Proc. of the 8th IEEE Intl. Conference on Peer-to-Peer Computing (P2P'08)
8-11 Sep 2008, Aachen, Germany

# IgorFs: A Distributed P2P File System

Bernhard Amann, Benedikt Elser, Yaser Houri, and Thomas Fuhrmann
Computer Science Department
Technical University of Munich
Munich, Germany
{ba|elser|houri|fuhrmann}@net.in.tum.de

## Abstract

*IgorFs is a distributed, decentralized peer-to-peer (P2P) file system that is completely transparent to the user. It is built on top of the Igor peer-to-peer overlay network, which is similar to Chord, but provides additional features like service orientation or proximity neighbor and route selection. IgorFs offers an efficient means to publish data files that are subject to frequent but minor modifications. In our demonstration we show two use cases for IgorFs: the first example is (static) software-distribution and the second example is (dynamic) file distribution.*

## 1 Igor

The **I**nternet **G**rid **O**verlay **R**outing network "Igor" is a structured P2P overlay network that provides a *key based routing service* similar to Chord [7]. Unlike a distributed hash table, which offers a publish/subscribe service only, Igor is a service oriented overlay network. It routes messages based on a destination key and a service identifier. It is up to the respective service implementation how these messages shall be handled. Igor efficiently combines multiple services into one overlay network [3]. Nevertheless, only those nodes participate in the routing of a message that run the respective service. This guarantees that different services don't interfere with each other.

Igor uses Proximity Route Selection (PRS) and Proximity Neighbor Selection (PNS) to exploit the proximity of nodes in the underlying Internet topology. Thereby, services can easily benefit from local nodes running the same service.

## 2 IgorFs

The Igor File System (IgorFs) [5] is one of the applications which have been built on top of the Igor overlay network. It uses a completely distributed approach without any central or special nodes. IgorFs provides applications with transparent access to remote storage resources by using Fuse [4], i.e. it can be mounted like any other file system.

IgorFs cuts all the files that it contains into blocks of differing sizes. The cutting marks are determined automatically based on the file's content, e.g. by using a rolling checksum. As a result, modifications to a part of a file affect only a few blocks in the vicinity of the modified part. In particular, IgorFs does not use static block sizes like traditional file systems. Hence, insertions or deletions can be handled efficiently, too.

Each data block $B$ in IgorFs is identified by a 256 bit wide identifier $I$ which is assumed to be globally unique. Each block is encrypted with its own 256 bit key. Both, the ID and the key are obtained by hashing the block with a cryptographic hash function $H$. Thus, independent IgorFs instances map the same clear text block to the same cipher text block. Blocks with identical content have the same identifier $I = H(E_{H(B)}(B))$ where $E$ denotes the encryption function. This is similar to the implementation of *Content Hash Keys* in Freenet [2].

In order to be able to read a block, an IgorFs instance needs to have the block's (ID, key) tuple. In order to be able to read a (sub-)tree in IgorFs, the user needs to know the (ID, key) tuple of the root block of the respective (sub-)tree. From there, the IgorFs instance can then recursively access all files and directories in this (sub-) tree. To this end, directory records contain the (ID, key) tuples of all the files and sub-directories in the directory. IgorFs serializes directory records into blocks and stores them like data blocks.

As a result, the root block changes everytime a file in the file system tree changes. To see this, consider a file $A$ in a directory $/b/c$. If we change a few bytes in this file, at least one of $A$'s blocks changes and $A$ is represented by a different (ID, key) tuple. This change has to be reflected in the block representing the directory $c$. The resulting change then affects $b$ and finally the root block.

This behavior would introduce a lot of overhead for file

operations. In order to reduce this overhead, IgorFs limits the publishing frequency. Modifications only become persistent upon a so called snapshotting event. Other IgorFs instances can only access a persistent state of the file system. Thereby, IgorFs regularly preserves consistent snapshots of the entire file system tree. Each snapshot produces only a limited number of new blocks, namely those blocks that reflect the parts of the file system tree that have been modified since the last snapshot.

When reading from the file system, IgorFs is not restricted to the most up-to-date persistent state. On the contrary, a user may want to explicitly mount an older version to retrieve a file in a previous version. Writing in such a reverted version, forks an independent tree that cannot be merged back again. This feature, i.e. the ability to mount different versions of a file system that share their common part is interesting for many applications. It allows, for example, the efficient provision of software distributions (see below).

Other applications of IgorFs require the instances to always track the most up-to-date version of a file system. Unlike the first case where published content is assumed to remain published, IgorFs provides an efficient means to add and remove subscribers. To accomplish this, IgorFs provides a secure way to exchange the (ID, key) tuple of the current file system revision. This allows the owner of a file system tree to control who may access new file system revisions [1]. This feature is based on the subset revocation method [6].

## 3 Demonstration

In our demonstration we will give a short overview of the main features of IgorFs and possible ways to use them in real-world applications.

In the first part of our demonstration, we present the *central software distribution* scenario. We copy a virtual machine image into IgorFs. This image contains the software distribution for a specific research or development project. The identifier and key for the image are distributed to all members of the research group. Now each group member can use the virtual machine and make his or her own modifications to it. These changes do not affect the base version that we have created. If a user screws up his version, he can always revert to the original version. If a modified version turns out to be more useful than the previous version, this modification can become the new base version.

This usage scenario is particularly suited to distribute static views of files to many people. Besides only sharing static content it enables those people to change these files on their own. Yet, this scenario is not suited to distribute dynamic content that frequently changes over time.

In the second part of our demonstration, we demonstrate the *publish–subscribe capabilities* of IgorFs. We create an IgorFs file system and start several subscriber instances (clients) that track the current file system revisions. Changes to the "master" file tree will then appear at the clients after a few seconds. This feature can be used to publish data that changes over time, for example, data files from an ongoing measurement.

We continue our demonstration with demonstrating the security features of IgorFs. To this end we revoke the access of one of the clients that tracks the file system revisions. After being revoked the client will not be able to access the new revision while all the other clients still have full access.

## 4 Future work

IgorFs and Igor are a subject of ongoing research in our group. The main goal is to make IgorFs support more usage scenarios. The two main fields of our work are multiple writer instances and Unix file system access rights. Multiple writers that concurrently access the same file system pose the problem of maintaining consistency in a fully decentralized setting. This is a hard problem which still awaits a scalable, fully decentralized solution. Access rights for users and groups that reflect the regular Unix file system access rights require new, cryptographically secured data structures besides the current (ID, key) tuple approach. Especially the case where the files in a directory and the directory itself have different owners is difficult to handle. To the best of our knowledge, this case has not been handled by any P2P file system so far.

## References


[1] B. Amann. Secure asynchronous change notifications for a distributed file system. Diplomarbeit, Chair for Network Architectures, Munich University of Technology, 2007.

[2] I. Clarke, O. Sandberg, B. Wiley, and T. w. Hong. Freenet: A distributed anonymous information storage and retrieval system. In *Proc. International Workshop on Design Issues in Anonymity and Unobservability*, volume 2009 of *LNCS*, pages 46–64. Springer Berlin / Heidelberg, 2001.

[3] P. Di, K. Kutzner, and T. Fuhrmann. Providing KBR service for multiple applications. In *Proc. IPTPS '08*, 2008.

[4] File system in userspace. *http://fuse.sourceforge.net*.

[5] K. Kutzner and T. Fuhrmann. The IGOR file system for efficient data distribution in the GRID. In *Proc. Cracow Grid Workshop CGW 2006*, 2006.

[6] D. Naor, M. Naor, and J. Lotspiech. Revocation and tracing schemes for stateless receivers. In *Proc. CRYPTO 2001*, volume 2139 of *LNCS*. Springer, 2001.

[7] I. Stoica, R. Morris, D. Karger, M. F. Kaashoek, and H. Balakrishnan. Chord: A Scalable Peer-to-peer Lookup Service for Internet Applications. In *Proc. ACM SIGCOMM '01*, pages 149–160. ACM Press, 2001.