The Decentralized File System Igor-FS
as an Application for Overlay-Networks

zur Erlangung des akademischen Grades eines

Doktors der Ingenieurwissenschaften

der Fakultät für Informatik
der Universität Fridericiana zu Karlsruhe (TH)

genehmigte

Dissertation

von

Dipl.-Ing. Kendy Kutzner

aus Karl-Marx-Stadt

Tag der mündlichen Prüfung: 14. Februar 2008
Erster Gutachter: Dr. Thomas Fuhrmann
Zweiter Gutachter: Prof. Dr. Klaus Wehrle

Contents

Chapter 1

Introduction

Jedem Anfang wohnt ein Zauber inne
Hermann Hesse

1.1 Motivation

Working in distributed systems is part of the information society. More and more people and organizations work with growing data volumes.

Often, part of the problem is to access large files in a share way. Until now, there are two often used approaches to allow this kind off access. Either the files are tranfered via FTP, e-mail or similar medium before the access happens, or a centralized server provides file services. The first alternative has the disadvantage that the entire file has to be transfered before the first access can be successful. If only small parts in the file have been changed compared to a previous version, the entire file has to be transfered anyway. The centralized approach has disadvantages regarding scalability and reliability. In both approaches authorization and authentication can be difficult in case users are seperated by untrusted network segments.

1.2 Zusammenfassung in deutscher Sprache

Das Arbeiten in verteilten Systemen ist Bestandteil der Informationsgesellschaft. Immer mehr Menschen und Organisationen arbeiten mit immer größer werdenden Daten.

Dabei geht es auch häufig darum, gemeinsam auf diese große Datenmengen zuzugreifen. Bisher gibt es zwei gebräuchliche Ansätze um diese Art der Zusammenarbeit zu ermöglichen. Entweder werden die Dateien via FTP, E-Mail, oder Ähnlichem vor dem Zugriff transportiert oder ein zentraler File-Server stellt die Datei-Dienste zur Verfügung. Die erste Variante hat den Nachteil, dass immer

11

erst die gesamte Datei transportiert werden muss, bevor der erste Zugriff erfolgen kann. Wenn sich nur kleine Teile der Dateien ändern, muss trotzdem
der gesamte Datenbestand erneut kopiert werden. Der zentralisierte Ansatz
mit einem File-Server hat Nachteile bei der Skalierbarkeit und bei der Ausfallsicherheit. In beiden Ansätzen kann es Probleme mit der Authorisierung der
Zugriffe und Authentifizierung der Daten geben, wenn zwischen den Nutzern
nicht vertrauenswürdige Netzsegmente liegen.

Das Promotionsvorhaben verfolgt das Ziel ein Gesamtsystem zu beschreiben,
welches die beschriebenen Nachteile nicht mehr aufweist und zusätzlich den
Ansprüchen an Vertraulichkeit und Sicherheit genügt. Diese Arbeit ist dazu in
drei Teile untergliedert. Im ersten Teil werden die existierenden Grundlagen
dargelegt, die für die Arbeit relevant sind. Dabei liegen die Schwerpunkte auf
Kryptographie, Overlay-Netzen und Dateisystemen.

Im zweiten Teil wird das neu entwickelte Overlaynetz Igor beschrieben.
Strukturierte Overlaynetze bieten ein skalierbares Substrat für viele Anwendungen. Igor baut auf den im ersten Teil eingeführten Ideen auf, fügt serviceorientiertes Routing hinzu und bringt neuartige Wege zur Adaption an das
bestehende Netz ein. Serviceorientiertes Routing ist notwendig, damit mehrere
Applikationen das Overlaynetzwerk gleichzeitig nutzen können. Mehrere solche
neu entwickelten Anwendungen werden beschrieben, die wichtigste darunter ist
das Dateisystem aus dem dritten Teil der Arbeit.
Die Adaption an das darunterliegende Netz ist für Overlaynetze eine wichtige
Eigenschaft um hohe Datenraten und kleine Latenzen zu erreichen. Die vorliegende Arbeit beschreibt ein neues Verfahren, welches schnellen Erfolg mit
wenig Overhead kombiniert.

Der dritte Teil der Arbeit beschreibt schliesslich das verteilte und dezentralisiert Dateisystem IgorFS. Dieses nutzt das Overlaynetz Igor als Kommunikationssystem. Daten in IgorFS werden in einer besonderen Art in Blöcke zerteilt,
die das verteilte Erkennen und Vermeiden von Redundanzen ermöglicht. Solche
Redundanzen werden sowohl zwischen Dateien als auch zwischen verschiedenen Versionen der gleichen Datei erkannt. Datenblöcke werden ausschliesslich
verschlüsselt gespeichert und übertragen, so dass die Vertraulichkeit gewährleistet ist. Der Besitz des zur Dechiffrierung notwendigen Schlüsselmaterial wird
als Authorisierungsmerkmal benutzt. Dies erlaubt die sichere verteilte Authorisierung: Ohne Schlüssel sind keine Daten lesbar. Zusätzlich werden Blöcke
sowohl für Verzeichnisse als auch für Dateien über ihre kryptografische Prüfsumme addressiert, wodurch jeglicher Inhalt authentifiziert werden kann.

Die Datenblöcke werden nach Anforderung im System verteilt, was dazu
führt das von häufig angeforderten Blöcken vielen Repliken existieren. Dies
führt zu sehr skalierbaren Systemen, ohne dass zentrale Server notwendig sind.
Andererseits kann ein Zugriff bereits durchgeführt werden, sobald die dafür
notwendingen Blöcke lokal vorhanden sind. Es ist nicht notwendig, vorher die
gesamte Datei zu transferieren.

Die Benutzung von IgorFS erfordert an vorhandenen Applikationen keine
Änderungen, da über ein Kernel-Modul allen Linux-Anwendungen die gewohnte
POSIX-Schnittstelle zur Verfügung gestellt wird. Die vorstellbaren praktischen

Einsatzszenarien von IgorFS reichen von der Verteilung von Softwaredistributionen bis zur gemeinsamen Nutzung großer pharmazeutischer Datenbanken.

1.3 Structure of this work

In order to reach these goals, this work is subdivided in three major parts. In the first part, the existing foundations relevant for this work are laid out. A special focus is set on cryptography, overlay networks and file systems.

The second part describes the newly developed overlay network Igor. Such structured overlay networks form a rich substrate for many different kinds of applications. Igor is based on ideas presented in the first part and extends these ideas by service oriented routing and adds new ways to adapt to the underlaying network. Service oriented routing is essential if more than one application can use the overlay network at once. Some newly developed applications based on Igor are described, the most important one the file system IgorFS presented in the third part.

The adaption of the overlay network towards the underlaying system is an important property to reach higher data rates and lower latencies. This work presents a new method, which combindes faster success with less overhead.

The third part describes the distributed and decentralized file system IgorFS. This file system uses the overlay network Igor as communcation primitive. In IgorFS, data is cut into blocks in a special way in order to detect redundancies and avoid them. Such redundancies are detected between different versions of the same file as well as between different files on remote systems. Data blocks are stored and transfered in an encrypted form only. This way the confidentiality is ensured at all times. Possession of a decryption key is used as proof of authorization, allowing a distributed authorization scheme: No data is readable without the proper decryption key. Additionally, all data blocks are identified by their cryptographic hash sum, which automatically ensures authentication of all content.

Data blocks are distributed on demand, which leads to more replicas for objects requested more often. With this property, the overall system is very scalable, without the need for a centralized server system. On the other hand, an access can already be fulfilled as soon as the required block is available. It is not necessary that the entire file has to be transfered before.

To use IgorFS it is not neccessary to modifiy applications. Using a kernel module, IgorFS is avalible via the POSIX interface to all Linux applications. The imaginable usage scenarios for IgorFS reach from software distribution to shared usage of large pharmaceutical data bases.

Part I

Foundations

Chapter 2

Cryptography

Trying to stop this is like trying to legislate the tides and the weather.
Philip R. "Phil" Zimmermann Jr.

Later chapters in this work rely on cryptography to ensure security. Therefore this chapter will introduce some cryptographic primitives like encryption and hash functions. To explain things in cryptography, virtual characters introduced by [142] are often used. These virtual characters play standard roles and therefore can make understanding the description of cryptographic processes easier.

The most often used roles are listed below.

- The sender of a message *Alice*.

- *Bob*, the intended receiver of that message.

- In between Alice and Bob, *Eve* tries to eavesdrop the communication and learn as much as possible from doing so.

- Even more malicious, *Mallory* can change messages and tries to hamper the communication.

This work follows the tradition and uses the canonic virtual characters to describe cryptographic processes.

Cryptography deals, among others, with the following problems:

- Traditionally, protecting the confidentialiality of information was one of the major goals of cryptography. To keep information confidential, a larger secret (the *clear text*) is *encrypted* with a (usually) smaller secret (the *key*). This process yields the *ciphertext*. The reverse process, i. e. transforming the ciphertext into the clear text with the help of the key, is called *decryption*.

- *Authentication* is a procedure to ensure that a given data object is genuine, i. e. the user is really the one she claims to be or a data packet is from the assumed source.

- *Anonymity* tries to prevent that communicating parties are identifiable. To do that, *pseudonyms*, i. e. additional identities that are not easily linkable to the real identities, can be created.

- Sometimes it is necessary to hide the fact that communication happened at all. Cryptography can be a building block to implement *steganography*. A weaker concept is *plausible deniability*, where the communicating parties can deny knowledge about the communication and third parties cannot prove the contrary.

Note that the cryptographic algorithm can be public. The security of the system rests on the secrecy of the key and the computational infeasability of all known attacks. For few algorithms such an infeasability has been proven, nevertheless it is often assumed (*Standard Hardness Assumption*).

2.1   Hash Functions

A hash function in general is a deterministic function $h()$ that maps a number of input values m to a smaller number of integer output values h:

$$h = h(m)$$

Since the output h of the hash function can be smaller than the input m, hash function can not be bijective. Usually, it is desired that these output values are evenly distributed, so that the hash function can map keys to values in hash tables.

As such, they are related to CRC functions (see section 2.1.1).

2.1.1   Cyclic Redundancy Check

Cyclic Redundancy Checks (CRC) are used as a general purpose checksum algorithm. They are based on polynomial residue division. To apply polynomial residue division to bit streams, both the input bit stream as divisor as well as the dividend are taken as binary coefficients of the polynomials.

CRCs can be viewed as shift registers with feedback, where the dividing polynomial denotes the positions of feedback lines. Shifting the input data stream through the register yields the desired CRC value.

Cyclic redundancy checks are linear functions. As such, they are useful to discover unintended changes (e. g. transmission errors) of the message in transit. In general, they are not useful to protect messages in a cryptographic sense for integrity protection or authentication. Discovered flaws in the IEEE 802.11b WEP protocol [15] show this impressively.

2.1.2 Cryptographic Hash Functions

Cryptographic hash functions are hash functions that are sometimes also called message digest or compression functions. In the cryptographic context, hash functions are required to have the following additional properties:

- Given a value of h, it is hard to find any m that hashes to the value of $h = h(m)$ (*preimage resistance*).

- It is hard to find any two input values m_1, m_2 that hash to the same output value h (*collision resistance*). This means because of the birthday paradox the hash output size must be at least twice as large as required for preimage resistance.

- For any given input m_1 it should be hard to find a second input m_2, such that $h(m_1) = h(m_2)$ (*second preimage resistance*). Note that this property is implied if collision resistance and preimage resistance are given.

One weaker requirement is that every output bit must depend on every input bit and for every input bit flip, every output bit has a probability $p = 0.5$ of changing. Internally, cryptographic hash functions use the avalanche effect (every bit in the computation influences many other bits) to create these dependencies.

In order to be practically useful, cryptographic hash functions should also

- be reasonably fast and

- require a reasonable amount of memory.

What is reasonable in this context depends on the available computing infrastructure and the required security. Furthermore these considerations change over time because of Moore's Law [116] and advances in cryptography.

Algorithm	Bytes processed per second
md5	$1.86 \cdot 10^8$
sha1	$1.69 \cdot 10^8$
rc4	$1.56 \cdot 10^8$
aes-128 cbc	$9.76 \cdot 10^7$
aes-256 cbc	$7.78 \cdot 10^7$

Table 2.1: Speed of cryptographic operations. Computed blocks of 8192 bytes of data on a AMD Opteron 248 Stepping 8 with 1024 KiB Cache running at 2.2 GHz. Implementation of the algorithms: OpenSSL 0.9.7e with standard SuSE 9.3 compiler flags.

Cryptographic hash functions have different intended usages. They can be used to implement integrity checks, because if the hashed content changes there is a very high probability that the hash value changes too (otherwise one would

have discovered a hash collision). Cryptographic hash functions are also a building block for many digital signature schemes (e. g. [41], [85], [4]):

Computing digital signatures on large amounts of data is not practical because of the large integer numbers involved in many signature algorithms. Computations with such numbers are relatively slow (see table 2.2).

The same holds true for verifying such signatures. On the other hand, hash functions are faster (see table 2.1). Therefore only a cryptographic hash over the message content is signed, and only the signature over the hash has to be verified.

Besides in signature algorithms (for integrity checks and authentication), cryptographic hash functions are used for message authentication codes (MAC). There, the hash function is computed over a secret k shared between Alice and Bob and the message m itself. For example [93] defines the Hash-MAC (HMAC):

$$\mathrm{HMAC}_k(m) = h\Bigg((k \oplus \texttt{0x5c0x5c0x5c} \cdots) \| h\Big((k \oplus \texttt{0x360x360x36} \cdots) \| m \Big) \Bigg)$$

Cryptographic hash functions can be constructed from encryption functions. Since current encryption functions are slower than current hash functions (see table 2.1), this approach is rarely used.

Algorithm	Signatures per second	Verifications per second
rsa 1024 bits	$9.87 \cdot 10^2$	$1.63 \cdot 10^5$
rsa 2048 bits	$1.68 \cdot 10^2$	$5.45 \cdot 10^3$
dsa 1024 bits	$1.96 \cdot 10^3$	$1.60 \cdot 10^3$
dsa 2048 bits	$6.16 \cdot 10^2$	$4.98 \cdot 10^2$

Table 2.2: Speed of asymmetric cryptographic operations. For details of parameters see table 2.1

2.1.3   Hash Function MD5

The message digest algorithm MD5[141] was designed to be a fast and secure cryptographic hash function. It was used widely for integrity checks and digital signatures.

The MD5 algorithm works on chunks of 512 bits. The input is padded (see section 2.4) so that the length of the input is evenly divisible by 512:

- A binary '1' is appended.

- Binary '0's are appended until the length is congruent to 448 modulo 512.

- The length of the input before padding is added as a 64 bit integer.

MD5 has an internal state of 128 bits. After this state is initialized, for every 512 bits of input, 64 rounds are executed. Each round is a function of the state, the input data and some internal constants. The function is changed every 16 rounds.

The cryptographic hash function MD5 must be considered broken, because it has been demonstrated[37, 170] that two meaningful messages can have the same MD5-value. Tables 2.3 and 2.4 show two different messages. Processed with MD5, both result in the hash value 0bcfc4ded8b9a153f8c59b7c19598138.

25215053	2d41646f	62652d31	2e300d0a
2525426f	756e6469	6e67426f	783a2030
20302036	31322037	39322020	20202020
20202020	20202020	20202020	200d0a28
e842a66a	de4d00e0	d5895ff8	e59cc1a7
$2fcab717$	0a467eaa	c003543e	b1fb7f08
456d3305	01fc53e0	5befa083	13237952
ed5a33ce	36990d9c	$076e45da$	528479eb
2fbd0f95	f557e576	3aecbfaa	0f0bd9ca
$b5735948$	32f47d0b	2cb9a376	d4361e20
fdef3b83	a1f27deb	ca361c53	$86586bc8$
f494f44e	611f7c84	$8060cfef$	94b50390

Table 2.3: Message 1 with MD5-Hash 0bcfc4ded8b9a153f8c59b7c19598138

25215053	2d41646f	62652d31	2e300d0a
2525426f	756e6469	6e67426f	783a2030
20302036	31322037	39322020	20202020
20202020	20202020	20202020	200d0a28
e842a66a	de4d00e0	d5895ff8	e59cc1a7
$2fcab797$	0a467eaa	c003543e	b1fb7f08
456d3305	01fc53e0	5befa083	$13a37852$
ed5a33ce	36990d9c	$076e455a$	528479eb
2fbd0f95	f557e576	3aecbfaa	0f0bd9ca
$b57359c8$	32f47d0b	2cb9a376	d4361e20
fdef3b83	a1f27deb	ca361c53	$86d86bc8$
f494f44e	611f7c84	$8060cf6f$	94b50390

Table 2.4: Message 2 with MD5-Hash 0bcfc4ded8b9a153f8c59b7c19598138

2.1.4 Hash Function SHA

Very similar to the MD5 hash function, SHA1 [121] is also based on the Merkle-Damgård-Construction [36]: A collision resistant compression function is used

to create a cryptographic hash function for arbitrary sized inputs. Since the design is similar, most weaknesses are shared.

Germany's Bundesnetzagentur has announced [11] that SHA1 is believed to be secure until 2009. The same announcement published that the successors of SHA1, SHA224 and SHA256 (up to SHA512) are believed to be secure until the end of 2012.

2.1.5   Hash Functions for authentication – Merkle Hashes

Cryptographic hash functions allow the integrity check of a data block: The hash value is transmitted over a trusted channel and the data block can be transfered in an untrusted manner. The receiver can than compute the hash function over the received data block and is convinced that the block has not been tampered with if the computed value matches the one received over the trusted channel.

This scheme has the disadvantage that the receiver can check the integrity only after the full block has been received. Such a check can be too late if the data block is large or if the application in question has a streaming character.

To overcome this problem, hash lists were introduced. The data block m is split into chunks $m_0, m_1, \ldots, m_n$. The hash function is applied to each chunk $h_i = h(m_i), i \in (0, n)$ and to the list of all hash values computed: $h_r = h(h_0|h_1|\ldots|h_n)$. This *root hash* h_r is transmitted over the trusted channel, the list itself does not need to be protected. The receiver can, given the hash list and the authenticated root hash, check the integrity of arbitrary chunks by comparing the hash of the chunk with the hash in the list and by computing the root hash over the list and comparing it with the one received over the trusted channel.

If the message size grows or the chunk size is decreased, hash lists grow and become impractical too. Hash trees as an extension to hash lists can solve this problem, even if they were invented to make Lamport-Signatures [97] more efficient [112]. Again, each chunk is hashed separately. In a tree with degree k, every k hash values are combined to a list and the hash is computed over this list. If the resulting list of intermediated hashes h_i has more than k entries, the procedure is repeated on this intermediate list. This process continues until the root hash h_r is found (see figure 2.1 for an example with $k = 2$).

2.1.6   Rabin hashes as rolling checksums

Originally invented as a tool for fast string search [83], Rabin hashes are a way of implementing a *rolling checksum*. The term rolling checksum refers to the fact that given the checksum over the bytes $n \ldots m$, it is relatively easy to compute the checksum over the bytes $n + 1 \ldots m + 1$. The Rabin checksum is given by

$$h(n, m) = c[n] * a^{m-n-1} + c[n+1] * a^{m-n-2}$$
$$+ c[n+2] * a^{m-n-3} \cdots + c[m] * a^0 \pmod{q}$$
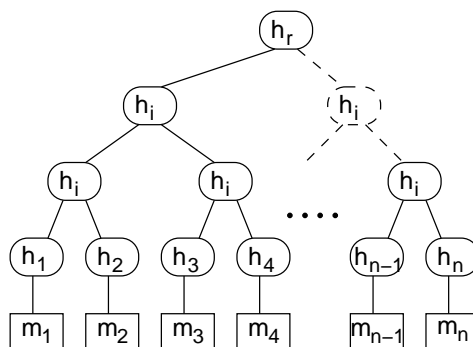
Figure 2.1: Merkle Hashes

where $c[n]$ is the byte at position n in the input. The parameters q and a influence the quality of the algorithm. In [99] good values for q and a are listed. To move one step forward, just

$$h(n+1, m+1) = h(n,m) * a - c[n] * a^{m-n} + c[m+1]$$

needs to be computed.

Rabin hashes, like CRC values, are not cryptographically secure, but have applications in pattern matching (see section 11.4).

2.2 Encryption and Decryption

There is a distinction between symmetric and asymmetric encryption algorithms. The distinctive property is that symmetric algorithms use the same key for encryption and decryption, where asymmetric algorithms use different key material for both operations.

Symmetric ciphers can be block ciphers and stream ciphers. Stream ciphers encrypt/decrypt each input byte independently. Block ciphers on the other hand operate on a number of bytes at once. It is easy to use a stream cipher as block cipher. Some cipher modes (see section 2.3) allow a block cipher to be turned into a stream cipher.

2.2.1 One Time Pad

The One Time Pad is a symmetric stream cipher. It works by XOR-ing every clear text bit P_i with a key bit k_i:

$$C_i = P_i \oplus k_i$$

If the key k is randomly selected and never used twice, this algorithm is provably secure [153]. The obvious disadvantage is that the key must have the same length as the plaintext. Furthermore the key must not be used twice.

2.2.2   Rijndael as Advanced Encryption Standard – AES

The block cipher AES [122, 34], also known as Rijndael after its Belgian designers Joan Daemen and Vincent Rijmen, has been selected as the encryption standard by the National Institute of Standards and Technology (NIST) of the United States of America. The standardization process was started in 1997 after the previous standard DES (Data Encryption Standard, standardized in 1981) was increasingly considered insecure. Weaknesses of DES include short keys (effectively 56 bits) and the existence of weak keys. Further progress in linear and differential crypto analysis showed success against DES. In the year 2001 Rijndael was announced as the new standard for symmetric block ciphers, because it is considered secure (NSA has approved the use of AES for top secret material) and reasonably fast both in hardware and software implementations.

Rijndael can work on any key and block sizes which are multiples of 32 between 128 and 256, but the AES standard prescribes that the block size is always 128 bits while the key size can be 128, 192 or 256 bits. AES is a substitution-permutation network. After generation of the round keys from the input key, the following round is executed 10 to 14 times, depending on the size of the input key.

AddRoundKeys All operations of AES work on a matrix of 4×4 octets. Every round key has a length of 128 bits that are arranged as 4×4 octets. In the AddRoundKeys step the current data matrix is combined with the round key by a XOR operation.

SubBytes This step substitutes every byte by means of a lookup in a substitution table (S-box). Here, substitution is a highly non-linear operation and the basis for the security of the algorithm.

ShiftRows The ShiftRows step shifts the n-th row by $n-1$ bytes, e.g. the first row is not touched and the 4th row is shifted 3 times. With this step, information is transported between columns, i.e. columns become dependent on each other during the next rounds.

MixColumns Similarly, the column mixing step creates interdependencies between rows. This step works by using the four bytes from one column as input to a function over Rijndael's Galois field $GF(2^8)$, which yield four output bytes that replace the original column.

However, some new weaknesses have been discovered. Implementations of the algorithm with a reduced number of rounds can be broken [53]. Further, AES can be described in closed algebraic form [54]. This is a new property of block ciphers, and whether this will lead to real attacks is still unknown. Despite these weaknesses, the algorithm is in widespread use since the standardization as AES.

2.2.3 RC4

Ron's Code Number 4 (after Ronald L. Rivest), also called RC4, is a stream cipher. That means the cipher itself will output a sequence of octets that are XOR-combined with the cleartext to form the ciphertext. The decryption operation is analogous.

The algorithm works in two steps, initialization and usage. The first step is the initialization of a 256-octet working area s. At starting time, this working array contains all octets from 0 to 255 in this sequence (i.e. $s_i = i, i \in (0, 255)$). Further, an array k of the same size contains the key (if the key is shorter than 256 octets, it is repeated). Then, a variable j is initialized to 0 and for $i = 0 \ldots 255$ the two operations

1. $j = j + s_i + k_i \pmod{256}$

2. Exchange s_i and s_j

are executed. The second step is the usage of the array s to produce the key stream. For every byte of the key stream the following operations are required:

1. $i = i + 1 \qquad j = j + s_i \pmod{256}$

2. Exchange s_i and s_j

3. $k = s_i + s_j \pmod{256}$

The generated key stream is the sequence of s_k.

RC4 excels by its simplicity and speed. However, there exists some weak keys and further attacks on the algorithm. A prominent usage of RC4 is probably the encryption in the IEEE 802.11 standard (WEP/WPA[128]).

2.3 Modes of Block Ciphers

Block ciphers as described above work on a single block of data.

$$C = E_k(P) \qquad P = D_k(C)$$

where P and C are blocks of plain- and ciphertext and E_k and D_k are encryption and decryption algorithms, respectively. The key k is identical to encryption and decryption (symmetric cipher). Both P and C usually have the same length. C should not be larger than P, otherwise the cipher would become impractical for many applications. On the other hand, C can not be smaller than P because the cipher must be invertible. The size of such blocks is typically in the order of 128 bits (e.g. DES uses 64 bits, Rijndael, which is a super set of AES, supports 128, 192 and 256 bits). A block size of 64 bits is unsuitable for many applications today, since there are only 2^{64} possible blocks and because of the birthday paradox, a collision can be expected with a probability 0.5 after 2^{32} blocks, even if more advanced operation modes (see below) are used.

Since the overall input data size can be larger than the block length of the cipher, the input data is split into multiple blocks. The last block may be padded (see section 2.4). If multiple blocks have to be encrypted, there can be different operation modes how encryption of one block influences the other. Such operation modes differ in their complexity to implement, attack possibilities, susceptibility to bit errors, how they can be executed in parallel and other factors.

2.3.1   Electronic Codebook Mode

Electronic codebook mode (ECB) is the simplest of all modes, because different blocks do not influence each other at all. Each block is encrypted separately:

$$C_i = E_k(P_i) \qquad P_i = D_k(C_i)$$

This mode is very easy to implement. The execution can be completely parallel for all blocks, both for encryption and decryption. Bit errors in the ciphertext (e. g. because of transmission errors) will only influence one block of the clear text after decryption.

Figure 2.2: Unencrypted bitmap

Since each block is treated completely separately, the same input block is always encrypted in the same way (given the same key is used). This is not alway desirable, as figure 2.3 shows. The image is the ECB-encrypted version of figure 2.2 shown before[1]. Even if AES256 is a strong encryption algorithm, the content of the image is clearly perceivable.

As figure 2.4 shows, other modes [2]do not show this weakness. Of course the inverse conclusion is not true: The fact that the image is not directly perceivable in figure 2.4 does not say much about the strength of the encryption.

---

[1]The uncompressed input image with 8 bit color depth was encrypted using AES256 with the randomly selected key `fc7d bd38 5b1a 57bf 9cca 447b df0a ee13 4638 8b23 1a2f d7aa 2963 1b7d b271 5584`.

[2]Encrypted with AES256 in CBC mode (see section 2.3.2) with same key as figure 2.3 and a initialization vector 0.

Figure 2.3: Bitmap from figure 2.2, encrypted with AES256 in ECB mode

2.3.2 Cipher Block Chaining

To overcome the weaknesses mentioned above for the Electronic Codebook mode, Cipher Block Chaining links the result of one block's encryption with the next:

$$C_i = E_k(P_i \oplus C_{i-1}) \qquad P_i = D_k(C_i) \oplus C_{i-1}$$

Different to ECB, every single bit error distorts all blocks starting from the block containing the bit error. This can be seen as an advantage, because transmission errors (caused by noise or by an adversary) are clearly visible. Encryption in CBC can not be parallelized, because handling block i requires that the treatment of block $i-1$ is complete. The decryption operation can start as soon as two consecutive cipher blocks are available.

To encrypt the first block P_1, the value C_0 is necessary. This value, also called Initialization Vector IV, is an additional input parameter to the algorithm. The initialization vector should be chosen at random for every message to ensure that even if the same key is used, the cipher texts differ. The IV C_0 must be known at decryption time because P_1 depends on it.
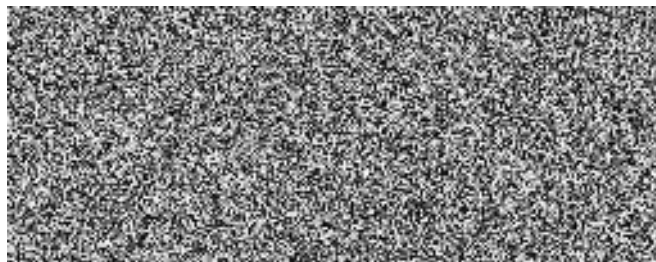
Figure 2.4: Bitmap from figure 2.2, encrypted with AES256 in CBC mode

2.3.3   Feedback Modes

Both Cipher Feedback Mode (CFB) as well as Output Feedback Mode (OFB) effectively turn block ciphers into stream ciphers. The advantage of this approach is that the last block of input data does not need to be padded to the block length of the algorithm.

In cipher feedback mode, the plaintext is xor-ed with the encrypted previous cipher block.

$$C_i = E_k(C_{i-1}) \oplus P_i \qquad P_i = E_k(C_{i-1}) \oplus C_i$$

Note that for both operations the encryption direction of the block cipher is used. CFB has the nice property that it is self-synchronizing. That means that with any two consecutive blocks of ciphertext the decryption of the second block is possible. This makes CFB suitable for continuous data streams where decryption must be possible even without listening since the beginning of the stream. The encryption operation can only be done sequentially, whereas the decryption can be parallelized.

The Output Feedback Mode (OFB) on the other hand does not use the plain text but computes a stream cipher. The stream cipher for OFB is given by:

$$O_i = E_k(O_{i-1})$$

With that stream encryption and decryption is the XOR with plain- and ciphertext respectively:

$$C_i = O_i \oplus P_i \qquad P_i = O_i \oplus C_i$$

Again, O_0 acts as initialization vector. The other values of O_i can be precomputed, so the actual encryption/decryption is only the XOR-operation.

2.3.4   Counter Mode

Counter mode is called counter mode because a nonce value is increased by one for every block encryption. Nonce values are number used only once and are written as N_{once} from time to time. Sometimes counter mode is also called Segmented Integer Counter (SIC) mode. In theory, every non-repeating sequence of nonce values would be sufficient. In practice, a counter is used because of its simplicity for many applications:

$$C_i = E_k(N_{once} + i) \oplus P_i \qquad P_i = E_k(N_{once} + i) \oplus C_i$$

Similar to the feedback modes described above, counter mode effectively turns a block cipher into a stream cipher (with the notation from above, here $O_i = E_k(N_{once} + i)$). This has the advantage that the last block does not need to be padded to multiples of the cipher block length.

Again, the block cipher is used in the encryption direction only.

To be effective, the value of N_{once} must be chosen in a way not predictable by an attacker. Further it is important that the nonce values are never reused,

otherwise the scheme is compromised: If the sender transmits $C_1 = E_k(N_{once}) \oplus P_1$ and $C_2 = E_k(N_{once}) \oplus P_2$ and the attacker computes $C_1 \oplus C_2 = E_k(N_{once}) \oplus P_1 \oplus E_k(N_{once}) \oplus P_2 = P_1 \oplus P_2$. If the attacker further knows P_1, he can compute P_2 easily.

Another important advantage of counter mode is the random access property. Each block of the ciphertext can be decrypted independently, which allows easy parallelism. Together that makes applications like disk block encryption easy to implement.

Given the constrains above, the security of counter mode is believed to be equivalent to the security of the underlying block cipher (see for example [111], which also shows how counter mode can be used for authentication).

2.4 Padding

In general, padding is the process of adding symbols to the clear text. This can serve a number of purposes. First, many plain-texts start or end with predicatable sequences of symbols. For example, documents in the Adobe Portable Document Format PDF contain always the sequence `0x25 0x50 0x44 0x46 0x2d` (the string "%PDF-") at the beginning of the file or PostScript files often end with `0x25 0x25 0x45 0x4f 0x46 0x0a` (the string "%%EOF<line end>"). Since in such cases the plaintext is know to a potential attacker, known plaintext attacks can be mounted. However, todays ciphers are believed immune to such attacks so this kind of padding is not common.

Some block ciphers or block cipher operation modes require that the length of the clear text is evenly dividable by the block length of the cipher. There are methods to circumvent this (e.g. ciphertext stealing [151, 5] or residual block termination [151]).

Both methods add an significant amount of complexity. If such a complexity increase cannot be justified and the application can deal with the fact that the ciphertext is a little bit longer than the clear text, padding can be used. Here, padding is the process of adding information the the clear text so that the length of the clear text becomes evenly dividable by the block length.

Many cryptographic hashing functions (see section 2.1.2) also require the input data to be a multiple of an internal block length and use padding to achieve this.

One easy way to do it is to extend the clear text with $N, N, N, \ldots$, where N is the number of padding octets. That also means that this padding method is only applicable if the padding length is a whole number of bytes. To do it in a unambiguous manner, the padding is always added, even if is not necessary (if the input data length is already evenly dividable by the cipher length). That means, for a cipher with block length 64 bit there are exactly eight different paddings possible: `0x01`, `0x02 0x02`, `0x03 0x03 0x03`, ..., `0x08 0x08 0x08 0x08 0x08 0x08 0x08 0x08`. This kind of padding is for example suggested in [76], [67], [78] and [77].

One other way to pad clear text blocks is to first append one 1 bit, then

a sequence of 0 bits. This padding method can be applied to messages whose
length is not a whole number of octets and is recommended by [44].

There is a similar alternative that includes the total length of the clear text.
For that, again one 1 bit is followed by a number of 0 bits until the condition

$$\text{length of input} + \text{length of padding} + \text{length of length field} \equiv \text{block length}$$

is true. Then the length of the input data before padding is appended. For
example [141] uses this type of padding with a 64 bit length field.

Table 2.5 shows applications of the last three padding schemes.

block length	8	16
Original	`CAFEBABE`	`CAFEBABE`
RFC2315[77]	`CAFEBABE04040404`	`CAFEBABE0C0C` ($8\times$`0C`) `0C0C`
NIST SP-800-38A[44]	`CAFEBABE80000000`	`CAFEBABE8000` ($8\times$`00`) `0000`
RFC1321[141]	`CAFEBABE80000004`	`CAFEBABE8000` ($8\times$`00`) `000C`

Table 2.5: Padding Examples

2.5   Summary

This chapter introduced important concepts from the field of cryptography,
encryption, decryption and hashing. These will be important building blocks
in later chapters. Also some non-cryptographic algorithms were presented, e. g.
cyclic redundancy checks. The next chapter will deal with the foundations of
overlay networks and peer-to-peer networks in particular.

Chapter 3

Overlay Networks

It is always possible to add another level of indirection.
Ross Callon

Computer networks connect a set of communicating computers or devices so that the participating nodes can exchange messages over the established links. The topology of the network is determined by the physical links (wired or wireless). Overlay networks are virtual networks which are built on top of other networks. As such, overlay networks can choose their own topology. Probably the most prominent example for overlay networks is the Internet itself, which was initially deployed to a large extend on top of the plain old telephone network (POTN).

Since overlay networks as considered in this and the following chapters create a their connections with the help of the transport layer from the OSI/ISO standard model[178][71], they conceptually reside on top of OSI layer 4.

This chapter first covers peer-to-peer networks, which are a related, but not identical, concept. Then, the idea of structured overlay networks is introduced together with key based routing. The chapter concludes with an important application for key based routing systems, distributed hash tables, and optimization opportunities for overlay networks.

3.1 Peer-to-Peer Networks

The term peer-to-peer is used for interaction among equals. In computer science, it is the opposite of the client-server approach. Client-Server communication is characterized by a clear distinction of roles between a requestor (client) and responder (server). On the other hand, in peer-to-peer communication systems, all participants can take on (at least in theory) all responsibilities. In reality, many systems are neither pure peer-to-peer systems nor pure client-server approaches, but are based on both concepts and called *hybrid* systems.

The goal of peer-to-peer systems is the sharing of resources. Resources in this context can be network connectivity, storage space or computing power, sometimes also the generation of content or human presence and attention. For examples see [158]. This sharing of resources makes peer-to-peer networks *distributed*.

Another important aspect of peer-to-peer networks is *decentralization*. That means the system tries to avoid a single central component, whose failure would be fatal for the system. Decentralized peer-to-peer systems try to avoid such single point of failures.

The next subsections will show examples of peer-to-peer systems, one more hybrid and one more pure peer-to-peer.

3.1.1   BitTorrent

The distribution of (large) files between users was one of the main drivers to increase popularity of peer-to-peer systems in the last decade. In such systems, bandwidth and storage capacity are the shared resources. In centralized approaches for data distribution both can become a limit.

BitTorrent [21] is one of the more hybrid approaches. A BitTorrent system consists of a tracker, an initial seeder and potentially many downloaders. The initial seeder is the publisher of the file. It splits the file in parts and computes a cryptographic hash over each part. These hashes are used for identifying the data blocks and later for integrity-checks after download. The initial seeder compiles all these hashes and the address of the tracker into a so called *torrent-file*. Such a torrent-file is distributed separately from the publisher to the downloaders. Traditional methods like a HTTP-server are used regularly, since the torrent-file is small compared to the to be transfered file. Then the initial seeder announces to the tracker that it has all the blocks of the file.

In order to fetch the file, the downloader first needs the torrent-file. As mentioned above, web-servers are a common source for them. In the torrent-file, the downloader finds identifiers of all blocks of the file as well as the address of the tracker. From the tracker the downloader gets the address of the initial seeder. As soon as the first blocks are transfered from the initial seeder to the first downloader, the first downloader informs the tracker of the successful download and the new source. This way, successive downloaders can fetch blocks either from the initial seeder or the first downloader. By implementing a rarest-block-first strategy, BitTorrent ensures that the load on the initial seed is quickly distributed over all downloaders.

The tracker component in the BitTorrent system is a centralized component, since all downloaders need the tracker to find the seeder and each other. This centralized component is a single point of failure: If communication to the tracker is not possible (for whatever reason), the BitTorrent system ceases to work. Later implementations of the BitTorrent protocol do not have the requirement of the centralized tracker. Instead of the tracker, they use a distributed hash table (see section 3.3) based on Kademlia (see section 3.2.1).

3.1.2 Freenet

Freenet [19] is a peer-to-peer network for distributed data storage and access. One major design goal was to have the network as anonymous as possible and make censorship of content in the network as hard as possible.
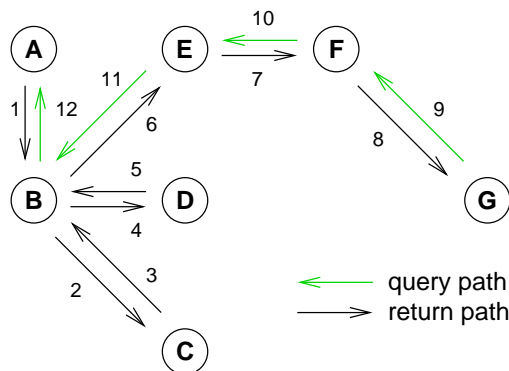
Figure 3.1: Routing in Freenet

The two main operations of Freenet are the storage (put) and the retrieval (get) of single data items. Both work by first assigning each data item a unique identifier. The Freenet node generating a get request chooses among its neighbors which one is the most likely to serve the request. Then the request is forwarded to this node, which does the same recursively, again evaluating the likelihood that the next hop has the requested data[1]. At each step, every node inserts itself as the originator of the request with a given probability. That makes it impossible for the receiver of a request to be sure where the request originally came from and creates the anonymity properties of Freenet.

If a node does not have any possibility left for forwarding requests (either because such forwarding would create loops or because such forwarding would decrease the probability of success), the request is returned. Upon such a non-successful return of a request, a node forwards the request to the second best neighbor. In effect, the network is searched as a tree in a depth-first manner (see figure 3.1).

Once a data item is found, it is returned on the same path the request took. Again, each hop on the path has a probability to set itself as the source of the item and preserves by that the anonymity of the real origin. The protection of the origin of data is an important part in the censorship resistant design of Freenet. To further strengthen censorship and denial-of-service resistance, intermediate nodes on the path from data origin to requestor save a copy of the item. Once a data item request returns successfully, the probabilities for subsequent similar requests to take this path are increased.

---

[1]Since version 0.5 of Freenet not only the probability of success is evaluated but also the estimated time until success. This is a form of proximity route selection, see section 3.4.2.

Storage requests are routed in similar manner, again with a probability that intermediate nodes disguise the identities of the publisher and the storage nodes.

For the request routing and forwarding process it is important that data items are identified in a unambiguous way. Freenet provides at least three ways to do so:

- For *Keyword Signed Keys* (KSKs), the publisher describes the data item with a descriptive text. From this text, an asymmetric key is computed in a deterministic manner. The public part serves as identifier for the data item, the secret part is used to sign the data. Note that since the key is produced in a deterministic way, such a signature provides integrity protection but no authentication. As such, the security of KSKs is pretty weak, but they are used nevertheless. The content of the data item does not influence the generation of the identifier, so it is possible for an attacker Eve to modify the data.

- *Content Hash Keys* (CHKs) are used for the bulk of data transported over Freenet. These identifiers are computed as a cryptographic hash function over the content of the data block. By this method, Eve the attacker can not modify data associated with a CHK without Bob noticing it. Further, the data block is encrypted with a symmetric cipher. This encryption ensures that intermediate nodes can not read the data they transmit. In fact, these intermediate nodes can *plausibly deny* that they have knowledge about the data. The publisher and the requestor of the data item share knowledge about the cryptographic hash as well as about the symmetric key.

- Since the required transportation of the symmetric key from publisher to reader is a disadvantage of CHKs, *Signed Subspace Keys* (SSKs) were introduced. First, the publisher randomly creates an asymmetric key pair. Then the identifier of a data block is computed as $ID = h(\text{public key}) \oplus h(\text{text})$. The text can be arbitrarily chosen by the publisher. In order to check the integrity of the stored data, the data block is signed with the private key of the publisher. Since asymmetric cryptographic is more expensive than symmetric one, SSKs are often used as a forwarding mechanism to CHKs. For the descriptive text some conventions have been established, e. g. the text can include a version number of the data item or a time stamp when the data has been published. Both conventions allow a receiver to deduce future description strings.

Freenet is designed to be anonymous and censorship resistant, but it does not conceal the identities (network (i. e. IP-) addresses) of nodes running Freenet nor the fact that they running Freenet. This made Freenet susceptible to scanning and banning. Since version 0.7 Freenet tries to address this problem by changing the way new connections are established. Previously, nodes could be queried for connections to other nodes. Newer versions create new connections only after a human has introduced the new peer. With this strategy, Freenet is modeled

after the human trust relationships network. This network is a *Small-World*-network and [145] showed that routing in such networks is possible. Since it can be difficult to detect and enter such a network, they are sometimes called *darknets*, a term allegedly introduced by [7].

3.2 Structured Overlay Networks and Key Based Routing

The peer-to-peer overlay networks just described do not have a special structure in their connection topology (besides all BitTorrent nodes connecting to a central tracker). The established links between peers are created on demand. Therefore these networks belong to the class of unstructured overlay networks. Structured overlay networks on the other hand use the freedom to choose their own topology to their advantage. The idea is to create invariants that are helpful for the tasks at hand. One task for overlay networks is to forward messages.

Key based routing systems are networks specialized in forwarding messages. In general, this works as follows:

1. Each node in the network is assigned a unique identifier.

2. Each message has a destination identifier from the same identifier space. Typically, the number space where node identifiers and message destinations are taken from is large, e.g. 2^{128} or 2^{256}.

3. A metric $d = f(\text{ID}_1, \text{ID}_2)$ is defined.

4. The messages are routed until $d(\text{ID}_{\text{node}}, \text{ID}_{\text{message}})$ becomes minimal.

The topology of the network is chosen in a way that facilitates finding the minimum in a fast, efficient and guaranteed manner. The process of routing can take place in two different manners:

Recursive routing forwards the message hop by hop from sender (S) over intermediate nodes (I) to the final receiver (R). Each hop evaluates the destination of the message and takes an appropriate forwarding decision. Figure 3.2 shows the concept.

Iterative routing on the other hand does not forward the message hop by hop. The sender of the message computes the next hop and queries this next hop for the target of the message. The next hop replies with an even better next hop, which is queried again (thin arrows in figure 3.3). This process of iterative querying continues until no better next hop is found. The message itself traverses only this single hop (bold arrow).

The ideas behind such strongly structured networks are not recent. For example [148] explores the usage of de Bruijn-graphs for message routing in multi-processor machines. Later, the idea was picked up for distributed and decentralized systems [40, 104, 132, 81].

Figure 3.2: Recursive Key Based Routing
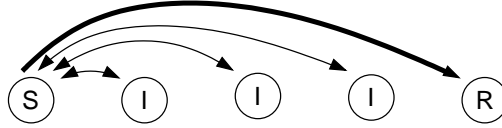
Figure 3.3: Iterative Key Based Routing

Since there is no specialized central component in such networks, they work
in a peer-to-peer fashion. However, not all key based routing schemes are nec-
essarily overlay networks, as [12] and [59] indicate. They showed that key based
routing schemes can also be used as normal routing/forwarding protocols at the
network layer.

Figure 3.4: Message Forwarding Protocols

Traditionally, message forwarding protocols are assessed in terms of required
state. Flooding (i. e. forwarding all messages to all nodes) requires $O(1)$ state,
but on the other hand flooding imposes high load on the network. Topology
based protocols need to consider the full topology of the network ($O(N)$), but
message forwarding itself is much more efficient. There are approaches to reduce
the amount of required state or required flooding by introducing hierarchies in
the network. These hierarchies are either in the topology itself (mostly trees),
or in the address space (like the subnets in IP-networks). Both approaches,
flooding and topology based, have in common that messages travel on a very
short path, often the shortest possible path. Figure 3.4 shows the tradeoff:
Either a small state and a high network load or large state and lower network
load. In any case, the shortest path is found.

On the other hand, key based routing protocols forward messages via in-
termediate hops. These intermediate hops are not necessarily on the shortest
path, but the key based routing protocols have the advantage that they neither
need flooding nor a large state on the forwarding nodes. Figure 3.5 shows how
key based routing protocols open up a new dimension in the design space for
message forwarding protocols. While giving up the shortest path property, key
based routing protocols are able to route efficiently even with a small local state.
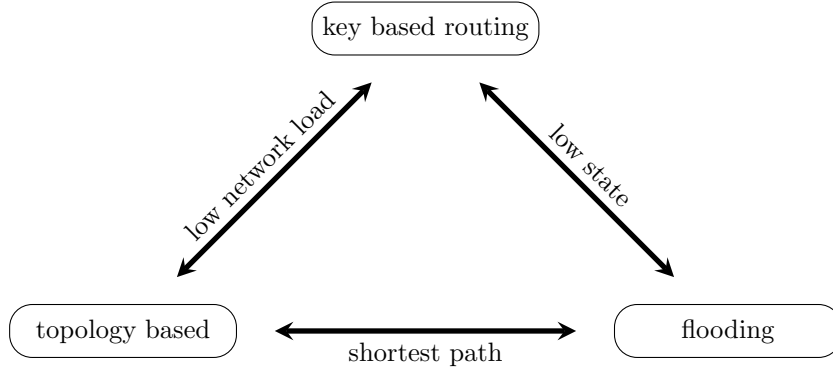
Figure 3.5: Message Forwarding Protocols and Key Based Routing

Key based overlay routing protocols themselves can also be assessed in terms of required state and the message overhead they create. Figure 3.6 shows some popular protocols and how they relate in terms of scalability ([158, 137, 160, 144, 148]).

The following sections describe some selected protocols in more detail. Others have been left out (e.g. [106] and [101]), either because the system is very similar to a presented one or the system lacks practical relevance.
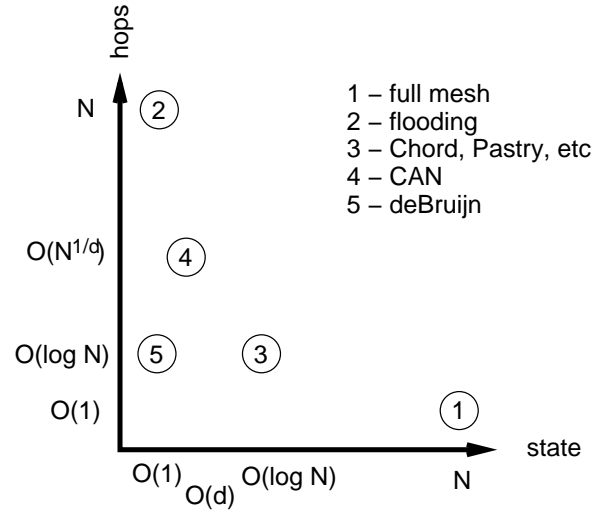
Figure 3.6: Scalability of Overlay Networks

3.2.1   Kademlia

Kademlia [110] is a key based routing protocol and uses XOR as the metric, i.e.
the distance of two identifiers is given by $d(a,b) = a \oplus b$. The protocol message
`FindNode` contains an identifier of the searched-for-target t and is sent to the
node i where $d(t,i)$ becomes minimal. This node may have knowledge of other
nodes even closer to t, and returns node identifiers and transport addresses
(i.e. Kademlia IDs as well as IP address and UDP port number). This process
continues iteratively until no node closer to the target is found. The XOR metric
has the useful property that it is commutative $(d(a,b) = d(b,a))$. In this way,
nodes quickly learn about their neighborhood: If A is close to B and sends B a
message, B can learn the existence of A from that message. In order to guarantee
the convergence of the lookup process, each node has to keep information about
$O(k \log_2 N)$ specific other nodes, where N is the number of nodes in the network
and k is a design parameter. This routing table is divided into buckets by the
Kademlia specification. If the distance $d(A,B)$ of two nodes A and B has a
prefix of i digits of zero, A and B store each other in the ith bucket. Each
bucket has a size of up to k entries, hence they are called k-buckets.

On top of the simple `FindNode`-procedure Kademlia uses the messages `Store`
and `FindValue` to implement a distributed hash table (see section 3.3). A fourth
message, `Ping`, is used to check the liveliness of nodes.

On startup (see section 3.5.2), a Kademlia node performs a lookup for its own
identifier. This lookup has the effect of informing other nodes of the existence
of the new node. In particular, the node with the ID closest to the ID of the
new node is informed too.

3.2.2   Pastry

Pastry [144] uses a similar metric as Kademlia. It also tries to increase the prefix
match between node identifier and to-be-searched-for key. To do so, it employs
a more strict routing table than Kademlia. First, every m bits of the identifiers
are grouped into digits. The routing table has 2^m columns and $\frac{n}{2^m}$ rows, if n
is the number of bits per identifier (usually larger than 100 bits). Table 3.1
shows a much smaller table as example. To insert new nodes in the routing
table, first the largest common prefix is computed. The number of common
digits gives the row of the routing table, the first non-equal digit the column. A
lookup for a next hop of a message works similarly: First the largest common
prefix between the current node and the target identifier is determined. The
quantity of matching digits specifies the table row, the first non-matching digit
the column. If node A uses row i for lookup and forwards a message to node B,
node B uses at least row $i + 1$, otherwise A would not have sent the message
to B. In this way, the lookup mechanism is guaranteed to be deterministic and
bounded.

Even if the routing table contains $\frac{n}{2^m}$ rows, the forwarding process is ex-
pected to complete after $O(\log_{2^m}(N))$ steps. The reason behind this is that
many of the lower routing table entries are empty, because the number of nodes

	0	1	2	3	4	5	6	7
1	0xxxx	1xxxx	—	3xxxx	4xxxx	5xxxx	6xxxx	7xxxx
2	20xxx	21xxx	22xxx	23xxx	24xxx	—	26xxx	27xxx
3	250xx	251xx	252xx	—	254xx	255xx	256xx	257xx
4	2530x	2531x	2532x	2533x	2534x	2535x	2536x	—
5	25370	25371	25372	25373	—	25375	25376	25377

Table 3.1: Example Pastry Routing Table for a node with identifier 25374, assuming $m = 3$ and 15 bit identifiers. For brevity, octal numbers have been used.

N in the network is expected to be much lower than 2^n.

3.2.3 Chord

The metric in Chord [160] is a little bit different to that in Kademlia and Pastry. Chord assumes a circular address space and uses subtraction as metric: $d(a,b) = b - a \pmod{2^n}$. Again, n is the number of bits in each identifier. Contrary to XOR, this metric is not commutative. Every node x tries to establish a routing table entry to the nodes $x + 2^i$ with $i \in (0, n]$.

Again, for small i the entry is likely to be empty. Overall, there are $O(\log_2 N)$ entries expected in the table. The routing table in Chord is called the finger table, because if one draws the circular address space together with the routing table entries of one node, these entries are spread like the fingers of a hand. Note that each node knows its two direct neighbors on the address ring.

Message forwarding in Chord works by using the "longest" finger, i.e. the finger just before the target. This way it is ensured that the forwarding process terminates at the node closest (according to the metric) to the target. The expected number of hops is $O(\log_2 N)$, because on average, each hop halves the distance to the destination.

3.2.4 Content Addressable Network – CAN

With CAN (Content Addressable Network) [137], the address space is not thought of as a ring as with Chord, but as a d-dimensional torus[2]. To do so, each identifier (node identifiers as well as message identifierss) is split into d components.

Message forwarding works in the same way as with geographic routing in ad-hoc routing protocols [82]. The next hop is selected based on the euclidian metric. If a_i are the d components (i.e. $i \in (1, d)$) of the identifier of node a, then the distance d between a and b is given by

$$d(a,b) = \sqrt{(a_1 - b_1)^2 + (a_2 - b_2)^2 + \cdots + (a_d - b_d)^2}$$

---

[2]The number of dimensions d is a design parameter.

Because of the d dimensions, there are on average $N^{1/d}$ nodes per dimension and the expected number of hops to the destination are $O(d \cdot N^{1/d})$. For the routing to work, it is necessary that each node in the network knows its direct neighbors in all dimensions. This number can be larger than $2d$ in case nodes are not evenly distributed.

To establish these connections, a new node A first does a lookup of its own position in the torus. This lookup will yield a node B, which will be queried for its neighbors. The area of responsibility, in CAN called zone, will be split between A and B and all the neighbors will be notified of this event. By choosing d to be $d = \log N$, the performance characteristics of CAN can be brought to the hop count and state amount of Chord and Kademlia: $O(\log N)$.

3.2.5   de Bruijn Networks

Overlay networks based on de Bruijn-graphs [148] have a constant node degree, i.e. the state at every node is in the order of $O(1)$. The node identifiers are made of k letters from an alphabet with size m. Each node selects m outgoing (i.e. the graph is directed) connections, one for each symbol of the alphabet. The next node is selected by shifting one letter out at the left hand side of the node identifier and shifting one new symbol in at the right hand side. Figure 3.7 shows such a network with $m = 2$ and $k = 3$. Note that some edges are reflexive by construction.

Routing in such a network works by taking the node identifier of the message origin and the routing key for the destination and shifting out the symbols of the source and shifting in the symbols of the destination. Each of these shift operations traverses one edge in the network. After at most m shift operations, the destination is reached.
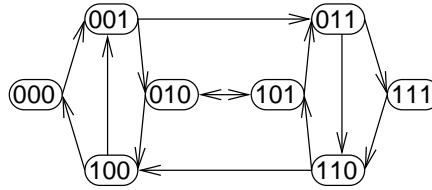
Figure 3.7: de Bruijn-graph with the alphabet $\{0, 1\}$ and node identifiers with three letters

In its original design, de Bruijn graphs exists only for exactly m^k nodes in a network. However it is possible to weaken this condition [163, 164].

3.3   Distributed Hash Tables

As hash tables are an efficient data structure to associate keys with values on a local computer, distributed hash tables are an efficient way to do so in

distributed systems. Here, some of the key-value associations are stored on every participating machine.

Key based routing systems are a very suitable substrate to implement the functionality of a hash table on top of the distributed key based routing network. Basically, this works with two types of messages:

Put This is the message to store a key-value pair in the system. Sometimes the message is also called *store* or *publish*. There is either no answer at all or an acknowledgment (possibly with a handle to later modify or delete the entry). The routing key (i.e. the destination) of the message is the key of the key-value pair.

Get The routing of this message works in the same way as the routing of the put message, which is the main idea of distributed hash tables on top of key based routing systems. The node receiving the get message replies with the value stored by a previous put message.

Possibly other messages are implemented to query properties of stored elements without retrieving them or to modify and delete entries.
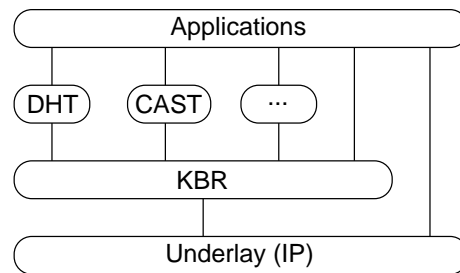
Figure 3.8: Relation of Key Based Routing and Distributed Hash Tables

On the other hand, key based routing systems can be seen as a distributed hash table themselves. Key based routing systems map routing keys to nodes (e.g. the nodes IP-address). Communication can thereafter occur directly. For both ways it is important that the allocation of objects to nodes does not change rapidly when nodes join or leave the system. This requirement can be satisfied with consistent hashing [81].

Figure 3.8 shows the former way to construct a distributed hash table on top of a key based routing system. Of course applications can access the network layer as usual. Furthermore, the key based routing system offers a service to the application, i.e. applications can send messages targeted at given routing keys and these messages are delivered to the node with the identifier closest to the key. On top of the key based routing layer, services like distributed hash tables can be implemented. [33] mentions multicast, anycast (CAST in figure 3.8), publish/subscribe systems and distributed object location and referencing [65] as other examples. These higher level services can then be used by the application.

3.3.1 Bamboo and other DHT Services

Bamboo [139] is a DHT implemented on top of the ideas from Pastry [144] with the help of [171]. However, it is optimized to work well under churn, i. e. it is more stable if many nodes join and leave per unit of time.

Bamboo is deployed on PlanetLab (see section 8.1.1) as OpenDHT [140] and running there continuously. It implements some basic security measures against malicious deletion of key-value pairs: During store operations, the cryptographic hash of a secret is attached. Later, a delete operation is only executed if the secret is revealed.

Many other distributed hash table concepts have been proposed, e. g. [32], [101] and [107]. Kato et. al compares different designs and implementations with special attention to return rates of get queries, reply delay and required network bandwidth [84]. These parameters are tested with static networks as well as dynamic ones. Furthermore it is checked how the distributed hash table reacts to multiple put requests with identical keys but different values. Additionally the scalability of designs and implementations is evaluated.

3.4 Optimizing Overlay Networks

Overlay networks as described in [160, 137, 144] and others in their basic form do not consider that connections in the underlying network differ in their properties. However, there are big disparities in latency, reliability, bandwidth and other parameters among network connections. The parameters may vary over time and direction, e. g. the bandwidth of a consumer link is asymmetric and latency depends on network usage and route selection. Since the network properties are not considered, "bad" connections occur are as often as "good" connections.
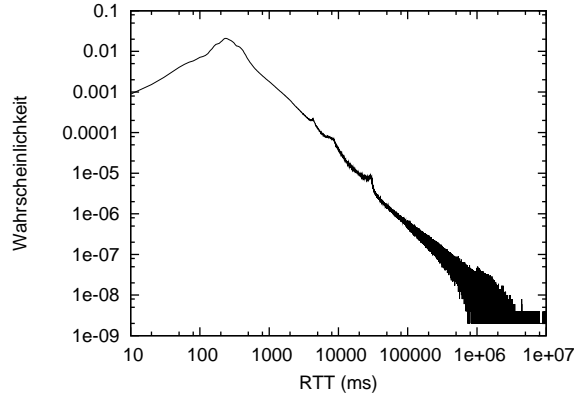
Figure 3.9: Round trip time distribution of replies in Overnet

Figure 3.9 from [96] shows that the distribution of round trip times in the

measured network[3] follows a power-law distribution over many orders of magnitude. This shows that "bad" connections are not a rare event. An overlay network without optimizations would choose connections from such a distribution at random, because it does not consider latency during connection establishment. Even if power-law distributions have a small median value, the mean value of random draws can be surprisingly large[4] because of the "heavy tail" [125]. Since overlay networks have to some extend the flexibility to choose their connections and routing decisions, they can optimize the performance even in the face of uneven and changing conditions.

3.4.1 Degrees of freedom

The key based routing systems and distributed hash tables described above have some degrees of freedom on

- how to select neighbors

- how to forward messages once neighbors have been selected.

For example, Pastry (see section 3.2.2) has great freedom on which neighbors are used to fill the upper columns of the routing table. Table 3.1 shows this freedom marked with an 'x'. As a second example, Chord has the liberty to choose any node from the finger table as next hop as long as the distance to the destination is decreased (i. e. there is no jump over the target). Many of these degrees of freedom and how they impact the performance and stability of routing in different topologies are studied in [62].

3.4.2 Proximity Route Selection

Proximity Route Selection (PRS) can be employed everywhere the forwarding rules in overlay networks allow to choose from more than one entry in the routing table For example Chord and Kademlia have this kind of freedom, whereas de Bruijn networks and Pastry do not. From the set of all possible next hops PRS first selects all next hops allowed by the routing metric, i. e. all next hops that decrease the distance to the destination in the identifier space. All next hops in this set are then evaluated using a second metric, e. g. latency. The message is then forwarded to the next hop where the second metric is optimal.

Optimizing overlay networks for latency is shown to have a significant effect on average end-to-end latency [62]. This is a useful goal for many overlay applications, especially if user interactions are involved. However, latency is not the only possible metric, other metrics are possible as well, e. g. bandwidth, reliability, trust or uptime. There is an interaction between bandwidth and

---

[3]The experiment measured the Overnet overlay network for two weeks during July 2004. During that time, the Overnet had between $2 \cdot 10^5$ and $2.65 \cdot 10^5$ participants. The experiment collected over $4.82 \cdot 10^9$ round trip time measurements. A round trip time in this measured as the time it takes to receive an application layer answer.

[4]The mean value for round trip times the Overnet experiment was more than 5 seconds, where as the median value was $4 \cdot 10^{-1}$ seconds

latency: If the bandwidth is saturated, the latency can increase because of filled buffers in the responsible routers.

Employing PRS has disadvantages too. First, there is no guarantee that the *overall* performance is enhanced. Second, the number of necessary hops to the destination will likely increase. With this increase, the reliability of the overall transfer can decrease. Advanced PRS schemes therefore strive to balance improvements in the identifier space metric with improvements in the performance metric.

3.4.3   Proximity Neighbor Selection

As seen in the previous section, PRS tries to optimize the use of connection after these connections have been established. Complementary to PRS, *Proximity Neighbor Selection* (PNS) tries to optimize the selection of entries in the connection table. As with PRS, not all key based routing schemes and distributed hash tables can freely select the routing table entries. For example for Pastry the freedom is obvious: All entries ending with an 'x' in the routing table (see section 3.2.2) have more than one possibility to be assigned. If the entry has i out of n identifier bits set to 'x' and the network has N nodes, then the position can be set to one of $\lfloor N \cdot \frac{2^i}{2^n} \rfloor$. Especially in the upper rows a high number of nodes is eligible. On the other hand, de Bruijn networks do not have any flexibility at all when it comes to neighbor selection. With Chord, the requirement for the i-th finger to point to the closest node after $\text{ID} + 2^i$ can be loosened to the requirement that this finger should point to any node in the interval $(\text{ID} + 2^i, \text{ID} + 2^{i+1}]$. With this weaker condition, a similar degree of freedom as in Pastry is achievable, because for each finger, a similar number of possibilities exists.

The difficulty with PNS is to find these proximity-optimal neighbors, because a potentially large number of nodes are eligible. These problems and possible solutions are discussed in the section 3.4.4.

3.4.4   Finding Close Neighbors

To implement proximity route selection (PRS, see section 3.4.2) is a straightforward task. The distance to all routing table entries has to be determined. Since one of the design goals of structured overlay networks is to limit the number of routing table entries (there are $O(\log N)$ entries or less), the amount of work to be done to estimate all the distances is limited.

On the other hand, proximity neighbor selection (PNS, see section 3.4.3) is more difficult. Here, a potentially large[5] fraction of all members of the overlay network are possible candidates for routing table entries.

The order of magnitude of todays overlay networks prohibits the methodic probing of all distances. Since finding the best routing table entry has been

---

[5] For example, in Kademlia (see section 3.2.1), the first half of all nodes are eligible for the first bucket. Similar, Chord allows half of all nodes to be the target of the largest finger.

shown to be difficult, a number of heuristics to find good neighbors have been developed. Hildrum [64] describes an algorithm to find the nearest neighbor in growth restricted metrics with $O(\log N)$ steps on average. Clarkson [20] gives a good overview over other approaches.

During Bootstrapping

Nodes can already actively select close neighbors in the early phase of joining the network. In [28] it is proposed to use common IP-prefixes as hint for topological closeness. This paper and [96] show that it is common for overlay nodes to share large prefixes.

Measuring

Most approaches to finding and using neighbors close by have in common that the distance to the neighbors has to be measured[6]. The protocols used at the network and transport layer influence how this can be done.

If the transport protocol does its own latency measurements, these measurements can of course be used. Some transport protocols need to do their own latency measurements in order to compute expectations for packet arrivals. For example the Transport Control Protocol (TCP [135]) keeps an estimation of the round trip time (RTT) and its variation in order to estimate arrival time for acknowledgments. Since such measurements are part of the implementation of the transport protocol, the application (i. e. the overlay network) needs access to the internals of the network stack. Linux for example allows such access with the system call

```
getsockopt(int socket s, SOL_TCP, TCPINFO,
     struct tcp_info *, socklen_t *)
```

The structure `struct tcp_info` contains values required for PRS and PNS.

Other transport protocols like the user datagram protocol (UDP [133]) do not implement such features and leave it to the user application to realize it if it is necessary. Transport protocols like Stream Control Transport Protocol (SCTP [129]), Datagram Congestion Control Protocol (DCCP [91]), Real-time Transport Protocol (RTP [152]) need to perform some of the measurements to implement congestion and flow control, either by the protocol or by the application.

It is also possible to measure one level below the transport protocol at the network level. IP networks provide the Internet Control Message Protocol (ICMP [134, 24]). This protocol contains a pair of messages (Echo Request and Echo Reply) that are used to measure the latency between two hosts.

Note that the systems described above all measure the round trip time and not the one-way delay.

---

[6]Measuring means comparison. Here, the latency to neighbors is either compared to other latencies or to the local system clock.

Coordinate Approaches: from GNP to Vivaldi

In order to predict latencies between Internet hosts, the idea of a coordinate system was proposed. Such a system tries to embed the Internet and the latencies between hosts into a metric space (the model). These systems depend on measurements to or from known locations (e. g. Global Network Positioning GNP [126], IDmaps [56]) or between participating hosts (e. g. Vivaldi [27]). After a measurement takes place, the result is compared to the values estimated by the model. If they differ, the model is adapted. Challenges in these systems include:

- The right model has to be selected. Possibilities from [27] include d-dimensional Euclidean spaces, similar with an additive constant and non-Euclidean spherical spaces.

- Measurement traffic has to be balanced with the quality of results. Higher quality coordinates can be achieved with more measurements, but more measurement traffic also means more overhead.

- The model adaption rate has to be balanced with the prediction quality. If the model is quickly adapted, the system can react fast to changing conditions in the network, but tends to overreaction and oscillation. Damping can become necessary. If the model is not adapted quickly enough, it may produce useful results far too late.

Recent systems do not take the Internet as a black box, but incorporate the structure of the network into the model in order to improve the quality of predictions [105].

Meridian

Coordinate based systems have at least two drawbacks: First, to keep the coordinates up-to-date, a constant measurement traffic is necessary. Second, once the coordinates are estimated, they are only partially helpful in finding close neighbors. Greedily following to the next best neighbor does not work because of the high dimensionality and local minima[7].

Meridian [172] circumvents the need to create or maintain artificial coordinates. Each node keeps the latency to a number of neighbors. These neighbors are grouped into so called rings. A ring consists of neighbors with similar distance. The radii of the rings grow exponentially.

The rings are populated with a gossiping protocol. From each ring one member is selected at random. Then a list of representatives from each ring is compiled and sent to the selected node. Upon reception of such a list, the receiving node measures its distance to each of the nodes in the list and adds them, according to their distance, to the local rings.

---

[7]A greedy algorithm chooses the local optimum at every step. Note that such an algorithm may fail to find the global optimum.

To find close neighbors, a member of the outermost ring is queried. It will reply with nodes from the next innermost ring. The distance to each of these nodes is measured and the node with the smallest distance is queried for the next ring.

3.5 Other issues

3.5.1 Route convergence

Route convergence is a property of many key based routing schemes and describes the fact that messages from two different origins to a common destination share the later part for their routes. Since the in-degree of every node (here especially the destination node) is limited, routes are bound to converge.

In many cases, there is more than one route possible from source to destination. If all nodes optimize routes in the same way (either for the identifier-distance metric only or with proximity route selection), routes converge early. This is a very useful property for some applications on top of the key based routing system:

- Storage and caching systems can locate replicates at the points where routes converge. This has the advantage that the node at the convergence point can already reply to requests and reduce the load on the real destination.

- If PRS (and potentially PNS) is in use, routes have a high probability to converge locally. This property is useful because local communication is often cheaper and/or better.

- The convergence points can aggregate messages, a useful property if the key based routing system is used for more complex tasks.

On the other hand, there are overlay structures where route convergence is not prevalent. In de Bruijn-Graphs (see section 3.2.5), which can also be used as substrate for overlay networks [163, 164], routes do not converge early. This can be used as an advantage. Some security sensitive applications might want to route a single message on multiple paths. An adversary which wants to tamper with the message need to intercept and modify the message on all paths to stay undetected.

3.5.2 Bootstrapping

Since overlay networks are built on top of other networks, a newly joining node does not have an a priori connectivity to the network. In order to join the overlay network, at least one other overlay node must be known.

There are different solutions to the problem, including:

Static Nodes Used for example in the early Gnutella network [79]. There, the static DNS name of some Gnutella nodes were hard-coded into each Gnutella client. These nodes were called *pong caches* because they stored the replies to ping-messages. When newly joining nodes queried for nodes in the overlay, the pong cache replied with recently received pongs.

The telephony network Skype uses such a central entry point, too [8]. These central entry points are again hard-coded in the Skype client.

Native IP Multicast/Anycast By using IP anycast it is easy to join an overlay network. It requires that all overlay nodes are part of the anycast group. The joining node sends a request to the group and one node answers. The answering node becomes the bootstrap node of the joining one. If multicast is used, a scheme like [60] must be used in order not to overwhelm the requestor with replies. Both approaches have the disadvantage that neither IP multicast nor anycast is available at a large scale in the current Internet.

Out of Band Here, the bootstrap node is known to the joining node by a mechanism outside the scope of the overlay network. Examples include Gnutella web caches and the Freenet node list.

The Gnutella web caches [35] are CGI-scripts that store addresses of recently seen Gnutella Hosts. However, the Web Caches are either overloaded or the informations are outdated: According to [80] the fraction of nodes effectively online and responding to connections requests can be as low as 16%.

During runtime, a Freenet node (see section 3.1.2) periodically dumps its current table of connections. This dump is used to connect to the network the next time the node is started. On deployment, an initial node list is provided.

Meta-Overlay networks can be used to do a service lookup to find a bootstrap node. Such meta-overlay networks can also be used to implement multicast or anycast (see above). Note that a meta-overlay just introduces a level of indirection [14] and moves the problem to how to join the meta-overlay.

Random Address Probing can be successful if there are already many participants in the searched-for network. In [96] it is shown that in some parts of the IPv4 space mainly used for consumer networks, a large fraction of hosts are part of certain overlay networks. However, random address probing is also the strategy used by viruses and worms, so using this technique may trigger intrusion detection systems.

3.5.3   Implementation Issues

[119] reports on experiences while deploying the node manager service on PlanetLab and draws conclusions on the do's and dont's for developing highly distributed applications. It emphasis especially that networks and systems are

neither reliable nor work consistently. This includes clock synchronization and DNS name resolution. The paper then encourages developers to reconsider their assumptions made on the system, handle all corner cases correctly and gracefully, and pay attention to the overall resource consumption of the system.

3.6 Summary

This chapter introduced overlay networks and peer-to-peer systems. A special focus was on structured overlay networks and key based routing systems and how these systems are a good substrate for scalabe distributed applications, e. g. distributed hash tables. Later sections showed existing approaches to optimize overlay networks by adapting them to the underlaying Internet infrastructre with proximity route and neighbor selection and showed how to bootstrap overlay networks in the first place.

Chapter 4

File Systems

But something's wrong. The files are too big.
Abby, Minimum Security, Navy CIS, CBS

File systems have the task of organizing (or to let the user organize) data into logical units called files. Often, file system can also group files into directories and store meta-data along with file data. Some file systems store the content on permanent storage like hard disks, optical drives or flash memory. Others keep the entire state in transient memory. Some file systems can be used over a communication network, some with central servers, some without. Another distinguishing feature of file systems is the level of the operating system at which they are implemented.

4.1 Local File Systems

Local file systems (the opposite of network- or distributed file systems) serve different requirements. For example they need to store data on permanent (magnetic systems, flash) or volatile (RAM) storage, with constant (flash, RAM) or variable (disks-based) access times. Some of them allow free write access, others incremental and some no write at all. Another kind of local file systems are virtual file systems, which do not manage data on a storage device but represent (virtual) objects as files to the application.

The following sections will describe some selected local file systems.

4.1.1 File Allocation Table – FAT

The name of the FAT file system is derived from one of its main data structures, the *File Allocation Table* (FAT). The system was developed for DOS (Disk Operating System) from Microsoft. It can operate on files, directories and subdirectories. The initial main use was for floppy and small hard disks. Later,

the FAT was improved to be useful on larger hard disks and today its main use is for portable flash disks that are mounted on different computers with potentially different operation systems.

A storage medium consists of many sectors. A FAT file system groups the provided sectors into four main areas.

Boot sector The boot sector is always the first sector of a FAT file system. It stores descriptive information, including especially the position, size, number of the following components and a designator that the file system on this volume is a FAT file system.

File Allocation Table The file allocation table contains one entry for every *cluster*. Because the number of sectors can be large, especially on larger hard disks, a number of sectors are combined to form a cluster. One cluster is the allocation unit in FAT file systems. The file allocation table can encode the following values for each cluster:

- Free clusters
- Bad clusters, i.e. clusters where read/write problems have been detected and which should not be used anymore
- Entry ends. This special value indicates that the entry (file or directory) which this cluster belongs to ends inside this cluster.
- Entry continuation. Here, the value indicates the next cluster number for the entry. In effect files are stored as a linked list of clusters.

Root Directory The root directory is the starting point for usage of the file system. It can contain file entries and entries for subdirectories. Along with each entry (which consists of file name, file date and six flags (archive, read-only, system, hidden, device label and directory)) the number of the starting cluster is stored. Following clusters can be determined with a lookup in the file allocation table.

Main Storage Space Data clusters start after the root directory. The data clusters contain, if allocated, either file data or subdirectories.

Early versions of the file system FAT were designed for floppy disks. As such, they had limitations when used on larger hard disks, e.g. only 4096 clusters were allowed (cluster numbers were encoded in 12 bit numbers), file names were limited to 11 characters (8 for the name, 3 for the extension). Later versions removed these limits. Since most operating systems support the FAT file systems, it is a common file system for data exchange, even if it has disadvantages like missing user and permission management or its tendency to fragment files over time. FAT is standardized in [45] and [72].

4.1.2   Process File System

The process file system (`procfs`) is an example of a purely virtual file system. It is supported on many Unix-like operating systems (Solaris, BSD, Linux, AIX).

Once mounted, often at `/proc`, the file system contains a virtual directory for each process [86, 51]. Virtual files in this directory can be read to obtain information about this process, for example its working directory, resource consumptions and running statistics. Linux' version of the process file system offers also the ability to gather information about open file descriptors and many non-process related items. Such items include the current state of the system, power management and other hardware related information.

4.1.3 Fourth Extended File System

The fourth extended file system (`ext4`) and its predecessor `ext3` are based on the second extended file system [16]. The main data structures are:

Boot Record Here the fact is marked that the file system is a fourth extended file system.

Block Group The entire volume is split into blocks. A typical block size ranges from 1KiB to 8KiB. Blocks are grouped into one or more block groups. Each block group has its own superblock, group descriptor, block and inode bitmap and inode table, which are described below.

Superblock The superblock collects all important information about the file system. The sizes of the other data structures and their position are crucial. Also included are an unique identifier for the file system, the creation time and how often the file system has been mounted.

Group Descriptors There is a structure in each block group describing all the block groups. This kind of redundancy makes it easier to recover from problems in case the file system is damaged. Each group descriptor contains the layout inside the group (i. e. where block/inode bitmaps and the inode table start) as well as some statistical data about the group.

Block Bitmap In the block bitmap every data block in the group is represented as one bit. If the bit is zero, the block is free, if the bit is one, the block is allocated.

Inode Bitmap The inode[1] bitmap works similarly, but does indicate free/used inodes in the inode table instead of blocks.

Inode Table The inode table is an array of inodes. Each inode represents one object in the file system. The attributes of the object are described in the inode e. g. the type of the object, ownership, size, access attributes. The inode also contains data block pointers. Most data block pointers are direct pointers, i. e. the entry points directly to a data block. The third last entry contains an indirect pointer, i. e. it does not point to a block with data, but to a block with an array of block identifiers. The last but one entry contains a double indirect pointer, i. e. the block which it

---

[1]Inode is an abbreviation for Index Node.

points to contains indirect pointers. The last entry is a triple indirection. Figure 4.1 shows the concept of indirect pointers to data blocks. Another important field is the link counter. UNIX-like file systems allow that more than one directory entry points to an inode, in this case the files are *hard linked*. The link counter may also be zero if the file was deleted but is still opened by a process.

Data Blocks Data blocks finally contain the actual file date.

Note that directories are also stored as files. Directories are basically a sequence of (inode, name, type)-tuples and associate an object name and type with an inode.
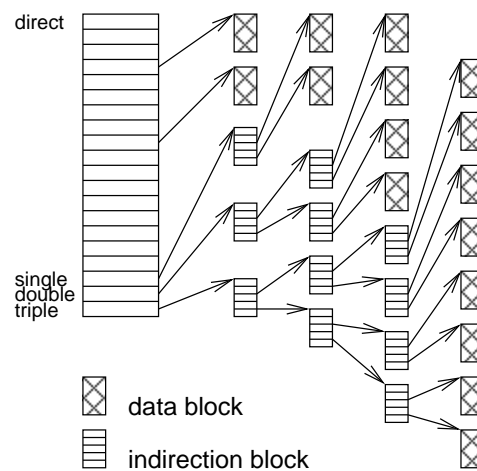
Figure 4.1: Direct and indirect pointers in inodes

The third extended file system `ext3` [166] added the journalling capability in a backward and forward compatible manner. Journalling means that write access to the file systems happens in transactions. A transaction consists of the actual data write and the meta data updates. Journalling ensures that the transaction either happens completely or can be rolled back if the execution of the transaction is interrupted. In this way, the expensive file system check after a non-clear unmount can be omitted. This increases the availability of systems because the time for a reboot is reduced.

The fourth generation `ext4` [109] increases the scalability of the system (file size, file system size). Further it adds *extends*, a new way to allocate blocks.

4.2   Distributed File Systems

In contrast to local file systems, in distributed file systems the machine using a file system is not necessarily identical to the machine providing the permanent

storage or file system implementation. Both machines can be connected over a network.

The following sections exemplarily introduce two distributed network file systems. Of course this selection leaves out important others, for example SMB/CIFS ([69, 70]).

4.2.1   Network File System

The *Network File System* (NFS [115]) was standardized in 1989. It is based on remote procedure calls (RPC [114]) from a client to a server. In order to be independent of details of the participating operating systems, all calls and replies are encoded according to a standard called eXternal Data Representation (XDR [113, 48]).

The network file system protocol tries not to create additional state at the server side (of course the files themselves are stateful). This makes clients robust against transient failures in the network and at the server side, i. e. if a server must be restarted, the client just notices a longer delay. To further increase robustness, most operations in the protocol are idempotent, i. e. they can be safely repeated.

To access a file on a server, a client has to execute the following steps:

1. The client needs to mount a file system from the server. The client issues a mount request, containing a string describing what it wants to mount. Usually this string is a path name interpreted by the server. The server checks if the mount request is authorized (by looking at the IP address of the client). If it is, the server returns a file handle for the root directory. For the client, such a file handle is an opaque sequence of octets (32 octets in NFSv2). Because of the stateless nature of the server, it must encode everything the server needs to know to access the object. Usually the device and the inode numbers are encoded. Once the client has a reference to the root directory, the file system is considered mounted.

2. With the handle of the root directory, the client can issue `read-dir` calls. The server replies with a linked list of entries.

3. The client can traverse the directory tree, potentially via other directories or by following symbolic links.

4. Once the client has read the directory where the file is located in, it can use the lookup call to find the file handle for the file in question.

5. With this file handle, the client can perform the actual read or write operation.

The stateless nature of the server makes the implementations of some semantics difficult. For example some Unix systems support deleted but still open files. Since NFS does not have a notion of open files, the semantic can not be directly supported. Most clients however implement a workaround: As soon as an open

file is deleted, it is transparently renamed. When the file is finally closed, the
client deletes the (renamed) file.

The version two of the NFS protocol (NFSv2 [115]) works solely on UDP.
Version three (NFSv3 [13]) allows the use of TCP, which allows to use NFS on
links where packet drops due to congestion are likely. Further, NFSv3 intro-
duces asynchronous writes. Before that, all calls from client to server have been
synchronous, i. e. the client had to wait for the result from the server. NFSv3
allows write calls to return early. To compensate for the uncertainty at the
client, a synchronous new call *commit* was introduced.

Other changes include better permission checking with an *access* call, better
handling of the end-of-file situation and informational calls to allow the client
to get more information about the mounted file system.

The fourth version of the network file system (NFSv4 [154]) introduces many
new features missing in previous standards:

- Authentication, integrity and privacy protection. This is achieved by using
  established standards like GSSAPI (Generic Security Services Application
  Programming Interface [103]), and Kerberos 5 [177].

- Compound operations. The client can pack many separate operations into
  one compound and send this compound to the server. This speeds up the
  operation on links with higher latency, since only one round trip time is
  required to fulfill many requests.

- Support for UTF-8 [174] in file names allows easier use in international
  contexts.

- Open, Close and more detailed Locking. With these operations, the NFS
  standard gives up the statelessness of the server, but decreases latencies
  because fewer calls are required to read/write one file.

- Delegations. Servers can hand out temporary rights to clients to work
  with files without notifying the server at all.

- Callbacks. In order for a server to be able to revoke delegations, a call
  back system was introduced.

After NFSv4, further development [155] went into the parallelizing of the
protocol. Currently, the following features are designed and implemented for
NFSv4.1 (sometimes also called pNFS for parallel NFS):

- The file server is split into one meta-data server and one or more data
  servers.

- Files can be striped over multiple data servers, thus aggregating their
  input/output performance.

- For the client, the meta-data server provides a unified view of all the data
  servers. Clients see all files in a single namespace.

These extensions are aimed at larger installations like file server clusters.

4.2.2   Other Distributed File Systems

Andrew File System (AFS[68][117]) is based on the Kerberos[124] security infrastructure and allows for extensive client side caching. The Coda file system[147] added even more distributed operations and conflict management. The distributed file system of the distributed computing environment (DCE DFS[130, 100, 74]) is a successor of AFS and shares most of the design principles. In IBM's Global Parallel File System (GPFS[150]) all nodes have equal access to all disks through a switching fabric. Lustre[10] seperates meta data servers (MDS) from object storage servers (OSS). By this, it achieves good IO-performance and it is popular within high performance computing centers.

The nature of all these systems is centralized, therefore they are not elaborated further in this document.

4.3   Decentralized File Systems

In contrast to distributed file systems, decentralized file systems do not have central components. This design paradigm avoids single points of failures, but has the additional problem that no trust anchor is directly available.

That such decentralized systems are possible has been shown for example with the Coorperative File System (CFS[30][31]). There, all content is stored in DHash, a distributed hash table running on the Chord[160] overlay network. Nearly all of the file system data is stored in self-authenticating content hash blocks, with the exception of the identifier of the root directory, which is signed asymmetrically. By checking this signature, clients can verify the data integrity.

In the Ivy file system[120], this trust is introduced with the participants public keys in the content-hashed view blocks. Contrary to to CFS, Ivy supports writing to the file system. All transactions in the Ivy file system are written in a log and the public keys are used to verify the log of other users. Using the transaction logs, users can work locally even in case of network partitions. After the network split ended, some possible conflicts have to be resolved manually. File system data is stored in DHash, too.

OceanStore[94][138] is decentralized in the sense that there is no single server but multiple ones. It also supports distributed modification access to the file system and uses a Byzantine-fault tolerant commit protocol[98] to achieve a consistent view on the global state. Because of massive replication and fault tolerant storage OceanStore excels as archive system. The system includes an economic utility model to allocate resources between consumers and among providers.

4.4   User Space File Systems

Traditionally, file systems are part of the operating system and as such implemented in kernel space. This has a number of reasons and advantages:

- The file system has to access the hardware devices that finally store the data blocks.

- Since file systems are relevant for security, they have been implemented in the trusted kernel zone.

- File systems offer resources to user processes and are as such part of the operating system.

In recent years, there is a trend away from this monolithic way to implement file systems. Micro-kernels or modularized kernels separate functionality. A new way for implementations are user space file systems. They offer some new advantages:

- Development is easier because a programming error in the file system code does not crash the operating system.

- Deployment is easier because the operating system does not have to be modified.

- More complex file systems are possible because user space offers much more libraries.

- Networked file systems need access to the network layer. Creating TCP-connections from the kernel is difficult.

The major disadvantage of user space file systems are security and safety because they are harder to guarantee in user space.

An important precondition to implement modular file systems is the existence of a central point to intercept calls from applications. Examples for such hooks include:

- Microsoft Windows systems offer *Installable File Systems* (IFS [26]).

- Linux has a *Virtual File System* (VFS [61]) layer.

- If applications use a common library to do file system access, this library can be replaced.

4.4.1   FUSE

FUSE [161] stands for file system in user space. It exports a virtual file system layer in the kernel back to the user space (see figure 4.2). FUSE has been ported from Linux to BSD systems (including MacOS X). It works by implementing a new file system at kernel level. This new file system does not reply to the calls from user space directly, but forwards these calls back to user space again. There, the real user space file system can be implemented.
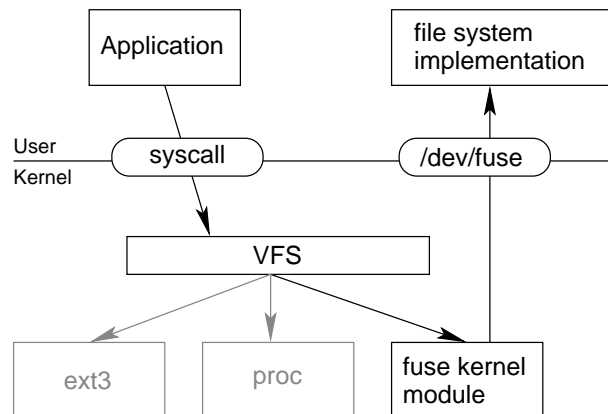
Figure 4.2: FUSE architecture

As mentioned above, user space file systems have security implications. If users are allowed to run file systems (and FUSE does so by default by providing a special setUID[2] mount helper), they can do harm in the following ways:

- Operations by other processes can be arbitrarily delayed.

- The amount of data returned by system calls can now be arbitrarily large, potentially larger than an application expected.

- Permission checks are not enforced.

- Arbitrary device files can be created.

To prevent exploitation of these problems, FUSE restricts user space file system in two ways. First, a mounted file systems is only accessible by the mounting user. That means the mounting user (any user, not necessarily the super-user) can mount FUSE file systems, but only this user can access it afterwards. Second, the creation of device file is not allowed. Both restrictions can be lifted by the super user.

4.4.2 Parrot

The Parrot [162] approach works differently. Here, the user space file system implementation intercepts system calls before they really reach the kernel level. Parrot uses the process tracing facility (ptrace), found in many Unix-like operating systems, for this.

Figure 4.3 shows the concept, which can be transparently [90] applied to any process except

---

[2]A binary executed with the permission of its owner, instead of the permissions of the executing user.

- SetUID-processes, where the ptrace call is not allowed for security reasons,

- Processes where the ptrace call is used for something else, e. g. debugging or monitoring.

An important advantage of the parrot concept is that it can work without any interaction from the super-user, i. e. no kernel modification is necessary and no setUID-program has to be installed. On the other hand, the ptrace interception has to be set up for each process separately, while mounting with FUSE is visible for all processes.
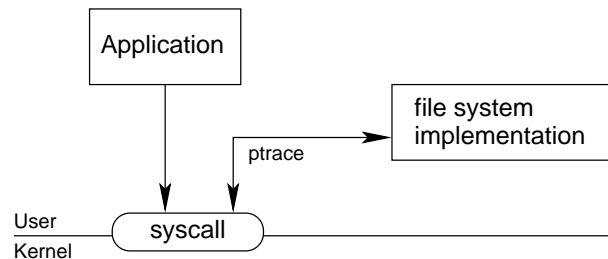
Figure 4.3: Parrot architecture

4.4.3   Gnome-VFS etc

Another possibility to implement user space file systems is shown by Gnome and similar projects. There, a virtual file system layer [127] is introduced in the user space. Note that this layer is independent from the VFS layer in the kernel which is used by FUSE. Here, applications do not use direct system calls any more. Instead, they use the functionality from the VFS library (see figure 4.4). This has the advantage that there is no dependency on the operating system. On the other hand, applications are not allowed to directly use the interface from the operating system any more, that means they are dependent on the user space VFS library.

4.5   Summary

This chapter gave an overview over file systems in todays computing infrastructure. First of all, local file systems that are used for single computers are coverd. Second, this chapter introduced distributed file systems. Such file systems are used to share data between different hosts. Additionally, virtual file systems and file systems in user space have been discussed.
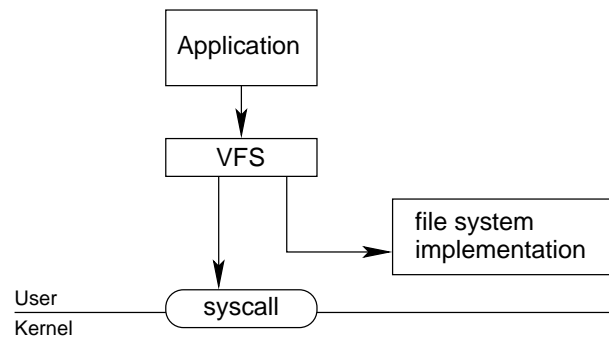
Figure 4.4: Gnome VFS architecture

Part II

The Overlay Network Igor

Chapter 5

Goals

The project Igor was started with the aim of gaining experience in overlay networks and key based routing systems. Igor itself is a reverse acronym and stands for Internet Grid Overlay Routing. Over time, quickly more objectives were included in the design guideline for Igor. On the one hand, these goals came from experience gained from practical uses and by developing applications using Igor. On the other hand, by developing overlay networks and key based routing systems one can gain a deeper understanding.

As an overlay network, Igor should run on top of existing networks, i. e. in the OSI model on top of other transport protocols (see figure 5.1). It offers services for applications running on top of it. The next sections describe the design goals for Igor related to applications running on top of Igor, the overlay network itself and issues regarding the network it is running on.

Application
Igor
Transport Protocol
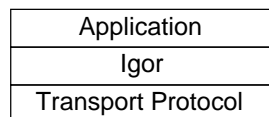
Figure 5.1: Igor Stack

5.1   Application

Overlay networks and especially key based routing systems allow a new class of distributed applications. Key based routing systems offer new semantics for message forwarding (forwarding a message to the node closest to the target), and Igor should offer these semantics to the applications. Further it will be designed to support a wide range of such applications and put as few restrictions on them as possible. In particular more than one application will be supported, i. e. Igor should be able to multiplex different uses.

For the development of applications using overlay networks and key based routing systems it is important that the *Application Programming Interface* (API) is simple and easy to use. To achieve this goal, an API similar to existing ones should be selected. Further it is important that Igor is easy to integrate in applications.

5.2   Igor

One reason to introduce structured overlay networks in the first place is their scalability. A network with many thousands of nodes must work as well as a network with only a couple of them. Second, the Igor-network must scale with the resources of its nodes. Powerful nodes will be able to take over more work, whereas weaker nodes must still be able to participate.

The implementation of Igor itself must be resource efficient. Since it is a research project, the Igor implementation must be easy to extend and modify.

5.3   Network

The Igor overlay network must be designed to be used by many different kinds of applications, among them applications that transfer larger amounts of data. Therefore, the overlay network must not put undue stress on the underlying network, especially it must obey the rules for regarding congestion and it must be fair to other users of the network.

If the conditions in the underlying network change, the Igor network should quickly adapt to these changes.

For network and firewall administrators the implications of running overlay networks must be clear.

Chapter 6

Design

What goeth around, cometh around
Terry Pratchett

This chapter describes the design of the Igor overlay network. It details the interface to the applications as well as some internal design decisions for the implementation of the daemon.

6.1 Application Interface

6.1.1 The Service Concept

One of the main concepts of Igor is that a single overlay network can be used for different applications. These different applications are called services and are distinguished by their service identifier I_S. The service identifiers are from the same number space 2^l as the node identifiers, where l is the length of the identifier. Why this is necessary is explained in more detail in section 6.2.4.

Applications can register with Igor that they are of a given type I_S. During message delivery it is ensured that messages of type I_{S_1} are delivered only at nodes where a service of type I_{S_1} is registered. While it is possible that multiple applications I_{S_1} and I_{S_2} register at the same Igor node, it may also happen that at a given application/service is not registered at one node at all.

6.1.2 The Upcall Concept

For recursive overlay networks (and Igor is recursive, see section 6.2.1) another dimension in the design space opens. In iterative overlay networks, the source of a message is responsible for discovering a route to the destination. After a route is discovered, the messages are forwarded directly from source to destination. In recursive overlay networks, messages are forwarded hop-by-hop through the
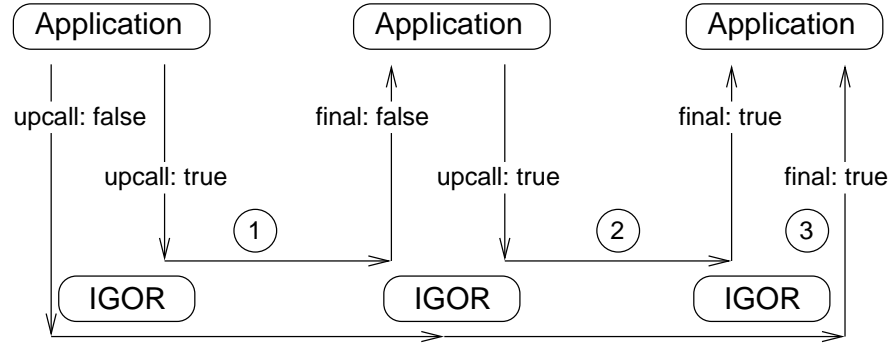
Figure 6.1: The upcall concept

network, and the route to the destination is found along the forwarding path. That has the consequence that messages are handled by nodes besides source and destination. If on these nodes the same application is registered (see section 6.1.1), it can be useful that these applications are given a chance to handle the message, too. Because this is not always useful, an additional parameter *upcall* is attached to every message[1]. The value of this parameter can be either *true* or *false*. If the value of upcall is true, the message is handed to the first application of the correct type along the way from source to destination, providing that such an application exists. If such intermediate application does not exist or the upcall flag is set to false, the message is transfered normally to the destination. The intermediate application (if it exists) can decide how to handle the message:

- Forward the message unmodified.

- Forward a different message (potentially a modified version of the received one).

- Delete the message.

- Delete the message and send a reply.

Applications may want to adjust their behavior depending on whether there is another application they potentially can forward the message to. For this purpose, the flag *final* is introduced. This flag is set by Igor on messages sent to applications. The flag is set to *true* if this Igor-node is the final destination for this message and set to *false* if this delivery is to an intermediate application. If upcall is set to false, the message is forwarded directly (potentially via different Igor-hops) to the destination and delivered there with the final flag set to true. Applications on intermediate nodes can not exert influence on this process.

---

[1] In [33] a similar problem is solved by introducing to down-calls: `route()` and `forward()`.

	upcall: true	upcall: false
final: true	message delivered directly, no intermediate applications exist (corresponds to message 3 in figure 6.1)	message delivered directly by request (corresponds to message 3 in figure 6.1)
final: false	message delivered to an intermediate application (corresponds to message 1 in figure 6.1)	not possible

Table 6.1: Possible combinations of the flags *upcall* and *final*.

Figure 6.1 shows the concept again and table 6.1 gives all possible combinations of the flags upcall and final. Note that the upcall flag is set by the application, whereas the final flag is set by Igor.

One main reason for sending messages with the upcall flag set to true is aggregation, i. e. intermediate nodes receive the message collect more messages to the same target and send on a summary towards the real destination. The other main reason is the possiblity of caching and early answers. Since intermediate nodes are expected to be closer (latency-wise) to the source of the message because of PRS/PNS (see section 6.4), answers from these nodes reduce the overall latency experienced by the requestor. Early answers also reduce the load on the real destination, which is important to releave hot spot behaviour (slash dot effect[1]).

On the other hand, applications should set upcall to false in case delivery speed to the final destination is important. The less intermediate systems are involved, the faster the message reaches the target. Also, if the message content is of no interest to the intermediate nodes (either by application design of for privacy reasons), the upcall flag should be set to false.

6.2 Message Routing and Forwarding

6.2.1 Iterative vs. Recursive

As shown in section 3.2 there is a distinction between iterative and recursive message delivery. Table 6.2 summarizes some of the differences. The big advantage for iterative implementations is that the message is transfered only once, from the sender to the final receiver. For every message, the final receiver is computed by iterative lookups in the routing tables of other nodes. This leads to a large number of connections. Each connection has to be set up, which takes some time and adds up to a significant delay for every message. Furthermore, this frequent set up of new connections is a burden for the operating system and all middle boxes, for example NAT-gateways, connections-oriented

firewalls and flow-accounting routers. NAT systems have to be traversed for every new connections, adding more connection problems and additional delay. Also proximity route and neighbor selection are more difficult with iterative message forwarding. PNS is useful, because in an iterative forwarding chain A-B-C, if A is close to B, then B is usually close to C too. However, PRS is more difficult to employ, because even if C is the closest node to B, it is not necessarily the closest node to A too. The only important disadvantage of recursive message forwarding, the fact that the message has to traverse many intermediate systems, is also an argument in favor of recursive routing: These intermediate systems can also process the message, for example caching the message content, answering requests early, or aggregating many messages into a single upstream message (see section 6.1.2). This is especially useful for networks where routes converge early (see section 3.5.1). Igor therefore implements recursive message forwarding.

Property	iterative	recursive
message sending overhead	no	yes
message delay	yes	no
number of new connections	many	few
PRS	more difficult	yes
PNS	yes	yes
firewall/NAT traversal	more difficult	solvable
upcall	no	yes
make sure message arrives	easy	difficult
connection to the destination	easy	difficult

Table 6.2: Some comparison between iterative and recursive routing

6.2.2   Metric

Every message is handed to a neighboring intermediate system that is closer to the destination than the current node. This process continues until the message arrives at the destination. To evaluate the closenesses between a given overlay neighbor and the destination of a message a metric is required. Igor uses $d_{igor}(ID_1, ID_2) = f(ID_1, ID_2) = |ID_1 - ID_2|$ as its metric. This metric has the properties that it is:

- symmetric, that means $d(ID_1, ID_2) = d(ID_1, ID_2)$,

- forms a ring, which XOR does not and

- easy to compute.

Message forwarding is guaranteed to work if the following invariant holds: Assume the node identifiers are laid out on a virtual ring like in Chord (see section 3.2.3). Each node needs to know its left-hand neighbor and its right-hand

neighbor on the ring. If this invariant is given, then there is a path between each pair of nodes where the distance (according to the metric) to the destination decreases strictly monotonicly. These two connections ensure consistency in the routing algorithm. To ensure efficiency too, shortcut connections [89] are necessary. These shortcut connections are opened to nodes with exponentially increasing distance.

6.2.3 Aggregation Tree

Due to the limited number of connections and the strict monotonic routing towards the message destination, a implicit tree is formed for every message destination. The node finally responsible for the message is the root of the tree. All nodes connected to this node are by construction more distant to the destination and form the second level of the tree. The third and all subsequent levels consist of nodes that would route messages with the given destination to the node in the level above. On the lowest level there are nodes that do not receive message with the given destination, because for all other nodes there is a better routing decision. Not that the aggregation tree is constantly changing because of changes in the Igor topology and due to changes in the underlaying network.

6.2.4 Service Routing

There are different ways to ensure that messages are delivered to

- one of the nodes where the service as indicated in the message is running *and*

- to the node whose identifier is closest to the destination of the message.

First, it is possible to enforce that the service is running on the destination node, either permanently or on demand. Both variants are not feasible in the Igor scenario.

Second, a multi-dimensional routing seems to be viable: one dimension for service identifiers, one dimension for node identifiers. Routing in such multi-dimensional spaces is not straight forward since local minima for a greedy algorithm may exist. Nevertheless, geographic routing [82] or CAN [137] show that it is possible. The disadvantage of such a scheme is that nodes that are not part of a service still have to forward messages for it.

Therefore the third alternative [88, 173] was chosen for Igor[167]. Here, all participants of a given service form a subgraph of Igor, where the invariant that neighbors on the virtual ring know each other holds for each subgraph. To find and join such a subgraph, the global Igor network is used as a lookup service (see section 6.3.4 and figure 6.2).
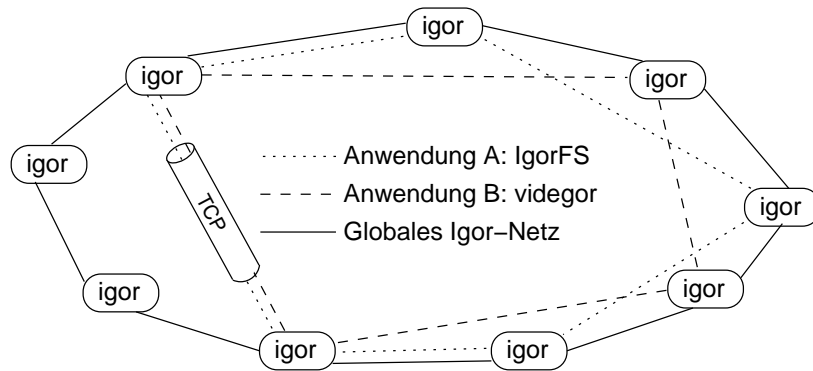
Figure 6.2: Service Sub-Graphs in Igor

6.2.5   Connections

As overlay network, Igor runs on top of a transport protocol. In todays Internet, two transport protocols (ISO/OSI layer 4) are in wide use: TCP and UDP. Table 6.3 compares the requirements of the Igor design (see section 5) with the features TCP and UDP provide. Both protocols have their advantages and disadvantages. UDP excels with its simplicity in the protocol stack, while TCP already implements more features required for Igor. Given these properties, Igor was implemented on top of TCP.

Property	Igor requirement	TCP	UDP
timely error detection	nice-to-have	not easy	not easy
connection oriented	yes	yes	no
flow control	yes	yes	no
congestion control	yes	yes	no
packet sending	yes	no	yes
delay measurement	yes	yes	no
bandwidth measurement	yes	no	no
TCP-friendliness	yes	yes	no
simplicity	yes	yes	yes
message size > 64k	yes	yes	no
NAT and firewall traversal	yes	possible	possible

Table 6.3: Requirements of Igor and features of available transport protocols

6.3 Routing Table Maintenance

All connections of an Igor-node are kept in a table called routing table (sometimes also referred as finger or connection table). This section describes how new entries are added to and how old entries are removed from this table.

6.3.1 Creation of New Entries

Periodically, each Igor-node tries to optimize its routing table. This works by selecting a random destination on the ring in a way that far away destinations are selected exponentially less often than local destinations. Next a message is sent to this destination, using the current routing table. The receiving node replies with a list of other nodes close to the selected random destination. This way, every node learns about new nodes, with a strong bias to local (on the virtual ring) ones and a small world network [89] will form.

Secondly, the two neighbors on the virtual ring are queried periodically for their virtual neighbors. If the network is in a consistent state, one of the neighbors' neighbor must be the querying node. This stabilization procedure is there to detect inconsistencies in the virtual ring and to quickly correct them.

6.3.2 Creation of the First Entry

Bootstrapping is the process of joining an existing network of Igor nodes (see section 3.5.2). To join an overlay network, at least one node of the existing network must be known. Since Igor is an overlay network, this holds true for Igor. There are different ways to gain knowledge of this first other node:

- Outside channel. The user gives the first node(s), for example via a configuration file or on the command line.

- Scanning. In [23] it has been shown that it is feasible for reasonable deployed networks to scan whole address ranges for other peers.

- Multicast/Anycast. When multicast or anycast is available, all existing nodes can join such a multicast or anycast group. The new peer also joins that group and sends a query. If multicast is used, a feedback suppression scheme (e. g. [60]) should be used to prevent too many answers to the newly joined peer.

Multicast and anycast are not readily available in the Internet (overlay multicast systems only shift the problem [14] to "How to bootstrap the overlay multicast system?") Since Igor is not sufficiently widely deployed (yet) and scanning is considered offensive, the "Outside Channel" method is used to bootstrap Igor.

6.3.3 Eviction of Old Entries

With the mechanism described above, the routing table would quickly become too large. To prevent this, there is a periodic mechanism to erase less useful connections. The algorithm works as follows:

1. Order all connections according to the distance (using the distance metric) from the node.

2. Compute the logarithm of the distance.

3. For each node in the ordered list, compute the difference between the logarithms of the succeeding and the preceeding node.

4. Close the connection to the node where this difference has the least value.

Some connections are exempt from closing by this algorithm: Direct neighbors in the virtual ring, direct neighbors in one of the service rings, and connections that have been opened recently. The last constraint was introduced to bring more stability into the routing table.

6.3.4   Connections for Services

For each service offered locally, Igor must maintain the invariant that connections to the left- and righthand neighbors in the virtual ring for that service are established and kept. The connection eviction algorithm prevents that these connections are closed after they have been opened.

To open these connections in the first place, a lookup mechanism is required. For this lookup mechanism to work, a minimal soft-state DHT is implemented inside Igor: After an application has registered with its local node, the node periodically announces this fact. To do so, it sends an announcement message, containing the identifier of the local node and the identifier of the service, towards the identifier of the service. The upcall flag is set to true, so intermediate nodes can aggregate these messages and limit the number of nodes announced per service.

In parallel, the node queries this minimal DHT for other nodes where the same service is registered. With this mechanism, nodes with the same services are able to find each other and establish the invariant that neighboring nodes are mutually connected per service.

6.4   Proximity

As described in section 3.4, proximity neighbor selection (PNS) and proximity route selection (PRS) are important for the performance of an overlay network. They exploit the basic property of overlay networks that they can choose their own topology. PNS forms the topology of the network in a way that the performance is optimized. Good links (low latency, high bandwidth) are favored. Links with low latency are local links, since the speed of light limits the lower bound for latency of long distance links. This way, Igor will prefer connections in the network neighborhood.

As soon as the topology of the overlay network is established, multiple paths from a source in the overlay to a given destination are usually possible. Proximity route selection ensures that a good one (again with respect to

latency/bandwidth) is selected. It works by including link quality in the routing decision.

Igor employs both PNS and PRS with latency as metric to improve the quality of the message routing process.

6.4.1 Proximity Route Selection

For PRS, Igor employs the following algorithm.

1. All nodes are evaluated using the identifier metric (see section 6.2.2). Each connection is assigned a score $\in (0, 1)$. Connections that lead closest to the destination receive the highest scores. All connections that are farther away from the target than the current node are evaluated with 0. The last condition ensures that no routing loops can occur.

2. The set of all nodes is evaluated given the proximity metric (see section 6.4). Each connection is given a score between 0 and 1, where connections with the best proximity value receive the value 1.

3. Then, connections are evaluated regarding services. If the next hop offers the service the message indicates, it receives a score of 1 and 0 otherwise.

4. The product of all scores is computed and the connection with the highest product is selected.

Note that all operations and required measurements can be done locally. That means that PRS can be employed easily and no protocol extension is required.

6.4.2 Proximity Neighbor Selection

Contrary to PRS, proximity neighbor selection is not a local process. Here, many nodes must work together to find closest neighbors for each of them.

Combination of Vivaldi and Meridian

Vivaldi (see section 3.4.4) excels at embedding a network like the Internet into a metric space. Coordinates are assigned to every node in the network. However, it does not prescribe a way to find close neighbors.

On the other hand, Meridian (see section 3.4.4) shows how to find close neighbors in networks, but it has to do a number of measurements to execute the protocol.

A combination of these two protocols, called Merivaldi[87], can merge the advantages of both. In order to do so, Vivaldi is executed to compute the coordinates of the nodes. The traffic to compute the coordinates can be neglected as it is part of normal protocol operations as in TCP. The information required are piggy-backed in every message. Meridian can then use the computed coordinates to find close nodes without further measurement actions.

Finding new neighbors

To compute the Vivaldi coordinates, the coordinates of neighboring nodes and
the latency to these nodes are required. The coordinates of neighboring nodes
are piggy-backed in regular messages. The protocol TCP is used for the mea-
surement. TCP needs to do similar measurements already, so there is no need
to re-implement it at the Igor-level.

To find better neighbors using the combination of Meridian and Vivaldi,
the Meridian find-closest-neighbor protocol is executed regularly. Since Vivaldi
keeps up-to-date coordinates for all nodes and the coordinates are communicated
regularly, Merivaldi ([87]) does not need to do measurement during its execution.
Merivaldi can resort to distance computations based on the Vivaldi coordinates
instead of measurements.

6.5   Interface to Applications

6.5.1   Connections

The message exchange between Igor and applications can be implemented with
a number of inter-process communication (IPC) methods like shared memory
or pipes. Table 6.4 compares the advantages and disadvantages of some of
them. The easiest would be a direct linking of Igor and the application. This
option has the downside that only one application can use the Igor network
at a time, other applications would have to start their own instances. Shared
memory was not chosen because then the application is bound to run on the
same machine as the Igor daemon. If, from a network administrator point
of view, the Igor daemon has to run in a demilitarized zone (DMZ) because
it opens connections to and receives connections from the outside world, the
placement of applications is limited. Usage of pipes has the same restrictions as
shared memory, but additional disadvantages of the need to serialize messages
in between. If the application and Igor communicate through TCP-sockets, the
restrictions regarding placement of the components are lifted. Many different
applications can connect to one Igor daemon, and the daemon can run in a
DMZ.

IPC	pro	con
single process	easy	inflexible
shared memory	fast	single machine
pipes	same interface as network	single machine, serialization
TCP socket	very flexible	serialization

Table 6.4: Comparison of different inter process communications between Igor
and applications
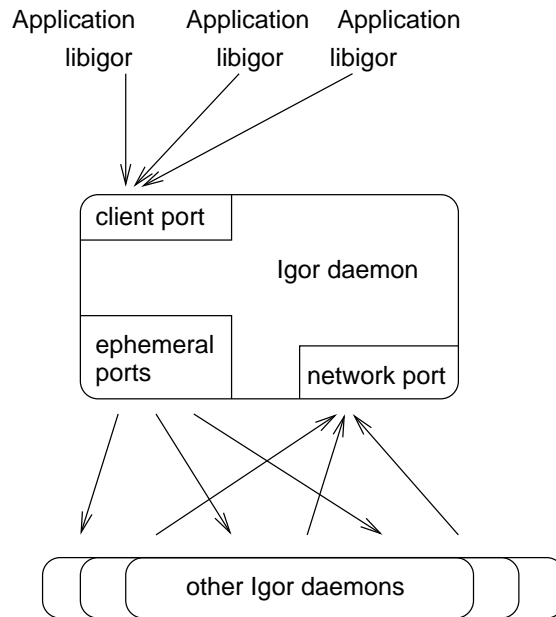
Figure 6.3: TCP port usage in Igor

Because of the advantages of TCP at the node-client interface and the similarities between node-node and node-client communictation, it was decided to use TCP connections in Igor for both purposes:

- First, connections between two Igor instances are TCP connections.

- Connections between the Igor node itself and the library use TCP too. This makes the Igor network accessible by applications.

A TCP socket in listening mode is opened for both purposes. Figure 6.3 summarizes both ways TCP connections are used. In section 6.2.5 the usage of TCP connections for inter-Igor links is explained. Note that application-to-Igor connections are always opened by the application towards the client port of the Igor daemon. On the other hand, node-to-node connections are opened by Igor processes from ephemeral ports and received by Igor-processes on the network port.

6.5.2   Library `libigor`

In order to make the Igor network easy to use for application developers, an interface with which many programmers are familiar with was chosen. The BSD socket interface [159] is available in many operating systems to interact with the network stack.

Figure 6.4 shows the similarities and some of the differences between programming regular BSD sockets and using Igor sockets.

Applications using Igor first need to open a socket. This socket is used for all further communication between Igor and the application. Once the socket is created, it can be bound to a specific service (traditionally called "giving the socket a name"). If this is done successfully, send and receive calls can be used to transmit messages. Finally, the function `igor_close()` terminates a connection between the application and Igor.

The structure `sa_igor` is used to communicate addresses and to name the service an Igor socket is bound to. This structure allows to encapsulate the large identifiers used by Igor. Furthermore, some convenience functions exist to hash arbitrary strings into the Igor identifier space.

A library called `libigor` provides the socket and convenience functions. Further the library encapsulates the TCP connection to the Igor daemon.

BSD-Sockets	Igor-Sockets
`int s = socket(sock_dgram);`	`int s = igor_socket();`
`bind(s, struct sa_inet);`	`igor_bind(s, struct sa_igor);`
`send(s, msg);`	`igor_sendto(s, ...);`
`recv(s, msg);`	`igor_recvfromto(s, ...);`
`close(s);`	`igor_close(s);`

Figure 6.4: Comparison of regular BSD-sockets with Igor sockets

6.6   Summary

This chapter described the design of the overlay network Igor. It covered the services offered by Igor via a library to the applications as well as the service oriented design of Igor itself. Important concepts like the upcall or final flag were introduced. The chapter explained how Igor forwards messages, regarding to services, identifiers and proximity.

Chapter 7

Implementation

7.1 The Call Back List

In order to keep Igor simple, it was decided to abstain from multi-threading. Igor is implemented as a single threaded application. As such, all actions are driven by a single event loop. Actions can be triggered by

- events on file descriptors. Such events can be the transition of a file descriptor from the blocked state into a readable or writable state. Other events include a successful connection setup or termination of a connection.

- timers. Timers can be used for periodic actions (e. g. a call `stabilize()`, every $t_{\text{stabilize}}$ seconds, see section 6.3.1 every) or single events (e. g. resend message if no reply is received within t_{timeout}).

If one event (either on a file descriptor or a timer) happens, a function is executed. This function is registered on demand (creation of the timer or file descriptor) and is called *call back function*.

Call back functions in object oriented languages are more difficult than in non-object oriented ones because functions can be called on objects only[1]. Since Igor is implemented in C++, the problem occurs. To circumvent this, a abstract base class (figure 7.1) and a template (figure 7.2) have been introduced.

This construction allows to create call backs to every method inside classes, given the method takes two integers as parameters and returns nothing.

---

[1]Functions on objects are called methods in object oriented languages.

```
class cCallbackBase {
        public:
                virtual void Go(int vHandle)=0;
};
```

Figure 7.1: Abstract Base Class for call backs

```
template <class C>
class cCallback : public cCallbackBase {
        public:
                void Init(C* rTargetObject,
                        void(C::*rTargetMethod)(int,int));
                virtual void Go()
                {
                        (*mTargetObject.*mTargetMethod)
                                        (vHandle, mState);
                }
        private:
                C* mTargetObject;
                void (C::*mTargetMethodFunction)(int,int);
};
```

Figure 7.2: Call Back Template

7.2   Plugins

As a research project, the Igor implementation must be easily extendable. To achieve this, a plugin concept was introduced. Such plugins can be included at compile time. A mechanism to select plugins at load or run time was not deemed necessary. To extend Igor, there are four different types of plugins:

cMessagePlugin Plugins of this type handle incoming messages, i. e. the plugin is responsible for the message. That can include unmodified forwarding, modified forwarding, generation of a reply or even a silent discarding.

cPeekPlugin This type of plugin may have a look at every incoming message, but has no possibility to influence the further fate of the message. Uses of this plugin type include the evaluation of piggy-backed information and statistical purposes.

cPolicyPlugin Policy plugins can influence decisions in an Igor node. Such decisions include

- the forwarding/routing of node-to-node messages. For every message to be transmitted, policy plugins are queried.

- connection opening. Before a new connection is actually opened, policy plugins can give their evaluation of the new connection.

- connection closing. Policy plugins are queried which connection they would close first.

- delivery of node-to-client messages. If more than one client can receive an incoming message, policy plugins are included in the decision.

Furthermore policy plugins are notified when connections are opened or closed.

`cTaskPlugin` In order to do regular (periodic) tasks, task plugins can be registered.

With the C++ multiple inheritance mechanism, a plugin can be of more than one kind. For example, a combination of `cPeekPlugin` and `cTaskPlugin` could record statistical information from incoming messages and output these statistics in regular intervals. Another example is a plugin derived from the two classes `cPolicyPlugin` and `cMessagePlugin`. Such a plugin can register for incoming connection recommendations and evaluate which of these will be opened.

7.3 IPv6

The Igor daemon has been implemented in a way that works equally well with IPv4 and IPv6 addresses. This was done by strictly encapsulating all network related functionality in the two classes `cTransport` and `cConnection`. The former is responsible to carry a OSI layer 4 address, the latter to handle all connection related calls into `libc`. All network related system calls that are specific to one address family, especially name resolution functions, have been avoided.

Chapter 8

Test and Deployment

If the implementation is hard to explain, it's a bad idea.
The Zen of Python

8.1   Testing

8.1.1   PlanetLab

Igor has been successfully tested on PlanetLab[131], the distributed (for a node distribution, see figure 8.1) overlay network research platform. To this end, specialized testing tools have been developed[92].
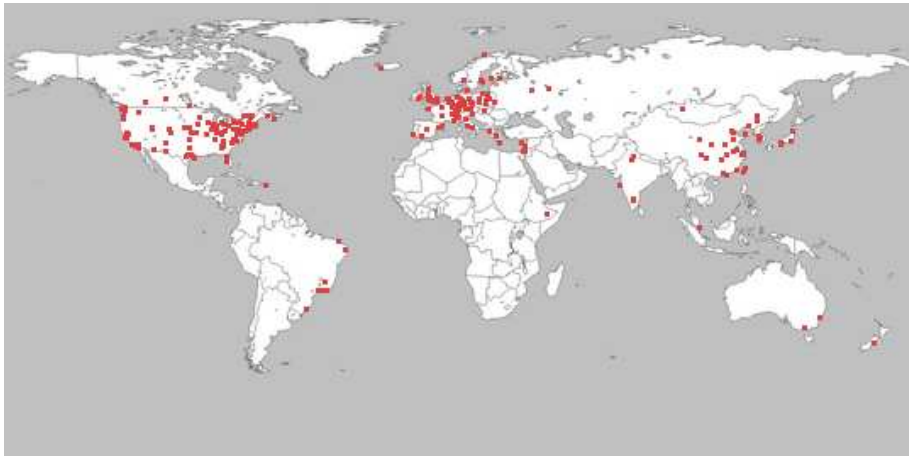
Figure 8.1: Node Distribution in PlanetLab on 2007-02-18

8.2   Build Process

Igor is built using the GNU[1]tool chain of `autoheader`, `automake` and `autoconf`.
For the user this creates the familiar environment to compile, link and deploy
Igor by using `configure`, `make` and `make install`. As maintainer of Igor, one
uses the shell script `bootstrap.sh` which executes all necessary tools of the
tool chain in the correct sequence. Note that the name of this shell script has
nothing to do with the bootstrapping process described in section 6.3.2

8.3   Running Igor

This section will gives details on how to start and stop Igor. This includes
configuration parameters on the command line as well as in the configuration
file. The log file and the bootstrap process are also described.

8.3.1   Start

Since Igor is intended to be used as a daemon (background) process. There-
fore the system call `fork()` is executed immediately after starting the binary.
The process called by the user quits soon, while the process in the background
detaches all terminals and stays in the background.

   During start, the command line of Igor is read. The command line may
contain the options shown in table 8.1.   Many options need a parameter.
Parameters to the long form of an option are mandatory to the short form too.
Network port and client port are explained in section 6.5.1.

8.3.2   Stop

During startup, the daemon writes a file containing the process identifier (PID)
of the newly started Igor instance. The location and name of this file is de-
termined at configure and compile time, but can be changed at startup, ei-
ther via command line parameter or via configuration file settings. The script
`igor-stop.sh` uses this PID-file to stop the running Igor. Alternatively, Igor
may be killed with the signals SIGINT or SIGTERM, i. e. by either hitting
Ctrl-C on foreground processes or terminating background processes with `kill`.

8.3.3   The Configuration File

In addition to options on the command line, the behavior of Igor can be in-
fluenced via a configuration file. The file names searched for as configuration
files are `igor.conf` and `<name of the binary>.conf`. The files are searched
for in the `/etc`, `/usr/local/etc` and the current directory. The format of the
file is relatively common: Empty lines and lines starting with a hash mark are
ignored, all other lines are expected to be of the form "key colon space value

---

[1]A recursive acronym: "GNU is not Unix"

short and long form of option		meaning
-h	–help	Show usage
-v	–version	Print version numbers and exit
-b	–bootstrap=TRANSPORT	Transport address of one bootstrap node (host name or IP address and port name or number)
-t	–cont_if_no_conns	Continue even if their are no connections left
-i	–id=ID	ID of the local node (in hex form)
-p	–netport=PORT	TCP port where to listen for other nodes (in decimal)
-u	–clientport=PORT	TCP port where to listen for clients (in decimal)
-c	–config=FILE	Name of a configuration file
-d	–debug=STRING	Debug options (see section 8.3.4)
-l	–logfile=FILE	Log file to use
-n	–no-daemon	Do not start Igor as a daemon and log to standard output
-s	–silent	Do not print daemon information to standard output
-P	–pidfile=FILE	Write process identifier to this file

Table 8.1: Command Line Parameters of Igor

newline" (e. g. `id: da4de85db275...`). Possible keys and the meaning of their corresponding values are listed in appendix A.

8.3.4 The Log File

In order to track execution, inform the user about important events and aid debugging, the Igor daemon can output log information. These are configurable and are either written to a log file or to standard out.

Every line in the log file consists of four items:

1. Current time. Format for time values is the number of seconds since the Unix epoch (1970-01-01T00:00:00) (as most RFCs, maybe with the exception of [136]), here given as a decimal floating point number.

2. Facility. Current facilities are listed in table 8.2.

3. Priority. Priorities range from 0 to 9, where 0 indicates fatal error messages and 9 signals verbose debug output.

4. Message. The last item on every line is the message itself.

Facility	Meaning
DBG	The debugging system itself. Print why some messages are printed while others are suppressed. Debug configuration of the debugging system.
DFLT	Default. When the developer did not mention a facility, this one is chosen automatically.
CBL	Print messages related to the call back list.
NODE	Information regarding the Igor node itself. Message routing is one example.
CONN	Connection establishment, usage and teardown to other nodes are logged using this facility.
PARSE	Parsing of incoming messages into raw messages.
MESG	Handling of parsed messages.
CFG	Reading configuration files and parsing command line options.
ID	Dealing with node identifiers. This includes parsing and metric computations.
RAWMSG	Transportations of raw messages.
BUF	Buffers are the entities for memory management. Debug the allocation, usage and destruction of such buffers.
LIB	Information regarding libigor.
PLUGIN	Information from plugins.
SERVICE	Service oriented message routing.
MERIVALDI	PRS/PNS and latency oriented message routing.

Table 8.2: Igor Debug Facilities

8.4   Application Examples

This section will detail some applications we implemented and that are deployable on top of Igor to show the usability of the system. Figure 8.2 shows some instances of the application described in the next sections together with an Igor overlay network. Since each application opens its own service in Igor, the figure is similar to figure 6.2.

8.4.1   Filesystem

One application implemented on top of Igor is the decentralized file system IgorFS. The third part of this document covers IgorFS in detail.

8.4.2   Videgor

Videgor [58] is a peer-to-peer hard disc video recorder based on the overlay network Igor. Our design goal was the seamless integration into the video disc
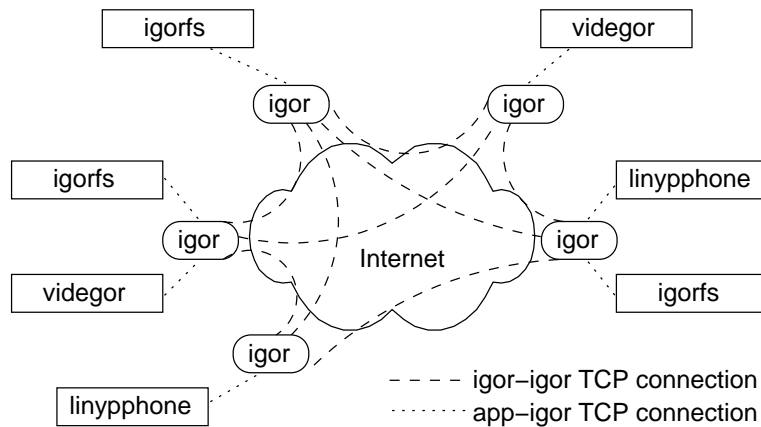
Figure 8.2: Application Connectivity with Igor

recorder user interface users are already familiar with[95, 29]. With its help it is possible to record missed broadcasts from the past and coordinate recordings in the future, i. e. much more user requests can be fulfilled with videgor than with non-networked video disk recorders. This works by using the distributed resources of many networked Videgor systems in a completely decentralized way. The distribution and decentralization was possible by using the key based routing interface from Igor.

Videgor is an extension to the Video Disk Recorders (VDR) from Klaus Schmidinger [149]. VDR is a program that enables a Linux PC to act as a video disc recorder with time shifting capabilities. Videgor extends the VDR to a distributed video recorder by providing three plugins and by automatically connecting the participating devices via Igor. The three plugins are presented below. With this peer-to-peer system the Videgor recorders can share past recordings as well as program data and can further coordinate future recordings (similar to [156] and [118]).

Scheduling

Users can request broadcasts to be recorded. If on different channels, these broadcasts can overlap. Users on other systems may have other requests, and may attach different priorities to each requests. The task of the distributed scheduling algorithm is to maximize the fulfilled user requests with the available receiver cards in the distributed system. The algorithm should also consider the priorities users gave different recordings. Furthermore, the number of recordings scheduled should increase with the number of users requesting this broadcast to be recorded.

Each Videgor system may have a number of receiver cards. Such a receiver card can usually be tuned to a single transponder (frequency), but multiple

channels can be broadcasted on one transponder. Therefor channels on one transponder can be recorded in parallel. Channels which are transmitted on different transponders usually can not be received simultaneously, because receiving nodes usually have only one tuner[2].

The problem of distributed scheduling can be mapped to the Knapsack problem, which is NP-hard [108]. In [95, 29] some approximation have been presented, which can easily be implemented with the Igor overlay network.

The scheduler plugin for the VDR implements these ideas. With this plugin it is possible to virtually record multiple broadcasts, even if the local receiver card is not capable of that. In order to compute the estimation of the schedule, the scheduler plugin uses two kinds of messages (see figure 8.3):

Request/Commit The RC-message is used to communicate the (aggregated, see below) user wishes (requests) as well as the current scheduling decision.

Suppress Nodes further up in the hierarchy (see figure 8.3) can change commitments of nodes further down by issuing suppress messages.

RC-messages have a target as the destination address and are sent via Igor with upcall flag set to true. This target is computed as a hash over the minute the schedule is computed for, i.e. for every minute, there is a seperate tree. This way, the load for figuring out the schedule approximation is distributed among all nodes. To further reduce the load on nodes further up in the hierarchy, intermediate nodes aggregate the RC-messages. With such an aggregation, the load on all nodes in the aggregation tree (see section 6.2.3) is similar.
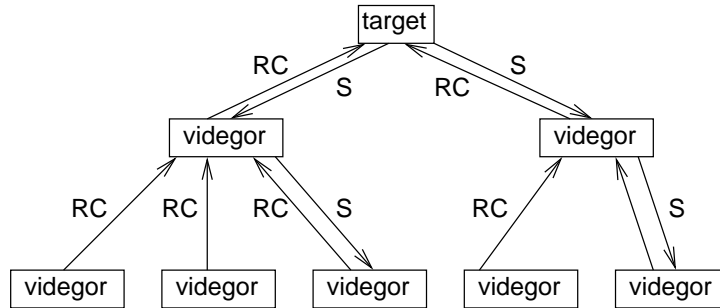

Figure 8.3: Example of Videgor Scheduling. The tree shown is also an Igor aggregation tree (see section 6.2.3).

If a node receives multiple RC-messages and detects imbalances between them (i.e. the node can improve the schedule for its subtrees), it sends suppress messages downwards. These suppress messages have the downwards node address as destination and are sent with upcall=false. Upon reception of suppress messages, further downward suppress messages may be triggered.

---

[2]As mentioned before, it is possible to have more than one receiver card per system.

The described scheduling algorithm uses some important features of the Igor key based routing system to approximate an optimal schedule for the distributed video recorder videgor. First, the route convergence of key based routing systems is used to do efficient aggregation. Since Igor employs PRS/PNS, the aggregation is likely to happen early on a nearby node. Second, the upcall concept is used to do computation on intermediate nodes, which would be impossible with iterative key based routing systems or DHTs.

Video Data Transport

A second plugin is the video data transport plugin. The task of this plugin is to fetch recordings from other videgor nodes. This is especially useful in conjunction with the scheduler plugin: After the scheduler plugin coordinated different recordings, the video data transport plugin can exchange the results.

The basic idea is similar to the scheduler plugin. Each node announces its local recordings in multiple aggregation trees as follows. The trees are rooted at the hash of the concatenation of the channel name and the minute the recording was done (e.g. the value of hash("CNN" + "2007-09-23t22-30") will become the root of one aggregation tree). This way, for every channel and every minute there is a separate tree. Since the hash values are distributed evenly, the overall load is distributed evenly too. Further, all intermediate nodes in these trees aggregate the announcement messages to reduce the load. Each announcement contains the node identifier from the announcing node and a list of hashes of groups of pictures in that minute. The video stream broadcasted is subdivided into GOPs (group of pictures). The video data transport plugin computes hash values over these GOPs and announces the hash values (see figure 8.4).

To transfer video data, the node requesting data sends a message in the appropriate aggregation tree. The first node with an answer sends a reply containing GOP-hashes together with node identifiers. The requesting node then directly contacts the announcing node and the transfer of the video data is the reply, as shown in figure 8.4.

Electronic Program Guide

Some channels broadcast the Electronic Program Guide (EPG) together with their normal program. The overview contains at least title of the broadcast, start time and duration. Additionally, EPG can also transport other meta data like detailed description, aspect ratio and sound format. The task of the EPG plugin is to transport this EPG data between videgor instances. Similar to video data, EPG data can be received only for the transponder currently tuned to. However, with this plugin, videgor nodes are able to receive all EPG data that they have channel identifier entries for. This also includes EPG entries from the past. Such old EPG entries are especially useful in conjunction with the video data transport plugin, which can fetch such past broadcasts from other videgor nodes.

The EPG plugin sends two different kinds of messages over the Igor network:
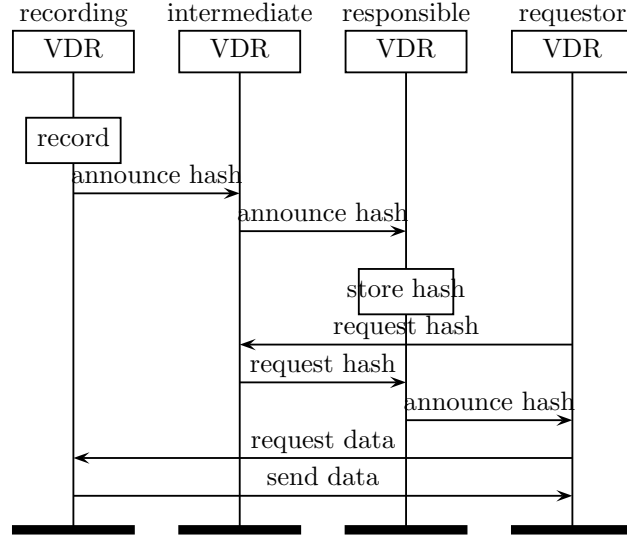
Figure 8.4: Message Sequence for Videgor Video Data Transport

Data and request. Request messages indicate for which channel and which time the message requests data. The granularity of time within the EPG plugin is one day, i. e. EPG data can be requested per channel per day. This decision leads to reasonable sized messages. A hash over channel and time is again the destination identifier of such requests. The upcall flag is set to true for request messages in order to allow intermediate nodes with proper knowledge to answer early.

Data messages on the other hand contain EPG data. They are sent for two different purposes: Announcements and replies to requests. First, regular announcements are sent. These regular announcements have the upcall flag set to true so that intermediate nodes can aggregate many announcements and take load from the aggregation destination (hash value of channel and day). Second, data messages are used as replies to request messages. Here the destination is set to the requesting node and the upcall flag is false. In this way, the message is delivered directly to the requestor.

8.4.3   LinyPhone

Linyphone is a peer-to-peer voice-over-IP application[46, 47]. It is mainly targeted at mesh networks, but is in parts based on key based routing and can therefore run on top of Igor. To initiate a voice-over-IP connection, the Session

Initiation Protocol (SIP [157, 143]) is used. SIP uses a central server (the registrar) to map the identities of users to an IP address where they are currently reachable. Linyphone uses a structure similar to distributed hash tables on top of key based routing to deliver a similar service as the SIP registrar.

8.5 Summary

This chapter dealt with the deployment of Igor. First some details of the build process, then the usage of the Igor daemon have been described. The file system IgorFS as an application to Igorhas its own part in this work. Videgor, another application we have designed and implemented to run on top of Igor, is explained in this chapter and finally Linyphone is mentioned.

Chapter 9

Conclusions

There yet remains but one concluding tale
Aleksandr Pushkin

9.1 Future Work

The development of Igor is not finished. Many more applications are imaginable for an overlay network like Igor.

9.1.1 Integration with Scalable Source Routing

Scalable Source Routing (SSR[59]) could use Igor as tunneling network between smaller mesh networks as shown in figure 9.1. Key based routing systems could be used to store larger amounts of messages, and together act as a NNTP[52, 66] server. Content distribution systems [169, 57] already use key based routing systems. Another possible application are anonymizing systems like TOR[42], where the directory service might benefit from the distributed nature of key based routing systems as well as the anonymized routing itself.

At the backend side, an integration with key based routing protocols running at lower layers (like SSR[59]) could be possible. For applications, this would have the advantage that they do not have to care on which key based routing system they run on. Applications could seamlessly communicate even if they are attached to very different networks and key based routing systems, like application 1 and application 2 in figure 9.2.

9.1.2 Control Plane

To ease the debugging, a control protocol similar to ICMP[25] might be included in the Igor message system. Tracing routes and controlling latencies (echo request/echo reply) and the transport of error message are possibe applications
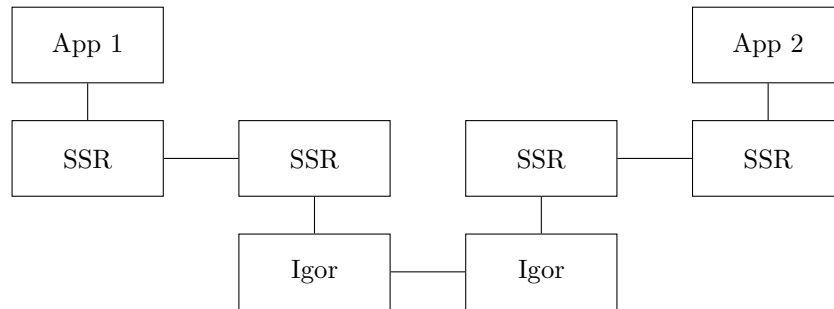
```
┌─────────┐                                                    ┌─────────┐
│  App 1  │                                                    │  App 2  │
└────┬────┘                                                    └────┬────┘
┌────┴────┐   ┌─────────┐        ┌─────────┐   ┌─────────┐     ┌────┴────┐
│   SSR   ├───┤   SSR   │        │   SSR   ├───┤   SSR   │
└─────────┘   └────┬────┘        └────┬────┘   └─────────┘
              ┌────┴────┐   ┌─────────┐
              │  Igor   ├───┤  Igor   │
              └─────────┘   └─────────┘
```

Figure 9.1: Possible Tunneling of SSR trough Igor

here.

```
┌─────────┐                                                    ┌─────────┐
│  App 1  │                                                    │  App 2  │
└────┬────┘                                                    └────┬────┘
┌────┴────┐   ┌─────────┐   ┌─────────┐   ┌─────────┐
│  Igor   ├───┤  Igor   ├───┤   SSR   ├───┤   SSR   │
└─────────┘   └─────────┘   └─────────┘   └─────────┘
```
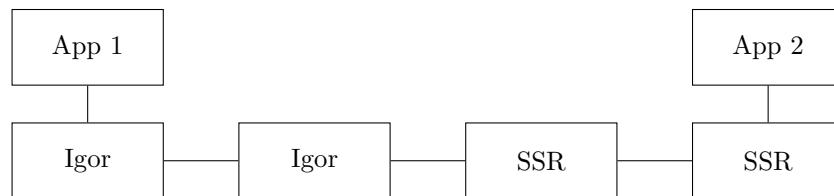
Figure 9.2: Possible Interaction between Igor and SSR

9.1.3   Firewall and NAT Traversal

Although the traversal of firewalls and Network Address Translation (NAT) devices was part of many design decisions, it is not fully implemented yet. However, there exists libraries that ease this task and integration of such libraries should be not difficult. Further, since message forwarding is the main task of the Igor overlay network, indirect message forwarding is not a problem in case NAT/firewall traversal is not possible. Through the plugin concept, such indirect connection can be penalized during normal routing.

9.1.4   Bootstrapping

The current Igor implementation uses just one node for bootstrapping. In case this bootstrapping node is not available for any reason, the startup of the Igor daemon fails. This behaviour will be changed in the way that multiple nodes can be given for bootstrapping purposes. During normal run time, the Igor process will memorize good (either latency-wise, because they offer similar services or because they are close in the virtual metric) neighbors and try them during the next startup.

9.2 Summary

The second part of this work introduced the overlay network Igor. After the design goals were laid out, the design decisions of the key based routing system has been described. Adaption to the Internet and service oriented routing have been important points here. The implementation and test of Igor has been reviewed. Applications based on the key based routing system have been illustrated, including applications developed by us. In this section, it has been shown how development on Igor could continue.

Part III

The Decentralized File System IgorFS

Chapter 10

Goals

This chapter will describe and justify the IgorFS design goals.

The aim of a file system in general is to map the user interface (system calls like `open()`, `seek()` and `read()`) of individual files to the underlying infrastructure, often a device with fixed block size. Further it is the task of the file system to allow the organization of files into directories and grant or deny read and write access to these files and directories. Often the tasks of the file system are accomplished by defining directory blocks, data blocks and meta-data blocks (in some file systems called inodes, see section 4.1.3), where information on how to find other blocks are stored.

The idea behind the distributed file system is to share access to data between multiple systems. IgorFS will prove that there are more efficient ways to share files than to copy the entire file over the network.

The IgorFS system should work without a central authority. Such a central entity would introduce a single point of failure and a performance bottleneck. Further this work assumes that the data handled is confidential by default. Therefore the file system should ensure that the data stays confidential. Unauthorized access to information is prevented and the authenticity of information is checked.

The scenario that is envisioned has large files, which change often but the changes are small compared to the overall file size. Every file has a designated writer (the creator/maintainer of that file), while there are potentially many readers of each file. The owner of the file must be able to determine who is able to read the file.

10.1   Security

IgorFS should work over untrusted networks, so security is a crucial part of the design. One important point is the assurance that the data stays confidential while it is in transit. Confidentiality of the data while it is on permanent storage would be an additional benefit. Such confidentiality should be ensured by using state-of-the-art encryption functions.

For readers of data it is important that they receive the data as it has been written, i. e. that the data item is authentic. This is not obvious in distributed systems and requires integrity protection mechanisms. The integrity assurance mechanism should cover every data and meta-data item and must therefore effienciently cope with large amounts of data. It is not feasible that the reader verifies each read data item with the publisher.

On the other hand, the writer (publisher) of data needs some control over who is able to read the data. Reading data without permission from the publisher should not be possible, i. e. the publisher must authorize reading its data from IgorFS. However, IgorFS should not be able to solve the Digital Rights Managment (DRM) problem, i. e. once a user has copied the data to a location outside IgorFS, the file system should no longer protect the data.

10.2   Distributedness and Decentralization

IgorFS should be a distributed system by the basic usage scenario, because it should be designed to exchange data between systems. These systems are connected via the overlay network Igor.

Further, IgorFS should also be a decentralized system, in order to avoid single points of failure and to increase robustness in case some nodes in the IgorFS network fail. Also, no central trust issues should be introduced, because different readers and writers may not necessarily agree on shared trust anchors.

Last but not least IgorFS is also a research project. One objective of this project is to prove the usefulness and performance of the overlay network Igor.

10.3   Scalability and Efficiency

The distributed and decentralized file system IgorFS should be designed in a way that utilizes the scalability properties of structured overlay networks and key based routing systems. This means that IgorFS must function efficiently with a couple of nodes as well as with thousands of nodes. Moreover, if thousands of nodes join the network, the resources of these nodes must be used in a shared way to increase the overall performance of the network.

Data in the file system IgorFS is expected to be large and changed often, however the changes are small compared to the overall size of the files. That means, between different versions of data in IgorFS there is a lot of redundancy. The system must therefore efficiently detect such duplications and avoid double

storage and double transmission of such duplicates. Figure 10.1 shows how current systems handle such situations and how IgorFS can do.
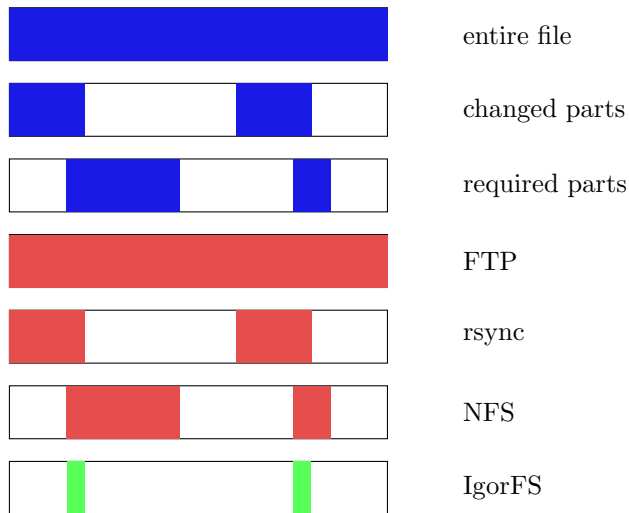

entire file

changed parts

required parts

FTP

rsync

NFS

IgorFS

Figure 10.1: Efficiency Improvement with IgorFS

10.4   Easy Deployment

In order to be useful for practical applications, IgorFS must not have difficult installation procedures or extensive external requirements. The system should not require undue modifications, neither on the target system nor on the applications using the file system. In order to use unmodified applications on top of IgorFS, the systems strives for compatibility with the POSIX standard.

It must be possible to do a gradual deployment, i. e. IgorFS must be useful from the first installation with one writing and one reading instance towards thousands of deployed systems.

Chapter 11

Design

> *Digital files cannot be made uncopyable,*
> *any more than water can be made not wet.*
> **Bruce Schneier**

This section will explain the design decisions made for IgorFS, based on the design goals laid out in the previous chapter. It covers the interface to the applications using the file system as well as security aspects. Also the internal representation of file system objects is explained together with their handling in memory and on disk. One of the key issues of IgorFS, the cutting of data blocks is introduced and the modularization of the implemented system is justified.

11.1   Interface to the Applications

Traditionally file systems are implemented as part of the operating system kernel. Section 4.4 showed the disadvantages of this option and how user space file systems offer alternatives. For IgorFS, the Parrot (see section 4.4.2) way to implement the file system was ruled out because exclusively using the `ptrace()` system call was deemed to be too intrusive for applications. To use a virtual file system layer like Gnome-VFS (see section 4.4.3) was not an option either, because that would restrict IgorFS to one special framework.

Therefore FUSE is used as interface between IgorFS and applications. This way, applications use the regular kernel interface and do not have to be modified (neither by introducing a VFS library nor by intercepting the `ptrace()`-call) at all.

On the file system side, FUSE works by registering a number of functions, mostly equivalents of functions at the VFS layer like open, read, write, and calling the main FUSE event loop. Then, FUSE creates a thread for each new request from the kernel and calls one of the registered functions.

11.2   Security

Data and meta-data stored in the file systems will, at some point, be put into chunks of data. To reach the security objectives of Igor (see section 10.1), cryptography is used at this block level.

We assume the following standard cryptographic primitives:

- Cryptographic hash function and

- Symmetric encryption and decryption.

Every single piece of data is part of a *block*. Note that at this point the size of the block does not need to be fixed. Each block has an unique identifier, gained by hashing the content of the (encrypted) block with a cryptographic hash function. The result of the hash function will be called the *identifier* or the *ID* of that block. This implies that the content of a block can not be changed (given standard hardness assumptions of the hash function). When something in the file system changes, a new block is created. The old block is available until everybody has deleted it. This means that it is an inherent feature of the file system that access to snapshots from previous points in time is possible, as long as the data blocks are cached in the system.

The content of every block will be symmetrically encrypted[1]. The key for the encryption process is derived from the hash result described above, i.e. the key is depedent of the content of the block. With this procedure, the same block is always encrypted in the same way, independent of other factors. The encryption key is not guessable by an attacker, but deterministic in case the same block is encrypted somewhere else.

To read anything from the file system, one needs first the identifier of a block. This is necessary because of the properties of the used hash function: The identifiers are not guessable. Note that if the content of a block is present, the ID can be computed. Since in that case the content of the block is known (but still encrypted as we will see later), there is nothing to be gained.

The second thing necessary to read a block is the key to decrypt the block. Table 11.1 formalizes the process.

11.2.1   Encapsulation of Cryptographic operations

All the cryptographic operations that are performed in IgorFS are encapsulated in a separate module. This module provides the following cryptographic services:

1. Symmetric encryption and decryption (see section 2.2). Given a fixed length key k, the module transforms arbitrary length of clear text input data B into encrypted data (encryption): $B_E = E_k(B)$. Because the the reverse operation $B = D_k(B_E)$ (decryption) uses the same key k, the cipher is called symmetric. Some symmetric cipher algorithms work in

---

[1] Asymmetric cryptography would be possible, too

Description	Formalization
A block (clear text)	B
Encryption function with key k	$E_k()$
Hash function	$H()$
Block, as transported or stored	$B_E = E_{H(B)}(B)$
Block identifier	$\text{ID}(B_E) = H(B_E)$
Block flag	f_b: `i` – indirection; `d` – data block
Indirection block	List of $(H(B_E), H(B), f_b)$-tuples
Human readable file name	N
Directory flag	f_d: `d` – directory; `f` – file entry
Directory entry	$V_i = (N, H(B_E), H(B), f_d)$
Directory	List of V_i
Decryption function	$D_k()$

Table 11.1: Formalization

a block-by-block manner. If the input data is not a integral multiple of the cipher block length, then the input data is padded to the next integer multiple (see section 2.4). The padding will include the original length of the input data. The way padding is done, the output data may be larger than the input data, but at decryption time the original data can be restored.

2. Hashing and verification. Hashing (as described in section 2.1) is the procedure of compressing an arbitrary amount of input data into a fixed sized output: $h = H(B)$. Further properties of cryptographic hash functions inlcude that it is hard to (a) create two sets of input data that hash to the same value and (b) hard to create a second set of input data that hashes to the value of a first one. Verification is the complementary operation. Given a block of input data B and a hash value h, the module checks whether the result of the hash function is equal to the one provided: $h \stackrel{?}{=} H(B)$

3. Combination of the above. Some cryptographic operations frequently occur together in IgorFS. Input data is often hashed to get an encryption key, then encrypted with that key and hashed again to derive the block identifier. This module provides the most often needed combinations as functionality: $(B) \Rightarrow (E_{H(B)}(B), H(E_{H(B)}(B)), H(B))$. The reverse operation is also supported. Here, the module expects an encrypted block as input along with a block identifier and a decryption key: $(E_{H(B)}(B), H(E_{H(B)}(B)), H(B)) \Rightarrow (B, \text{bool})$. The boolean value is set to true if all verification operation have been successful. It will decrypt the block and verify the outer and the inner hash.

The encapsulation of all cryptographic operations has a number of advantages. To increase the processing speed of IgorFS, cryptographic co-processors, once they are widely available, can be used. Also parallelism for normal multi-CPU systems can be implemented in this central facility.

A second advantage is that the selection of cryptographic algorithms is pooled in one place. This is useful because not all combinations of cryptographic algorithms have the same level of security. Useful combinations of algorithms are grouped into crypto suites. Cryptographic algorithms also need to be exchanged from time to time, either because of advances in crypto analysis or because of legal changes (export, import or usage restrictions).

Currently, IgorFS is designed to use SHA-256 as the hash function and AES-256-CBC (see sections 2.2.2 and 2.3.2) to encrypt and decrypt. Both algorithms are considered secure right now and in the near future [11]. Once one of the algorithms is considered broken, a change is easily implemented by defining a new crypto suite. The cipher block chaining (CBC) mode (see section 2.3.2) is appropriate, because it does not have the weaknesses of electronic code book and blocks are always encrypted and decrypted at once. Random access into blocks is not necessary. CBC mode requires an initialization vector to encrypt its first block (note that this block has nothing to do with data blocks from IgorFS, but with the block cipher characteristics of AES). Regarding initialization vectors, it is important that the same initialization vector IV (see sectionsec:ciphermod) is never used with the same key and different data. Because each block is encrypted with a different key, the initialization vector can be kept constant.

11.2.2   Authentication

As described above, the blocks building up files and directories are authenticating themselves in the way that the identifier of the block ensures, with the use of the hash function, that the block has not been tampered with. This implies that the authenticity of the block identifiers must be protected. These block IDs are contained either in indirection blocks or directory entries. Recursively, these indirection blocks or directory entries are themselves saved in blocks which are identified by self-authenticating IDs. In the end, everything is authenticated by the identifier of the root directory. This again means that a trusted anchor point of entry is necessary. Such an anchor point can be created by at least two ways:

- The first block identifier is transported over a trusted out-of-band channel. Such a channel can be personal contact, cryptographically secured e-mail transport or a secure web page.

- The first block identifier is transported over an untrusted out-of-band channel, but secured with other authentication schemes like a signature with a trusted key. How this key is trusted is outside the scope of IgorFS.

11.2.3 Authorization

The authorization system inside the file system is based on the fact that every single piece of data and meta-data is encrypted. The authorization to read a block equals the ability to decrypt that block. This means the security of the authorization rests on the security of the block encryption and not on the trust in a (third party) authorization system. There is no explicit authorization mechanism to allow or deny to change a file. Since blocks are immutable (changing the content of a block changes the identifier), it is not possible to change an existing file and therefore an authorization for change is not necessary. Changing a file means creating a new file, which in turn is a change to the parent directory (i. e. creation of a new parent directory). The additional required storage is limited because blocks shared between the old and the new version are encrypted and therefore stored in the same way. The ability to write new blocks is not limited by the authorization system but by the amount of local free disk capacity. Older blocks are not affected by newly written blocks.

11.2.4 Confidentiality

All data in the system is regarded and treated as confidential by default. There are no unencrypted blocks of data, every single piece of information is only stored and transfered in encrypted form. To decrypt and read a block in order to serve local user requests, the matching key is necessary. Such a key can be learned by two mechanisms:

- Receiving it via an out-of-band mechanism (already mentioned in section 11.2.2).

- Reading it in an upper level structure. For example the key to a file is listed in the directory together with a block identifier for that file.

Note that intermediate systems do not have the key for the blocks they transport and therefore move around opaque data. The handling of unencrypted data is limited to two points. First, the initial publisher writes the data it wants to publish towards the daemon. The daemon encrypts the data and creates the block structures described above. Second, the final user (reader) of the data receives the blocks in an unencrypted form. Again, the daemon does the decryption and hands the data to the user.

11.2.5 Trust Issues

A publisher of data wants to keep the data confidential except for legitimate users. The publisher has to trust:

- first and foremost, in the strength of the cryptographic algorithms.

  - If a known-plaintext attack to the encryption function is possible, not much is gained for an attacker, since the keys are based on the plain text in many contexts.

- If a known-ciphertext attack is possible, the file system can be considered broken.

- If the collision resistance of the hash function is not given, attackers may have the possibility to publish different data in different contexts.

- If the used hash function is not second-pre-image resistant any more, attackers may introduce non-authentic data into the system.

- If the one-way property of the hash function is weakened, attackers may learn confidential keys to data.

- legitimate users of data. These users have access to the unencrypted data (that is the purpose of the whole system) and the keys that let them decrypt the data. It must be ensured by measures outside the IgorFS that these users stick to the rules set by the publisher.

- administrators of the system of said users. Since the user have access to the data, the administrators can gain access, too.

On the other hand, the publisher of data does explicitly not have to trust

- intermediate systems, as they merely pass on encrypted data blocks.

- administrators of intermediate systems, for the same reason.

- other users of the same system as the legitimate user. Here the local (local to the legitimate user) file system rules (enforced by the local administrator) ensure that no access is granted.

11.2.6   Examples

This section walks through the encryption and decryption mechanism and uses the formalization shown in table 11.1. A producer of two files (1 and 2) wants to publish them in the above described file system. Both files are smaller than one block (to skip the indirection block steps). The producer then computes: The two encrypted data blocks $B_E^1 = E_{H(B^1)}(B^1)$ and $B_E^2 = E_{H(B^2)}(B^2)$, the two block identifiers $H(B_E^1)$ and $H(B_E^2)$. Then a directory is created with the two entries $(N^1, H(B_E^1), H(B^1), f_d^1)$ and $(N^2, H(B_E^2), H(B^2), f_d^2)$. This directory forms a third block B^3, of which $H(B_E^3)$ and $H(B^3)$ are computed as above. The three blocks B_E^1, B_E^2 and B_E^3 are pushed into the network.

The producer may then decide to sell access to the two files to an user A. The user pays and receives the tuple $(H(B_E^3), H(B^3))$ together with the indication that this is a directory. It may also receive a signature of the producer to check the authenticity of the tuple.

The user is then able to fetch the block B_E^3, since she knows the block identifier $H(B_E^3)$. With this identifier she is able to verify the authenticity of the block by re-computing the block identifier $H(B_E^3)$. She can then decrypt the block, since the decryption key $H(B^3)$ is also known. The decrypted block

is the directory containing further file names N, block identifiers and decryption keys. With these information the user is able to fetch the blocks, verify, decrypt and use them.

At some other point in time, the producer decides to sell another directory with files 1 and 4 in it to user B. The directory then contains the two tuples $(N^1, H(B_E^1), H(B^1), f_d^1)$ and $(N^4, H(B_E^4), H(B^4), f_d^4)$. Note that the entries for file 1 are the same as for user A. User B, however, is not able to read file 2, since user B is not able to decrypt B_E^2 because the key is not known to her. Even the mere existence of file 2 and the directory published to user A is kept secret. Everything user B can learn by chance is that the blocks B_E^2 and B_E^3 exist and have been transfered.

11.3 Handling of File System Objects

To answer requests from the FUSE layer efficiently, IgorFS needs an in-memory representation of file system objects. Otherwise, for each request from FUSE, the full process of encryption/decryption would be necessary, which is not feasible. This representation needs to fulfill at least two requirements: First, application requests received via FUSE must be dealt with efficiently. Second, the transformation into blocks and back should be simple.

11.3.1 Files

Files consists of, potentially many, blocks. The size of each block will be chosen

- large enough so that the overhead of requesting and transferring the block is small compared to the transfer of the block itself.

- small enough so that random-access read patterns can be served fast.

- large enough so that the overhead of maintaining the block structure is reasonable.

As desribed above, every block is encrypted: For each block, a distinct and unique encryption key is selected. This key is based on the content of the block[2] and derived by applying the hash function to the content of the block.   See table 11.1 for a formalization of this procedure. This ensures that if two entities store the same data in the system, they encrypt it with the same key. The result is that every unique block is stored only once. Otherwise it would be possible that the same block occupies storage with many different encryption keys. Again see table 11.1 how this key is derived.

Every directory entry (see section 11.3.2 below) points to a single block and contains the decryption key to that block. The block contains either the final file data, given the entire content of the file fits into one block or a list of identifiers

---

[2]This is not strictly required. The key can also be selected randomly to avoid unforeseen weaknesses by the combination of hash function and encryption function. Then, however, the property that every block is stored only once is lost.

of other blocks and the keys to decrypt them (*indirection block*). The latter allows the construction of a B-tree[6] like structure to implement large files, since indirection blocks can point to other indirection blocks again.

Figure 11.1 shows how indirection influences the maximum allowed file size. The figure assumes that each pointer requires 72 octets (8 octets offset, 32 octets block identifier, 32 octets decryption key). Note the logarithmic scales of the plot.
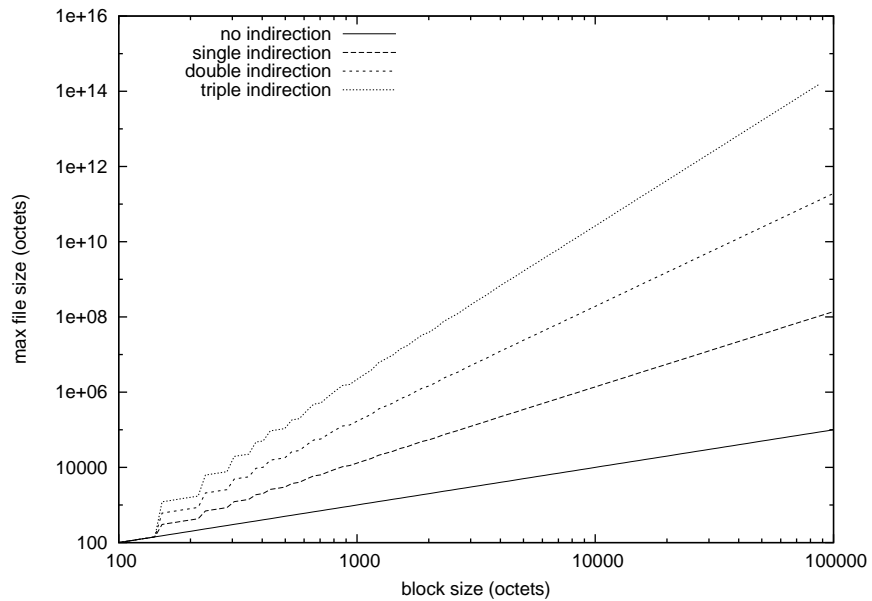

Figure 11.1: Influence of Indirection on Maximum File Size

Since these blocks are identified by the result of a cryptographic hash function, they are self-authenticating in that way. Because of the second pre-image resistance of the hash function it is not possible for an intermediate system (read: Eve the attacker) to modify the block without changing the block identifier. As mentioned above, by changing the block identifier an entirely different block is created. The same logic applies to indirection blocks and directory blocks. The idea of encoding files in this way is loosely based on content hash keys (CHK) described in [18].

A specialty of files in Unix-like operating systems is that files can contain holes. A hole in a file is an unallocated area where no write access ever happened. Such areas can emerge if applications `seek()` over the end of the file and write data. IgorFS can efficiently deal with such holes because indirection blocks always store the offset where the pointed-to-block belongs. Missing bytes between successive blocks represent holes in the file. Note that such holes neither require hard disk storage capacity nor network bandwidth.

11.3.2 Directories

At the first sight directories are indistinguishable from other files: They are made of encrypted blocks and one needs the identifier of these blocks and the key(s) to fetch and decrypt them. The difference between directories and ordinary files lies in the fact that directories are interpreted by the file system itself. Directories contain pointers to other blocks (block identifiers), keys to these blocks and human readable names to identify these pointers together with a flag indicating whether the pointed-to-item is an ordinary file or another directory.

11.3.3 Directory Layout

In memory, directories are handled as tree, using the entry name as key. The C++ standard template library (STL) allows efficient implementation. The in-memory representation as a tree avoids slow reactions to FUSE requests if directories contain many entries.

Once directories are converted to blocks (serialized), the content is a linear list of entries. This has the advantage that directories can be treated exactly like files and all lower layers do not need to distinguish between files and directories.

Serialized directory entries consist of the following items:

- The first item is the name of the the the entry. IgorFS does not need to deal with encoding problems of entry names, since this problem is already handled at the FUSE layer.

- A flag indicates the type of the entry. Currently supported types include files, directories and soft links. Hard links are not supported because their POSIX semantic is difficult to map to the IgorFS context. Hard links are two directory entries (potentially in different directories) that point to the same file. If the file is changed, the change is visible through both directory entries. In IgorFS, directory entries point to block identifiers. Altough it is possible that two directory entries point to the same block identifier, changing a file creates a new block (including a new block identifier). The change would not be visible through the other directory entry, because the block identifier would not change their. Device files and named pipes have been left out too because they are highly dependent on the host system, which may vary in a distributed system like IgorFS.

- The block identifier and decryption key to read the entry are serialized next. This is only required for files and directories. Soft links are stored directly in the directory entry.

- For files, the overall length of the file is noted.

- Other meta-data including the creation and modification time of the entry follows. Storing access time is not useful because that would require very frequent updates of block identifiers and associated operations. Also access

rights are stored, however their usefulness is limited. Anyone in possession of the key to decrypt the directory can locally alter the access rights. The most meaningful of the POSIX access bits is the executable bit, because it allows to store executable files on IgorFS.

Once a directory is serialized, it is treated like any other serialized content, i. e. it is subject to block cutting and cryptographic operations. The result will be an identifier and decryption key for this directory and *all* of the content below. These numbers can then either be embedded in the parent directory or be transported to the subscriber by another mechanism.

11.4   Block Cut

Block cutting is the operation of taking data written by the user processes and chop it up into pieces which are transferable through the network. The next section lays out some goals to select the best option for IgorFS.

11.4.1   Requirements

Some requirements for the cutting operations are:

- The block size must be reasonable. Blocks that are too large take too much time to transfer and thereby generate latencies. If users want only parts of that block, other parts are transfered without any need. On the other hand, if block sizes are too small, a lot of overhead is created by transferring lots of small blocks. The optimum depends on expected usage patterns, expected bandwidth and the overhead to transport one block.

- Equivalent pieces in different files should be detected, i. e. they should be chopped the same way.

- Equivalent files should be cut in the same way at different hosts. That means the algorithm should be deterministic.

- Small modifications in a file should influence the cutting algorithm only locally. That means that when the content of one block changes, most block boundaries in the file should stay the same.

- Insertion or deletion of some data should influence block boundaries as slightly as possible. This requirement is related to the previous.

- Application behavior can be honored. The application may know how to store the file: IgorFS could use the block boundaries used by the application writing the file.

- Knowledge about the general file structure can be incorporated. For example:

- Cut out consecutive 0x00s, a pattern which occurs in binary files as well as in files with holes. Further "natural" cutting points in binary files are the transitions between different sections (e. g. text segment and data segment) or the boundaries of function code.

- Text files can be cut preferably at line endings. This concept is especially useful for source files, where changes happen often in one or many consecutive source lines. Version control systems like CVS [49] use this concept.

- Video files could be cut at Group of Pictures (GOP) boundaries.

- Some databases used by the Sequence Retrieval System (SRS [50]) have the marker `\n\n` as separator between data items. It is useful to cut SRS files at these points, because data items are always read as a whole.

- Implement an interface for the application to explicitly give hints to the cutting algorithm.

- Because the final algorithm must be fast enough to run on large amounts of data.

These requirements in part contradict each other, e. g. the algorithm can not honor application behaviour and be completely deterministic on data at the same time. The next sections will list possible solutions and explain which tradeoffs have been taken.

Possible variants to cut data include:

- Cut data at fixed positions (fixed block length). File systems designed to work on hard disks cut files at sector boundaries, often 512 octets.

- Cut data at fixed, but user specified positions (allow adaption to `struct` lengths). Applications with fixed record lengths would benefit most from this option.

- Cut data at some (not all) line endings to improve storage for text-based files. Since identical blocks are stored only once in IgorFS, this option would be especially efficient if text files with small changes are the major workload for IgorFS.

- Cut data at `0x00` bytes in the file to ease storage of files with holes in them.

- Cut data at positions determined by an algorithm on the data.

A combination of the variants is possible.

Cutting at fixed data patterns (like `0x00` or `\n`) is ruled out because it would restrict IgorFS to specific file formats. The same holds true for approaches that try to understand the data at a higher level than octet sequences. Honoring application behaviour is difficult for two reasons. First, this would require an

interface between the application and IgorFS beyond the POSIX file system interface. The write pattern can not be used as such an interface, because the specifics how an application writes can be hidden by the kernel and by FUSE. The second reason against honoring application behaviour is that the data cut algorithm should be deterministic. Even if one file is written by two different applications (e. g. the orginal producer and a copy process), it should be cutted in the same way.

11.4.2   Rolling Checksums

To achieve these goals, it became clear that the cutting algorithm had to be data dependent only. However, format dependency was abolished because of the number of possible formats and the paradigm to design a general-purpose file system. Therefore a data dependent and format independent way to cut files is required.

Rolling checksums (e. g. Rabin hashes (see section 2.1.6)) have the property that they are

- easy to compute in general.

- easy to compute for a sliding window which is moved over the data.

The second property is important here: data outside the sliding window does not influence the computation of the checksum. This way, the checksum of similar strings in the data is identical, even if the data is preceeded by different strings.

The rolling checksum is used to find cutting marks in the data stream. Given the checksum algorithm outputs the values from 0 to x with a uniform probability and the desired average block size is b. Then a cutting mark is inserted every time the rolling checksum stays below the threshold of $t = x/b$.

Before blocks are actually cut, small holes in the data stream are replaced with a sequence of '0'-octets. At the same time, larger sequences of '0'-octets are replaced with real holes in the file.

After the block cutting is done, very small segments are joined again to avoid unnecessary overhead. Similarly, segments that are too large are split by decreasing the threshold.

Fixed Block Size

A fixed block size is a simple and deterministic algorithm to cut files. Available commodity hard disks use 512 octets as sector size and file systems designed for these split file content in (multiples of) the sector size. This scheme keeps most blocks constant if some data is overwritten. However, when some data (not a multiple of 512 octets) is inserted or deleted, many blocks (e. g. most of the blocks behind the modification) have to be changed. For the same reason, if two files differ only by an insertion/deletion early in the file, most of the

blocks stored are different. As stated above, one of the goals of IgorFS is to find similarities in independent files, the fixed block size approach is not suitable[3].

Adler32

The Adler32 (defined in [39], based on [55]) checksum is computed as

$$A = 1 + \sum_{i=0}^{n-1} d_i \quad (\mathrm{mod}\ 65521)$$

$$B = (1 + d_0) + (1 + d_0 + d_1) + \cdots$$
$$+ (1 + d_0 + d_1 + \cdots + d_{n-1}) \quad (\mathrm{mod}\ 65521)$$
$$= n + \sum_{i=0}^{n-1} (n - i) \cdot d_i \quad (\mathrm{mod}\ 65521)$$

$$\mathrm{Adler32}(d) = B \cdot 65536 + A$$

Note that the final computation does not need to be computed modulo 2^{32} to fit in a 32 bit value, because the result is always below or equal to $65520 \cdot 65535 + 65520 \approx 2^{31.9996}$. The definition used in this work is slightly different for two reasons. First, the checksum has to be easily computable from previous values. Second and more important, the checksum must be independent of the position of the data in the file. Therefore Adler32* over b octets with the last octet at position x (with $x \geq b$) is defined as:

$$\mathrm{Adler32}^*(d_x) = (\overbrace{b + \sum_{i=x-b}^{x} (x - i) \cdot d_i}^{B})\ \mathrm{mod}\ 65521 \cdot 65536$$
$$+ (\overbrace{1 + \sum_{i=x-b}^{x} d_i}^{A})\ \mathrm{mod}\ 65521$$

CRC and Rabin

Both CRC checksums as well as Rabin hashes (see sections 2.1.6 and 2.1.1) quickly achieve the desired equal distribution of hash values. The associated graphs have been omitted for that reason. However, both share with Adler32* the disadvantage that all octets in the sliding window need to be memorized in order to remove them once the windows is moved. Both checksums have some computation overhead compared to the XOR32 algorithm introduced next.

---

[3]This may not be true for very small blocks. However, very small blocks are not suitable because of the large overhead introduced as explained in the goals in section 11.4.1.

XOR32

Taken the file as continuous data stream of octets d_i, the XOR32 rolling check sum with bit width b at position x is defined as

$$\mathrm{RC}_x = \bigoplus_{k=0}^{b-1} 2^k \cdot d_{x-k} \pmod{2^b}$$

With this definition, the continuous (rolling) checksum is easy to compute. Given the checksum at position x, the checksum at position $x + 1$ follows $\mathrm{RC}_{x+1} = (2 \cdot \mathrm{RC}_x) \oplus d_{x+1} \pmod{2^b}$, i.e. no previous data is required. Both operations, one shift and one XOR, are very cheap to implement and fast to execute. The effects of data patterns are fading out: After b shift operations the influence of a single byte is gone. On the other hand, every bit position of the input data can affect every bit of the checksum.

11.4.3   Conclusion

The previous sections showed the design issues around splicing a continuous stream of data into blocks. Fixed size blocks can not cope with insertions or deletions of data amounts which are not a multiple of the block size. Section 11.4.2 also explained why it is not feasible that IgorFS is aware of multiple file formats. A similar argument holds true for direct hints by the application.

Therefore a data dependent but deterministic algorithm to cut data into blocks is the right design for IgorFS. Previous sections together with section 14.1 compare CRC32, Rabin Checksum, Adler32* and the XOR32 algorithms. Table 11.2 compares the results. CRC32 is not easily computable in a sliding window manner and is more complex than the other solutions. The Rabin and Adler32* approaches work with sliding windows. The computed values for Adler32* are not distributed evenly[4], which is required for proper operations. Rabin has a noticable computation overhead, too. Therefor the XOR32 algorithm is most suitable one to find cutting marks, given the desired block size is larger than 32 octets.

Feature	CRC	Adler	Rabin	XOR
Sliding Window	no	yes	yes	yes
Equal Distribution	yes	no	yes	yes
Small Overhead	no	yes	no	yes

Table 11.2: Comparison of Checksum Algorithms for Block Cut

---

[4]The reason behind this is the fact that the sums A and B do not overflow early enough

11.5 Snapshot

This section will described the goals of snapshotting in general. A snapshot of the file system conserves the state of the file system at one point in time. This is useful to export exactly this state to other peers or to revert to this state in case something goes wrong. Some Logical Volume Managers (LVM) provide a similar service. The difficulty in creating such snapshots exists because additional modification requests for the file system may arrive from the application while creating the snapshot. The goal is to design a snapshot mechanism that can deal with the problem efficiently. Generation of snapshots should not hamper normal performance significantly. It must also be possible to generate multiple snapshots in parallel.

11.5.1 Data Structures Necessary

To implement the snapshotting process several additional data structures are necessary. First of all, an epoch designator is introduced. The name and concept are borrowed from Java garbage collector systems [43, 102]. The variable `epoch` starts with the value 0 at program start-up and is incremented whenever a snapshot is started. One instance of this variable must be kept per mount point. The counter may overflow if sufficiently many snapshots have been generated. IgorFS handles this situation properly, however, no more than 2^{32} (assuming integers with a length of 32 bit) snapshots may run at once. Note that the epoch counter is an in-memory structure and has a local meaning only.

Second, all in-memory representations of file system objects (e.g. files, directories, links, ...) keep their current epoch. Each instance of an object belongs to exactly one epoch. On the contrary, not every object exists in every epoch.

Last, file system objects (FSOs) from previous epochs must be accessible somehow. For that, all file system objects can keep references to previous representations of themselves if such a previous object exists.

FSOs can contain multiple references to other FSOs (e.g. directories point to their content). The semantic of these references is now slightly changed: The reference points to an instance in the same or an older epoch. Figure 11.2 shows how FSOs and epochs are related. Note that not every FSO exists in every epoch.

If the file system object is current with respect to the global epoch (see section 11.5.2 how this is detected), then references to the child objects are always to the current epoch of the child object. On the other hand, if a FSO is not current any more, references to older epochs of children may exists. Figure 11.2 shows the concept.

11.5.2 Process

Read-Only Access

Accesses to the file system or to file system objects that do not modify the objects are not changed at all. That means that the introduction of the snap-
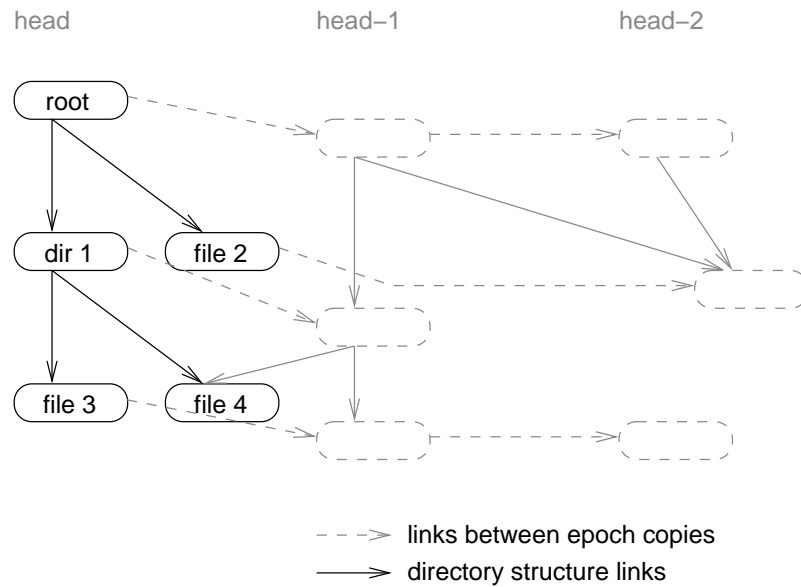
Figure 11.2: Connections Between File System Objects and their Epoch Copies

shotting process has no implication for read-only access. The reading process can lock the current head epoch of the object and has read access for it. This is true since the access time is noted in IgorFS. Otherwise each read access would be equal to a write access at least to the meta data of a file system object. Efficient snapshotting is one reason why the access time is not tracked.

Write/Modify Access

There is a distinction between memory blocks and disk blocks. During (over-)write access from user space, first the disk block has to be read from disk, decrypted and converted to a memory block. The the write call from user space can affect more than one memory block, therefore it is checked which memory blocks it will modify. Then the write call from user space is dissected into chunks, one chunk for every affected memory block. These chunks are written into the memory blocks and the blocks are marked dirty. The boundaries of memory blocks may be affected during this process, i.e. new memory blocks can appear because old ones are cut.

For write access (any access that modifies the object), two cases have to be distinguished:

1. The object was *not* modified since the object was made persistent the last time. In this case the epoch counter of the object is set to the current epoch and the write access can occur as normal.

2. The object *was* modified since it has been written to disk the last time. In this case, the previous version of the object is still required, because it has to be written out. That's why an in-memory copy of the object is created. After the copy is complete the objects' epoch counter is set to the current global one and the object modification can occur as described before.

The algorithm is shown in figure 11.3. If the object is not dirty (i.e. it has been written to disk since the last snapshot), no special handling occurs: The epoch counter is incremented and the write request is fulfilled. In case the object has been modified after the global epoch has been incremented the last time (i.e. the last snapshot is not yet complete), a copy of the object is created in memory. The old snapshot process can continue on the copy, while the current write request is served on the original. The object is dirty after both cases.

```
if (fso.is_modified()) then {
    fso.clone = clone_object();
}
fso.epoch_counter = current_epoch_counter;
/* write as normal */
```

Figure 11.3: Modifying Access with Epochs

A good moment in the write process to implement the epoch counter check is when the object is locked for write access. The write lock is a procedure that is necessary for all modifications of an object, because the implementation needs to deal with concurrency.

Writing to child objects invalidates the block identifiers for parent object, because the parent object in its serialized form contains the block identifier of the child object. This block identifier of the child object changes if the child object is modified. Therefore all parent objects need to be marked as 'dirty' when a child object is modified. This recursive setting of objects to dirty requires a lock on the parent object. To avoid lock contention on the objects closer to the root of the tree, two techniques are employed:

1. Truncation of the recursion. As soon as a dirty object on the way upwards is discovered, the recursion is stopped. This is justified because this algorithm ensures that dirty objects have dirty parents.

2. Overlapped locking. Locks on intermediate objects (between the object to-be-written-to and the root) are acquired as late and given up as early as possible. Figure 11.4 shows the algorithm used for that.

Start of a new Epoch

To start a new epoch (i.e. to induce the creation of a new snapshot), the epoch counter (kept per mount moint) is incremented. After the increment, the root

```
object.setdirty() {
    metawritelock(object)
    bool objectwasdirty = object.isdirty;
    if (object.isdirty)
        object.dirtyflag = true;
        parent = object.parent;
        unlock(object);
    if (objectwasdirty)
        parent->setdirty();
}
```

Figure 11.4: Algorithm for overlapped locking during `setdirty()`

block is queued to be made persistent. Note that after these two operations, normal file system calls can resume. Because of the mechanisms described in section 11.5.2, data that has not been written out yet is preserved. The persistence-making of objects can happen asynchronously. That means, that normal file system operations are not delayed because of the snapshotting mechanism. A special case occurs if the root object is not marked dirty. That means the file system has not been modified since the last snapshot and the result of this snapshot is the same as the result of the last snapshot.

Process Snapshot / Make Persistent

File system objects in main memory are volatile. In order to be persistent, they have to be stored to hard disk. Objects saved from main memory onto a hard disk drive or into the network are called persistent objects. As described above, the persistence making of objects is an asynchronous process, i. e. normal file system operations are not affected by it and continue in parallel. The process is invoked by the request to make the root object (the root directory) persistent. This request triggers requests to make the childen objects (files and directories) persistent too. To make the in-memory representation of a directory persistent, the following steps are necessary.

1. Serialization of the in-memory representation. That means, all fields of every directory entry (as described in section 11.3.3) have to be converted into a linear sequence of octets. Note that this includes the block identifier and decryption key for the sub-entries.

2. The linearized content of the entry is then subject to the block cutting procedure. The process has been described in detail in section 11.4. The main properties of the result are that it is deterministic and that octet sequences with similarities are cut in a similar way.

3. Encryption and hashing. Per design requirements (see section 10), data

blocks[5] are stored in encrypted form only. The process of encryption and hashing (see section 11.2) yields the block identifier and the decryption key for this directory itself. In case the directory was a sub-directory to some other directory, the new block identifier and decryption key can now be used in the parent directory.

4. Disk storage and network publication. Once the cut block is encrypted, it can be stored on the local disk. Blocks on the local disk are then published in the Igor overlay network (see section 11.7) in order to allow other nodes to fetch this block.

If the sub-object in the above mentioned directory is not a sub-directory but a file, the required steps are similar:

1. Serialization. Here, all data octets of the file need to be serialized and organized. The serialization of data octets themselves is straightforward: Each octet in a file is represented by that octet in the serialized stream. One exception are holes in files (see section 11.3.1). These are either represented by a sequence of octets with value 0x00 (if the hole is short) or are encoded by the number of consecutive zero-valued octets. The former are stored as regular data, the latter as a hint in the file's block list by the fact that the end of one block (given by the start offset of this block and its length) does not coincide with the start offset of the next block.

2. Block cutting. The process of block cutting splits the file's content in discrete blocks. This process is described in detail in section 11.4. If the block cutting process decides the file is not stored as a single block, an additional data structure becomes necessary. After all plain data blocks have been made persistent (that means encryption/hashing has computed a block ID and a decryption key and the block has been stored to disk), a list of (block-identifier, decryption key)-pairs is compiled. This list contains next to the just mentioned pair the offset of the block. The result is serialized. If the compiled list is too large to fit in one block (as again decided by the block cutting algorithm), the process is repeated recursively, i. e. the serialized block is cut in sub-blocks, which are stored separately and their block identifiers are listed. As soon as the list is complete, it is subject to encryption and hashing.

3. Encryption and hashing. Depending on the size of the file, either the entire file or its separate blocks are (maybe recursively) encrypted and hashed. In any case, as soon as the result of the encryption and hashing process is available, the parent object can be serialized.

4. Storage and publication. Encrypted data blocks are handled the same way as encrypted directory blocks. After storage on local disk other nodes are informed via the Igor network.

---

[5]Directory content is regarded as data in this context.

The algorithm in figure 11.5 shows that there are no differences between directories and files with regard to the process of creating persistence at this abstraction level. The distinctions are in the types of sub-items and the way the objects are serialized. After serialization, files as well as directories are linear streams of data. Directories are streams that are interpreted by the file system itself. As such, they are comparable to indirection blocks with names attached to each indirection. In figure 11.5 sub-items for files are data blocks or indirections blocks, sub-items for directories are other directories or files.

```
make_persistant(x) {
    foreach (subitem y of x)
        make_persistant(y);
    serialize(x)
    x'=block_cut(x)
    encryption(x')
    store_persistant(x')
}
```

Figure 11.5: Algorithm to Create Persistent Objects

11.5.3   Summary of Snapshot

The preceding sections showed the design of the snapshot system. This system allows the creation of fixed states of the file system without interruption of regular access.

The result of the snapshot process is a block identifier and a decryption key for the root directory. With these two values, the fixed state can be accessed at a later point in time. Since the block identifier of the root directory is indirectly based on the entire content of the file system via cryptographic hash functions, this block identifier also indirectly authenticates the entire content of the file system.

11.6   Block Cache

The amount of data handled by IgorFS in the scenario it is designed for is larger than the typical main memory of nodes. Therefore, and because data should be persistent across system restarts, an additional storage medium is required.

Second, IgorFS is designed to cache data on intermediate nodes. This creates no security problems for confidential data, since all blocks are encrypted. This caching on foreign nodes increases performance (data is available from multiple sources) and robustness (if one source fails, another one can be selected).

This section describes the design of the block cache component [146] of IgorFS. The block cache is responsible for permanent storage as well as intermediate caching.

An important design parameter for such a caching component is the cache strategy, which controls the sequence items are evicted if the cache becomes full. The least recently used (LRU) strategy is an often used one. Similar projects to IgorFS, for example [19], [18], [94] or [138], all use LRU as their main caching strategy. More advanced strategies like least value based on caching time (LVCT [73]) or object LRU (OLRU [63]) include the time a data item already stays in the cache and the size of the object. Both can increase the hit rate in the cache, at the expense of a more difficult maintenance. Since the success of a caching strategy highly depend on the usage patterns and the usage patterns for IgorFS are not yet clear, the more robust strategy LRU is used in the IgorFS block cache.

11.6.1 Requirements

In order to scale with the expected load, the block cache component must meet a couple of requirements. First of all, the block cache must be able to store at all, i. e. the block cache should not have artificial limits on data size. Since the overall data volume is high and the size of blocks is limited in order to restrict latency, the number of blocks is large too. The speed of all operations on the block cache (store block, retrieve block, reference block, expel block) must scale well with large number of blocks.

As mentioned above, there is the distinction between permanent storage and intermediate caching. At least for these two different kinds of storage a separate LRU-list should be kept. Blocks cached can be expelled from the block cache at any time, since it is expected that the block can be reloaded from the network on demand. Blocks which are in the block cache for permanent storage should only be evicted with consent from the file system user. If the limit for permanently stored blocks is exceeded, IgorFS will signal "no space left on device" to the user application.

The block cache component stores data on hard disks. Todays hard disks are relatively slow compared to main memory. That's why it is important that the block cache uses parallelism and asynchronous communication to the hard disk in order not to slow down the overall system.

11.6.2 Design

The IgorFS block cache stores encrypted data blocks on disks using a local file system. The following design allows that to be done efficiently. Blocks are stored in the file system using the hexadecimal representation of the block identifiers as file name. Many local file systems have scalability problems with many files in one directory, therefore the files are distributed into directories. For that, the first two octets of the block identifers are used as directory and sub-directory names, e. g. the block with the identifier `1234567890...` will be stored in the path `12/34/1234567890....` That means the number of directories per level is fixed to 256. This trick reduces the number of entries per directory in the

local file system by a factor of 2^{16}. It can easily be extended to three or more prefixing octets.

This scheme allows the efficient storage and retrieval of blocks. However, to implement a LRU caching strategy more is necessary. LRU requires that two operations can be performed efficiently: Finding the oldest entry and updating a timestamp of an arbitrary entry. To do so, a secondary tree is used in the local file system, this tree is ordered by time. For example, if the block mentioned above was last used at 2007-07-09t23-59, the (symbolic) link `2007/07/09/23/59/1234567890...` will be created. This technique allows directly finding the oldest entry by traversing the directory structure in a depth-first manner. Updating such time stamps requires that the stored block also memorizes the last time stamp, which can be done easily.

Reading a block already in the block cache therefore consists of the actual transfer from disk to memory and a update of the time stamp, done in the following steps:

1. Given the block identifier, the path name to the block file is clear. Using this path name, the local file system will quickly find the file, because the number of entries per directory is limited.

2. Reading the block will yield a time stamp telling when the block was last read.

3. Using this time stamp, the link in the secondary tree can be found and removed.

4. Using the current time, a new link in the secondary tree will be created.

5. Using the same current time, the time stamp in the primary tree will be updated.

Using the local file system as storage solution for the block cache of course introduces overhead. In [146] the overhead is estimated to be less than 2%[6].

To implement different storage classes with separate LRU parameters, the time stamp in the time stamp tree and in the block itself will include the storage class. Currently, permanent storage and intermediate caching are used as storage classes.

11.7   Igor Interface

For the distributed components, IgorFS relies on the overlay network Igor. To do so, messages between IgorFS instances are sent via Igor. All these messages have to be serialized before Igor can handle them. After transmission, each message has to be deserialized again.

In order to limit the required bandwidth in the idle case, it was decided to transfer encrypted data blocks on demand only. This requires that there is a way

---

[6]The average block size is assumed to be 2^{16} octets, the used local file system was ReiserFS.

to find nodes that are currently in possession of a given block. The association of a block identifier and one or more node identifers is called a pointer.

11.7.1 Pointer Cache

The task of the component pointer cache is the distributed management of these pointers. Initially, once a block is stored into or evicted from the block cache, the block cache notifies the local pointer cache. Because of this, pointer cache and block cache are assumed to be synchronized, i. e. the pointer cache knows at least one pointer for each block in the local block cache. These pointers refer to the local node identifier.

The pointer cache publishes the availability of blocks (i. e. pointers) via Igor. The destination of the messages is the identifier of the block. The upcall flag (see section 6.1.2) is set to true in these announcements. This allows intermediate nodes (nodes with identifers between the publishing node and the node with the identifier closest to the block identifier) to aggregate multiple announcements for the same block. This aggregation prevents the root of the aggregation tree to be overloaded.

The announcements happen

- once when the block is written locally.

- once when the block is requested and successfully transfered from a remote IgorFS instance.

- periodically when the local block cache is the original writer of the block.

- every time the neighbor set changes and the final flag was set during reception of that pointer. This happens at the root of the aggregation tree and ensures that pointers are stored in the correct location even if membership in the Igor network changes.

Pointers in the pointer cache can be replaced or deleted at any time. Exception to this rule include:

- If the block is available in the local block cache. This ensures that the block can be found by other IgorFS-instances.

- The final flag was set during reception. This ensures the root of the aggregation tree always keeps a pointer to the block.

The pointer cache is kept as a soft-state cache. That means that received pointers are deleted after a certain period of time, and that nodes in possession of blocks must re-announce their pointers before this interval expires.

11.7.2 Data Transfer

Since the file system IgorFS will only transport encrypted blocks, they are not regarded as confidential material, but rather as opaque data. The file system

will, to improve performance, cache encrypted blocks in the block cache and may retrieve encrypted blocks in advance for anticipated read-calls. The file system will *never* move around unencrypted blocks.

Transported blocks may and should be cached at intermediate nodes. The cache itself knows or can learn the identifier of the block. This is useful because the cache than can verify the authenticity of the block. Note that this however does not imply that the intermediate node can decrypt the block, it remains opaque data. Only encrypted blocks will be cached, since clear text blocks are never transfered and therefore never cached.

The transfer between different instances of the file system (daemons) can be secured additionally. This, however, does not increase the security of the transported data, but helps to establish trust between the systems running the file system (see section 13.1.2).

11.7.3   Block Transmission

As mentioned earlier, in IgorFS it is sufficient to have the block identifier (and a decryption key) to read a file systems object. To decrypt such an object, it must be available locally. If the block is not in the local block cache yet, it has to be fetched from another IgorFS instance. For that, the following steps are performed (see figure 11.6):

a The orginal possessor of a block announces that it has the block locally available by announcing a pointer. This pointer is stored on the node responsible for this block (as determined by the Igor metric). Intermediate nodes may also store (cache) this pointer.

b The local read request contains the identifier of the required block. A request for a pointer to that block identifier is sent towards the responsible node.

c An answer including one or more pointers to possessors of that block is received. Intermediate nodes may also answer if they have the pointers available.

d A block transfer request is sent to one of block possessing nodes.

e The node replies with the block.

a After the block is received and stored in the local block cache, the local pointer cache will be informed. The local pointer cache will send announcements to other pointer caches. After this is done, the local node will act as an additional source for this block.

Since in this process many network operations with potentially many other Igor and IgorFS nodes are involved and these nodes are not completely reliable, reasonable timeouts have to be installed and operations need to be repeated if they are not completed within the timeout.
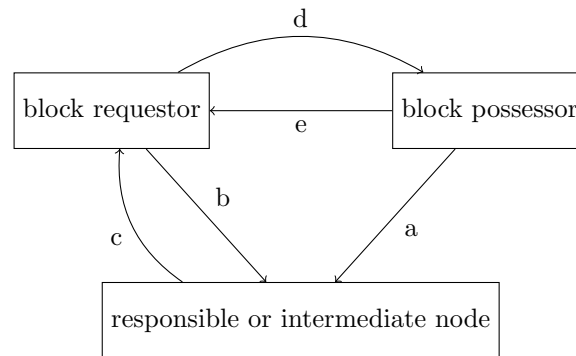
Figure 11.6: Block Transfer

11.8   The Proc System

Similar to the Linux `proc` file system, IgorFS has such a file system, too. Here, different modules and components of the system can offer information about their current state. The user may access these information via normal POSIX file system calls, because the information is offered inside the IgorFS file name space.

11.8.1   Registering

In order to use the proc system, the components have to register. Together with the registration they have to provide a file name (their information will be published using this file name) and a callback for retrieval of the information. Registration for writable files is also possible. In this case, the proc file system is used to transfer information from the user (application) towards IgorFS.

11.8.2   Reading and Writing

Currently, there are some limitations in using the proc file sub-system. Directories have not been implemented. Write requests need to deliver their entire information in one request. For read requests, the full information is retrieved in each request, even if the user (application) asks for only a small part of it. However, these restrictions do not limit the usability of the proc system inside IgorFS.

11.8.3   Example Uses

Two example should illustrate the usage of the proc file system. First, writing anything to the file `proc/debug_clear` will set the debug logging system in a state where no restrictions regarding logging are in place. Second, reading the

file `proc/pointercache_params` will result in a list of parameters for the pointer cache currently in force.

Chapter 12

Implementation

These obviously are not your ordinary bugs, to say the least
Doug Spinney, The X-Files, Darkness Falls

This chapter will detail some implementation issues regarding IgorFS. First, a general overview is given, then the tools to control IgorFS are explained and finally some important modules are delineated.

12.1 Overview

The IgorFS implementation is divided in four major layers shown in table 12.1. The previous chapter explained why FUSE is used as interface to the kernel. FUSE forwards system calls from applications via the kernel VFS interface back to the user space. Here, IgorFS installs handlers for these calls. In order to treat these calls, IgorFS needs an in-memory representation of all file system objects. The file system objects are encapsulated as C++-objects.

For persistence, all file system objects are put in a block store. This storage is much larger than the memory requirements of IgorFS. The block store, here called block cache, is implemented using regular file systems. Finally, blocks from the block store are distributed in the IgorFS network. The interface is similar to distributed hash tables, but not identically. Most important, IgorFS transports blocks on demand only.

From table 12.1 and figure 12.1 it is visible that IgorFS has three major interfaces to other systems. First, FUSE is used to intercept file system calls from the kernel VFS layer. Second, IgorFS needs a way to store blocks persistently, here a local file system is used. Third, to interoperate with other IgorFS instances and thus creating the distributed file system, the Igor overlay network is used.

In addition to the interfaces mentioned above and shown in figure 12.1, out-of-band channels are required. This mechanism is required to establish first

block identifiers and keys to decrypt these blocks. Section 12.3 shows how this
channel works technically. Business and legal aspects of the system can be
attached to this channel, see section 13.1.7 for details about this.

Layer	Data	Interface
FUSE	en-clear	syscall
in memory	en-clear	file system objects
on disk	encrypted only	block store
network	encrypted only	distributed hash table

Table 12.1: IgorFS Layers

IgorFS is implemented as a process network [75], where independent modules
communicate via messages. This design was chosen because

- modules are independent. They can be separately developed and tested.

- it is easy to separate modules into different processes. These processes
  can even run on different machines, given that messages are serialized and
  deserialized in between.

- the design makes it easy to port IgorFS to simulation environments using
  discrete event simulators, e.g. [168][123][176].

12.2   Logging

In order track execution of IgorFS, a logging system has been established. This
system is useful for debugging (even post-mortem), profiling and can give hints
to the user. Since C++, the language IgorFS is implemented in, does not have
advanced inspection and stack examination capabilities, the following function-
ality has been implemented.

The GNU C++ compiler can be queried about the current file, line num-
ber and method and in IgorFS these information are encapsulated in the class
`cCodeLocation`. To remember these code locations, the class `cCodeSegment`
(more precisely: the constructor of this class) pushes one of these code locations
on a stack. There is one stack for each running thread. Since most non-trivial
methods in IgorFS contain a local variable of type `cCodeSegment` (via the pre-
processor macro `BLOCK()`, the constructor and destructor of this variable take
care of maintaining the stack.

To trace IgorFS or output messages of varying importance, objects of types
`cDebug`, `cWarning`, `cError` and `cFatalError` can be instanciated. All of them
can take additional input via the `<<` operator and `cVariable` objects.

All messages are written to an output file. Because the amount of output can
be enormous, a filtering system has been established. Messages can be switched
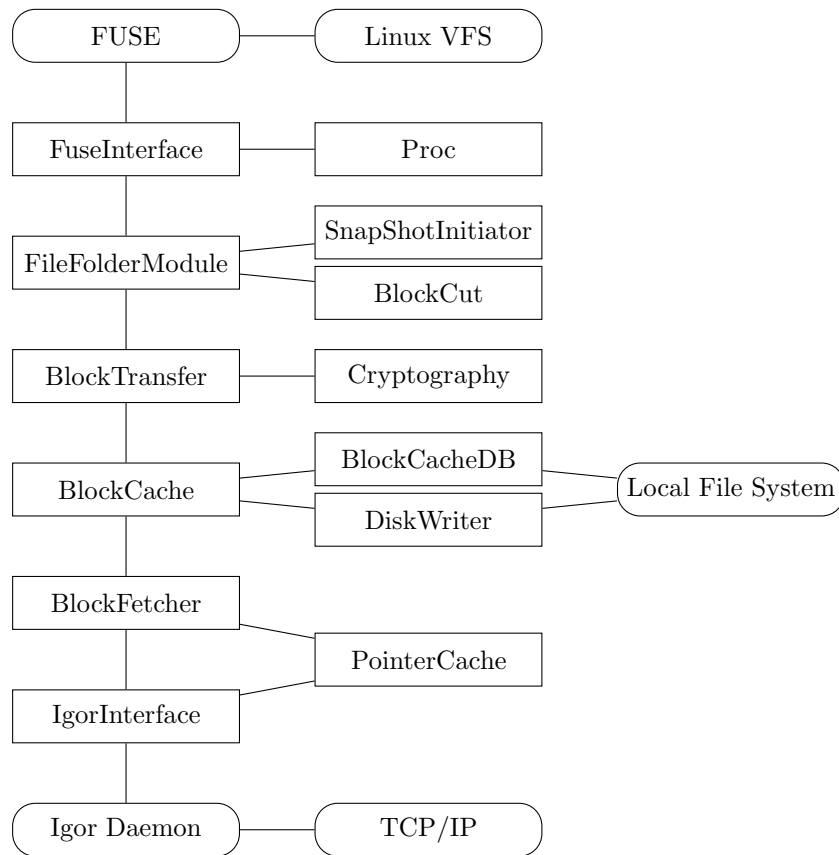
Figure 12.1: Modularization and Interfaces in IgorFS

on and off by adding regular expressions to match the message location to `cDebug::Hide()` and `cDebug::Show()`. To modify this behaviour at run time, the proc files `debug_hide` and `debug_show` give an interface to this message filtering system.

12.3   User space tools

The IgorFS daemon runs in the background and does not interact with the user except via the POSIX file system interface. User space tools allow two operations that are not possible otherwise. First, communication with the running daemon is possible. Second, file system functions not available throught he POSIX interface can be used.

12.3.1 Export Key

During operation, especially during the snapshot operation (see section 11.5), IgorFS computes a block ID which identifies and authenticates the entire content of the locally mounted file system. This identifier is required later to mount the file system again. In order to retrieve the newly computed identifier, the tool `exportkey` is used.

12.3.2 Mounting and Unmounting

Mounting a file system is the process to communicate the operating system the existence of the new file system, while unmounting does the reverse. With FUSE, starting the file system implementation is equivalent to mounting the file system. IgorFS can either be mounted as an empty file system, or using a previously available block identifier and decryption key. Unmounting happens either during termination of the binary, or with the FUSE tool `fusermount -u`.

12.4 File System Daemon IgorFS

The file system is implemented using a *daemon* running on a computer system. There is one daemon per computer using the file system. All these daemons can communicate equally among each other via the Igor overlay network, hence the name peer-to-peer file system.

There are at least three threads:

- Threads created by FUSE wait for requests from kernel. FUSE can create a separate thread for each request[1]. Each of these threads sends a message into the IgorFS process network and blocks until it receives an answer. This answer is delivered back to FUSE by returning from a call.

- The Igor-receiver thread blocks and is unblocked every time a message is received via the Igor overlay network.

- All other actions within IgorFS are executed from the message queue driver thread. There can be more than one queue driver. In the IgorFS process network, modules communicate by sending messages to each other. Timers are modeled as messages from one module to itself. These messages are sorted by delivery time in the main message queue. Queue driver threads take the first message from the message queue and call the message handler from the destination module.

12.5 Modules

The IgorFSis designed to be modular. Modules communicate through messages only. This enables that modules can be located in different processes and even

---

[1]For debugging purposes, FUSE can be instructed to use just one thread for all requests.

in different machines. Further this design makes integration in a discrete event simulator (e. g. [168][123]) more easy.

This section describes the more important modules IgorFSis made of. All modules are based on a C++ class called cModule. Modules are the entities that can send and receive messages.

12.5.1   Module BlockingModule

The BlockingModule is not a real module but an abstract base class for all modules that need to wait synchronously for messages from other modules. This is required at the FUSE interface, because calls from FUSE must not return until the request is fulfilled. Second this feature is useful for testing other modules. The BlockingModule is used for thread synchronization and simply waits (blocks) until a specified message arrives.

12.5.2   Block Cache

For a description of the design criterias of the block cache, refer to section 11.6. To implement the block cache, the problem was divided in four sub-modules. One reason for this approach is that it is expected that many tasks of the block cache require disk I/O, and these system calls may block for a longer period of time.

The module BlockCache receives the request regarding block cache tasks from other modules. Such requests include loading of blocks from and storing blocks to disk. The module BlockCache first checks for duplicate requests and then wraps the request in a block cache internal data structure called job container.

These job containers are then handled by the block cache workers (BCWorker). The advantage of this approach is that there can be multiple block cache workers in parallel. This parallelism is used to counter the effect of blocking system calls.

The seperate module BCLinkDatabase keeps track of the LRU time stamps. As described in section 11.6, these time stamps are kept as soft links in the underlaying file system.

Since the time stamps and the content of the block cache can become inconsistent (time stamps to non-existing blocks or blocks without time stamp), there is a the module BCFSCheck. This module implements a file system check for the block cache data structures on disk. Since the execution of this check can be expensive in terms of run time and I/O load, it is ran on request only.

12.5.3   Module BlockCut

The design and functionality of the block cut module has been described in section 11.4.

12.5.4   Module BlockFetcherModule

When the block cache gets the request to deliver a block to the local file system but does not have this block available, it forwards the requst to the block fetcher module. This module in turn has the task to retrieve the block from remote IgorFS instances. Figure 12.2 shows the process in the following steps:

a The block cache has received a load request it can not fulfil. Therefore the block cache forwards the request (including the identifier of the block to retrieve) to the block fetcher, so the block fetcher could get the block from another IgorFS.

b In order to retrieve the block, the block fetcher first needs a node identifier where to fetch this block from. The association of a block identifier with node identifiers of nodes in possession of that block is called a pointer. To get such a pointer, the request is forwarded to the pointer cache module. The pointer cache either has such a pointer locally available or it has not. In case the pointer is there, the pointer is returned in the process at step **i**.

c If the pointer is not available locally, other pointer caches have to be queried. In order to do so, a request is generated and sent to the igor interface module.

d Here the request is serialized and sent via Igor to other instances of IgorFS.

e Upon reception (the request is sent with the upcall flag set to true), the request is deserialized again and sent to the remote instance of the pointer cache. This remote pointer cache either has a cache miss too (not shown in figure 12.2), in which case the request is forwarded to other pointer caches towards the root in the aggregation tree rooted at the identifier of the block.

f Or the wanted pointer is available. In this case a reply with the pointer is generated and sent back.

g The reply is sent with the upcall flag set to false, because intermediate nodes are not likely to be interested.

h The local pointer cache receives the reply and stores the results. From this time onwards, the local pointer cache can answer requests from other pointer caches for this block identifer. The pointer cache is cleaned again after a timer expires.

i The pointer cache has successfully acquired pointers and sends them to the block fetcher.

j The block fetcher extracts the node identifiers from the received pointer and sends request to another IgorFS instance to get the block. If there is no answer to this request after a specified amount of time, the next node identifier is extracted and the next request is sent.

k This serialized request is forwarded through the Igor network directly to the node with the identifier mentioned in the pointer.

l After reception, the request is sent by the igor interface module to the remote block fetcher. Note that the remote pointer cache and the remote block fetcher do not run in the same instance of IgorFS necessarily.

m The blockfetcher gets the block from the block cache. The requested block should be in this block cache, otherwise the pointer would have been invalid.

n The block cache reads the block from disk and sends a reply back to the remote block fetcher module.

o Again, the block fetcher just forwards the reply.

p A message containing the block is sent through the Igor network. The omit uneccesary overhead, the reply is always sent directly, i. e. the upcall flag is set to false.

q At the local igor interface, the message is deserialized and the block is extracted.

r Finally, the block requested in step **a** is sent to the local block cache. Here, it is stored on disk and the request that initially triggerd the block cache can be fulfilled.

During operation of the block fetcher, two failure modes can occur. First, the pointer cache may not return a pointer. This can happen if the request or the reply got lost for any reason, or if the pointer (and the block) does not exists at all. In both cases, the request from block fetcher to pointer cache (step **b** in figure 12.2) is repeated up to three times. The second failure mode occurs if the pointer cache returned a list of node identifiers where the block should be available, but node of the requests successfully returned the block (step **j**). This case is handled identically to the first failure mode, i. e. it is treated as if the pointer cache would have returned an empty pointer containing no node identifiers.

After block fetcher failed three times, this failure is reported back to the block cache.

12.5.5   Module BlockTransModule

The block transfer module is an intermediate module that moderates between the FSO handling, the cryptographic module and the block cache (see figure 12.3).

Storage requests from the FSO handling are first encrypted and hashed and then sent to the block cache, where the encrypted result is stored on disk. In parallel, the result of the hashing (the block identifier) is sent back to the FSO handling module.
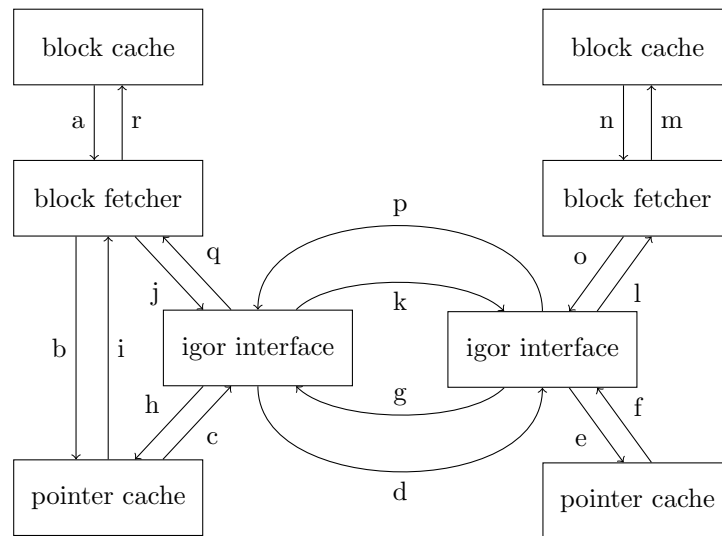
Figure 12.2: Block Fetcher Operation

During read requests, FSO handling asks with a block identifier and a decryption key. The block transfer module uses the block identifier to retrieve the block from the block cache. Using the block identifier, the authenticity of the (still encrypted) block can checked by the crypto module. Since the decryption key is also provided by the FSO handler, the block can be decrypted too. With the same key, the authenticity of the decrypted block can be checked. As soon as all authenticity checks are passed and the block has been decrypted, the clear text block is transfered from the block transfer module to the FSO handler.
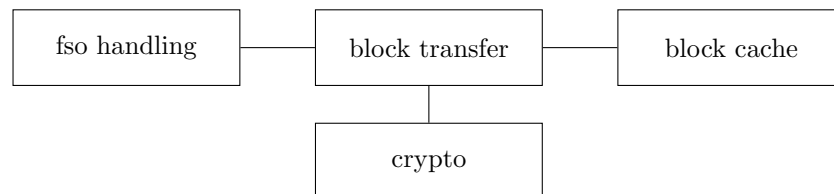
Figure 12.3: Interaction of the Block Transfer module

12.5.6   Module FileFolderModule

The module is responsible for storing the in-memory representations of all file system objects and executing all operations on them. These file system objects include directories, regular files and soft links. Section 11.3.3 explained why

hard links, device files and named pipes are not handled.

Further the transformation of these in-memory objects to blocks on disk and vice versa is initiated here. Part of the forward transformation is the snapshot process. The snapshot related procedures are described in section 11.5.

12.5.7 Module IgorInterfaceModule

The module IgorInterfaceModule is the interface between IgorFS and Igor. Messages from modules in the local IgorFS are sent to the IgorInterfaceModule, where they are forwarded via Igorto other instances of IgorFS. On the other hand, messages from other IgorFS nodes are received and handed to the module in charge. Since Igor expects messages to be octet streams, the IgorInterfaceModule has to serialize and de-serialize all messages.

Currently, the pointer cache module (see section 12.5.8) and the block fetcher module (see section 12.5.4) make use of the services offered by the IgorInterfaceModule.

The IgorInterfaceModule is further responsible to retrieve the identifier of the local node from Igor. This is necessary to embed this identifier in pointer announcements made by the pointer cache (see next section).

12.5.8 Module PointerCache

A Pointer in the context of IgorFS is an association between a block identifier and a node identifier. The semantic behind is that the block with the given ID is stored on the node with the ID in the pointer. It does not neither imply anything about the permanence of that storage nor why this node stores the block. The node may be the initial writer, an intermediate caching system or the finally responsible node (see section 6.2.3) for this pointer.

Pointers for blocks in the local block cache are kept as long the block is kept in the block cache. If the pointer has been acquired as an intermediate caching system, the pointer can be evicted at any point in time without influence for other IgorFS-nodes. In case the node is the finally responsible node for this pointer, the pointer is kept without limits.

The entire system is designed to be soft-state: Owners of blocks (nodes with the block in the block cache) regularly re-publish their pointers. This way all pointers (and by that all blocks) can be found even if some systems fail non-gracefully. The nodes which are finally responsible for pointers re-publish these pointers if Igor signals that the network neighborhood changed. Changes in the network neighborhood could imply that the responsibility for pointers changed.

All publications of pointers (either initial publications or one of the repetitions mentioned above) are sent with the Igor flag upcall set to true. This means that intermediate systems can aggregate these announcement messages and aggregate them. Aggregation here means that if multiple pointers with for the same block identifier arrive at a node, only one pointer with a subset of these nodes is forwarded in the aggregation tree. This way it is ensured that the root of the aggregation tree (a) receives at least one pointer if such a pointer

exists and (b) is not overloaded if many pointers exists in the overall system of all IgorFS instances.

12.5.9   Module SnapShotInitiator

The module SnapShotInitiator initiates the creation of snapshots. Periodically the appropriate message `cSnapShotRequest` is sent to the file/folder handling module. As soon as the answer `cSnapShotReply` is received, the snapshot is successful and the computed pair of block identifier and decryption key can be exported. Even if the produced pair of block ID and key is not exported, periodic snapshotting ensures that all written data is periodically made persistent. That means the snapshotting process also triggers the local write to hard disks and with that the freeing of memory resources.

12.5.10   cProcModule

As described in section 11.8, IgorFS features a interface similar to the proc interface similar to Linux. Here, the proc interface is used to query internals from the running IgorFS system as well as to modify its behaviour. The proc modules is responsible for accepting registrations of proc file system entries from different modules. Then, the fuse interface module forwards requests to the proc module. The proc modules has to decide how to handle the request (see figure 12.4). If an appropriate handler has been registered by another module (here called proc user), this handler is called. The answer is reported back to the fuse module, where it is delivered to the requesting process.
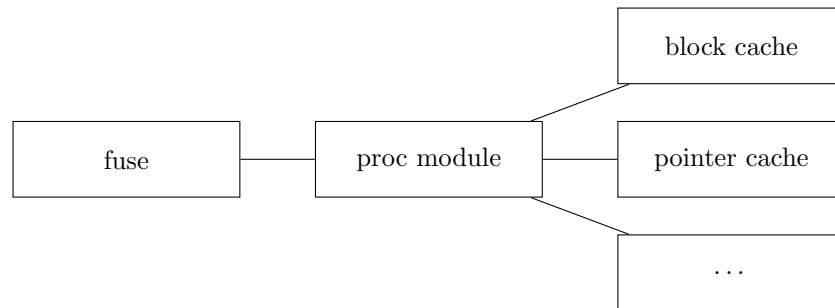
Figure 12.4: Interaction of the proc module

12.5.11   Fuse Interface Module

Overview

The Linux kernel sends all file system related requests towards the VFS layer. Here it is decided which Linux file system should deal with the request. If the

request regards the IgorFS file system, the fuse kernel modules forwards them to the user space, where libfuse receives them. This modules receives the requests from libfuse and serves them. It does so by registering for C function callbacks and forwarding them into a C++ object.

The requests are either answered directly or encapsulated in IgorFS internal messages and forwarded to the responsible module. The libfuse library can create a new thread per request. Since the fuse interface module is a blocking module (see section 12.5.1), each of these threads blocks until an answer is received from other modules. This parallelism is important so IgorFS can work on multiple requests from fuse at once.

Supported Operations on File System Objects

```
int (*mknod) (const char *, mode_t, dev_t);
int (*mkdir) (const char *, mode_t);
int (*symlink) (const char *, const char *);
```

The call `mknod()` is used to create ordinary files, device nodes and named pipes. Since IgorFS does not support the latter, only regular files may be created. The file is created empty.

After return of `mknod()`, the file exists as an in-memory object only. The creation of this in-memory objects makes all parent directories dirty, if they had been clean previously. The next snapshot operation makes the new file persistent, as well as all new versions of the parent directories.

Directories and softlinks are created with calls to `mkdir()` and `symlink()`, respectivly. Since both dirty the parent directory, all comments from `mknod()` apply too.

```
int (*unlink) (const char *);
int (*rmdir) (const char *);
```

These operations remove files, softlinks (`unlink()`) and directories (`rmdir()`) from the directory hierarchy. Removing an objects changes the parent directory, see comments for `mkdir()` above.

The local block cache (and therefore the pointer cache too) is not affected by the operation. This is useful because other instances of IgorFS may require the blocks, even if the objects are deleted locally.

```
int (*rename) (const char *, const char *);
```

The `rename()` call is used to change the visible name (absolute path) of a file system object. The object may be transfered to a different directory during this operation, change its name in the directory, or experience both changes at once.

The renamed object is not dirtied during this operation, but containing directory/directories is/are.

```
int (*chmod) (const char *, mode_t);
int (*chown) (const char *, uid_t, gid_t);
int (*utime) (const char *, struct utimbuf *);
```

These requests change the meta information about file system objects: ownership, access permissions and creation/modification times. Since the data items themselfes are not changed, only the containing directory gets dirty. The access time is kept locally, but not transfered over the network. Note that the access right also have local significance only.

```
int (*write) (const char *, const char *, size_t,
              off_t, struct fuse_file_info *);
int (*truncate) (const char *, off_t);
```

Both operations have influence on the contents of files. To overwrite parts of a file, it may be necessary to request other parts from the file from the network first.

```
int (*getattr) (const char *, struct stat *);
```

The kernel requests meta information about a file system object. Such information include file size, hard link counter, access time and many other details. Since this call is issued often, quick operation is essential here. In order to fulfil the requests, it may be necessary to fetch directory blocks, including parent directory blocks.

```
int (*read) (const char *, char *, size_t,
             off_t, struct fuse_file_info *);
int (*readlink) (const char *, char *, size_t);
int (*readdir) (const char *, void *, fuse_fill_dir_t,
                off_t, struct fuse_file_info *);
```

These operations read the content of files, softlinks and directories, respectivly. First the local block cache is checked for information and if nothing is available the block is fetched from the network. As above, all requests dealing with file system objects may need to fetch many blocks and therefore block for a some time.

Supported Operations on the Entire File System

```
int (*statfs) (const char *, struct statfs *);
```

This operation is called in order to obtain general information about the mounted file system. Such information inlcude overall file system used and free space.

```
void *(*init) (void);
void (*destroy) (void *);
```

These two operations are the main hooks to start and stop the internal working of IgorFS. Fuse calls these just after the file system has been mounted and just before it is going to be unmounted. Here all the C++ objects required to run IgorFS are constructed and destructed.

Operations not Supported

```
int (*opendir) (const char *, struct fuse_file_info *);
int (*releasedir) (const char *, struct fuse_file_info *);
int (*fsyncdir) (const char *, int, struct fuse_file_info *);
```

These operations are useful for stateful operations. Since IgorFS is right now implemented in a stateless manner (regarding the interface to fuse), these calls are not required. Later version of IgorFS may support these. One reason to support these functions would be to pass state from `opendir()` to related `readdir()` calls. Another would be to trigger prefetching of other blocks, because once `readdir()` has been called it is expected that other calls to this directory or its children will follow.

```
int (*open) (const char *, struct fuse_file_info *);
int (*release) (const char *, struct fuse_file_info *);
int (*flush) (const char *, struct fuse_file_info *);
int (*fsync) (const char *, int, struct fuse_file_info *);
```

The same holds true for file objects. Right now, the IgorFS operates stateless with regard to the fuse interface. Later version may pass state between calls and use prefetching to improve performance or use `flush()` and `fsync()` as signals to start snapshot operations.

```
int (*setxattr) (const char *, const char *,
                 const char *, size_t, int);
int (*listxattr) (const char *, char *, size_t);
int (*getxattr) (const char *, const char *, char *, size_t);
int (*removexattr) (const char *, const char *);
```

Extended attributes are not supported at the moment, but proably are a way to integrate IgorFS-specific interfaces.

```
int (*link) (const char *, const char *);
```

As explained in the previous chapter, IgorFS does not support the semantic of hard links, therefore this call is not supported either.

```
int (*getdir) (const char *, fuse_dirh_t, fuse_dirfil_t);
```

This interface to fuse is depreciated and therefore not supported.

12.6   Summary

This chapter gave an overview over the IgorFS implementation. First the logging and debugging facilities in the IgorFS process have been covered. Next, the user space tools that are necessary to use IgorFS were discussed. The chapter concluded with a description of the more important modules IgorFS is comprised of.

Chapter 13

Conclusions

Gaebe es die letzte Minute nicht, so wuerde niemals etwas fertig
Mark Twain

13.1 Open Issues

13.1.1 Obfuscation

If required, traffic and usage patterns may be obfuscated by the following measures:

- Proactive Caching: The daemons may, when idle, fetch blocks where future requests by neighboring daemons are anticipated. Random blocks may be fetched when available disk space and bandwidth permit it. This has the further advantage that proactive caching helps to increase the performance of the overall system. Participants requiring anonymity can gain anonymity by paying with bandwidth and hard disc space.

- Onion Routing: Based on the ideas in [17], every request in wrapped in multiple asymmetric encryption layers.

- Random Routing: Every request is either routed correctly or, with a given low probability, sent in the wrong direction. Every peer in the system can act as proxy, thereby hiding the real source of a request.

- A combination of the above.

Note that none of the above mentioned options provides real anonymity.

13.1.2 Denial of Service

Denial of Service by resource exhaustion is one of the most difficult attacks to resist for an open system. Possible defenses include:

- Ignore the problem. This is probably the most common route.

- Try to provide more resources than the attacker can consume. Static WWW-Content can be distributed by resource-providers like Akamai[2]. This is working well but can be expensive.

- Deny some requests to serve others well. For example some FTP- or SMTP-Servers take this path. Only a given number of connections are allowed, others are denied/deferred. In many cases, there is no distinction between good and bad requests.

- Require explicit resource reservations. This is the basis of many Quality-of-Service proposals.

- Try to block bad requests, like e. g. firewalls do.

- Admit that one can not solve the problem entirely, but make attackers traceable, for example [3].

In this context, we suggest that the last alternative is the way to go.

This requires some out-of-band administrative effort. Either all daemons are authorized by a certificate based from a central certification authority. File system daemons can then identify each other in a secure way. An alternative is that daemons are introduced to each other by the local administrator, and by this act, the local administrator 'vouches' for the remote daemon.

13.1.3   Hash Collisions

It is open how to deal with hash collisions. Since cryptographic hash function (see section 2.1) compress input data of arbitrary length into a fixed number of bits, a collision (the event that two different inputs map to the same output) is not avoidable. The birthday paradox [9] is the reason that the expected number of invocation of a hash function with N bits before experiencing a collision is $2^{N/2}$. Table 13.1 gives some examples on how soon the first hash collision is expected for some popular hash widths. Note that this table assumes hash functions with no weaknesses, where the output is a uniform random distribution. The table also shows how many blocks of 100 kibibytes can be hashed without expecting a collision.

Since the probability of a hash collision is very small but not zero, a hash collision avoidance field can be added to every block. The write of that block checks whether the block identifier exists and if it yields to the same block. If not, the writer can change the hash collision avoidance field and gain a different block identifier.

13.1.4   Reliability and Block Deletion

Right now, blocks on disk are managed by the LRU mechanism in the block cache module. In the future, other possibilities could be useful. First, the LRU

Width of hash	Possible results	Expected first collision	Data
128	$\approx$3.40e38	$\approx$1.84e19	1638400 exbi-bytes
160	$\approx$1.46e47	$\approx$1.21e24	1.07e11 exbi-bytes
256	$\approx$1.16e77	$\approx$3.40e38	3.02e25 exbi-bytes

Table 13.1: Expected hash collisions. Data assumes the block size is 100 kibibytes

mechanism does not guarantee any reliability on block availability. This was a deliberate decision based on the design goal that block transfers should happen on demand only. To create reliability, possessors of a given block are in charge of ensuring that x copies of that block exists at any point in time. The larger x, the larger the overall reliability. On the other hand, such a scheme would never delete blocks. However, because of limited storage space, block deletion might be necessary. Here it would be useful that publishers could either explicitly mark blocks for deletion, or that the block cache could give a limit which snapshots are still fully available.

13.1.5 Read Ahead

Network latencies are large compared to today's computing speeds. This is especially severe since most applications do read in a synchronous manner, i.e. they block until the read result returns. To overcome this, a read-ahead mechanism can be employed. FUSE does so already at the file level. The overall read performance for applications could be further improved by implementing a read-ahead mechanism in IgorFS itself. Here, x pointers and y blocks could be fetched in advance, where usually $x > y$, because it is less resource intensive to fetch pointers than it is to fetch blocks.

13.1.6 XML and Other Interfaces

The IgorFS design aims at the POSIX file system interface. However, other interfaces to access data exists and are in use. For example, data bases and especially XML backed data bases need a different API. Such APIs include functions to insert, delete and move sequences of bytes. Similar functions are not available in the POSIX interface, but can be easily supported by the IgorFS design.

13.1.7 Business/Legal Aspects

The initial keys and block identifiers have to be transported by some out-of-band (with respect to the file system) means.

This is the point where we envision some money can be made. This is also the point where we envision some legal contracts can be established. Such a contract must include the things that the receiver of the key is allowed to do with the decrypted data (and with the key). This contract may include among others:

- The admission or interdiction to re-sell the key.

- The admission or interdiction to re-sell keys decryptable by the key mentioned above.

- The level of admission to use the decrypted data.

- The admission or interdiction to publish derivate work.

Note that these are entirely legal aspects. The abilities of the receiver of a key are not (and can not be) limited by technical means.

There is also some trust needed between instances of the file system. Such trust can be established by creating a central certification authority (CA) and requiring every participating daemon to provide an identity signed by the CA. This would enable the CA to control participants of the system and earn money by issuing these signatures.

Furthermore, these certificates can be used to sign the distributed keys (as described in section 11.2.2) to authenticate them. This way illegal content can be traced back to its origins.

13.2   Summary

After the the first part of this work laid out the basics and the second part covered the overlay network Igor, the third part of this work dealt with the distributed and decentralized file system IgorFS. After identifying the design goals of the file system, the overall architecture has been described. Then some more important concepts have been presented more detailed. The implementation idea was explained along with again a more detailed coverage of some fine points. This part concluded with some open issues, the next part of this work continues with tests and evaluations of IgorFS.

Part IV

Evaluation and Summary

Chapter 14

Evaluation and Testing

One test result is worth one thousand expert opinions.
attributed to Wernher von Braun

This section shows how Igor and IgorFS is evaluated and tested. It covers tests of system details like network measurements and the checksumming algorithm to tests that check the entire system.

14.1 Checksum Algorithm

Section 11.4.2 discussed different approaches to compute cutting marks with the help of rolling checksums. One main selection criteria is the uniform distribution of checksum values. This section compares different algorithms and shows their distribution functions.

Figure 14.1 presents the complementary culumlative distribution function (CCDF) of Adler32[*] and with that a major disadvantage of this checksum. For each line in the graph 10^5 equally distributed random octet strings of the given length were generated and the checksum were computed over each of them. The figure shows the distribution of the results and the desired equal distribution of checksum values. For shorter sliding windows, the distribution of possible checksums is not uniformly distributed, i. e. some checksums are much more likely than others. A secondary disadvantage not visibile from the graph is that in order to move the sliding window, all octets in the windows need to be memorized in order to remove them once the window is advanced.

Figure 14.2 shows clearly that the distribution of checksum values is close to the optimum except for sliding windows shorter than 32 octets. The plots for checksum lengths with 32 octets and more are all located on top of each other toghether with the desired equal distribution. Note that for clarity abscissa is shown with a logarithmic scale.
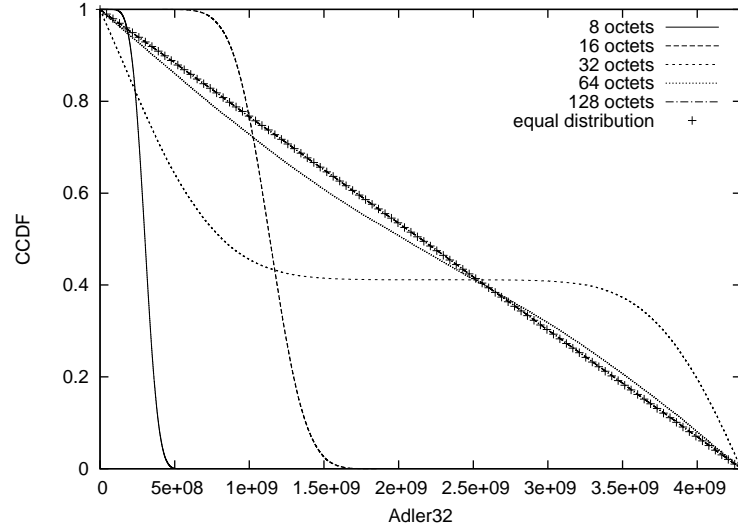
Figure 14.1: Checksum Distribution for Adler32[*]

14.2   Adaptive Block Size

This section evaluates the usefulness of the adaptive block cutting mechanism introduced in section 11.4 and especially 11.4.2. In order to do this, 218 files from the IgorFS source code repository were taken and cut with two different algorithms. The first algorithm simply cuts the file to the desired fixed block size. The second algorithm is the one that has been described in the design section. Figure 14.3 shows that with the IgorFS approach more storage space than with fixed block size is conserved by omitting duplicate blocks. It plots the (normalized) required storage over (average normalized) block size, where

Property	Value
Measurement host	`onelab03.inria.fr`
Reflector	`iraclyde.iralab.uni-karlsruhe.de`
Records	15467
Time span (s)	$1.000 \cdot 10^3$
Average measurement by TCP (s)	$5.308 \cdot 10^{-2} \pm 9.257 \cdot 10^{-4}$
Average measurement (s)	$5.312 \cdot 10^{-2} \pm 3.326 \cdot 10^{-3}$
Average relative error (s)	$2.271 \cdot 10^{-3} \pm 5.017 \cdot 10^{-2}$
Min/max relative error (s)	$-4.352 \cdot 10^{-1}/5.758 \cdot 10^{-1}$

Table 14.1: TCP round trip time measurement experiment 1
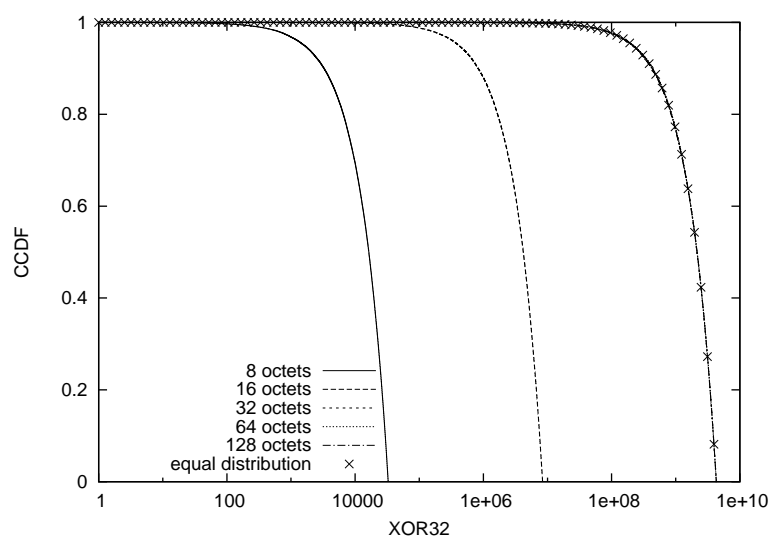
Figure 14.2: Checksum Distribution for XOR32

required storage less than one means saved space. Both algorithms safe significant storage for very small block sizes. However, as discussed in section 11.4, small block sizes lead to more overhead and are therefore not desireable. The IgorFS algorithm is able to preserve much more storage already at larger blocks. Note that the IgorFS algorithm is able to save storage even when the normalized block size is equal to 1. The reason behind this is that file sizes differed significantly, so larger files have been cut in a way that duplicates have been discovered.

Property	Value
Measurement host	`planetlab2.dtc.umn.edu`
Reflector	`planetlab2.eecs.northwestern.edu`
Records	45199
Time span (s)	$1.142 \cdot 10^3$
Average measurement by TCP (s)	$1.399 \cdot 10^{-2} \pm 4.068 \cdot 10^{-4}$
Average measurement (s)	$1.385 \cdot 10^{-2} \pm 2.236 \cdot 10^{-3}$
Average relative error (s)	$1.513 \cdot 10^{-2} \pm 5.050 \cdot 10^{-2}$
Min/max relative error (s)	$-9.487 \cdot 10^{-1}/5.819 \cdot 10^{-1}$

Table 14.2: TCP round trip time measurement experiment 2
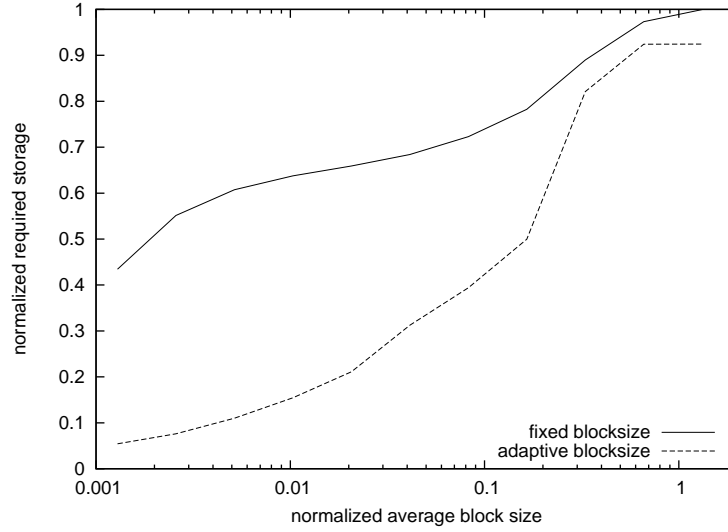
Figure 14.3: Storage savings through block cutting

14.3   TCP kernel information

In section 3.4.4, it has been described how Igor uses a system call to gather latency estimates from the kernel.

To evaluate how well suited this system works, real round trip time measurements have been compared with the values returned by the Linux TCP stack. Each experiment consists of a measurement hosts and a reflector. The measurement hosts sends out short sequences of octets[1]over TCP to the reflector. The reflector returns them as fast as possible over the same connection. Upon

---

[1]In this case, the sequence contained eight octets. These octets represented a floating point

Property	Value
Measurement host	`planetlab2.wiwi.hu-berlin.de`
Reflector	`planetlab2.larc.usp.br`
Records	3810
Time span (s)	$1.000 \cdot 10^3$
Average measurement by TCP (s)	$2.493 \cdot 10^{-1} \pm 7.807 \cdot 10^{-4}$
Average measurement (s)	$2.511 \cdot 10^{-1} \pm 1.105 \cdot 10^{-1}$
Average relative error (s)	$-2.695 \cdot 10^{-4} \pm 1.733 \cdot 10^{-2}$
Min/max relative error (s)	$-9.647 \cdot 10^{-1}/3.059 \cdot 10^{-2}$

Table 14.3: TCP round trip time measurement experiment 3

Property	Value
Measurement host	`planetlab-02.naist.jp`
Reflector	`planetlab-03.naist.jp`
Records	77791
Time span (s)	$1.000 \cdot 10^3$
Average measurement by TCP (s)	$2.108 \cdot 10^{-3} \pm 5.307 \cdot 10^{-4}$
Average measurement (s)	$8.499 \cdot 10^{-4} \pm 2.120 \cdot 10^{-3}$
Average relative error (s)	$7.316 \cdot 10^0 \pm 5.014 \cdot 10^0$
Min/max relative error (s)	$-8.344 \cdot 10^{-1}/4.335 \cdot 10^1$

Table 14.4: TCP round trip time measurement experiment 4

Property	Value
Measurement host	`77-57-169-132.dclient.hispeed.ch`
Reflector	`220-245-140-197.static.tpgi.com.au`
Records	15243
Time span (s)	$1.851 \cdot 10^4$
Average measurement by TCP (s)	$1.095 \cdot 10^0 \pm 9.514 \cdot 10^{-1}$
Average measurement (s)	$1.202 \cdot 10^0 \pm 2.258 \cdot 10^0$
Average relative error (s)	$1.601 \cdot 10^{-1} \pm 5.069 \cdot 10^{-1}$
Min/max relative error (s)	$-9.612 \cdot 10^{-1}/5.013 \cdot 10^0$

Table 14.5: TCP round trip time measurement experiment 5

reception, the measurement hosts determines the time the octet sequence was underway. At the same time, the measurement hosts queries the TCP stack about its estimation of the round trip time. Both values, the stacks estimation as well as the measured value, are recorded together with the current wall clock time.

Tables 14.1 to 14.3 show that the method works well. The measurements were done inside Europe (table 14.1), inside North America (table 14.2) and across the Atlantic (Europe – South America, table 14.3). Each experiment lasted for at least 1000 seconds. In all three cases, the measurements and the values from TCP agree. The relative error is small on average, and there are no outliers. The standard deviation of all values is small. The number of records is smaller in the trans-atlantic case (Humboldt University Berlin – University of São Paulo) because the number of records is (for constant experiment length) inverse proportional to the average round trip time.

The following two experiments (table 14.4 and table 14.5) show the limits of the technique. The run between two machines with very short latency in between them shows the problem of granularity in the `tcp_info` RTT informa-

---

number of double precision encoding the number of seconds since the UNIX epoch.
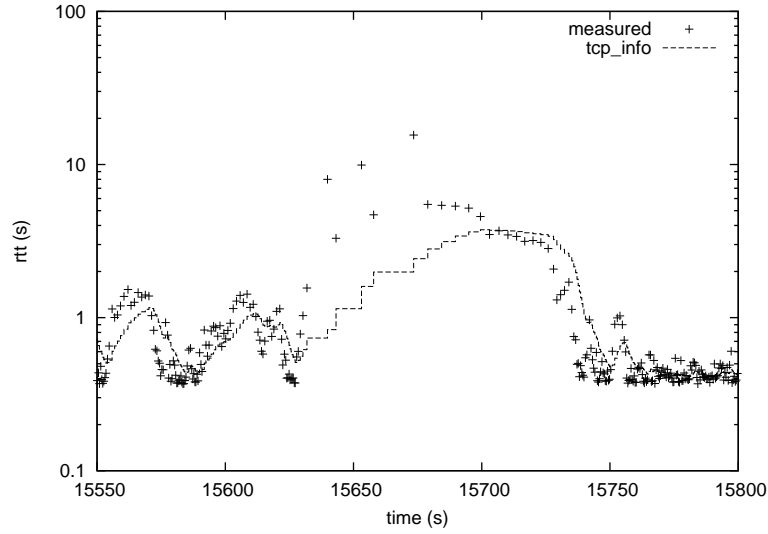
Figure 14.4: Effect of RTT smoothing

tion. The Linux kernel returns RTTs with a granularity of $\frac{1}{8000}s$. If the real values are in the same order of magnitude or lower, the measurement is bound to produce large relative errors. Figure 14.5 shows a part of the CDF of values returned by Linux from all experiments.

In figure 14.4 two other limitations of the approach are visible. The TCP standard[135] gives an example how TCP instances can measure the round trip time (RTT) and how to keep an exponentially smoothed version of the RTT as follows:

$$\mathrm{RTT_{smoothed}} = \alpha \cdot \mathrm{RTT_{smoothed}} + (1 - \alpha) \cdot \mathrm{RTT_{measured}}$$

The standard suggests a value of α between 0.8 and 0.9, the Linux kernel[2] uses $\alpha = \frac{7}{8}$ because of its efficient computability.   This smoothing is necessary for TCP in order to prevent overly fast reactions to network fluctiations. On the other hand this means that real changes in the network are reflected in RTT values only after the exchange of few more segments. The second limitation is that large RTTs (singular high value in figure 14.4) due to short network outages (in figure 14.4 visible as horizontal space between values) and package retransmissions are not accounted for. The latter limitation is not a real problem since these events have no predictive value for future RTTs.

---

[2]The   responsible   code   is   located   in   `net/ipv4/tcp_input.c`   in   function `tcp_rcv_rtt_update()`
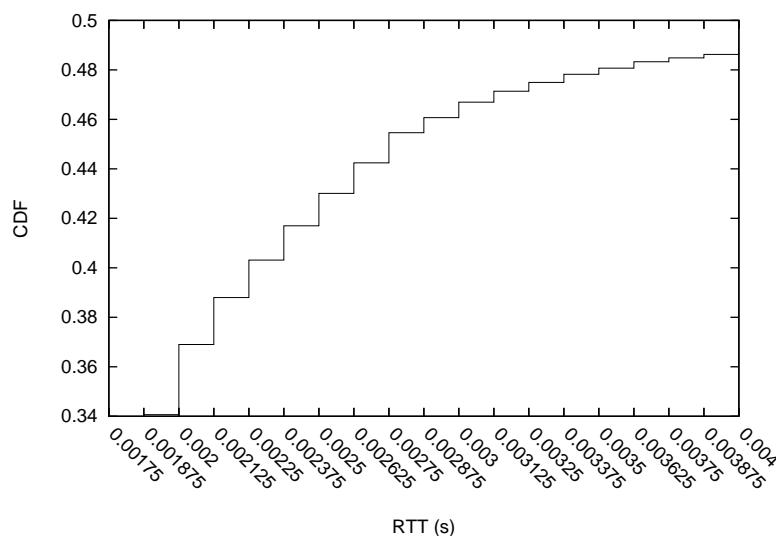
Figure 14.5: CDF of RTT values from TCP
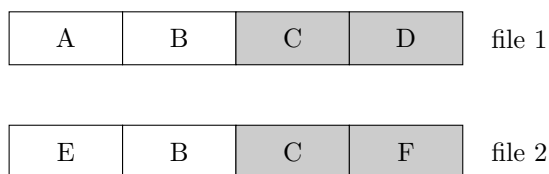
Figure 14.6: File Setup for Performance Comparison

14.4 End-to-End Evaluation

This section describes test that test the entire system, i. e. the file system IgorFS running on top of the overlay network Igor. PlanetLab would be a nice testing platform, but using IgorFS on top of PlanetLab is unfortunately not possible. PlanetLab uses VServer [38], an approach to virtualize the Linux kernel. With VServers, the kernel is shared among all virtualized machines. Since the PlanetLab kernel does not contain the FUSE module and IgorFS depends on this module, IgorFS can right now neither be deployed nor tested on PlanetLab.

In order to verify that IgorFS reached its designed goals, a comparison between scp (secure copy, based on [175]), rsync[165], sshfs (a file system based on [175]) and IgorFS was set up. Two files were set up as seen in figure 14.6. The first file consists of four large (larger than the IgorFS block size) parts A to D. The second file has the same size, also is comprised of four parts, but shares two parts with file 1, whereas the parts E and F are different from file 1.
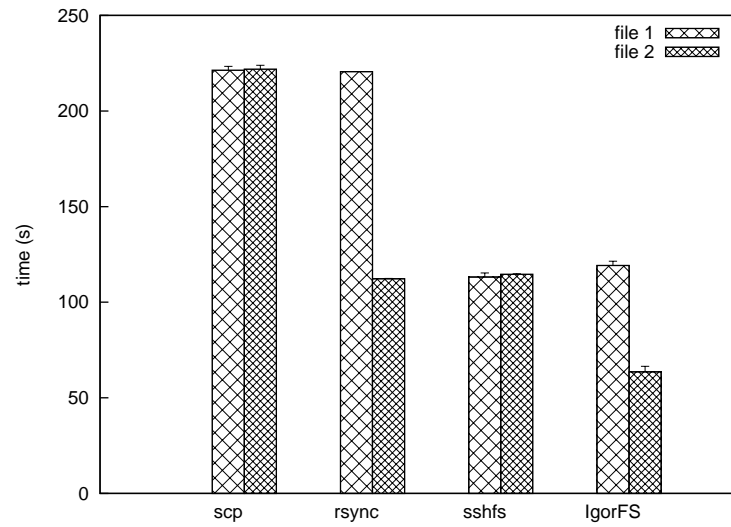
Figure 14.7: Overall performance comparison

Both files were created at one host. Afterwards, the second part of each file was read at a different host using scp, rsync over ssh, sshfs and IgorFS The results of this evaluation are shown in figure 14.7. It shows the times required to read the second part of both files.

With scp, there is no good way to read any part of the file if it has not been copied to the second host in full. It does neither exploit the fact that only parts of the files are read nor that parts of the files are identical. All blocks are transfered over the network, blocks B and C twice. The rsync protocol is able to detect the shared blocks between both files. In order to not to introduce another variable, the transfer for rsync was also ssh, but an independend experiment has shown that the transfer is dominated by the network time, so the additional CPU overhead for cryptography does not influence the transfer. Each block is transfered over the network once. The time savings compared to scp result in the fact that blocks B and C are not transferred twice. Using sshfs, only the blocks that are actually read are transfered over the network. Since only half of all data is read for both files, sshfs is able to complete the experiment in half the time compared to scp. Overall, blocks C, D and F are transfered, however block C is transfered twice. IgorFS is able to combine the advantages of rsync and sshfs. Here, only distinct blocks that are actually read need to be transported. That means for file 1, blocks C and D are transmitted, but for file 2 only block F needs to cross the network. One could expect that the read time of file 2 is exactly a quarter of the scp method and half of the rsync and sshfs times, however it is more for two reasons. First and foremost, the block boundaries in IgorFS do not necessarily exactly match the block boundaries in this experiment (and

user requests in real life). A second reason is that the IgorFS implementation is not yet fully optimized. Even with these two drawbacks, the savings in network capacity are still very significant.

14.5 Summary

This section showed evaluations of subsystems of Igor and IgorFS with an emphasis on new ideas presented in this work. The section concluded with a end-to-end test of the entire system where it was possible to show significant advantages over other systems that are in use today.

Chapter 15

Conclusions

Now this is not the end. It is not even the beginning of the end. But it is, perhaps, the end of the beginning.
Winston Churchill

This thesis presented the overlay network Igor and the secure, distributed and decentralized file system IgorFS. Together, they offer a new way to share access to potentally large files, even if these files change often. The distributed use is more secure and more efficient than before.

The design is open to new ideas. At the Igor level, the integration of overlay networks with network layer routing protocols seems promising. In the area of IgorFS, new interfaces offer complete new use cases even beyond file system access.

15.1 Acknowledgments

Alone, adj.: In bad company.
Ambrose Bierce, "The Devil's Dictionary"

Thanks for advice and criticism, for motivating me before and during every phase of this thesis, for reading and commenting on (even early) revisions of this document, for many hours of fruitful discussions on ideas and designs, for your help with implementing Igor and IgorFS, for help with design and implementation of Videgor, for your friendship and for your support.

Appendix A

Igor Configuration options

The following list describes possible values in the configuration file of the Igor daemon. The format of the file is relatively common among configuration files: Empty lines and lines starting with a hash mark are ignored. All other lines are expected to be of the form "key colon space value newline". For example:

```
# igor config file
id: da4de85db275...
bootstrap: host.example.org:11071
netport: 11071
# end igor config file
```

bootstrap Host name and port number of another Igor instance. This Igor instance is used to bootstrap into the network.

clientport TCP port where to listen for connections from the library `libigor` (see section 6.5.2).

cont_if_no_conns Influence the behavior when no connections to other Igor nodes are available (any more). Usually, an Igor instance quits if no connections to other Igors are left. This behavior is reasonable because a standalone Igor is not useful. However, to start a new network, at least one node must run initially without outside connections. If set to an integer with boolean value "false" (i.e. set to zero), Igor quits after the last connection closes. If set to a "true" (i.e. all other integers), Igor does not quit and waits for other incoming connections.

id The ID of the local Igor node, given in the hexadecimal form. If the ID is not provided during start, a new ID is randomly generated.

max_conns_allowed Upper bound on number of connections.

merivaldi Disable Merivaldi algorithm.

min_conn_alive_time Connections younger than this timeout are not closed. Useful to stabilize the routing table.

min_conns_wanted Lower bound on the number of desired connections.

netport At this TCP port Igor will listen for connections from other Igor nodes.

timer_fixfingers Time interval between two consecutive invocations of the fix-finger procedure (see section 6.3.1), given in seconds.

number_of_ringmembers Number of nodes kept per Meridian ring.

number_of_rings Number of Meridian rings.

prs Enable/Disable the latency oriented message forwarding.

timer_fixfingers Time between two executions of the finger fixing algorithm.

timer_improvement_consideration Interval between two executions of Merivaldi.

timer_lookup Time between two RTT-estimations of the Merivaldi plugins.

timer_pinger Number of seconds between two ping message. Ping messages are sent periodically on every network connection, given the connection is not idle.

timer_service Service Announcement Interval.

timer_stabilize Time between two executions of the stabilize algorithm.

vivaldi_dimensions Size of Vivaldi coordinates.

Appendix B

Messages in IgorFS

Section 12 showed, IgorFS is implemented as network of message handling modules (a process network). There are a lot of different messages, and most of them are exchanged between two distinct modules. This section lists the messages and explains their content very briefly.

Message `cBFUpdateStatus`

Sent from block fetcher to notify that the status of one fetch request changed.

Message `cCryptoMsg`

This is not a real message but a base class of all crypto messages.

`cMsgEncrypt` Request to encrypt something.

`cMsgEncrypted` Something has been ciphered/encrypted, sent as a result to `cMsgEncrypt`.

`cMsgDecrypt` Request to symmetrically decrypt a block of encrypted data.

`cMsgDecrypted` Result of decryption.

`cMsgHash` There is a need to compress some data with a cryptographic hash function.

`cMsgHashed` Some data has been processes with a crypto hash function.

`cMsgVerify` Check whether a hash is correct.

`cMsgVerified` The result of checking with `cMsgVerify`.

`cMsgHashNEncryptNHash`   This message is often needed by the block transfer mechanism (see section 12.5.5): hash some data, encrypt with the hash result as key, hash again

`cMsgHashedNEncryptedNHashed`   The triple operation has been done.

`cMsgVerifyNDecryptNVerify`   Reverse of the above: check the outer hash, decrypt the data, check the inner hash.

`cMsgVerifiedNDecryptedNVerified`   Reverse operation done.

Message `cFsoPersistenceRequest`

This message orders to make an FSO persistant.

Message `cIFSMessage`

This is the base class of all messages used in relation with the block cache.

`cMsgBlockRequestFromBfToBc`   The outside world needs a block, and requests the block with this message.

`cMsgBlockResponseFromBcToBf`   Maybe we have the block, then this message contains it. Otherwise the message transports the negative result.

`cBCJobContainer`   Messages of type `cBCJobContainer` transport requests between the block cache and its sub-modules.

`cBCDBUpdateMessage`   The block cache updated something on disk and needs to tell the block cache data base about.

`cBCDBResponse`   Confirmation that something in the data base changed.

`cBCFSCKCommand`   Start a check of all on-disk structures of the block cache.

`cBCFSCKSweepComplete`   Ends such a check.

`cBCShutdownMessage`   This is the end...

`cInventoryUpdateMsg`   The block cache state changed and the pointer cache or other modules need to know about this.

Message `cIgorInterfaceMessage`

All messages that are serializable and can be transfered via Igor are derived from this base class.

`cMsgBlockRequestFromBfToBf`   Block requests between different IgorFS instances are transfered with the help of this message.

`cMsgBlockResponseFromBfToBf`   Here, the actual block transfer (or a negative reply) happens.

`cMsgPointerAnnounceFromPcToPc`   We have a pointer and tell other IgorFS systems about.

`cMsgPointerRequestFromPcToPc`   This IgorFS needs a block. Before the block can be requested, a pointer to it is required.

`cMsgPointerResponseFromPcToPc`   With this message, pointer requests are answered.

Message `cMsgBlockCutRequest`

Request from the file system to the block cut algorithm to cut this collection of blocks.

Message `cMsgBlockCutResponse`

Answer to the above.

Message `cMsgBlockRequestFromBcToBf`

The local block cache has a search request for a block and can not answer itself so it needs outside help.

Message `cMsgPcCleanUp`

Timer message in the pointer cache to time out stale pointers.

Message `cMsgPcPeriodicAnnouncement`

Pointers are periodically announced. This message is the timer for this periodic action.

Message `cMsgPcQueueUpdateStatus`

Used to time out requests in the pointer cache.

Message `cMsgPointerRequestFromBfToPc`

Going to request pointers from other IgorFS instances.

Message `cMsgPointerResponseFromPcToBf`

Going to request pointers from other IgorFS instances.

Message `cMsgReadReqFFH`

A block needs to be read by the file system.

Message `cMsgReqBase`

Base class from for messages between fuse interface and file/folder module.

`cMsgReqModDev`   Request to change the mode of a FSO.

`cMsgReqTwoNames`   A request with two paths, e. g. link, rename, etc.

`cMsgReqOffSetSize`   A request with offset and size, e. g. read.

`cMsgReqTwoTimes`   Change access times of a FSO.

Message `cMsgResBase`

All answers to `cMsgReqBase` are derived from `cMsgResBase`.

`cMsgResName`   Return just a name.

`cMsgResReadOnly`   Something has been read.

`cMsgResStat`   The status of a file system object is returned.

`cMsgResStatfs`   The status of the entire file system is reported.

`cMsgResVector`   Directory content.

Message `cMsgWriteReqFFH`

A block needs to be written to disk.

Message `cSnapShotRequest`

Start a snapshot, i. e. a new epoch.

Message `cSnapShotReply`

A snapshot is complete.

Message `cSnapShotTimer`

Internal timer to snapshot initiator module: the next snapshot is due.

List of Figures

List of Tables

Index

Glossary

⊕ ....Exclusive Or
Igor ....Internet Grid Overlay Routing
AES ....Advanced Encryption Standard
AFS ....Andrew File System
Alice ....Canonic sender in cryptography
API ....Application Programming Interface
Bob ....Canonic receiver in cryptography
BSD ....Berkeley Software Distribution
CA ....Certification Authority
CAN ....Content Addressable Network
CBC ....Cipher Block Chaining
CCDF ....Complementary Cumulative Distribution Function
CDF ....Cumulative Distribution Function
CFB ....Cipher Feedback
CHK ....Content Hash Key
ciphertext ....The transformation of a clear text with an encryption function
CRC ....Cyclic Redundancy Check
CTR ....Counter Mode
DCCP ....Datagram Congestion Control Protocol
DCE ....Distributed Computing Environment
DFG ....Deutsche Forschungsgemeinschaft
DFS ....Distributed File System
DHT ....Distributed Hash Table
DMZ ....Demilitarized Zone
DNS ....Domain Name System
DOLR ....Distributed Object Location and Retrieval
DRM ....Digital Rights Management
ECB ....Electronic Codebook
EPG ....Electronic Program Guide
Eve ....Canonic eavesdropper in cryptography
exbi-byte ....2^{60} Bytes [22]
FSO ....file system object
FUSE ....File System in User Space
GNP ....Global Network Positioning
GNU ....GNU is not Unix

GOP . . . . Group of Pictures
Gossiping Protocol A protocol where neighboring nodes often chat with each other
GPFS . . . . Global Parallel File System
Greedy algorith . . . An algorithm that chooses the local optimum at every step
HMAC . . . . Hash-MAC
HTTP . . . . Hypter Text Transport Protocol
ICMP . . . . Internet Control Message Protocol
Inode . . . . Index Node
IPC . . . . Inter-Process Communication
IV . . . . Initialization Vector
kibi-byte . . . . 2^{10} Bytes [22]
KSK . . . . Keyword Signed Key
LRU . . . . Least Recently Used
LVCT . . . . Least Value based on Caching Time
MAC . . . . Message Authentication Code
Mallory . . . . Canonic malicious party in cryptography
MD5 . . . . Message Digest Function 5
MDS . . . . Meta Data Server
NAT . . . . Network Address Translation
NFS . . . . Network File System
OFB . . . . Output Feedback
OLRU . . . . Object LRU
OSS . . . . Object Storage Server
P2P . . . . Peer-to-Peer
Padding . . . . Adding bits or bytes for security or alignment reasons
PDF . . . . Portable Document Format
PNS . . . . Proximity Neighbor Selection
POTN . . . . Plain Old Telephone Network
PRS . . . . Proximity Route Selection
RPC . . . . Remote Procedure Call
RTP . . . . Real-time Transport Protocol
RTT . . . . Round Trip Time
S-box . . . . Substitution Box
SCTP . . . . Stream Control Transport Protocol
SIC . . . . Segmented Integer Counter
Snapshot . . . . Preserving the state of the system as it was at one point in time
SSK . . . . Signed Subspace Key
STL . . . . Standard Template Library
TCP . . . . Transmission Control Protocol
TLA . . . . Tree Letter Acronym
UDP . . . . User Datagram Protocol
WEP . . . . Wired Equivalence Privacy
WPA . . . . WiFi Protected Access

Bibliography

[1] Stephen Adler. The Slashdot Effect – An Analysis of Three Internet Publications, January 1999. `http://ssadler.phy.bnl.gov/adler/SDE/SlashDotEffect.html`, accessed 2007-10-12.

[2] Akamai: The Leader in Web Application Acceleration and Performance Management, Streaming Media Services and Content Delivery, 2007. `http://www.akamai.com`, accessed 2007-10-19.

[3] E. Allman, J. Callas, M. Delany, M. Libbey, J. Fenton, and M. Thomas. DomainKeys Identified Mail (DKIM) Signatures. RFC 4871 (Proposed Standard), May 2007.

[4] D. Atkins, W. Stallings, and P. Zimmermann. PGP Message Exchange Formats. RFC 1991 (Informational), August 1996. obsoleted by RFC 4880.

[5] R. Baldwin and R. Rivest. The RC5, RC5-CBC, RC5-CBC-Pad, and RC5-CTS Algorithms. RFC 2040 (Informational), October 1996.

[6] Rudolf Bayer. Binary B-Trees for Virtual Memory. In Edgar F. Codd and Albert L. Dean, editors, *SIGFIDET Workshop*, pages 219–235. ACM, 1971.

[7] Peter Biddle, Paul England, Marcus Peinado, and Bryan Willman. The darknet and the future of content protection. In Eberhard Becker, Willms Buhse, Dirk Günnewig, and Niels Rump, editors, *Digital Rights Management – Technological, Economic, Legal and Political Aspects*, volume 2770 of *LNCS*, pages 344–365, Berlin, November 2003.

[8] Philippe Biondi and Fabrice Desclaux. Silver needle in the skype. Black-Hat Europe, 2006.

[9] D. M. Bloom. A birthday problem. *American Mathematical Monthly*, pages 1141–1142, 1973.

[10] Peter J. Braam. Lustre File System. Technical report, Cluster File Systems, Inc., July 2007. Version 2.

[11] Bundesnetzagentur für Elektrizität, Gas, Telekommunikation, Post und Eisenbahnen. Bekanntmachung zur elektronischen Signatur nach dem Signaturgesetz und der Signaturverordnung (Übersicht über geeignete Algorithmen). Bundesanzeiger Nr. 69, Seiten 3759–3761, February 2007.

[12] Matthew Caesar, Miguel Castro, Edmund B. Nightingale, Greg O'Shea, and Antony Rowstron. Virtual Ring Routing: Network Routing Inspired by DHTs. In *Proc. ACM SIGCOMM '06*, Pisa, Italy, September 2006.

[13] B. Callaghan, B. Pawlowski, and P. Staubach. NFS Version 3 Protocol Specification. RFC 1813 (Informational), June 1995.

[14] R. Callon. The Twelve Networking Truths. RFC 1925 (Informational), April 1996.

[15] Nancy Cam-Winget, Russell Housley, David Wagner, and Jesse Walker. Security flaws in 802.11 data link protocols. *Commun. ACM*, 46(5):35–39, 2003.

[16] Rémy Card, Theodore Ts'o, and Stephen Tweedie. Design and implementation of the second extended filesystem. In *Proceedings of the First Dutch International Symposium on Linux*, December 1994.

[17] David Chaum. Untraceable electronic mail, return addresses, and digital pseudonyms. *Communications of the ACM*, 4(2), February 1981.

[18] Ian Clarke, Theodore W. Hong, Scott G. Miller, Oskar Sandberg, and Brandon Wiley. Protecting Free Expression Online with Freenet. *IEEE Internet Computing*, 6(1):40–49, 2002.

[19] Ian Clarke, Oskar Sandberg, Brandon Wiley, and Theodore W. Hong. Freenet: A Distributed Anonymous Information Storage and Retrieval System. *Lecture Notes in Computer Science*, 2009, 2001.

[20] Kenneth L. Clarkson. Nearest-Neighbor Searching and Metric Space Dimensions. In Gregory Shakhnarovich, Trevor Darrell, and Piotr Indyk, editors, *Nearest-Neighbor Methods for Learning and Vision: Theory and Practice*, pages 15–59. MIT Press, 2006.

[21] Bram Cohen. Incentives Build Robustness in BitTorrent. Technical report, bittorrent.org, 2003.

[22] International Electrotechnical Commission. Addendum to IEC 60027-2 on binary prefixes, 2000.

[23] Michael Conrad and Hans-Joachim Hof. A Generic, Self-Organizing, and Distributed Bootstrap Service for Peer-to-Peer Networks. In *Proceedings of New Trends in Network Architectures and Services: 2nd International Workshop on Self-Organizing Systems (IWSOS 2007)*, September 2007.

[24] A. Conta and S. Deering. Internet Control Message Protocol (ICMPv6) for the Internet Protocol Version 6 (IPv6) Specification. RFC 2463 (Draft Standard), December 1998. obsoleted by RFC 4443.

[25] A. Conta, S. Deering, and M. Gupta. Internet Control Message Protocol (ICMPv6) for the Internet Protocol Version 6 (IPv6) Specification. RFC 4443 (Draft Standard), March 2006. updated by RFC 4884.

[26] Microsoft Corporation. IFS Kit - Installable File System Kit. `http://www.microsoft.com/whdc/DevTools/IFSKit/default.mspx`, accessed 2007-08-19.

[27] Russ Cox, Frank Dabek, Frans Kaashoek, Jinyang Li, and Robert Morris. Practical, distributed network coordinates. In *Proceedings of the Second Workshop on Hot Topics in Networks (HotNets-II)*, Cambridge, Massachusetts, November 2003. ACM SIGCOMM.

[28] Curt Cramer, Kendy Kutzner, and Thomas Fuhrmann. Bootstrapping locality-aware p2p networks. In *Proceedings of the IEEE International Conference on Networks (ICON 2004)*, volume 1, pages 357–361, Singapore, November 16–19 2004.

[29] Curt Cramer, Kendy Kutzner, and Thomas Fuhrmann. Distributed Job Scheduling in a Peer-to-Peer Video Recording System. In *Proceedings of the Workshop on Algorithms and Protocols for Efficient Peer-to-Peer Applications (PEPPA) at Informatik 2004*, pages 234–238, Ulm, Germany, September 23 2004.

[30] Frank Dabek. A cooperative file system. Master's thesis, Massachusetts Institute of Technology, September 2001.

[31] Frank Dabek, M. Frans Kaashoek, David Karger, Robert Morris, and Ion Stoica. Wide-area cooperative storage with CFS. In *Proceedings of the 18th ACM Symposium on Operating Systems Principles (SOSP '01)*, Chateau Lake Louise, Banff, Canada, October 2001.

[32] Frank Dabek, Jinyang Li, Emil Sit, James Robertson, M. Frans Kaashoek, and Robert Morris. Designing a DHT for low latency and high throughput. In *Proceedings of the 1st USENIX Symposium on Networked Systems Design and Implementation (NSDI '04)*, San Francisco, California, March 2004.

[33] Frank Dabek, Ben Zhao, Peter Druschel, John Kubiatowicz, and Ion Stoica. Towards a Common API for Structured Peer-to-Peer Overlays. In *Proceedings of the 2nd International Workshop on Peer-to-Peer Systems (IPTPS '03)*, Berkeley, CA, USA, 2003.

[34] Joan Daemen and Vincent Rijmen. *The design of Rijndael: AES — the Advanced Encryption Standard*. Springer-Verlag, 2002.

[35] Hauke Dämfpling. Gnutella Web Caching System, July 2002. http://www.gnucleus.com/gwebcache/specs.html, accessed 11 March 2004.

[36] Ivan Bjerre Damgård. A design principle for hash functions. In *CRYPTO '89: Proceedings on Advances in cryptology*, pages 416–427, New York, NY, USA, 1989. Springer-Verlag New York, Inc.

[37] Magnus Daum and Stefan Lucks. Attacking Hash Functions by Poisoned Messages "The Story of Alice and her Boss", 2005. `http://www.cits.rub.de/MD5Collisions`, accessed 2007-03-28.

[38] Benoit des Ligneris. Virtualization of Linux Based Computers: The Linux-VServer Project. In *HPCS '05: Proceedings of the 19th International Symposium on High Performance Computing Systems and Applications*, pages 340–346, Washington, DC, USA, 2005. IEEE Computer Society.

[39] P. Deutsch and J-L. Gailly. ZLIB Compressed Data Format Specification version 3.3. RFC 1950 (Informational), May 1996.

[40] Robert Devine. Design and implementation of ddh: A distributed dynamic hashing algorithm. In *FODO '93: Proceedings of the 4th International Conference on Foundations of Data Organization and Algorithms*, pages 101–114, London, UK, 1993. Springer-Verlag.

[41] T. Dierks and E. Rescorla. The Transport Layer Security (TLS) Protocol Version 1.1. RFC 4346 (Proposed Standard), April 2006. updated by RFCs 4366, 4680, 4681.

[42] Roger Dingledine, Nick Mathewson, and Paul F. Syverson. Tor: The Second-Generation Onion Router. In *USENIX Security Symposium*, pages 303–320. USENIX, 2004.

[43] Tamar Domani, Elliot K. Kolodner, Ethan Lewis, Eliot E. Salant, Katherine Barabash, Itai Lahan, Yossi Levanoni, and Erez Petrank. Implementing an on-the-fly garbage collector for java. In *ISMM*, pages 155–166, 2000.

[44] Morris Dworkin. Recommendation for Block Cipher Modes of Operation. Technical report, National Institute of Standards and Technology, 2001.

[45] ECMA. *ECMA-107: Volume and File Structure of Disk Cartridges for Information Interchange*. ECMA (European Association for Standardizing Information and Communication Systems), Geneva, Switzerland, second edition, June 1995.

[46] Johannes Eickhold. Entwicklung einer SSR-basierten Peer-to-Peer-Telefonieanwendung für das Nokia 770. Master's thesis, System Architecture Group, University of Karlsruhe, Germany, 2006.

[47] Johannes Eickhold. Linyphone: Voice over P2P net with SIP on Nokia 770, 2007. `http://linyphone.net`, accessed 2007-11-01.

[48] M. Eisler. XDR: External Data Representation Standard. RFC 4506 (Standard), May 2006.

[49] Per Cederqvist et. al . *Version Management with CVS*, 2007. `http://ximbiot.com/cvs/manual/stable`, accessed 2007-07-08.

[50] Thure Etzold and Patrick Argos. SRS – an indexing and retrieval tool for flat file data libraries. *Computer Applications In The Biosciences: CABIOS*, 9(1):49–57, February 1993.

[51] Roger Faulkner and Ron Gomes. The Process File System and Process Model in UNIX System V. In *USENIX Winter*, pages 243–252, 1991.

[52] C. Feather. Network News Transfer Protocol (NNTP). RFC 3977 (Proposed Standard), October 2006.

[53] Niels Ferguson, John Kelsey, Stefan Lucks, Bruce Schneier, Mike Stay, David Wagner, and Doug Whiting. Improved cryptanalysis of Rijndael. In *Seventh Fast Software Encryption Workshop*, page 19. Springer-Verlag, 2000.

[54] Niels Ferguson, Richard Schroeppel, and Doug Whiting. A simple algebraic representation of Rijndael. *Lecture Notes in Computer Science*, 2259, 2001.

[55] John Georg Fletcher. An arithmetic checksum for serial transmissions. *IEEE Transactions on Communications*, 30:247–252, January 1982.

[56] Paul Francis, Sugih Jamin, Cheng Jin, Yixin Jin, Danny Raz, Yuval Shavitt, and Lixia Zhang. IDMaps: A Global Internet Host Distance Estimation Service. *IEEE/ACM Transactions on Networking*, 9(5):525–540, 2001.

[57] Michael J. Freedman, Eric Freudenthal, and David Mazières. Democratizing content publication with Coral. In *Proceedings of the 1st USENIX Symposium on Networked Systems Design and Implementation (NSDI '04)*, San Francisco, California, March 2004.

[58] Research Group Fuhrmann. Videgor, 2007. System Architecture Group, University of Karlsruhe, `http://www.videgor.net`, accessed 2007-02-22.

[59] Thomas Fuhrmann. A self-organizing routing scheme for random networks. In *Proceedings of the 4th IFIP-TC6 Networking Conference*, pages 1366–1370, Waterloo, Canada, May 2–6 2005.

[60] Thomas Fuhrmann and Jörg Widmer. Extremum feedback with partial knowledge. In Burkhard Stiller, Georg Carle, Martin Karsten, and Peter Reichl, editors, *Networked Group Communication*, volume 2816 of *Lecture Notes in Computer Science*, pages 181–192. Springer, 2003.

[61] Richard Gooch and Pekka Enberg. Overview of the linux virtual file system, October 2005. Part of the Linux kernel source at Documentation/filesystems/vfs.txt.

[62] Krishna Gummadi, Ramakrishna Gummadi, Steve Gribble, Sylvia Ratnasamy, Scott Shenker, and Ion Stoica. The Impact of DHT Routing Geometry on Resilience and Proximity. In *Proceedings of the SIGCOMM 2003 conference*, pages 381–394. ACM Press, 2003.

[63] Ulrich Hahn, Werner Dilling, and Dietmar Kaletta. Improved adaptive replacement algorithm for disk-caches in HSM systems. In *IEEE Symposium on Mass Storage Systems*, pages 128–140, 1999.

[64] Kirsten Hildrum, John Kubiatowicz, Sean Ma, and Satish Rao. A Note on the Nearest Neighbor in Growth-Restricted Metrics. In *In proceedings of the Symposium on Discrete Algorithms*, January 2004.

[65] Kirsten Hildrum, John D. Kubiatowicz, Satish Rao, and Ben Y. Zhao. Distributed Object Location in a Dynamic Network. In *Proceedings of the fourteenth annual ACM symposium on Parallel algorithms and architectures*, pages 41–52. ACM Press, 2002.

[66] M.R. Horton and R. Adams. Standard for interchange of USENET messages. RFC 1036, December 1987.

[67] R. Housley. Cryptographic Message Syntax (CMS). RFC 3852 (Proposed Standard), July 2004. updated by RFCs 4853, 5083.

[68] John H. Howard. An Overview of the Andrew File System. In *USENIX Winter Technical Conference*, February 1988.

[69] NetBIOS Working Group in the Defense Advanced Research Projects Agency, Internet Activities Board, and End to End Services Task Force. Protocol standard for a NetBIOS service on a TCP/UDP transport: Concepts and methods. RFC 1001 (Standard), March 1987.

[70] NetBIOS Working Group in the Defense Advanced Research Projects Agency, Internet Activities Board, and End to End Services Task Force. Protocol standard for a NetBIOS service on a TCP/UDP transport: Detailed specifications. RFC 1002 (Standard), March 1987.

[71] ISO/IEC. *ISO/IEC 7498-1:1994 Information technology — Open Systems Interconnection — Basic Reference Model: The Basic Model.* International Organization for Standardization / International Electrotechnical Commission, 1994.

[72] ISO/IEC. *ISO/IEC 9293:1994 Information technology – Volume and file structure of disk cartridges for information interchange.* International Organization for Standardization / International Electrotechnical Commission, 1994.

[73] Song Jiang and Xiaodong Zhang. Efficient Distributed Disk Caching in Data Grid Management. In *CLUSTER*, pages 446–451. IEEE Computer Society, 2003.

[74] Brad Curtis Johnson. A Distributed Computing Environment Framework: An OSF Perspective. Technical Report DEV-DCE-TP6-1, Open Software Foundation, June 1991.

[75] Gilles Kahn. The semantics of simple language for parallel programming. In *IFIP Congress*, pages 471–475, 1974.

[76] B. Kaliski. The MD2 Message-Digest Algorithm. RFC 1319 (Informational), April 1992.

[77] B. Kaliski. PKCS #7: Cryptographic Message Syntax Version 1.5. RFC 2315 (Informational), March 1998.

[78] B. Kaliski. PKCS #5: Password-Based Cryptography Specification Version 2.0. RFC 2898 (Informational), September 2000.

[79] Gene Kan. Gnutella. In Andy Oram, editor, *Peer-to-Peer. Harnessing the Power of Disruptive Technologies*, pages 94–122. O'Reilly, Sebastopol, CA, 2001.

[80] Pradnya Karbhari, Mostafa Ammar, Amogh Dhamdhere, Himanshu Raj, George Riley, and Ellen Zegura. Bootstrapping in Gnutella: A Measurement Study. In *Proceedings of the PAM 2004 workshop*, Antibes Juan-les-Pins, France, April 2004. Springer-Verlag.

[81] David Karger, Eric Lehman, Tom Leighton, Rina Panigrahy, Matthew Levine, and Daniel Lewin. Consistent hashing and random trees: distributed caching protocols for relieving hot spots on the World Wide Web. In *Proceedings of the twenty-ninth annual ACM symposium on Theory of computing*, pages 654–663. ACM Press, 1997.

[82] Brad Karp and H. T. Kung. GPSR: greedy perimeter stateless routing for wireless networks. In *Mobile Computing and Networking*, pages 243–254, 2000.

[83] Richard M. Karp and Michael O. Rabin. Efficient randomized pattern-matching algorithms. *IBM Journal of Research and Development*, 31(2):249–260, 1987.

[84] Daishi Kato and Toshiyuki Kamiya. Evaluating DHT Implementations in Complex Environments by Network Emulator. In *International workshop on Peer-To-Peer Systems (IPTPS 2007)*, Bellevue, WA, USA, February 2007.

[85] S. Kent and K. Seo. Security Architecture for the Internet Protocol. RFC 4301 (Proposed Standard), December 2005.

[86] Tom J. Killian. Processes as files. In *Proceedings of the Summer 1984 USENIX Conference*, 1984.

[87] Yves Philippe Kising. Proximity neighbor selection and proximity route selection for the overlay-network igor. Diploma thesis, Network Architectures, Technical University Munich, Germany, June 15 2007.

[88] Michael Klein, Birgitta König-Ries, and Philipp Obreiter. Lanes - a lightweight overlay for service discovery in mobile ad hoc networks. In *Proceedings of the 3rd Workshop on Applications and Services in Wireless Networks (ASWN 2003)*, Berne, Switzerland, July 2003.

[89] Jon Kleinberg. The Small-World Phenomenon: An Algorithmic Perspective. In *Proceedings of the 32nd ACM Symposium on Theory of Computing*, 2000.

[90] Sander Klous, Jaime Frey, Se-Chang Son, Douglas Thain, Alain Roy, Miron Livny, and Jo van den Brand. Transparent access to grid resources for user software. *Concurrency and Computation: Practice and Experience*, 18(7):787–801, 2006.

[91] E. Kohler, M. Handley, and S. Floyd. Datagram Congestion Control Protocol (DCCP). RFC 4340 (Proposed Standard), March 2006.

[92] Ivan Kostov. Entwicklung einer automatisierten Testumgebung fuer das Overlaynetz IGOR, 2006.

[93] H. Krawczyk, M. Bellare, and R. Canetti. HMAC: Keyed-Hashing for Message Authentication. RFC 2104 (Informational), February 1997.

[94] John Kubiatowicz, David Bindel, Yan Chen, Steven Czerwinski, Patrick Eaton, Dennis Geels, Ramakrishna Gummadi, Sean Rhea, Hakim Weatherspoon, Westley Weimer, Chris Wells, and Ben Zhao. OceanStore: An architecture for global-scale persistent storage. In *Proceedings of the Ninth international Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS 2000)*, November 2000.

[95] Kendy Kutzner, Curt Cramer, and Thomas Fuhrmann. A self-organizing job scheduling algorithm for a distributed vdr. In *Workshop "Peer-to-Peer-Systeme und -Anwendungen", 14. Fachtagung Kommunikation in Verteilten Systemen (KiVS 2005)*, Kaiserslautern, Germany, February 2005.

[96] Kendy Kutzner and Thomas Fuhrmann. Measuring large overlay networks - the overnet example. In *Konferenzband der 14. Fachtagung Kommunikation in Verteilten Systemen (KiVS 2005)*, Kaiserslautern, Germany, 2005.

[97] Leslie Lamport. Constructing digital signatures from a one-way function. Technical Report SRI-CSL-98, SRI International Computer Science Laboratory, Palo Alto, October 1979.

[98] Leslie Lamport, Robert Shostak, and Marshall Pease. The Byzantine Generals Problem. *ACM Transactions on Programming Languages and Systems*, 4(3):382–401, 1982.

[99] P. L'Ecuyer. A table of linear congruential generators of different sizes and good lattice structure. *Mathematics of Computation*, 1999.

[100] Norbert Leser. Towards a Worldwide Distributed File System. Technical Report DEV-DCE-TP4-1, DCE Evaluation Team, Open Software Foundation, September 1990.

[101] Jinyang Li, Jeremy Stribling, Robert Morris, and M. Frans Kaashoek. Bandwidth-efficient management of DHT routing tables. In *Proceedings of the 2nd USENIX Symposium on Networked Systems Design and Implementation (NSDI '05)*, Boston, Massachusetts, May 2005.

[102] Tim Lindholm and Frank Yellin. *The Java Virtual Machine Specification*. Addison-Wesley, Second edition, 1999.

[103] J. Linn. Generic Security Service Application Program Interface Version 2, Update 1. RFC 2743 (Proposed Standard), January 2000.

[104] Witold Litwin, Marie-Anna Neimat, and Donovan A. Schneider. LH* – a scalable, distributed data structure. *ACM Transactions on Database Systems*, 21(4):480–525, 1996.

[105] Harsha V. Madhyastha, Thomas Anderson, Arvind Krishnamurthy, Neil Spring, and Arun Venkataramani. A structural approach to latency prediction. In *IMC '06: Proceedings of the 6th ACM SIGCOMM on Internet measurement*, pages 99–104, New York, NY, USA, 2006. ACM Press.

[106] Dahlia Malkhi, Moni Naor, and David Ratajczak. Viceroy: A Scalable and Dynamic Emulation of the Butterfly. In *Proceedings of the 21st ACM Symposium on Principles of Distributed Computing*, 2002.

[107] Gurmeet Singh Manku, Mayank Bawa, and Prabhakar Raghavan. Symphony: Distributed hashing in a small world. In *Proceedings of the 4th USENIX Symposium on Internet Technologies and Systems (USITS 2003)*, 2003.

[108] Silvano Martello and Paolo Toth. *Knapsack Problems: Algorithms and Computer Implementations*. Wiley Interscience Series in Discrete Mathematics and Optimization. John Wiley & Sons Inc, 1990.

[109] Avantika Mathur, Mingming Cao, Suparna Bhattacharya, Andreas Dilger, and Alex Tomas Laurent Vivier. The new ext4 filesystem: current status and future plans. In *Proceedings of the Linux Symposium*, Ottawa, Ontario, Canada, June 2007.

[110] Petar Maymounkov and David Mazières. Kademlia: A Peer-to-Peer Information System Based on the XOR Metric. In *Revised Papers from the First International Workshop on Peer-to-Peer Systems*, pages 53–65. Springer-Verlag, 2002.

[111] David A. McGrew and John Viega. The security and performance of the galois/counter mode (gcm) of operation. In Anne Canteaut and Kapalee Viswanathan, editors, *INDOCRYPT*, volume 3348 of *Lecture Notes in Computer Science*, pages 343–355. Springer, 2004.

[112] Ralph C. Merkle. *Secrecy, authentication, and public key systems*. PhD thesis, Department of Electrical Engineering, Stanford University, 1979.

[113] Sun Microsystems. XDR: External Data Representation standard. RFC 1014, June 1987.

[114] Sun Microsystems. RPC: Remote Procedure Call Protocol specification: Version 2. RFC 1057 (Informational), June 1988.

[115] Sun Microsystems. NFS: Network File System Protocol specification. RFC 1094 (Informational), March 1989.

[116] Gordon E. Moore. Cramming more components onto integrated circuits. *Electronics Magazine*, April 1965.

[117] James H. Morris, Mahadev Satyanarayanan, Michael H. Conner, John H. Howard, David S. Rosenthal, and F. Donelson Smith. Andrew: A Distributed Personal Computing Environment. *Commun. ACM*, 29(3):184–201, 1986.

[118] John Morris, James Walker, and Bostjan Marusic. Share it! A rights managed network of peer-to-peer set top boxes system architecture. In *Proceedings of the International Broadcasting Convention (IBC) 2003*, Amsterdam, September 2003.

[119] Steve Muir. The seven deadly sins of distributed systems. In *First Workshop on Real, Large Distributed Systems*, 2004.

[120] Athicha Muthitacharoen, Robert Morris, Thomer Gil, and Benjie Chen. Ivy: A Read/Write Peer-to-peer File System. In *Proceedings of the 5th USENIX Symposium on Operating Systems Design and Implementation (OSDI '02)*, Boston, Massachusetts, December 2002.

[121] National Institute of Standards and Technology. Secure Hash Standard. Federal Information Processing Standards Publication 180-1, April 1995.

[122] National Institute of Standards and Technology. Specification for the Advanced Encryption Standard (AES). Federal Information Processing Standards Publication 197, 2001.

[123] The Network Simulator ns2. `http://nsnam.isi.edu/nsnam/`, accessed 2007-09-15.

[124] C. Neuman, T. Yu, S. Hartman, and K. Raeburn. The Kerberos Network Authentication Service (V5). RFC 4120 (Proposed Standard), July 2005. updated by RFCs 4537, 5021.

[125] Mej Newman. Power laws, Pareto distributions and Zipf's law. *Contemporary Physics*, 46(5):323–351, September 2005.

[126] T. S. E. Ng and Hui Zhang. Predicting Internet Network Distance with Coordinates-Based Approaches. In *Proceedings of the Twenty-First Annual Joint Conference of the IEEE Computer and Communications Societies (INFOCOM) 2002*, volume 1, pages 170–179, 2002.

[127] Seth Nickell. GnomeVFS - Filesystem Abstraction library. `http://developer.gnome.org/doc/API/gnome-vfs/` accessed 2007-07-06.

[128] Institute of Electrical and Electronics Engineers. IEEE Standard for Information technology-Telecommunications and information exchange between systems-Local and metropolitan area networks-Specific requirements - Part 11: Wireless LAN Medium Access Control (MAC) and Physical Layer (PHY) Specifications, 2007. IEEE Std 802.11-2007 (Revision of IEEE Std 802.11-1999).

[129] L. Ong and J. Yoakum. An Introduction to the Stream Control Transmission Protocol (SCTP). RFC 3286 (Informational), May 2002.

[130] Open Software Foundation, Inc. File Systems in a Distributed Computing Environment, July 1991.

[131] Larry Peterson, Tom Anderson, David Culler, and Timothy Roscoe. A Blueprint for Introducing Disruptive Technology into the Internet. In *Proceedings of HotNets–I*, Princeton, New Jersey, October 2002.

[132] C. Greg Plaxton, Rajmohan Rajaraman, and Andréa W. Richa. Access-
ing nearby copies of replicated objects in a distributed environment. In
*Proceedings of the ninth annual ACM symposium on Parallel algorithms
and architectures*, pages 311–320. ACM Press, 1997.

[133] J. Postel. User Datagram Protocol. RFC 768 (Standard), August 1980.

[134] J. Postel. Internet Control Message Protocol. RFC 792 (Standard),
September 1981. updated by RFCs 950, 4884.

[135] J. Postel. Transmission Control Protocol. RFC 793 (Standard), September
1981. updated by RFC 3168.

[136] J. Postel and K. Harrenstien. Time Protocol. RFC 868 (Standard), May
1983.

[137] Sylvia Ratnasamy, Paul Francis, Mark Handley, Richard Karp, and Scott
Shenker. A Scalable Content-Addressable Network. In *Proceedings of the
SIGCOMM 2001 conference*, pages 161–172. ACM Press, 2001.

[138] Sean Rhea, Patrick Eaton, Dennis Geels, Hakim Weatherspoon, Ben Zhao,
and John Kubiatowicz. Pond: The oceanstore prototype. In *Proceedings
of the 2nd USENIX Conference on File and Storage Technologies*, pages
1–14, San Francisco, CA, USA, 2003.

[139] Sean Rhea, Dennis Geels, Timothy Roscoe, and John Kubiatowicz. Han-
dling churn in a DHT. In *Proceedings of the 2004 USENIX Annual Tech-
nical Conference*, Boston, Massachusetts, June 2004.

[140] Sean C. Rhea, Brighten Godfrey, Brad Karp, John Kubiatowicz, Sylvia
Ratnasamy, Scott Shenker, Ion Stoica, and Harlan Yu. OpenDHT: a
public DHT service and its uses. In Roch Guérin, Ramesh Govindan, and
Greg Minshall, editors, *SIGCOMM*, pages 73–84. ACM, 2005.

[141] R. Rivest. The MD5 Message-Digest Algorithm. RFC 1321 (Informa-
tional), April 1992.

[142] Ronald L. Rivest, Adi Shamir, and Leonard M. Adleman. A method for
obtaining digital signatures and public-key cryptosystems. *Communica-
tions of the ACM*, 21(2):120–126, 1978.

[143] J. Rosenberg, H. Schulzrinne, G. Camarillo, A. Johnston, J. Peterson,
R. Sparks, M. Handley, and E. Schooler. SIP: Session Initiation Protocol.
RFC 3261 (Proposed Standard), June 2002. updated by RFCs 3265, 3853,
4320, 4916.

[144] Antony Rowstron and Peter Druschel. Pastry: Scalable, distributed object
location and routing for large-scale peer-to-peer systems. In *Proceedings
of the IFIP/ACM International Conference on Distributed Systems Plat-
forms (Middleware) 2001*, Heidelberg, Germany, November 2001.

[145] Oskar Sandberg. Distributed Routing in Small-World Networks. In *8th Workshop on Algorithm Engineering and Experiments (ALENEX06)*, January 2006.

[146] Axel Sanwald. Entwurf und Implementierung eines Block-Zwischenspeichers für verteilte Dateisysteme, July 15 2006. Study Thesis.

[147] Mahadev Satyanarayanan, James J. Kistler, Puneet Kumar, Maria E. Okasaki, Ellen H. Siegel, and David C. Steere. Coda: A Highly Available File System for a Distributed Workstation Environment. *IEEE Transactions on Computers*, 39(4), April 1990.

[148] Maurice Lorrain Schlumberger. *De-Bruijn Communications Networks.* PhD thesis, 1974.

[149] Klaus Schmidinger. Video Disk Recorder, 2007. `http://www.cadsoft.de/vdr`, accessed 2007-02-23.

[150] Frank Schmuck and Roger Haskin. GPFS: A Shared-Disk File System for Large Computing Clusters. In *Proc. of the First Conference on File and Storage Technologies (FAST)*, pages 231–244, January 2002.

[151] Bruce Schneier. *Applied Cryptography.* John Wiley & Sons, Inc., New York, NY, USA, 1996.

[152] H. Schulzrinne, S. Casner, R. Frederick, and V. Jacobson. RTP: A Transport Protocol for Real-Time Applications. RFC 3550 (Standard), July 2003.

[153] Claude Shannon. Communication Theory of Secrecy Systems. *Bell System Technical Journal*, pages 656–715, 1949.

[154] S. Shepler, B. Callaghan, D. Robinson, R. Thurlow, C. Beame, M. Eisler, and D. Noveck. Network File System (NFS) version 4 Protocol. RFC 3530 (Proposed Standard), April 2003.

[155] Spencer Shepler, Mike Eisler, and Dave Noveck. NFSv4 Minor Version 1, June 2007. Internet Draft draft-ietf-nfsv4-minorversion1-11.txt, expires 2007-12-13.

[156] Sergios Soursos, George D. Stamoulis, and Theodoros Bozios. Distributed Scheduling of Recording Tasks with Interconnected Servers. In *Proceedings of the Third International IFIP-TC6 Networking Conference*, pages 1483–1488, Athens, Greece, May 2004.

[157] R. Sparks. Actions Addressing Identified Issues with the Session Initiation Protocol's (SIP) Non-INVITE Transaction. RFC 4320 (Proposed Standard), January 2006.

[158] Ralf Steinmetz and Klaus Wehrle, editors. *Peer-to-Peer Systems and Applications*, volume 3485 of *Lecture Notes in Computer Science*. Springer, 2005.

[159] W. Richard Stevens. *UNIX Network Programming*. Prentice-Hall, Upper Saddle River, NJ 07458, USA, 1990.

[160] Ion Stoica, Robert Morris, David Liben-Nowell, David Karger, M. Frans Kaashoek, Frank Dabek, and Hari Balakrishnan. Chord: A scalable peer-to-peer lookup service for internet applications. Technical report, MIT Laboratory for Computer Science, PDOS Group, 2002.

[161] Miklos Szeredi. FUSE: Filesystem in userspace. `http://fuse.sf.net`, accessed 2007-07-06.

[162] Douglas Thain and Miron Livny. Parrot: Transparent user-level middleware for data-intensive computing. In *Workshop on Adaptive Grid Middleware*, New Orleans, Louisiana, September 2003.

[163] Manuel Thiele. Entwicklung eines sicheren Protokolls für ein Peer-to-Peer-Hotspot–Accounting-System. Diploma thesis, Institute of Telematics, University of Karlsruhe (TH), Germany, February 15 2005.

[164] Manuel Thiele, Kendy Kutzner, and Thomas Fuhrmann. Churn resistant de bruijn networks for wireless on demand systems. In *Proceedings of the Third Annual Conference on Wireless On demand Network Systems and Services*, Les Ménuires, France, January 18–20 2006.

[165] Andrew Tridgell. *Efficient Algorithms for Sorting and Synchronization*. PhD thesis, The Australian National University, 1999.

[166] Theodore Y. Ts'o and Stephen Tweedie. Planned extensions to the linux ext2/ext3 filesystem. In Chris G. Demetriou, editor, *USENIX Annual Technical Conference, FREENIX Track*, pages 235–243. USENIX, 2002.

[167] Dominik Vallendor. Service Oriented Message Routing for the Structured Overlay Network Igor. Study thesis, System Architecture Group, University of Karlsruhe, Germany, July 10 2007.

[168] András Varga. The OMNeT++ discrete event simulation system. In *Proceedings of the European Simulation Multiconference (ESM'2001). June 6-9, 2001. Prague, Czech Republic*, 2001.

[169] Limin Wang, KyoungSoo Park, Ruoming Pang, Vivek S. Pai, and Larry Peterson. Reliability and Security in the CoDeeN Content Distribution Network. In *Proceedings of the USENIX 2004 Annual Technical Conference*, Boston, MA, June 2004.

[170] Xiaoyun Wang and Hongbo Yu. How to Break MD5 and Other Hash Functions. In Ronald Cramer, editor, *EUROCRYPT*, volume 3494 of *Lecture Notes in Computer Science*, pages 19–35. Springer, 2005.

[171] Dave Winer. XML/RPC specification. Technical report, Userland Software, 1999.

[172] Bernard Wong, Aleksandrs Slivkins, and Emin Gün Sirer. Meridian: a lightweight network location service without virtual coordinates. In *Proceedings of the ACM SIGCOMM 2005 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications, Philadelphia, Pennsylvania, USA, August 22-26, 2005*, pages 85–96, 2005.

[173] Qi Xia, Ruijun Yang, Weinong Wang, and De Yang. Fully decentralized DHT based approach to grid service discovery using overlay networks. In *Fifth International Conference on Computer and Information Technology (CIT 2005)*, pages 1140–1045, Shanghai, China, September 2005. IEEE Computer Society.

[174] F. Yergeau. UTF-8, a transformation format of ISO 10646. RFC 3629 (Standard), November 2003.

[175] T. Ylonen and C. Lonvick. The Secure Shell (SSH) Protocol Architecture. RFC 4251 (Proposed Standard), January 2006.

[176] Xiang Zeng, Rajive Bagrodia, and Mario Gerla. GloMoSim: A Library for Parallel Simulation of Large-Scale Wireless Networks. In *Workshop on Parallel and Distributed Simulation*, pages 154–161, 1998.

[177] L. Zhu, K. Jaganathan, and S. Hartman. The Kerberos Version 5 Generic Security Service Application Program Interface (GSS-API) Mechanism: Version 2. RFC 4121 (Proposed Standard), July 2005.

[178] Hubert Zimmermann. OSI Reference Model — The ISO Model of Architecture for Open Systems Interconnection. *IEEE Transactions on Communications*, 28(4):425–432, 1980.