipher;
import javax.crypto.spec.SecretKeySpec;

/***
    One Publisher object handles publication of one file: meta data and parts.
*/
public class Publisher extends Thread
{
  private static AccordionApplication app;
  private Random rand = new Random();

  private String filename;
  private int m = 4;
  private int n = 8*m;
  private int packetsize;

  private PastryNode pn;

  public Publisher(AccordionApplication aa, String fn, PastryNode pn)
  {
    app = aa;
    filename = fn;
    this.pn = pn;
  }
```

```
public void run()
{
  //byte[] source = new byte[95];                                    //0 Generate random junk file
  //rand.nextBytes(source);

  String contents = "In␣the␣beginning␣God␣created␣the␣heaven␣and␣the␣earth.␣And␣the␣earth␣was␣
      without␣form,␣and␣void;␣and␣darkness␣was␣upon␣the␣face␣of␣the␣deep.␣And␣the␣Spirit␣of␣
      God␣moved␣upon␣the␣face␣of␣the␣waters.␣And␣God␣said,␣Let␣there␣be␣light:␣and␣there␣was
      ␣light.␣And␣God␣saw␣the␣light,␣that␣it␣was␣good:␣and␣God␣divided␣the␣light␣from␣the␣
      darkness.␣And␣God␣called␣the␣light␣Day,␣and␣the␣darkness␣he␣called␣Night.␣And␣the␣
      evening␣and␣the␣morning␣were␣the␣first␣day.";

  byte[] source = contents.getBytes();

  System.out.println("Contents␣of␣file␣to␣be␣published:␣\"" + contents + "\"␣(length␣=␣" + source.length
      + "␣bytes)");

  MetaPublication mp = new MetaPublication(app.appAddress, null);   //1 Create a MetaPublication
      message with no source
  mp.id = new Id(app.hash(filename.getBytes()));                     //2 Calculate the fileId
  byte[] encryptedfile = encrypt(source, mp);                        //3 Encrypt the file − and set the
      PD key in the MetaPublication message

  System.out.println("File␣Id:␣"+mp.id);

  System.out.println("Hexadecimal␣representation␣of␣contents␣in␣plaintext:␣" + HexDump.toHex(source)
      + "␣(length␣=␣" + source.length+ "␣bytes)");

  System.out.println("Hexadecimal␣representation␣of␣encrypted␣contents:␣" + HexDump.toHex(
      encryptedfile) + "␣(length␣=␣" + encryptedfile.length+ "␣bytes)");

  packetsize = encryptedfile.length / m;

  ArrayList pubparts = encode(encryptedfile);                        //4 Erasure code the file

  mp.n = n;                                                          //5 Set the MetaPublication
      message's EC params
  mp.m = m;

  for (Iterator i = pubparts.iterator(); i.hasNext();)               //6 Build the partIdList
    mp.partIdList.add(((DCPart)i.next()).id);

  app.forward(mp);                                                   //7 Publish meta data
  System.out.println("Publisher␣"+app.getNodeId() + "␣publishing␣meta␣data␣for␣file␣id␣" + mp.id.
      toString());

  PartPublication pp;
  for (Iterator i = pubparts.iterator(); i.hasNext();)               //8 Publish parts
  {
    pp = new PartPublication(app.appAddress, null, (DCPart)i.next());
    app.forward(pp);
    System.out.println("Publisher␣" + app.getNodeId() + "␣publishing␣part␣data␣for␣part␣id␣" + pp.
        dcpart.id.toString());
  }
}
```

```
//----------------
// ENCRYPTION
//----------------

private byte[] encrypt(byte[] source, MetaPublication mp)
{
  byte[] encryptedfile = null;

  try
  {
    Cipher bfc;
    MessageDigest md;
    SecretKeySpec myTripleDesKey;
    byte[] tripleDesKeyData = new byte[8];
    byte[] rawKey, hashedKey;

    rand.nextBytes(tripleDesKeyData);
    myTripleDesKey = new SecretKeySpec(tripleDesKeyData, "DES");
    bfc = Cipher.getInstance("DES/ECB/PKCS5Padding"); //Blowfish/CBC/PKCS5Padding
    mp.enckey = myTripleDesKey;

    bfc.init(Cipher.ENCRYPT_MODE, myTripleDesKey);
    encryptedfile = bfc.doFinal(source);
  }
  catch(Exception e) { System.out.println(e); }

  return encryptedfile;
}

//----------------
// ENCODING
//----------------

private ArrayList encode(byte[] encryptedfile)
{
  ArrayList parts = new ArrayList();

  if ((encryptedfile.length % m*packetsize) != 0)
  {
    System.out.println("- ERROR: Length of encrypted file: " + encryptedfile.length + " is not
        divisible by m*packetsize: " + m*packetsize);
    return null;
  }

  byte[] repair = new byte[n*packetsize]; //this will hold the encoded file

  //These buffers allow us to put our data in them
  //they reference a packet length of the file (or at least will once
  //we fill them)
  Buffer[] sourceBuffer = new Buffer[m];
  Buffer[] repairBuffer = new Buffer[n];

  for(int i = 0; i < sourceBuffer.length; i++) sourceBuffer[i] = new Buffer(encryptedfile, i*packetsize
      , packetsize);
  for(int i = 0; i < repairBuffer.length; i++) repairBuffer[i] = new Buffer(repair, i*packetsize,
      packetsize);
  int[] repairIndex = new int[n];
```

```
    for( int i = 0; i < repairIndex.length; i++ ) repairIndex[i] = i;

    FECCodeFactory factory = FECCodeFactory.getDefault();
    FECCode fec = FECCodeFactory.getDefault().createFECCode(m,n);
    fec.encode( sourceBuffer, repairBuffer, repairIndex );          //This is where the magic happens!

    // ***Naming confusion: the DCPart objects contain actual data
    // ***when publishing and random byte streams ("dc data") when
    // ***retrieving
    DCPart dcpart;
    byte[] part;

    for(int i=0; i<n; i++)
    {
      part = repairBuffer[i].getBytes();
      dcpart = new DCPart(part.length); // part.length = packetsize!

      dcpart.id = new Id(part);
      dcpart.data = part;
      dcpart.size = part.length;
      dcpart.repairIndex = i;
      parts.add(dcpart);
      System.out.println("Data␣for␣one␣part:␣" + HexDump.toHexShort(part) + ",␣part␣id:␣" + dcpart.id.
          toString());
    }

    return parts;
  }

}
```

A.11 RandNode.java

```
package accordion;

import accordion.messagetypes.AccordionMessage;
import accordion.messagetypes.MetaPublication;
import accordion.messagetypes.PartPublication;
import accordion.messagetypes.StoredPart;
import accordion.messagetypes.PublishedPart;
import accordion.messagetypes.SeedMessage;

import accordion.tabletypes.PartProviderRow;

import rice.pastry.Id;
import rice.pastry.NodeId;

import java.util.ArrayList;
import java.util.Iterator;
import java.util.Random;

class RandNode implements AccNode
{
  private static double PROB_SUBMIT = 0.4;
  private static double PROB_SAVE = 0.1;
```

```
private AccordionApplication app;
private PartProvider partprovider;

public RandNode(AccordionApplication aa, PartProvider pp) { app = aa; partprovider = pp; }

public void handle(AccordionMessage am)
{
  NodeId pplnode = null;

  if (am instanceof MetaPublication)
  {
    MetaPublication mp = (MetaPublication)am;

    if (doSubmit())
    {
      System.out.println("Random␣node␣" + app.getNodeId() + "␣submitting␣" + mp.toString() + "␣for␣file
          ␣id␣"+mp.id.toString()+"␣to␣the␣meta␣root␣node");
      app.submit(mp);
    }
    else
    {
      System.out.println("Random␣node␣" + app.getNodeId() + "␣forwarding␣" + mp.toString() + "␣for␣file
          ␣id␣"+mp.id.toString()+"␣to␣another␣random␣node");
      app.forward(mp);
    }
  }

  //----------------------Part publication----------------------

  else if (am instanceof PartPublication)
  {
    PartPublication partpub = (PartPublication)am;
    //System.out.println("Random node " + app.getNodeId() + " received message containing the part " +
        partpub.dcpart);
    //System.out.println("Node " + app.getNodeId() + " playing RandNode [ " + partpub.toString() + " ]");

    if (doSave())
    {
      partprovider.handle(partpub);
      System.out.println("Node␣" + app.getNodeId() + "␣is␣PartStorer.");
    }
    else
    {
      System.out.println("Random␣node␣" + app.getNodeId() + "␣is␣forwarding␣a␣PartPublication␣
          message");
      app.forward(partpub);
    }
  }

  else if (am instanceof StoredPart)
  {
    StoredPart sp = (StoredPart)am;

    if (doSubmit())
    {
      partprovider.handle(sp);
    }
```

```
        else
        {
          pplnode = app.getRandomPPLNode(sp.partProviderList);
          System.out.println("Random␣ppl␣node␣" + app.getNodeId() + "␣is␣forwarding␣StoredPart␣message
              ␣" + sp.dcpart.id.toString());
          app.forward(pplnode, sp);
        }
      }

      else if (am instanceof PublishedPart)
      {
        PublishedPart pubpart = (PublishedPart)am;
        partprovider.handle(pubpart);
        System.out.println("Node␣" + app.getNodeId().toStringFull() + "␣is␣now␣a␣part␣provider␣for␣" +
            pubpart + "\n");
      }

    }

    private boolean doSubmit() { if (Math.random() < PROB_SUBMIT) return true; else return false; }

    private boolean doSave() { if (Math.random() < PROB_SAVE) return true; else return false; }


}
```

A.12 Seed.java

```
package accordion;

import java.io.Serializable;
import rice.pastry.NodeId;

public class Seed implements Serializable
{
  public NodeId x, y;
  public long value;

  public Seed(NodeId x1, NodeId y1) { x = x1; y = y1; }

  public boolean equals(Seed seed)
  {
    //System.out.println((this.x.equals(seed.x) && this.y.equals(seed.y) || (this.x.equals(seed.y) && this.y.
        equals(seed.x))));
    return (this.x.equals(seed.x) && this.y.equals(seed.y) || (this.x.equals(seed.y) && this.y.equals(seed.x)
        ));
  }

  public String toStringFull()
  {
    String s = "Seed␣(x,y):␣(" + x + ",␣" + y + "),␣value:␣" + value + "\n";
    return s;
  }

}
```

A.13 AccordionMessage.java

```
package accordion.messagetypes;
import rice.pastry.Id;
import rice.pastry.NodeId;
import rice.pastry.messaging.Address;
import rice.pastry.messaging.Message;

public abstract class AccordionMessage extends Message
{
  public Id id;                    //Identifier for an Accordion message, e.g. a fileId or a partId
  public NodeId source;

  public AccordionMessage(Address address, NodeId source)
  {
    super(address);
    this.source = source;
  }

  public AccordionMessage(Address address, NodeId source, Id id)
  {
    super(address);
    this.source = source;
    this.id = id;
  }

  public String toString() { return "AccordionMessage"; }

  public String toStringFull()
  {
    String s = "AccordionMessage:\n";
    s += "\tid:␣" + id;
    s += "\tsource:␣" + source;
    return s;
  }
}
```

A.14 MetaPublication.java

```
package accordion.messagetypes;

import rice.pastry.Id;
import rice.pastry.NodeId;
import rice.pastry.messaging.Address;

import java.util.ArrayList;
import java.util.Iterator;
import javax.crypto.spec.SecretKeySpec;

public class MetaPublication extends AccordionMessage
{
  public ArrayList partIdList;
  public SecretKeySpec enckey;
  public int n, m;
```

```
  public MetaPublication(Address address, NodeId source)
  {
    super(address, source);
    partIdList = new ArrayList();
  }

  public String toString() { return "MetaPublication"; }

  public String toStringFull()
  {
      //super.toStringFull();

    String s = "Meta␣data␣includes␣the␣following␣part␣ids:␣\n";
    for (Iterator i = partIdList.iterator(); i.hasNext();)
    {
      s += (((Id)i.next()).toString()+",␣");
    }

    s += "and␣the␣encryption␣key␣is:␣" + HexDump.toHex(enckey.getEncoded());
    return s;
  }
}
```

A.15 MetaRNMessage.java

```
package accordion.messagetypes;

import accordion.tabletypes.MetaRNRow;

import rice.pastry.Id;
import rice.pastry.NodeId;
import rice.pastry.messaging.Address;

import java.util.ArrayList;
import java.util.Iterator;

public class MetaRNMessage extends AccordionMessage
{
  public ArrayList metaRows; //List of MetaRNRows
  public boolean isReply = false;
  public int returnCode = 0;

  public MetaRNMessage(Address address, NodeId source, Id fid) { super(address, source, fid); }

  public String toString() { return "MetaRNMessage"; }

  public String toStringFull()
  {
    super.toStringFull();

    String s = "MetaRNMessage:\n";
    for (Iterator i = metaRows.iterator(); i.hasNext();)
    {
      s += ("\tMetaRNRow:␣" + ((MetaRNRow)i.next()).toStringFull()+"\n");
    }
```

```
    s += "\tisReply:␣" + isReply + "\n";
    s += "\treturnCode:␣" + returnCode;
    return s;
  }
}
```

A.16 PartMessage.java

```
package accordion.messagetypes;

import accordion.DCPart;
import rice.pastry.messaging.Address;
import rice.pastry.Id;
import rice.pastry.NodeId;

public class PartMessage extends AccordionMessage
{
  public DCPart dcpart;
  public int sessionId;
  public int returnCode = 0;
  public boolean isReply = false;

  public PartMessage(Address address, NodeId src, Id id, int sid)
  {
    super(address, src, id);
    sessionId = sid;
  }

  public String toString() { return "PartMessage"; }

  public String toStringFull()
  {
    super.toStringFull();

    String s = "PartMessage:\n";
    s += "\tdcpart:␣" + dcpart.toStringFull();
    return s;
  }

}
```

A.17 PartPublication.java

```
package accordion.messagetypes;

import accordion.DCPart;

import rice.pastry.Id;
import rice.pastry.NodeId;
import rice.pastry.messaging.Address;
import java.util.ArrayList;

public class PartPublication extends AccordionMessage
{
```

```
  public DCPart dcpart;

  public PartPublication(Address address, NodeId source, DCPart dcp)
  {
    super(address, source);
    dcpart = dcp;
  }

  public String toString() { return "PartPublication"; }

  public String toStringFull()
  {
    super.toStringFull();

    String s = "PartPublication:\n";
    s += "\tdcpart: " + dcpart.toStringFull();
    return s;
  }

}
```

A.18 PartRNMessage.java

```
package accordion.messagetypes;

import accordion.tabletypes.PartRNRow;

import rice.pastry.Id;
import rice.pastry.NodeId;
import rice.pastry.messaging.Address;

import java.util.ArrayList;
import java.util.Iterator;

public class PartRNMessage extends AccordionMessage
{
  public ArrayList partRows; //List of PublishPartRows
  public int sessionId;
  public boolean isReply = false;
  public int returnCode = 0;

  public PartRNMessage(Address address, NodeId source, Id pid, int sid)
  {
    super(address, source, pid);
    sessionId = sid;
  }

  public String toString() { return "PartRNMessage"; }

  public String toStringFull()
  {
    super.toStringFull();

    String s = "PartRNMessage:\n";
    for (Iterator i = partRows.iterator(); i.hasNext();)
    {
```

```
      s += ("\tPartRNRow:␣" + ((PartRNRow)i.next()).toStringFull()+"\n");
    }

    s += "\tisReply:␣" + isReply + "\n";
    s += "\treturnCode:␣" + returnCode;
    return s;
  }

}
```

A.19 PublishedPart.java

```
package accordion.messagetypes;

import accordion.DCPart;

import rice.pastry.Id;
import rice.pastry.NodeId;
import rice.pastry.messaging.Address;

import java.util.ArrayList;
import java.util.Iterator;

public class PublishedPart extends AccordionMessage
{
  public DCPart dcpart;
  public ArrayList seedList;  //List of SeedRows

  public PublishedPart(Address address, NodeId source, DCPart part, ArrayList seedlist)
  {
    super(address, source);

    dcpart = new DCPart(part.size);

    dcpart.id = part.id;
    dcpart.size = part.size;
    dcpart.repairIndex = part.repairIndex;
    seedList = seedlist;
  }

  public String toString() { return "PublishedPart"; }

  public String toStringFull()
  {
    super.toStringFull();

    String s = "PublishedPart:\n";

    s += "\tdcpart:␣" + dcpart.toStringFull() + "\n";

    /*
    for (Iterator i = partProviderList.iterator(); i.hasNext();)
    {
      s += ("\tnodeId: " + ((NodeId)i.next()).toStringFull()+"\n");
    }
    */
```

```
    return s;
  }

}
```

A.20 StoredPart.java

```
package accordion.messagetypes;

import accordion.DCPart;
import accordion.Seed;

import rice.pastry.Id;
import rice.pastry.NodeId;
import rice.pastry.messaging.Address;

import java.util.ArrayList;
import java.util.Iterator;

public class StoredPart extends AccordionMessage
{
  public DCPart dcpart;
  public ArrayList partProviderList; //List of NodeId objects
  public ArrayList seedList;         //List of Seed objects

  public StoredPart(Address address, NodeId source, DCPart dcp, ArrayList ppl, ArrayList seedlist)
  {
    super(address, source);
    dcpart = dcp;
    partProviderList = ppl;
    seedList = seedlist;
  }

  public String toString() { return "StoredPart"; }

  public String toStringFull()
  {
    super.toStringFull();

    String s = "StoredPart:\n";

    s += "\tdcpart:␣" + dcpart.toStringFull() + "\n";

    Iterator i = null;

    for (i = partProviderList.iterator(); i.hasNext();)
    {
      s += ("\tNodeId:␣" + ((NodeId)i.next()).toStringFull()+"\n");
    }

    for (i = seedList.iterator(); i.hasNext();)
    {
      s += ("\tSeed:␣" + ((Seed)i.next()).toStringFull()+"\n");
    }
```

```
    return s;
  }

}
```

A.21 SubmittedPart.java

```
package accordion.messagetypes;

import accordion.DCPart;

import rice.pastry.Id;
import rice.pastry.NodeId;
import rice.pastry.messaging.Address;

import java.util.ArrayList;
import java.util.Iterator;

public class SubmittedPart extends AccordionMessage
{
  public ArrayList partProviderList;

  public SubmittedPart(Address address, NodeId source, Id pid, ArrayList ppl)
  {
    super(address, source, pid);
    partProviderList = ppl;
  }

  public String toString() { return "SubmittedPart"; }

  public String toStringFull()
  {
    super.toStringFull();

    String s = "SubmittedPart:\n";

    //s += "\tdcpart: " + dcpart.toStringFull() + "\n";

    /*
    for (Iterator i = partProviderList.iterator(); i.hasNext();)
    {
      s += ("\tnodeId: " + ((NodeId)i.next()).toStringFull()+"\n");
    }
    */

    return s;
  }

}
```

A.22 AccRow.java

```
package accordion.tabletypes;
```

```
import java.io.Serializable;
import rice.pastry.Id;

public class AccRow implements Serializable
{
  public Id id;
  public AccRow() {}
  public AccRow(Id id) { this.id = id; }

  public String toString() { return "AccRow"; }

  public String toStringFull()
  {
    String s = "[AccRow] id: " + id;
    return s;
  }

}
```

A.23 MetaRNRow.java

```
package accordion.tabletypes;

import java.util.ArrayList;
import javax.crypto.spec.SecretKeySpec;
import rice.pastry.Id;
import java.util.Iterator;

public class MetaRNRow extends AccRow
{
  public ArrayList partIdList;
  public SecretKeySpec enckey;
  public int n, m;

  public String toString() { return "MetaRNRow"; }

  public String toStringFull()
  {
    super.toStringFull();

    String s = "[MetaRNRow] ";
    for (Iterator i = partIdList.iterator(); i.hasNext();)
    {
      s += ("partId: " + (Id)i.next() + ", ");
    }

    s += "enckey: " + enckey;
    return s;
  }
}
```

A.24 PartProviderRow.java

```
package accordion.tabletypes;
```

```
import accordion.DCPart;
import rice.pastry.Id;
import rice.pastry.NodeId;
import java.util.ArrayList;
import java.util.Iterator;

public class PartProviderRow extends AccRow
{
  public DCPart dcpart;
  public ArrayList seedList;                    //List of SeedRow objects

  public PartProviderRow(int partsize)
  {
    dcpart = new DCPart(partsize);
    seedList = new ArrayList();
  }

  public String toString() { return "PartProviderRow"; }

  public String toStringFull()
  {
      //super.toStringFull();

    String s = "[PartProviderRow] id: " + super.id + "\n";
    s += "\t\t\t\t\t\t" + dcpart.toStringFull() + "\n";

    /*
    SeedRow row = null;
    for (Iterator i = seedList.iterator(); i.hasNext();)
    {
      row = (SeedRow)i.next();
      s += ("\t\t\t\t\t\t" + row.toStringFull() + "\n");
    }
    */

    s += "\n";

    return s;
  }

}
```

A.25 PartRNRow.java

```
package accordion.tabletypes;

import rice.pastry.Id;
import rice.pastry.NodeId;

import java.util.ArrayList;
import java.util.Iterator;

public class PartRNRow extends AccRow
{
  public ArrayList partProviderList; //List of nodeIds
```

```
  public String toString() { return "PartRNRow"; }

  public String toStringFull()
  {
    super.toStringFull();

    String s = "[PartRNRow] id: " + super.id + "\n\t\t\t\t\t\tPPL: ";
    for ( Iterator i = partProviderList.iterator(); i.hasNext(); )
    {
      s += ((NodeId)i.next() + ", ");
    }

    s += "\n";

    return s;
  }

}
```

A.26 SeedRow.java

```
package accordion.tabletypes;

import rice.pastry.NodeId;

public class SeedRow extends AccRow
{
  public long value;

  public SeedRow(NodeId nid) { super(nid); }
}
```