University of Zurich
Department of Informatics

Diploma Thesis March 31, 2008

Ubidas - A Novel P2P Backup System

"Temporary Stone Sculpture 001"
Stones from the Swiss Alps
Daniel Buchmüller 2008

Daniel Buchmüller
of Rüti ZH, Switzerland

Student-ID: 02-707-198
db@icontent.ch

Advisor: **Christoph Kiefer**

Prof. Abraham Bernstein, PhD
Department of Informatics
University of Zurich
http://www.ifi.uzh.ch/ddis

Acknowledgements

I would like to thank Prof. Abraham Bernstein, PhD for giving me the opportunity to write this thesis in his group and for his time discussing and asking challenging questions. Christoph Kiefer and Stefan Leuthold helped me in the last stage by proofreading this thesis and giving numerous helpful hints on style and language. Thank you!

A thank you also to Antony Rowstron of Microsoft Research for his thoughts and comments on the subject in general and on my idea.

Last but not least, I would like to thank my parents for their encouragement and support!

Abstract

This diploma thesis introduces Ubidas, a novel p2p backup system. The goal of Ubidas is to offer a distributed secure and cost-efficient platform for both individual and corporate backups by distributing copies intelligently across multiple participating nodes. New concepts such as the prioritization of local and close resources, an automated connection establishment algorithm and a highly redundant, distributed hash table (DHT) algorithm are presented and evaluated.

Zusammenfassung

Diese Diplomarbeit stellt Ubidas vor, ein neuartiges peer-to-peer Backup-System. Das Ziel von Ubidas ist es, eine verteilte, sichere und kostengünstige Plattform für Individual- und Firmenbackups bereitzustellen. Dafür werden die Daten von Benutzern verschlüsselt und auf verschiedene andere Teilnehmer des Systems verteilt. Bekannte Konzepte wie Verteilte Hashtabellen (DHTs) werden punktuell optimiert und neue Konzepte, wie die automatische Priorisierung von verfügbaren lokalen Ressourcen werden in dieser Arbeit eingeführt und evaluiert.

Table of Contents

1

Introduction

1.1 Motivation

Personal and corporate computer data storage requirements grow at a high rate, estimated to double every 3 years [Lyman and Varian, 2003]. For example with the shift from film to digital for both still and video cameras, the constant decline in prices for such devices, and the low cost to store digital memories, personal computing is a major contributor to this development.

The use of digital cameras is omnipresent in western countries and adapted by the majority. These devices generate so much data that it is hard to keep (i.e., pictures organized) and often the cost of going through and deleting bad shots is higher than keeping them.

On the other hand, hard disks that store our digital memories crash or are wiped by malicious software quite often. Especially, if compared with the old days where valuable moments of our lives caught on camera were kept in one shoe box or in a couple of photo albums. The probability that these originary photos are destroyed (i.e., through fire) is very low compared to the probability of a digital data loss.

A positive aspect of digital memories is that they can be copied and distributed without loss of information, therefore reproducing data for backup purposes becomes an easy task, technically. The problem with classic backups is that the owner has to take care of it—replace old backup medias with new ones and constantly perform complete or incremental backups of all his or hers computers. We humans are not especially good at such repetitive and tedious tasks, and technology can also help here.

This thesis introduces Ubidas, a reliable and safe peer-to-peer (p2p) communication stack for data distribution on other users' computers (nodes), including on the internet.

With the sinking cost for magnetic media such as hard disks, current computers are often over-equipped with storage. The average user only occupies a fraction of the computer's persistent storage space capacity. Also the number of broadband internet connections is high in western countries and still rising. For example in 2006, over 2 million households in Switzerland were already equipped with a high speed internet access.[1]

This over-provisioning and the unused capacities give a new power to the edge of the internet. The development from a few central internet servers providing mostly text and images to the current concepts of "Storage in the cloud" will ultimately lead to a democratization of internet nodes; the world's data will be spread evenly over all participating computers.

I was always fascinated by the given asymmetry of a need to backup your data on one side and the free space and unused internet access bandwidth on the other. Ubidas tries to bridge this gap, giving its users automatic, free, safe, and secure backup as a service.

1.2 Related Work

The following Section introduces related work in the fields of p2p backup, storage, and distributed hash table concepts.

1.2.1 pStore

pStore is a secure Chord-based [Stoica et al., 2001] p2p backup system that uses unused space of participating nodes to backup data chunks of other nodes in the system. It applies salting to the data chunk identifiers to spread chunks evenly over the distributed hash table (DHT) key space. A specialty of pStore is the support of a Concurrent Versions System (CVS)-like versioning enabling incremental backups and restores. Delete requests are authorized with a user-signed request [Batten et al., 2002].

pStore uses a DHT for both table of contents (TOC) storage and file chunk storage. There are two issues with this approach:

- Every node is responsible for a certain portion of the key space. If a node leaves, the DHT has to re-organize itself to cover the whole key space again and re-establish a predefined redundancy r. If the system has a high churn rate—the rate of joins and leaves of nodes

[1] `http://www.bfs.admin.ch`

to and from the system—a high re-synchronization effort is needed to keep the DHT well-organized.

- pStore stores the actual data chunks based on their IDs on the node, responsible for that key space range. If also the data is imposed to the classic Chord DHT, heavy synchronization is needed, causing a high bandwidth usage.

1.2.2 PeerStore

PeerStore has a reduced maintenance overhead compared to pStore. This is achieved by decoupling the metadata storage (TOC) from the actual data storage. PeerStore employs the nature of symmetric trades to achieve fairness and prevent freeriding: a user who requests a store has to offer the same amount to other users. Every storing node gets to store the same amount on the requesting node [Zhang et al., 2004].

The following key points are identified:

- A problem with symmetric trades occurs for uneven equipped requesters. There could be system users that want to backup considerably more data compared to what they are willing to offer for other users' backups.

- PeerStore uses convergent encryption [Bolosky et al., 2000] which uses the cryptographic hash of a byte sequence (chunk) to encrypt the chunk itself. This allows to detect identical files and safe storage space in the system while not exploiting the content itself. An attacker could easily identify which users store the same blocks by encrypting a chunk himself and requesting the DHT for all users that own that chunk.

1.2.3 Chord

In a distributed system with no mission-critical central servers a simple layer for storing *key/value* pairs is needed. This is generally called a distributed hash table (DHT). Usually *value* is a sequence of bytes and *key* the cryptographic hash of *value*. This creates a key space with a maximum capacity of $Keyspace_m = 2^m$, where m is the chosen *key* length in bits. If every participating node were to store all values, then lookup performance for *value* given *key* is $O(1)$. This does not scale to a large number of nodes n. Also the insert/update effort for a new *key/value* pair to n nodes is $O(n-1)$ which indicates again that it does not scale for a large n. Chord introduces a smart DHT for p2p systems. It has $O(log\ n)$ lookup performance. The novelty is that not every node needs

to know about every other node, but optimally only about $O(log\ n)$ neighbor nodes. *Key* lookup requests are sent from node to node, where every node forwards the request to the nodes with an *ID* closest to *key*. *ID* is also a cryptographic hash, representing the node uniquely in the *key* space. In Chord, *ID* is created from the IP address of the participating node [Stoica et al., 2001].

Chord has several drawbacks:

- Due to the mechanism of how a node *ID* is created—by hashing its IP address—no unique *IDs* for multiple nodes behind the same network address translation (NAT) device are possible. The DHT in this case still works but is less efficient.
- There is no spacial redundancy for failed nodes. In Chord the redundancy factor *r* is implemented by distributing *key/value* pairs to neighbor nodes. An attacker could critically split a Chord DHT by obtaining $> r$ nodes that are all neighbors of each other in the key space and then make all nodes he controls leave the system at once. The *key/value* pairs in that section would be lost.

1.2.4 PAST

A short comment on PAST which uses Pastry [Rowstron, 2001] for routing and locating *keys*. It has similar capabilities to Chord in terms of lookup and re-organizing performance but enhances the system's integrity by employing Smartcards to sign files and verify update requests. The Smartcard's unique ID is hashed to produce a unique node *ID key*. This makes forging a *key* much harder and prevents from the above mentioned DHT split attack [Druschel and Rowstron, 2001].

1.3 Ubidas Design Goals

Analyzing the previous work above reveals distinct highlights in different areas:

- Robustness, redundancy and persistency in DHT (Chord, Pastry)
- Decoupling of TOC and storage layer to reduce maintenance traffic (PeerStore)
- Using a strong identity provider such as Smartcards to improve authenticity and prevent malicious DHT modifications (PAST)
- Scalability—in the order of $O(log\ n)$ performance—for lookups and neighbor list storage (Chord, Pastry)

The goal of this thesis is to combine these individual highlights and introduce new ideas to achieve a holistic view of the p2p backup problem. Most systems concentrate on improving one aspect of performance such as minimizing storage and transfer volume. The combination of above highlights, the introduction of new real-world aspects, and the resulting modification of above concepts are the core philosophy of Ubidas.

1.3.1 Conflicting Goals: Choosing a Design Philosophy

In computation the landau—also known as "big O"—notation [Landau, 1909] is popular for describing an algorithm's performance. For example the Chord algorithm has an average lookup performance of $O(log\ n)$ as mentioned above. The $O(...)$ simply state that the algorithm scales with a performance stated within the brackets.

Designing a real-world system means in this case dealing with the conflicting goals of *performance* (i.e., storing all data on all nodes yields high performance) versus *storage* (i.e., storing data only on one other node yields low performance and poor data reliability but low storage requirements). And if the system is of a distributed nature, the third conflicting goal of *robustness* has to be considered, too.

Two examples illustrate this:

- **Example A:** Consider a distributed system should be optimized for best performance. It has a very sleek transfer protocol with only 5% metadata overhead and a lookup performance to find a desired *value* of only $O(1)$. Such a system could be implemented by deploying local hash tables to every node initially. In fact this could be the most efficient implementation for systems with a very low change rate in the hash tables and a comparably high lookup rate.

- **Example B:** Imagine a system that concentrates on persistent storage efficiency and only stores values on the disk for which the node is directly responsible. Now the storage requirement per node would be

$$Size_{PersistentStore}(N, V, size_v) = \frac{V * size_v}{N}$$

  where N is the amount of nodes, V the number of *values*, and $size_v$ the average size per *value* in the system. This is the optimum in terms of the storage requirement an average distributed system node could have.

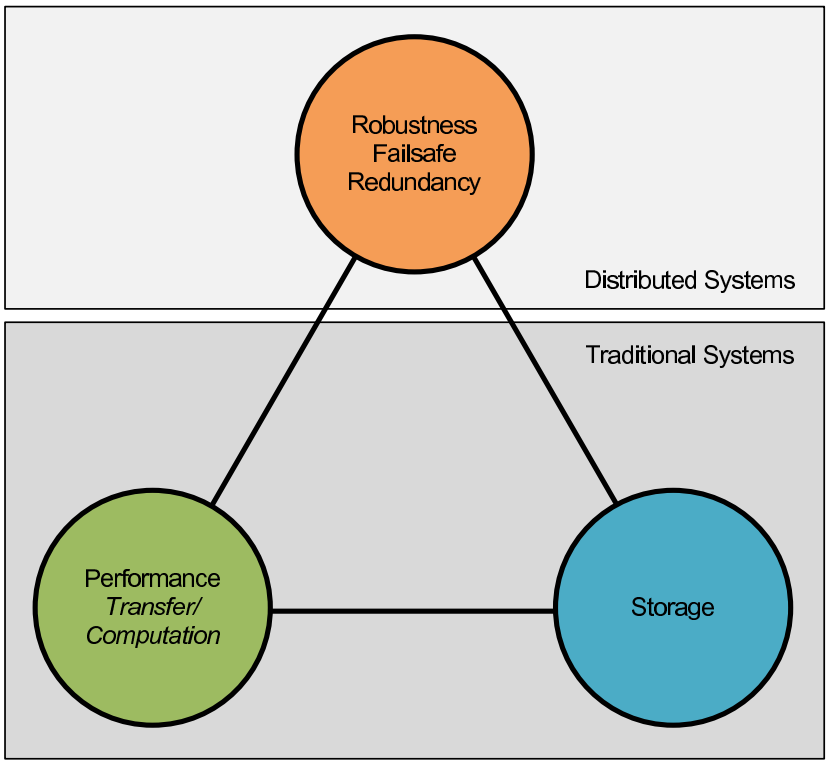

Figure 1.1: Conflicting goals space in distributed systems design

Above examples are extremes, concentrating on one of the conflicting goals, only. Figure 1.1 shows the conflicting goals space for traditional systems and expanded for distributed systems. Systems can be classified into this space. Clearly, Example A would be placed in the lower left and Example B in the lower right corner. Although they would suit very special cases, generally a distributed system's design should be placed carefully between the three conflicting goals, weighting the different design criteria to obtain ideal system characteristics and overall performance.

With this concept in mind, Ubidas' design goals are targeted to meet several real-world requirements. This makes it harder to evaluate the system because a good holistic performance can only be derived from individual performance aspects with a subjective weighting. The next section reveals the Ubidas design goals:

1.3.2 Internet-wide Reachability

All mentioned systems in Section 1.2 only work in scenarios with direct IP connectivity between all participating nodes. This does not reflect the real world with a multitude of hierarchically

separated public and private networks. Most nodes in private networks are not reachable from the internet. Some nodes on the other hand have a public IP address but are restricted by a firewall from being accessed from the public internet.[2] There are advanced methods such as UDP hole punching and others. Skype makes prominent use of those.[3]

Ubidas introduces a pragmatic method to overcome these barriers and achieve real end-to-end connectivity. This system component is called Ubidas End2End and explained in detail in Section 2.4.

1.3.3 DHT: High Reliability, Dynamic Redundancy Scaling

Most p2p systems these days use Chord or some variation of it for their DHTs. The problems with a standard DHT are:

- its hard bound to IP addresses as hash bases for the generation of node *IDs*,
- high synchronization efforts to cope with the churn rate, and
- an attack surface for DHT splitting attacks.

Ubidas is designed to address and overcome these weaknesses by:

- using per-node unique guids that are cryptographically hashed,
- maintaining a high redundancy and lazy update cycles to reduce maintenance overhead, and
- distributing DHT *key/value* pairs (DHT records) with two adjustable redundancy factors both evenly over the key space and to other nodes with a close *ID*.

A detailed description of the Ubidas DHT is given in Section 2.7.

1.3.4 Decoupling of TOC and Storage

PeerStore was the first system to decouple the TOC from the actual storage by implementing a symmetric storage trade layer. Ubidas goes one step further and removes the condition for symmetric storage trades. Additionally, data chunks are kept with a higher redundancy on participating nodes and therefore maintenance efforts for meeting full redundancy requirements after

[2] The uzh network is in fact designed like this, with port Transmission Control Protocol (TCP) 113 being the only publicly reachable.

[3] `http://www.skype.com`

some nodes have not joined in a longer period of time are relaxed and lowered. Section 2.10.1 explains this subject.

1.3.5 Central Nodes for Non-Mission-Critical Tasks

Often p2p systems stick to the goal of getting rid of any central system elements. This draws up synchronization efforts due to the nature of a fully meshed network connection requirement, i.e.,

$$Connection(n) = \frac{n(n-1)}{2},$$

where n is the number of nodes in the network. Normally a trade-off of not having to reach every node with a $O(log\ n)$ performance is chosen. This leads to the incapability of having a holistic view-state over the whole system with little effort. Ubidas Index is introduced as a non-mission-critical system component that helps for bootstrapping new nodes and can give updates on the total number of nodes recently present and available transit nodes for NAT-protected nodes. This substantially can lower synchronization efforts. Ubidas Index Server (UIS) is introduced in Section 2.4.4.

1.3.6 Consideration of Local Resources

All systems mentioned in the Related Work Section 1.2 make no differentiation between nodes across the Ocean and nodes in the same private network. Often these systems claim to achieve a high data redundancy by placing a *DHT record* anywhere on the key space and thus anywhere in the world. This decoupling of a *value* from its origin is an important concept to achieve robustness. In real-world deployments using such a system exclusively yields to poor performance because the underlying network and usage patterns are not considered. Therefore Ubidas introduces IPTree in Section 2.6. It takes available local resources into account to some extent when choosing nodes for DHT queries, storage requests, and in the transit node subscription process.

1.4 Design Problems not Addressed in this Thesis

Most of the time ideas go beyond the current implementation and the time available; so the following topics were excluded from this work and are left open for future work:

- Prevention of freeriding is excluded from the current Ubidas implementation. In a large deployment such a component should be included. Personally my vision would be to introduce a market-based approach where offering users could trade storage for an amount of a generic points currency and requesting users could cover their storage needs by buying quotas with points. Points could be purchased with a defined exchange rate with a real-world currency (e.g., USD, CHF, etc.). A central system entity called Market Server could manage the price-finding and act as a market intermediary, similar to Google's automated AdWords auction.[4] This would allow the system to suit unevenly equipped participants—such as companies with few client nodes compared to the number of their servers—to still use the system. On the other hand Ubidas as the intermediary could collect a small fee on every storage space trade which would allow to run it as a profitable business and maintain its future development. Currently, there is a simple authentication mechanism in place that will be discussed in detail in Section 2.7.
- Strong, public/private key-based authentication is excluded in Ubidas. A simple, private key-based mechanism is in place.
- Multiple users per machine capabilities are excluded in the current version of Ubidas. So far none of the above described systems has support for multiple users. Although in a future commercial application supporting more than one backup user per machine is crucial.
- Although actual simple transfers are working, due to time constraints Ubidas has a strong implementation up to and including the level of the TOC, but excluding the transfer layer in its current implementation.

1.5 Structure of this Thesis

After having introduced this thesis' motivation and covered the related work, the design goals and philosophies have been set in the right light. This builds the basis for this thesis and opens the stage for Chapter 2, titled *Concepts*. Further the implementation is discussed in Chapter 3. This thesis will close with an evaluation of selected system characteristics in Chapter 4, and an outlook to future work.

[4] `http://adwords.google.com`

2

Concepts

This chapter explains the concepts of Ubidas. When designing a distributed system, there is much more to care about than in traditional programs that run in one process on one computer.

The challenges involved when designing such a system are:

- **Underlying system architecture:** Some distributed systems are designed to run on different hardware and operating system platforms. Depending on the layer of abstraction between the operating system and the specific software, distributed systems may have to handle different mechanisms to interact with input/output (I/O) devices and change endianness (i.e., encoding a value with its highest-order bit first).

- **Resource locking:** In traditional systems basic locking of resources for exclusive access is easy. Some operating systems such as Windows support locking on the platform level, others (e.g., Linux, Unix) need an application level locking mechanism, such as using a separate LOCK file. In a distributed environment it is hard to achieve transaction-grade I/O operations. Due to no guarantees for availability of nodes, nodes can block a resource indefinitely by not removing a lock before leaving.

- **Reliability:** The reliability of an atomic operation in a distributed system is not as high as for operations within the boundaries of one process or machine. Especially, when using frameworks such as Java or .NET, the additional layer of abstraction comes with benefits such as memory violation protection. Therefore when designing a traditional system one does not have to take into account failures of common operations such as I/O, or when they occur, it is much easier to recover from and take proper action.

  When moving to a distributed computing environment one has to consider that failures occur with a relatively high probability and special care has to be taken of.

Typical examples are:

  - network interruption,
  - high latency for remote operation invocation, and
  - depending on the specific system: high churn rate (rate of joining and leaving nodes)

- **Reachability:** A system can be classified in terms of its communication scopes. The following list explains the different communication scopes and how interconnection can be achieved:

  - **Intra-Process:** Within the same process a class of a component can be instantiated to an object and public methods and members can be called.
  - **Inter-Process, within the same machine:** Named pipes are a common mechanism to communicate between different processes on the same machine. Their characteristics are high-bandwidth and low-latency communication. A different approach is message-based communication over files in designated "in" and "out" folders.
  - **LAN:** Moving beyond the scope of intra-machine to inter-machine communication demands for new channels, protocols, and means to communicate over. The local area network (LAN) is the first scope of a networked interaction and communication can be achieved on layer 2 theoretically and layer 3 practically allowing to connect to another participant's IP address and port.
  - **WAN:** Real end-to-end connectivity is—in most cases—still easily possible in wide area network (WAN) setups where participating LANs trust each other. Therefore firewalls do not restrict traffic and NATting becomes not an issue.
  - **Internet:** In an internet-wide deployed distributed system, real end-to-end connectivity is hard to achieve. In a traditional client-server system design a client pulls data from a server. Firewalls are configured to allow outgoing connection establishment for clients to pull information from servers. This approach does not scale as well as a system equipped with real end-to-end connectivity.

Moving further to a subclass of distributed systems leads to the group of peer-to-peer systems which have even more underlying constraints in their design:

- **Heterogeneous system architectures:** Participants of a p2p system, so called nodes, are often deployed on a totally heterogeneous hardware basis. It therefore has to be designed with these constraints in mind, allowing it to adapt to generic hardware.

- **Untrusted nodes:** If private content is distributed to other nodes for safekeeping reasons with redundancy, it has to be encrypted with state-of-the-art encryption algorithms that make it unfeasible to decrypt the data in terms of reasonable attack time. Today, Rijndael's method or generally known as advanced encryption standard (AES) provides strong protection with a key and block size of 256 bit each.

- **Churn rate:** When designing a p2p system that is targeted to be deployed to home and office computers, a high rate of joins and leaves to and from the system (churn rate) has to be considered.

  There is a positive correlation between the churn rate cr and the redundancy factor r:

  $$r(cr) = const * cr^x \mid const > 0,\ x >= 1$$

  Proactive re-synchronization of data from nodes that were not present in the system for $threshold_{activity\ timeout}$ helps to maintain a low r and a higher availability a of data chunk $chunk_{data}$.

- **Freeriding:** Coupled with the characteristic of an untrusted system environment special measures to prevent nodes from misusing the system have to be taken (i.e., asymmetrical system use, also known as freeriding). In Ubidas a central, non-mission-critical index server represent the marketplace for storage needs and offers (not present in the current implementation). Other approaches are distributed, market-based trust and rating systems to overcome freeriding.

Ubidas as a p2p system addresses several of these challenges. Its core concepts are explained below.

2.1 Communication Stack: a Layered Approach

After discussing the challenges and constraints that underlie p2p systems we must establish an architecture that can address all of these in a structured manner. I chose a layered approach making use of the power of abstractions in systems design.

Starting at the lowest level lies the operating system, providing the abstraction to the central processing unit (CPU), volatile, and persistent memory and devices such as—in this case important—a network card. .NET was chosen as the runtime middleware. Using C# as a managed language offers benefits such as process isolation, fault recovery, garbage collection, etc.

In Ubidas the .NET Framework forms the foundation for the three different system tiers:

- Communication,
- Sentinel, and
- Controller

A subset of the .NET Framework since its version 3.0 are the Windows Communication Foundation (WCF) classes which form the first layer of the communication tier. The evaluation of technologies used here, WCF and all other technologies used for the creation of Ubidas are explained in detail in Chapter 3, *Implementation*.

The following layers of the communication tier and the other tiers are explained in sections 2.2 through 2.10. Figure 2.1 gives an overview of the foundation layers, the three tiers, and the respective sub layers in the communication tier.

2.2 Sentinel

One of the goals of Ubidas is to provide instantaneous backups of selected directories. Ubidas Sentinel is the second tier in the Ubidas communication stack shown in Figure 2.1 and it is responsible for detecting new changes in the file system and for queueing pending, changed files for backup in the p2p system. Also, the creation of new distributed hash table (DHT) items for new folders should be triggered by Sentinel. For backups, a user might not want to backup every file type in every directory on every drive. A user might want to save all subdirectories under i.e., `c:\docs`, and `d:\pictures`, exclude files of type `*.tmp, *.db, *.ost`, etc. from being saved. The design of the Ubidas Sentinel filter is constraint-based, similar to the configuration of *rsync* [Tridgell and Mackerras, 1996]. It supports the following options:

- **Include list:** Specifies a list of directories to include for backup. Every directory of the list can be tagged whether to include its subdirectories or not.
- **Exclude list:** Specifies a list of directories to exclude for backup. This option helps to exclude one or more subdirectories from a directory specified in the include list.
- **Exclude file types list:** Global filter to exclude certain file types. For example all .ost files can be excluded from a backup because they represent an offline cache for a server-based groupware, including mail. Also a user might want to exclude .dbf database files that may

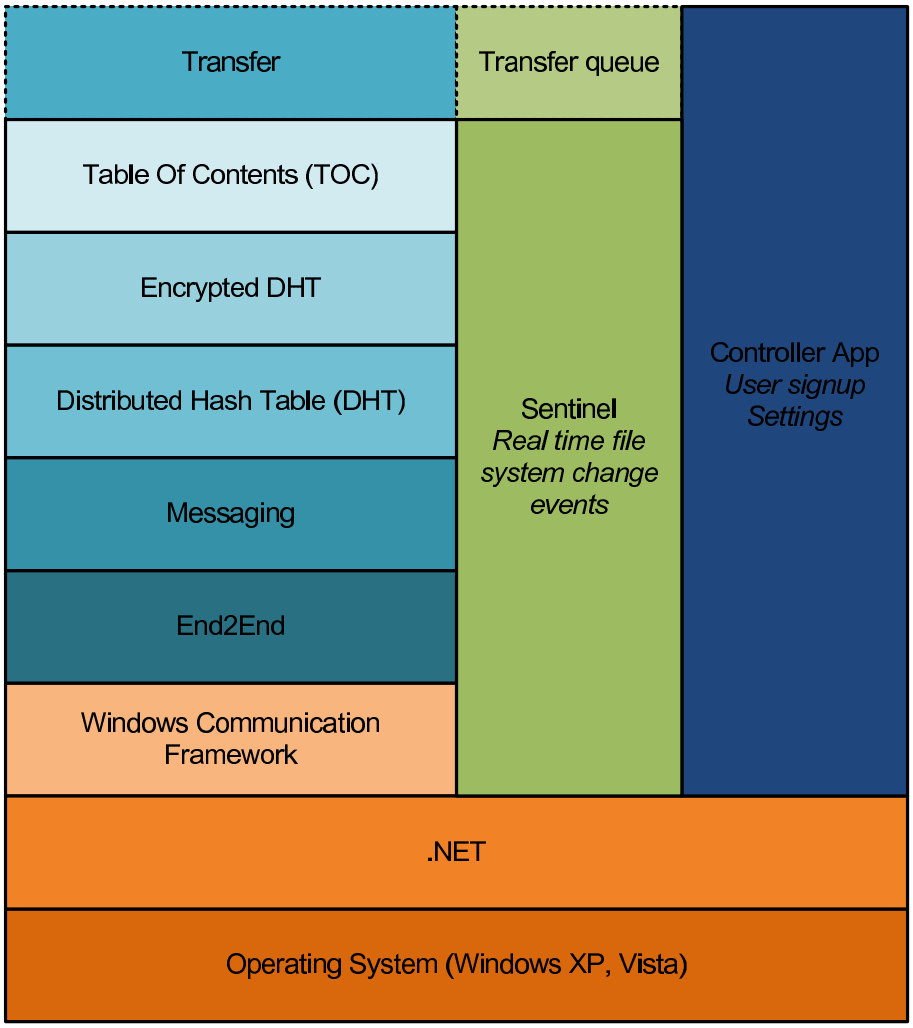

Figure 2.1: Ubidas Communication Stack: A Layered Approach

grow huge and change frequently which makes it unfeasible to backup with the real-time detection of Sentinel. In a future version of Ubidas, an application-level plug-in infrastructure could be introduced to handle file type-specific incremental backups.

To achieve realtime detection of file system changes on the drives of included directories, Sentinel takes advantage of a built-in feature of the operating system, called FileSystemWatcher.[1] Because it is limited to monitoring just one directory a dynamic composition of generic watchers was implemented. Figure 2.2 shows the orchestration of individual watchers; combined they feed a global feeder queue. The feeder queue is constantly being dequeued and its items filtered

[1] `http://msdn2.microsoft.com/library/system.io.filesystemwatcher.aspx`

against the mentioned filter lists. Finally the filter component fires events through the event interface to the layer on top, consuming the Sentinel. In the case of Ubidas, DHT generation and the transfer queue are consumers of the Sentinel events, as Figure 2.1 shows.

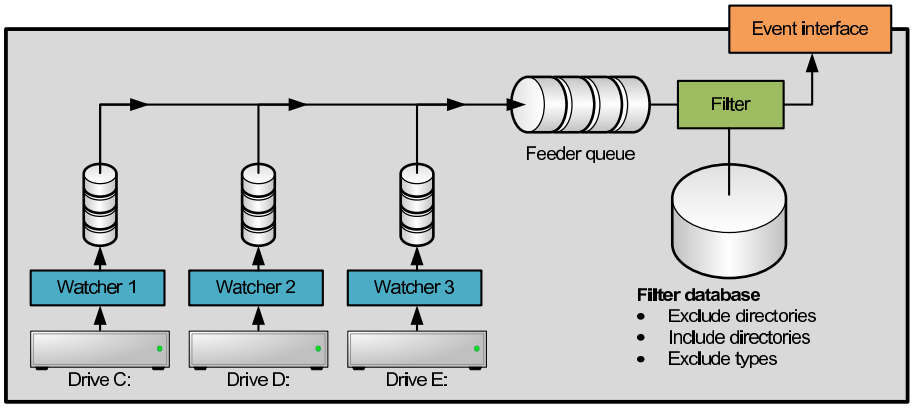

Figure 2.2: The second tier: Ubidas Sentinel

2.3 Controller

The controller application represents the third tier of Ubidas, as Figure 2.1 shows. The Controller is a simple Win32 application that can be configured to automatically launch with an operation system start. When launched the controller application neatly sits in the task bar region, indicating its running state as shown in Figure 2.3

Figure 2.3: The Ubidas Controller application icon indicating its state

It lets a new user sign up for Ubidas and existing users log in to Ubidas. After a successful user login, the connected state is displayed in the lower right corner of the controller application as shown in Figure 2.4. Because the user credentials are used to generate the user's private key, the password is never transmitted to the Ubidas Index Server (UIS). Instead it is hashed locally and then transmitted.

Figure 2.4: The Ubidas Controller application showing the user status

2.4 End2End

As stated in Section 1.3.2, internet-wide reachability is one of the major design goals of Ubidas. All mentioned related work papers in Section 1.2 do not implement real end-to-end connectivity. An ubiquitous backup system needs to deal with all relevant characteristics of the real world to be usable. With this holistic design approach, end-to-end connectivity is a major cornerstone to Ubidas' success. When evaluating we will see what benefits in terms of coverage and adaptiveness this brings. Ubidas End2End is therefore the abstraction layer enabling real end-to-end connectivity between all participating nodes in the system. Most home and office users are behind a network address translation (NAT) router device. Section 2.4.1 introduces the issues with NAT-traversal and the techniques to overcome them. As a consequence of different network reachability the three different types of Ubidas nodes are explained in Section 2.4.3 and 2.4.4. A detailed explanation of the connection establishment process is given in Section 2.4.5; this then leads to the network border crossing capabilities, laid out in Section 2.4.6.

2.4.1 Network Address Translation (NAT)

IPv4 was introduced in September 1981 in RFC 791.[2] An IP address consists of four octets of one byte length each. The total IP address length of 32 bits leads to a theoretical maximum of 4.3 billion unique addresses. The internet was originally intended to connect every participating computer by assigning one address to each node. With the fast expansion of the internet—way beyond connecting high-power physics labs around the world and university campuses—the need for a solution to prevent running out of IP addresses became apparent. The Internet Engineering Task Force's Network Working Group compiled RFC 3330 "Special-Use IPv4 Addresses".[3] which designates for private use reserved IP addresses. These addresses are not routed on the public internet and can be used in private networks. This allows for a setup with only at least one

[2] `http://tools.ietf.org/html/rfc791`
[3] `http://tools.ietf.org/html/rfc3330`

device with a public IP address, representing a NAT device for a whole organization. Within the organization computers are assigned IP addresses from the designated special-use address ranges.

To establish connectivity with nodes on the internet the NAT device maps a request from the inside to a designated outside address and forwards it to the host on the public internet. It remembers this temporary designation to eventually allow a response to find its way back to the originator in the private network.

For traditional client-server systems this concept works fine because most of these systems rely on clients pulling information from a server. With the rise of p2p concepts NAT traversal became an issue: Private hosts need to be reachable by hosts from a different private network. It is possible to map a port on the public side of the NAT device to a host in the private network. This approach needs manual configuration or semi-automatic with the help of standards such as Universal Plug and Play (UPnP).[4] Although UPnP works great for home setups it is not common in enterprise environments, in part because NAT prevents private hosts from becoming reachable from the internet and is thus an additional layer of security.

Ubidas introduces the approach of self-establishing transit nodes. The following sections explain how this works in Ubidas.

2.4.2 Node Characteristics Determination

Do determine what characteristics a node has in terms of reachability from the internet it launches a bootstrap method every time Ubidas starts.

The bootstrap process is outlined as follows:

1. Get listening port from configuration
2. Start listening on port
3. Ping Ubidas Index Server with listening port as argument
   (a) Index Server receives ping and determines client IP address from TCP context
   (b) Index Server opens connection to client IP address and given port
   (c) If no host answers ping-back request for a certain timeout, Index Server returns false, otherwise true.

[4] `http://www.upnp.org/`

NodeName	IsPublic	IsTransit	INetEP	LocalEP	IPTree
AsusVista	True	True	217.162.206.4:113	192.168.0.195:113	62.2...195

Table 2.1: Node Characteristics Record Example

4. Receive result from Index Server. Update own node characteristics record accordingly

This process is presented in Figure 2.5 with the result that the node cannot be reached directly from the internet. If the node is internet-reachable, the default behavior of being a transit node is applied by setting the `IsTransit` flag in the record. Another part which is determined in this process is the node's IPTree which is explained in detail in Section 2.6.

Finally the own node characteristics is sent to the Index Server for fast caching and additionally distributed with the DHT infrastructure to some nodes in the overlay network. An example Node Characteristics record is shown in Table 2.1.

This leads to the differentiation of node types in the Ubidas system. They are described in the next section.

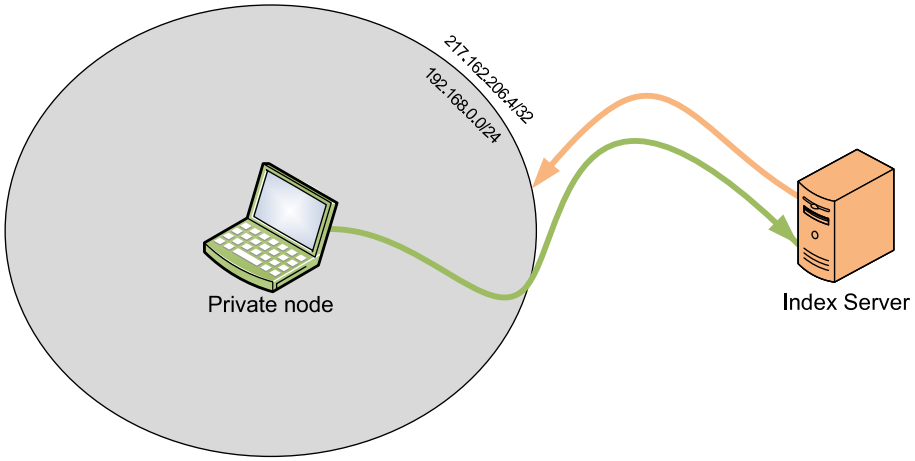

Figure 2.5: Node Characteristics Determination: Reachability Detection

2.4.3 Ubidas Node Types

Due to the nature of node internet-reachability different types of node dynamically emerge in the Ubidas system; through the node characteristics determination algorithm in the bootstrap

Type	Description
Private node	Typical node in an enterprise environment or in a home setup where a NAT router blocks access to nodes in the local subnet and/or where no NAT rule can be set to enable reachability.
Public node	Node that is internet-reachable, meaning it has a public IP address or an in between NAT router has been configured to forward traffic to the local node.
Transit node	Per default, every public node is set to be a transit node as well. As an additional task it is responsible to route traffic to private nodes that subscribed to this transit node.
Index server	A special case of a public node is the Ubidas Index Server. It handles node characteristics pings for clients to determine their internet-reachability and performs other non-mission critical tasks. In the current setup one Ubidas Index Server is deployed in the system but it is easy to add more to scale the system and add additional redundancy.

Table 2.2: The different node types in Ubidas

process, its type is determined. Table 2.2 gives an overview of the different node types in Ubidas. Ubidas Index Server is further explained in detail in Section 2.4.4.

2.4.4 Ubidas Index Server (UIS)

As briefly described in Table 2.2, the Ubidas Index Server (UIS) is a non-mission-critical component of the Ubidas system. In the current setup it is deployed once but more instances can be added to scale and improve redundancy.

In short, the UIS is a lightweight, internet-reachable, state-keeping node with the following functions:

- store system state information such as:

  - total system size in terms of number of nodes, available space, etc.,
  - node characteristics cache, and
  - system user IDs (email addresses) to prevent duplicates in the sign up process

- provide helper tools for nodes:
  - ping service for internet-reachability determination of nodes
  - sign up service. Note that the sign up service does not store the credentials of the participating user, but only his email address as an unique identifier
  - bootstrap service to serve a set of node characteristics to a new node
  - transit node assignments. To enable reachability on an abstracted layer, private nodes need at least one transit node they can subscribe to. UIS determines the best possible transit node for a node with the help of IPTree (see Section 2.6). This algorithm is explained in detail in Section 2.4.6 “Network Border Crossing”.

2.4.5 Connection Establishment

In a perfect networking world every node of a system can reach every other directly by connecting to it (i.e., through TCP to a node’s IP address and a designated port). As introduced in Section 2.4.1 due to the shortage of IPv4 addresses, not every participating node on the internet can be given a public IP address. Therefore real end-to-end connectivity is not given and it has to be re-established by finding a way through NAT devices or by building an abstraction layer that achieves push connection initialization for all participating nodes, again. Ubidas focuses on the later and this section explains how the connection establishment works in detail. Figure 2.6 shows the connection establishment process in a state-activity diagram.

Assume node A wants to send a message to node B. First A checks its local cache for a node characteristics record of B. This cache is being updated by incoming DHT broadcast node announcements. At this time it is only cached in RAM so it will completely be “forgotten” once a node shuts down. It would be a small change to the code to enable a persistent node characteristics cache. On the other hand the principle of forgetting is used in artificial intelligence systems as biologically inspired models [Correia and Abreu, 2004] and could benefit the system performance in this case, too (not tested).

If the node characteristics are not cached locally, a request to the UIS is made. Note that this could be implemented also making a DHT query but the performance of a lookup in the UIS

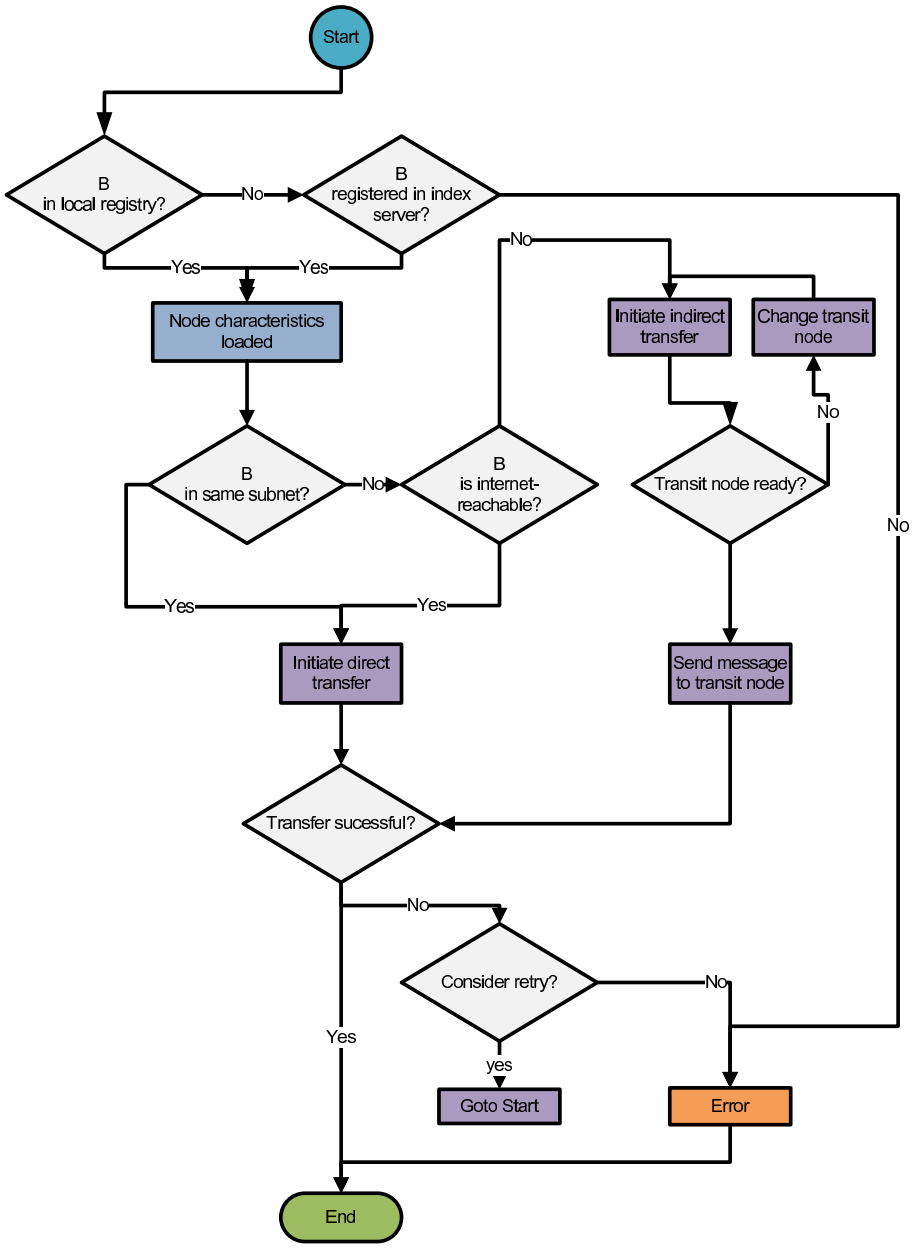

Figure 2.6: Connection establishment for a connection from node A to node B. A detailed explanation is given in Section 2.4.5.

is only of $O(1)$ and thus very fast compared to a slower $O(log\ n)$ lookup in a DHT. An ideal approach would probably be to first query the UIS and if that fails go ahead and query the DHT as a failover backup; this would take final responsibility weight off the UIS and make the system more robust.

One of these requests must succeed, otherwise the requested node B is treated as non-existent and the call fails. If at least one method succeeds, the node B characteristics record is loaded and inspected. Using the IPTree algorithm explained in Section 2.6, the process tries to evaluate whether nodes A and B are in the same subnet. Note that inspecting just the local endpoint in B's record is not sufficient as there are only a few different designated private network ranges and false matches would be numerous for nodes that are—in terms of their final IP address—in the same subnet, but in reality in different private networks with the same subnet. Therefore the IPTree algorithm analyzes the created IP tree of both nodes and looks for enough similarity. Enough similarity is given when two nodes' IP trees match the second and third last IP addresses. For an illustrative example see Table 2.6.

Typically nodes in the same subnet can communicate directly; so the next step is to initiate a direct transfer to B. If this fails, the process is restarted and this time, A tries to reach B over B's transit node. This happens rarely because in most setups, private nodes are not isolated against each other.

If the subnets do not match, the node B characteristics record field `IsTransit` is checked whether B is internet-reachable. If this is the case, also a direct transfer is initiated; otherwise and indirect transfer must be established. To initiate an indirect transfer, A checks B's `TransitServers` field in the characteristics record and starts by trying to send the message to the first registered transit node of B. If this fails, the current transit node is dropped and a next transit node from the list is used and so on. This helps if a transit node of B is temporarily unavailable. In that case node B, which polls the transfer server every 20 seconds realizes its fault or unavailability and moves on to the next transit node. It then re-broadcasts its new transit node selection via DHT. Due to the nature of DHT this can take several seconds so it makes sense that A is also changing B's transit node when a fault occurs to reduce overall failures.

All retry patterns use more than one retry and vary the retry timeout by increasing a hold time to prevent unnecessary flooding.

We now have real end-to-end connectivity established. Section 2.4.6 explains how this mechanism can actually be used for efficient routing.

2.4.6 Network Border Crossing

The previous sections established the concept of real end-to-end connectivity. Following, mechanisms are explained to optimize the actual message routing.

Node	Characteristics
A	Node A is a fully internet-reachable node that functions as a transit node for node D.
B	Being in the same subnet as node C, node B is also fully internet-reachable. This could be achieved by enabling a port forwarding on the NAT device.
C	Node C is local reachable, only. It is a typical example for a node in a home or office environment. C subscribes to B as its transit node to achieve abstracted end-to-end connectivity with other nodes of the system.
D	Finally, node D is also a local reachable node only and it neither has a transit node in the same subnet; which makes it subscribe to node A as its transit node.

Table 2.3: Node characteristics for nodes in Figure 2.7

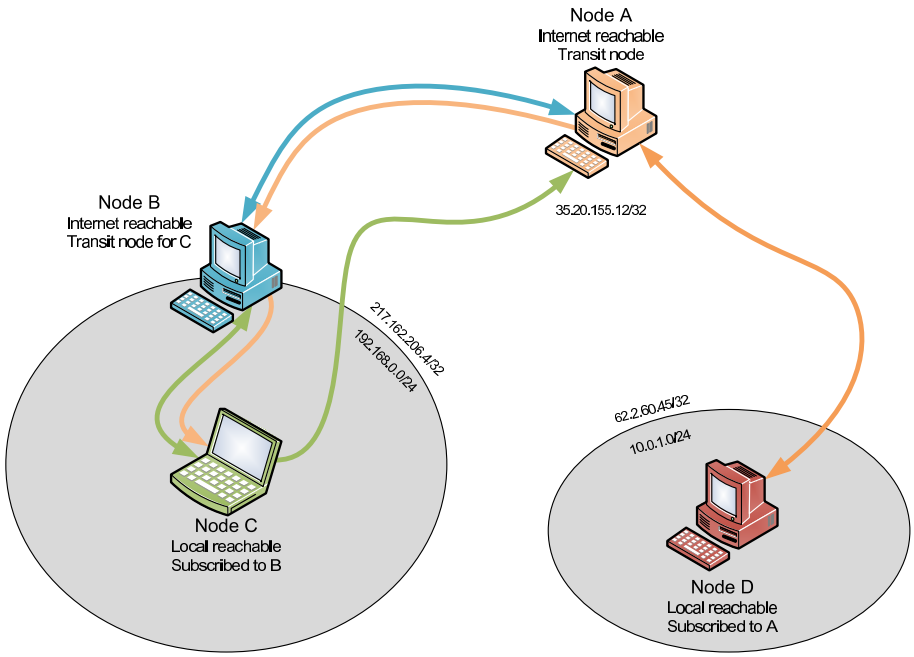

Figure 2.7: Ubidas network border crossing capabilities

As introduced in Section 2.4.2 and 2.4.5 the IPTree algorithm (see Section 2.6) is used to gain knowledge about a node's location in terms of its path to the internet. Figure 2.7 shows a typical setup of four nodes with different node characteristics. Table 2.3 lists and explains the different nodes and their types involved.

We can see that node C subscribed to node B which is an optimal solution for the given nodes infrastructure. But why is that? Table 2.4 reveals how different network zones are equipped with bandwidth.

There are three pillars we can identify:

- With this bottleneck in the internet access zone in mind, a system has to be built to perform optimal in this constraint-space.
- A typical subnet has more than one computer:
  - In home computing there could be a standard PC and a laptop installed
  - In the corporate environment there surely are a couple of computers
- The nature of backup is that a user who wants to safe his data will want to restore or access his data from within the same subnet with high probability.

Network zone	Characteristics
ISP backbones	Over-provisioned
Internet access	Bottleneck, asymmetric
Local subnet	Over-provisioned

Table 2.4: Typical network zones characteristics

These pillars lead to the need for local resource priorization. Of course a well-designed distributed backup system cannot rely only on local resources to safe a user's data set, but this system should well balance the given over-provisioning of the local subnet in terms of bandwidth with using storage of nodes farther away to improve safety. Ubidas' design is built on top of these two foundation stones. The principles are:

- Fast local restores
- Safe and secure remote backups

To come back to our example in Figure 2.7, node C having node B as its transit node is exactly the right thing to have. This configuration emerges through the use of the IPTree algorithm (see Section 2.6) in the bootstrapping process where for the local-only reachable node C, the best matching transit nodes in terms of IP tree equivalence are evaluated. In this case it is node B as the primary transit node. The performance gain by using close transit nodes over random transit nodes are significant and evaluated in Section 4.1.2.

For other nodes in the example, IPTree determined for node D to select node A as its transit node. Node A and node B can initiate direct transfers between them to send both messages to their own node and to their subscribed local nodes.

This concludes the abstraction layer of End2End and we now move upwards the Ubidas layers to Messaging.

2.5 Messaging

The Ubidas Messaging layer is the foundation for all application-level communication. After having abstracted the heterogeneous network scopes and different reachabilities in the previous sections we now have a common ground for the messaging infrastructure to build on. Real end-

From Guid	To Guid	Message Payload

Table 2.5: A generic message

to-end communication is given and directed messages arrive either directly or over a transit node to its specified destination.

It is important to note that Ubidas does not support multicast messaging on this foundation layer. This is a consequence from the abstraction sequence of layer 2 over TCP/IP to End2End. Multicast on IP-level cannot be achieved when some nodes are behind NATs and located in heterogeneous scopes. Because Ubidas is designed also to reach local and shielded nodes, too, multicast messaging must be re-introduced on a higher level of abstraction. The DHT layer accomplishes this and is explained in Section 2.7.

The concept of the basic messaging layer is to provide a generic type of a message. A generic message is displayed in Table 2.5. One notices the sender's and recipient's fields. As an addressing schema, IP addresses cannot be used because multiple identical local IP addresses will exist. Hashing the IPTree path would be an approach. But because some nodes will move to different networks scopes (i.e., notebook computers) Ubidas uses Globally Unique Identifiers (Guids[5]) to identify nodes. A Guid is a 128 bit number which is randomly generated with the SHA-1 cryptographic algorithm which guarantees a very low probability for collision. A node receives its Guid at first start-up; the new Guid is then stored persistently on the node's disk. This is important, because the whole Ubidas addressing and therefore storage lookup infrastructure builds on top of these Guids.

A generic message record thus only consists of the addressing fields `FromMachineGuid` and `ToMachineGuid` and one generic message payload field. In Chapter 3, we will see that the .NET-based implementation of Ubidas builds on top of a generic `UbidasMessagePayload` type by deriving subtypes for higher abstraction payload type implementations.

Messages on this level are not persistent and any consumer of the basic messaging infrastructure needs to take care of it. Before the introduction of Ubidas DHT in Section 2.7 IPTree, as an underlying concept is discussed in the following section.

[5] `http://www.itu.int/ITU-T/asn1/uuid.html`

2.6 IPTree - Localization Semantics

Ubidas IPTree is a tree-like structure and a set of generic algorithms that enable localization of a network node and basic query operations such as compares and sorts on the IPTree structure.

As stated in Section 2.4.6 a real-world p2p system is distributed over different network scopes, with different reachabilities and bottlenecks. Also it was noted that a backup system should place some of the redundant data set on close nodes to enable fast restores or possible fast data sharing.

IPTree was briefly mentioned in previous sections and is now introduced in detail. IPv4 addresses are assigned to ISPs or—in case of provider-independent addresses—individuals in the form of subnets. ISPs themselves assign one or more IP addresses to their clients. Because of the mentioned IPv4 shortage, NAT devices must be introduced to enable more computers to connect to the internet than the assigned number of public IP addresses.

This introduces the known hurdles for real end-to-end connectivity which are bridged by the Ubidas End2End layer. Now, like in all classical p2p systems, with the introduction of the overlay layer, knowledge about the underlying equipment and organisation of the network is lost due to abstraction. Ubidas re-introduces these semantics with IPTree, bringing back knowledge about every node's underlying network location. It is important to mention that IPTree makes no statement about the real location of a node, but rather about its location in terms of internet access. For data-centric systems such as p2p backup this is important knowledge that has to be integrated to optimize the system's performance (e.g., restores).

2.6.1 IPTree Structure Building

IPTree creation is initiated in the bootstrap phase of Ubidas. A traceroute to Ubidas Index Server (UIS) is run and its result is formatted in a path-like description of the tree. Figure 2.8 shows a client node's path to UIS. Traceroute returns a result packet for every network hop in between. The tree is built by adding all traceroute hops from top down and the node's own IP address as the leaf. An example is displayed in Table 2.6. A path-like representation of node A's IPTree looks like this:

62.2.30.1 **.** 62.2.30.65 **.** 192.168.0.1 **.** 192.168.0.195

Note the bold dots joining all tree IP addresses.d dots joining all tree IP addresses.

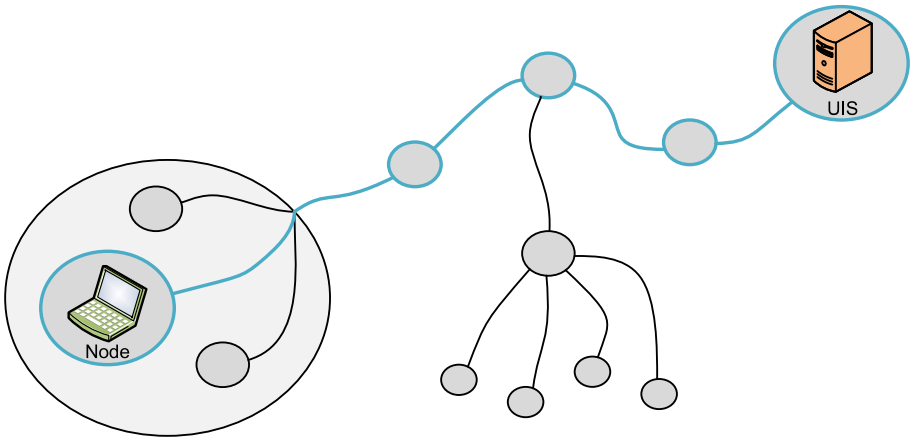

Figure 2.8: The IPTree concept illustrated: a node sends a traceroute to UIS to determine its IPTree.

Tree node	**1**	**2**	**3**	**4**
Type	ISP router	ISP gateway	Home router	Local IP address
Node A	62.2.30.1	62.2.30.65	192.168.0.1	192.168.0.195
Node B	62.2.30.1	62.2.30.65	192.168.0.1	192.168.0.194

Table 2.6: IP trees for two nodes in the same subnet

2.6.2 IPTree Algorithms

Once a node generated its IPTree, it sends it both to the UIS and via DHT broadcast to other nodes in the system embedded in in its Node Characteristics record.

There are two simple algorithms implemented that can work with the IPTree data structure: `IsInSameSubnet` and `GetClosestNodes`. The method's names speak for themselves and are used throughout Ubidas to optimize available local resources.

Comparing two given IPTrees to determine if they are in the same subnet is achieved by comparing the second- and third-last nodes of each node's IPTree for equivalence. Table 2.6 illustrates two node's IPTrees that are in the same subnet.

Currently IPTree is used for transit node assignments, local DHT routing, and in a future version on the transfer layer itself. Performance gains due to Ubidas are evaluated in Section 4.1.3.

2.7 DHT

With established real end-to-end connectivity throughout heterogeneous network scopes, the local resource prioritization with IPTree, and basic messaging, Ubidas is equipped with the tool set to build a real-world, adaptive p2p system.

The goal of Ubidas is ultimately to backup users' data in a safe, secure, and efficient manner. With these central design goals in mind, the resource lookup, storage, and restore component must be highly reliable and not negatively impacted by system growth. As introduced in the Related Work Section 1.3, distributed hash tables (DHTs) are well-suited for the task. They provide a basic key/value lookup service for p2p systems. Once basic key/value lookup is implemented, the upper layers of encryption, record persistency, and redistribution, table of contents, and the actual storage layer can be built on top of this solid foundation.

Ubidas DHT directly builds on the messaging layer. To the common message type with its fields indicated in Table 2.5 is extended with DHT-specific fields. The most important is the time-to-live (TTL) field. DHT algorithms use a special kind of flooding with $O(log\ n)$ complexity. To prevent infinite flooding of the system, the TTL field is decreased every time a node receives a DHT record query. If it is 0 after decreasing, the node does not redistribute the request further.

It has been shown that standard DHTs have several weaknesses which will be addressed in the Ubidas DHT implementation.

Key/value pair as the low-level storage entity in a DHT is somewhat not enough as a descriptive name because Ubidas will extend this. Define: *DHT Record* as a tuple of a key with a value in the given key space, a sequence of bytes as the value, and possible other fields. A key is called *ID* in Ubidas.

When designing a DHT, the first decision is of how large a key space one wants to have. The key space is the numeric range in which a valid DHT record ID must lie. Chord used a 128 bit key space which has already a very low probability for collisions. To further reduce the collision probability and better prevent splitting attacks, Ubidas introduces a 512 bit key space. A Ubidas DHT ID thus uses 64 bytes or 128 bytes in octet string representation. 2^{512} is a huge number and demands special types beyond built-in system types to work with. The implementation details are explained in Section 4.1.1.

Figure 2.9 illustrates the Ubidas key space ring. It starts at `0=0x000..` and ends at 2^{512} `=0xfff..` at the top of the circle. Now for a given sequence of bytes named value, a crypto-

graphically strong hash is created with the SHA-512 managed algorithm[6] to form the ID. Figure 2.9 also shows such an ID placed on the key space ring.

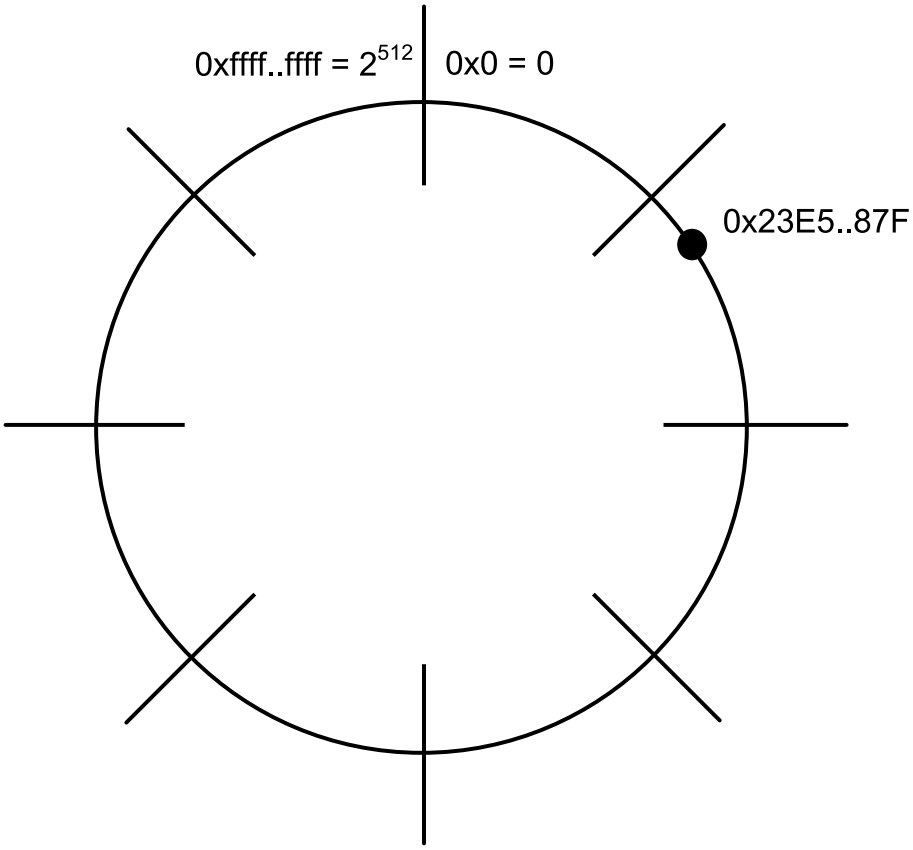

Figure 2.9: The DHT concept

The weaknesses of classic DHTs such as Chord were already stated in Section 1.2.3. Figures 2.10 shows a simplified classic Chord DHT.

In a classic DHT, the key space is divided into node responsibility segments. Consider the generated ID from before, again. Notice how the hash value comes to lie in the "upper-right to right" segment and the responsible node is A. If node A now leaves, node B and D close the gap and B will be appointed responsible for that ID. The rate of joins and leaves to/from the system is also called the churn rate. Management effort is defined as the effort needed to re-organize the responsibility ranges due to joins/leaves. In a real-world p2p system the churn rate is relatively high. A quick estimation for a system of size $N = 1'000'000$ nodes, with an average node behavior of two system starts per day for 4hours each, shows this:

[6] `http://msdn2.microsoft.com/library/system.security.cryptography.sha512managed.aspx`

Average running time of node $t_{running} = 4h$

System starts per node per day $n_{reboot} = 2$

Average total uptime per node per day $t_{up} = 8h = \frac{1}{3}d$

Joins/Leaves per node per day $Cr_{one\ node} = n_{reboot} * 2 = 4joines/leaves$

Joins/Leaves for the system per day $Cr_{system} = Cr_{one\ node} * N = 4'000'000 \frac{joines/leaves}{d}$

Churn rate (joines/leaves per system per hour) $cr = \frac{Cr_{system}}{24h} = 166'666.666 \frac{joines/leaves}{h}$

This is a huge number for the management effort and without basic optimizations it would not scale; the management overhead would be more than payload traffic. One quick optimization Chord adds, is the redundancy factor r. It assigns more than one node for each responsibility segment. This allows a relaxed management effort. Therefore gaps from a leaving node do not lead to a hole in the key space ring because other nodes are responsible for the same segment.

One issue with this kind of redundancy is the fixed start- and endpoints for a given segment. This static segmenting still allows for splitting attacks and there is a direct negative trade-off relation between the size of r and the needed management effort. Ubidas addresses both issues with two new concepts, *Cycle Redundancy* and *Derive Fraction Redundancy*.

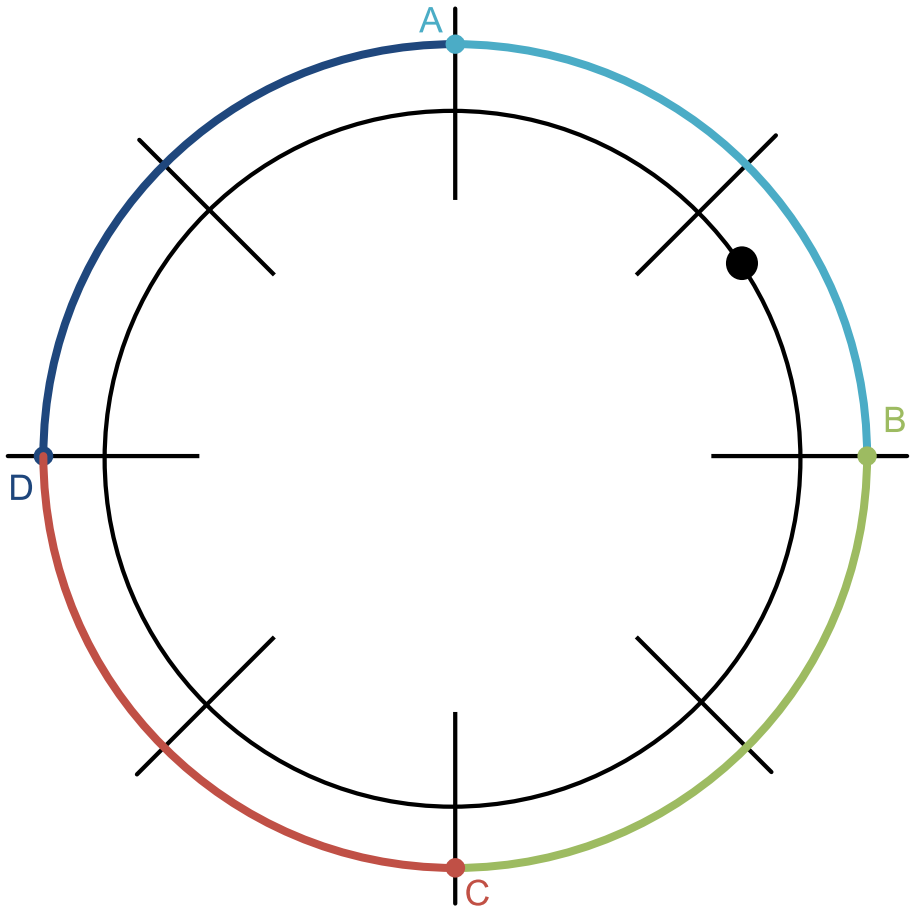

Figure 2.10: A Classic Chord DHT: static segments for node responsibility ranges.

2.7.1 Cycle Redundancy

Ubidas Cycle Redundancy is a similar concept like the redundancy factor r in Chord. the main difference is that it has not fixed segments that are replicated to other nodes to take care of. Instead, every node is responsible for a segment starting with his individual ID.

Chord also needs coordination with neighbor nodes to determine a node's responsibility range, whereas Ubidas uses metainformation about the system to achieve the same. In the bootstrapping process a node queries UIS for a set of system parameters such as the total system size N and the redundancy factor r. To determine its range length, the following formula is used:

$$\text{Key space max value } MAX_{key} = 2^{512} - 1$$
$$\text{Even distribution node range length } range_{even\ distribution} = \frac{MAX_{key}}{N}$$
$$\text{Node range length with redundancy } range_{redundant} = range_{even\ distribution} * r$$

Figure 2.11 illustrates the overlapping ranges of nodes A,B, and C. All these nodes are responsible for serving the illustrated DHT record.

This redundancy helps relax management efforts. A huge advantage is that it scales with the system. If for example $N <= r$, every node will be responsible for one full circle of the DHT key space ring. This allows Ubidas to be deployed to considerably small systems that run on a small set of nodes or to semantic overlay networks that sometimes partition DHTs down to a small size for semantic-optimized routing and resource discovery [Comito et al., 2006].

2.7.2 Derive Fraction Redundancy

The second optimization to the DHT is a completely new approach called *Derive Fraction Redundancy*. A major problem with classic DHTs is the relatively large attack surface for splitting attacks. In these attacks, an evil user uses a substantial amount of machines, all with the same or a similar node ID and joins the system. Once the system is reorganized, giving a segment of the DHT key space ring to be managed by these nodes, the attacker pulls the plug on all his instances at the same time. All DHT records in the respective segment are lost. Ubidas Cycle Redundancy makes it harder for an attacker because it uses dynamic range start points and nodes assign their ranges individually, not caring about any neighbors. It is still possible though to critically weaken the system. Therefore Derive Fraction Redundancy is introduced. The core idea is to replicate a DHT record over the DHT key space ring. Table 2.7 shows a list of evaluated key derivation strategies and their characteristics.

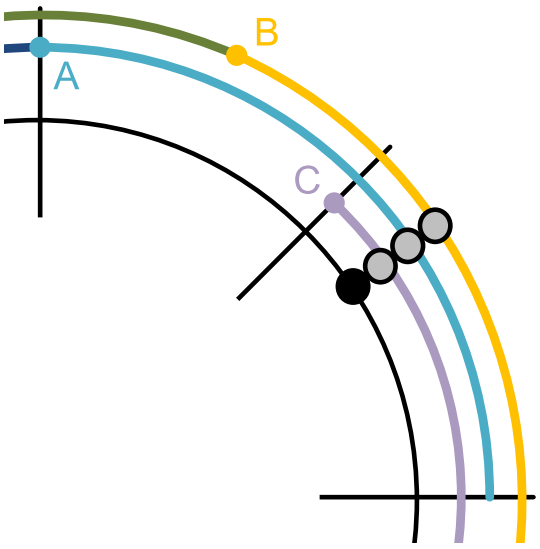

Figure 2.11: Ubidas DHT Cycle Redundancy: overlapping node responsibilities for the same ID on the key space ring.

The third strategy *Derive Fraction* offers the best approach, being computationally simple and covering the whole key space circle. The derive fraction strategy evenly distributes a copy of a DHT record in a polyangular way over the whole key space. Figure 2.12 illustrates the derive fraction redundancy strategy with a factor of 4. This approach applied, Additionally to nodes A, B, and C, nodes D, E, F, G, and H become responsible, too. The derived DHT records are indicated in gray. A special feature to prevent unnecessary management efforts is that derived records are marked with the fraction applied to reconstruct the so-called originary DHT record. A node periodically goes through all stored records and re-broadcasts all originary records to nodes it determined responsible. This relaxed, digest-style record management reduces heavy re-synchronization efforts. Derived records are not re-distributed as they only represent backups of the originary DHT records. This is of course a trade-off between storage requirements and redundancy. Keeping record size at a minimum, a large combined redundancy is feasible as the evaluation shows in Section 4.1.1.

Strategy	Characteristics	Key space coverage
Inversion	Invert a DHT key. Adds redundancy of 1	Full
Fraction	Create fractions of key such as $\frac{1}{f} * key$, $\frac{2}{f} * key$, .. $\frac{f-1}{f} * key$. Adds redundancy of $f - 1$.	Partial. $(0..key)$
Derive Fraction	Creates spacial redundancy by adding $\frac{x}{f} * MAX_{key}$ to key where f is a constant, $f - 1$ representing the added redundancy and x is varied in $[1..f)$.	Full

Table 2.7: DHT key derivation strategies

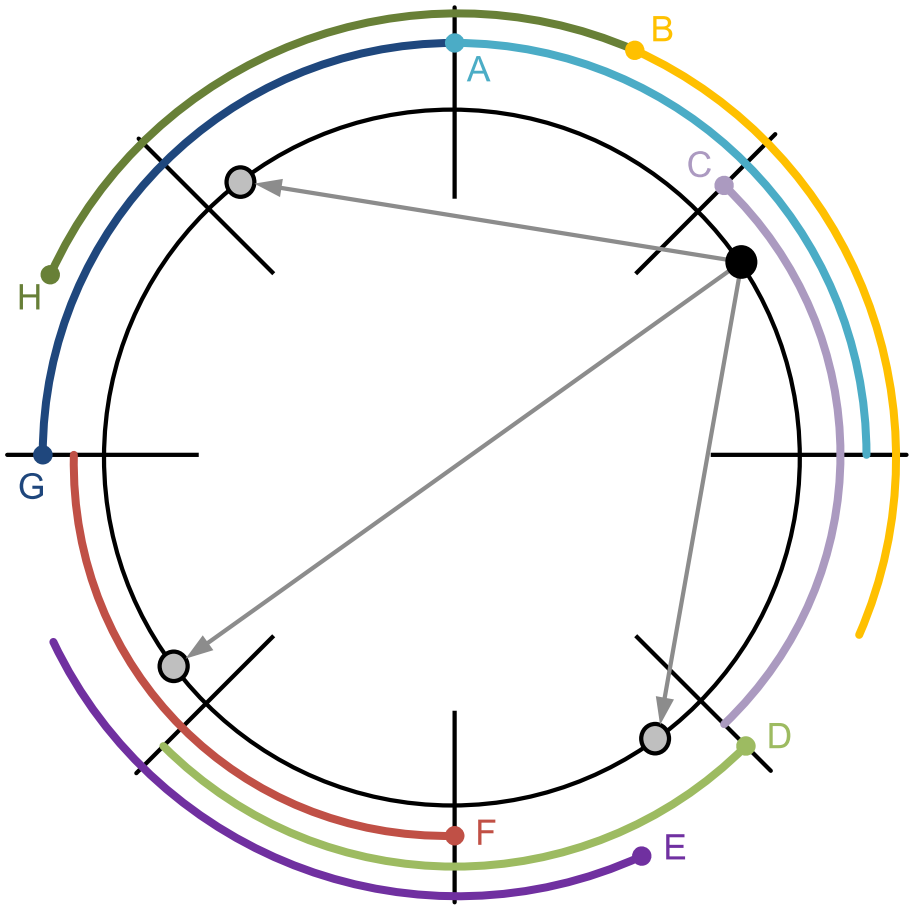

Figure 2.12: Ubidas DHT Fraction Redundancy: deriving a DHT record to distribute copies evenly over the key space ring.

2.7.3 Persistent Record Storage

Up to now, it was explained how DHT records are created, routed, and how a node's responsibility range is determined. To retain DHT records beyond a node's shutdown, a persistent record store has been implemented. If a node is responsible for a received node it needs to update its persistent store. First, the record is serialized with XmlSerializer[7]. This serialization process yields a XML string describing the record's instance but not a full description of its type and fields—rather it's the minimal needed description to recreate the instance, later.

A sample, unencrypted serialized record can be seen in Listing 2.1. This represents a simple DHT record with its payload "Paris". Note also the field *Fraction* with its value *0.3* which denotes that this current record is a derived record.

Storing the serialized string on the hard disk is technically straight-forward, but considering the possible 2^{512} records, writing every serialized record in its own file is unfeasible due to modern operating systems file system limitations. This issue has been evaluated in Section 4.1.1.

```
<?xml version="1.0"?>
<ArrayOfDHTPersistentRecord
        xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
        xmlns:xsd="http://www.w3.org/2001/XMLSchema">
        ..
  <DHTPersistentRecord>
    <LastModifiedDateTime>2008-02-29 ..</LastModifiedDateTime>
    <HashRecord>
      <HashKey>K8XPmS4n0R5Ku .. SSKRZioprdGVzBjVg==</HashKey>
      <HashBase>UABhAHIAaQBzAA==</HashBase>
      <TTL>3</TTL>
      <Fraction>0.3</Fraction>
      <RecordPayloadHash />
      <RecordPayload>
        <RecordName>Paris</RecordName>
      </RecordPayload>
    </HashRecord>
  </DHTPersistentRecord>
  ..
</ArrayOfDHTPersistentRecord>
```

Listing 2.1: Persistent, serialized, and unencrypted DHT record

When a node receives a DHT request it converts the given ID—the 512 bit hash key—to its persistent file representation. Listing 2.2 illustrates a typical path with its three groups of 16 bytes long folders. This grouping of the DHT ID into 4 segments yields 32 bytes in hexadecimal format;

[7] http://msdn2.microsoft.com/library/system.xml.serialization.xmlserializer.aspx

thus every ID byte is converted to its 2 byte hexadecimal representation. This is needed so that special characters and no case sensitivity is needed in the folders. To reduce overall file size, the last 16 bytes of the ID space are grouped into one file named *hash*. This can be seen by the XML tag *ArrayOfDHTPersistentRecord* used in Listing 2.1—multiple DHT records with the same leading 48 bytes (or 96 bytes in the hexadecimal form which is used in the folder names) are stored in the same file.

```
C:\ubidas\dht\5EF902CC615B04517DED110904CAA504\3A2590 .. A1C340\3CBB02 .. 23253B\hash
```

Listing 2.2: Persistent DHT Record Path

2.8 Encrypted DHT

Up to now, DHT records in Ubidas were cleartext, as illustrated in Listing 2.1. Every node can see the contents of a DHT record (no confidentiality) and a record's authenticity is not protected—a tampered node can not be distinguished from a authentic node. As a preparation for the *TOC* layer described in Section 2.9 encrypted DHT records are introduced in this section.

The *RijndaelManaged*[8] algorithm is used to encrypt data throughout Ubidas. Rijndael is a state-of-the-art encryption standard. Key and block sizes can be varied; Ubidas uses 256 bit.

The generic base class of *RecordPayloadBase* is extended so support encrypted record payloads. The derived type is *DHTEncryptedRecordPayloadBase*. Listing 2.3 shows an excerpt of a serialized, encrypted DHT record. Note that the derived type of the payload is specified. *DHTEncryptedRecordPayloadBase* is a pure encapsulation layer; the nested type of the encrypted payload is specified in the field *RecordPayloadTypeName* which must be a derived class from *RecordPayloadBase*, again. The actual payload is stored in the field *EncryptedRecordPayload* and can only be decrypted with a known key and initialization vector (IV).

2.9 TOC

The Ubidas table of contents (TOC) layer sits on top of the Encrypted DHT layer presented in previous Section 2.8. Where *DHT* and the secured *Encrypted TOC* are designed to store a flat structure of key/value pairs, TOC is an abstraction layer on top, introducing hierarchical data structure storage. The obvious use for that in Ubidas is the actual table of contents describing the

[8] `http://msdn2.microsoft.com/library/system.security.cryptography.rijndaelmanaged.aspx`

```
..
<HashRecord>
  ..
  <RecordPayloadHash>6gZStxt+bQ18 .. xz7VFzQ==</RecordPayloadHash>
  <RecordPayload xsi:type="DHTEncryptedRecordPayloadBase">
    <RecordPayloadTypeName>HierarchicalRecord</RecordPayloadTypeName>
    <EncryptedRecordPayload>w+Qg .. Uz89o</EncryptedRecordPayload>
  </RecordPayload>
</HashRecord>
..
```

Listing 2.3: Excerpt of persistent, serialized, and encrypted DHT record

user's directories and files. But its design is not limited to that and can be used to implemented a persistent, safe, and reliable hierarchical data store such as a genealogical tree, organizational structure, or a library classification system.

The base type to use for TOC records is *HierarchicalRecord*. Ubidas implements several derived types that represent the hierarchical organization of a user's resource structure. Table 2.8 illustrates the different TOC record types in logical, increasing order. References between the TOC records are established with simple DHT IDs to the respective records.

2.9.1 Hierarchical Key Sharing

To allow dedicated sharing of resources, Ubidas introduces a hierarchical key system. This allows a user to share one single key with another user to allow read access to the specific file system item and to all items below in the hierarchical tree. The detailed key generation algorithm is explained below.

2.10 Transfer

The top layer of the Ubidas communication stack is the transfer layer. In this thesis the main focus was set on a solid underlying communication stack up to and including the Ubidas TOC. A simple Win32 client application was written as a proof-of-concept, linking the Ubidas Sentinel queue together with the TOC to generate and replicate folder structures in real-time into the Ubidas system.

The foundation Ubidas supports with its communication stack can be used straight-forward

Type	Description	Children
HierarchicalRecord	*Base TOC record type. All following records derive from this.*	Virtual type
UserRecord	A user record identifies a user with his ID (Email).	A list of machines associated with the user.
MachineRecord	Machine records are associated with a user record. It represents a physical machine, identified by its machine guid.	A list of file system records as references to a machine's drive(s).
FileSystemRecord	Respresents a generic file system object such as a drive or directory. Its parent is a reference to either, in the case of it being a drive, a machine record or otherwise to another file system record.	The children records are pointers to sub directories (pointer to FileSystemRecord) and/or file records.
FileRecord	File records are leaves in a traditional file system tree. Here they represent a bucket structure for its file chunks.	A list of references to file chunk records.

Table 2.8: Derived TOC records represent a user's resource tree structure

to implement any distributed storage application including backup. It was originally intended but then considered beyond a realistic scope in terms of time available to fully implement the transfer layer and thus left open for future work.

2.10.1 Decoupling of TOC and Transfer Layer

The design of Ubidas decouples Transfer from TOC. This enables a stack-oriented approach to "plug-in" any distributed storage application layer on top and optimally take advantage of the Ubidas communication stack.

3

Implementation

After discussing the fundamentals of Ubidas in Chapter 2, the following *Implementation* chapter explains the technical background and implementation. It is structured into the following three core sections:

- **Technology Evaluation:** A detailed description of the technology evaluation that lead to the current implementation of Ubidas.
- **System Architecture:** Discussion of selected class structures and instance handling in Ubidas.
- **Implementation Highlights:** A few implementation highlights with basic code samples.

and closes with a technology conclusion section. The implementation of Ubidas consists of about 8000 lines of code, it is thus beyond the scope of this thesis to go into every detail of the implementation and therefore a selection had to be made on selected highlights. The complete source code is included in the accompanying CD-ROM.

3.1 Technology Evaluation

At the beginning of this diploma thesis a suitable technology platform had to be evaluated and selected. This section outlines the path that led to the final technologies used.

The criteria for the technology evaluation were centered around the following core principles:

- **State-of-the-art:** The technology platform should be up-to-date in terms of industry standards (e.g., full support for XML serialization, service abstraction, etc.)

- **Service-orientation**Services should be supported on a highly abstracted level and independent from its implementation—separation of function from configuration.
- **Solid platform:** Recent platform advancements such as memory protection, garbage collection, and an overall platform ecosystem with integrated development environment (IDE), broad APIs, extensive documentation, and community support are vital for fast development and reliable results. Further, more platform independence should be achievable with minimal effort.

These allow us to fulfill the goal behind the central ideas of this thesis; to actually build a solid system, rather than implementing a scripted prototype.

Microsoft's .NET Framework[1] (.NET) was chosen as the base technology platform. .NET covers all goals stated above: with Microsoft's major focus on shifting its own implementations toward .NET and broad industry adoption, it is a strong competitor to Java.[2] With its .NET version 3.0, Microsoft introduced Windows Communication Foundation (WCF), a cornerstone for networked applications. Details on WCF will be explained in Section 3.2.

Platform independence can be achieved by running the .NET code in the Mono environment that supports beyond Windows also Linux, Solaris, Mac OS X, and Unix.[3]

3.1.1 BitTorrent

The BitTorrent protocol is a popular p2p protocol mainly used for data distribution, such as file sharing.[4] It uses a few central index servers, also known as trackers, that hold information about which nodes store which data. An extension enables BitTorrent to function tracker-less with the help of a DHT. BitTorrent was first considered as a p2p layer to implement Ubidas. First, a suitable library had to be found.

MonoTorrent

MonoTorrent is an open and free—under MIT/X11 license—.NET BitTorrent library.[5]. Development of a p2p application based on MonoTorrent is straight-forward and would have made the implementation of Ubidas much easier.

[1] `http://msdn2.microsoft.com/netframework/`
[2] `http://java.sun.com`
[3] `http://www.mono-project.com/`
[4] `http://www.bittorrent.com`
[5] `http://www.monotorrent.com`

Evaluation

An early proof of concept prototype of Ubidas was developed using MonoTorrent. A known weakness of BitTorrent is, as in many other p2p protocols, its ultimate inability to prevent freeriding.[6] BitTorrent uses the so-called tit-for-tat principle to create an incentive not to freeride: nodes that request data from another node in the network need to offer some bandwidth on their own to the targeted node.

After conceptual considerations and extensive discussions with Alan McGovern, the creator of MonoTorrent, an alternative to BitTorrent in general had to be considered. The main reason is that tit-for-tat works great for resources that are accessed by a lot of users—users to files ratio acts like $m : 1$. In a backup system on the other hand, the principle is inverted because one user typically has numerous files that are not requested by other nodes—$1 : n$. This practically eliminates the positive effects of tit-for-tat in p2p file sharing applications for p2p backup systems and with the current specification of BitTorrent, there is no way to prevent freeriding ultimately.

3.1.2 Peer Name Resolution Protocol (PNRP)

An alternative to BitTorrent had to be found. WCF looked promising as it offers its own p2p stack, called Peer Name Resolution Protocol (PNRP).[7]

Introduction

PNRP is a DHT-like distributed key/value cache store. It is excellent for routing purposes to reach nodes behind firewall and NAT devices as it abstracts real end-to-end connectivity with the help of Teredo tunneling.[8]

Evaluation

PNRP is available in version 2 as of the date of writing of this thesis. Several tests were conducted using PNRP routing and issues were identified for special NAT/firewall configurations: when used for nodes with a public IP address, but shielded behind a port-restricting firewall, the PNRP stack showed timeout-like disconnection behaviors that make it unfeasible for this kind of p2p

[6] Freeriding is the act of a node only benefiting from a p2p network without adequately donating its resources (i.e., bandwidth) to other users.

[7] `http://technet.microsoft.com/library/bb726971.aspx`

[8] `http://www.ietf.org/rfc/rfc4380.txt`

system. Also the gateway at the university assigns a wireless node a public IP but restricts access to the internet before the node is authenticated. For authentication a user visits special catch-all websites where credentials are entered and verified by the gateway. PNRP determines within the first thirty second after a reboot which tunneling mechanism to use based on the assigned IP address. If the IP address is from a public range, PNRP uses 6to4, a IPv6 to IPv4 encapsulation technique[9] but because the user was not fast enough to authenticate, the correct tunneling method would be Teredo instead. It is hard to change the tunneling method for a normal user and requires administrator privileges. Therefore PNRP had to be dropped from consideration for the Ubidas implementation.

3.1.3 Technology Evaluation Conclusions

The p2p technologies evaluated showed that for a p2p backup system a tailor-made p2p layer is needed to fulfill the outlined design goals in Section 1.3.

3.2 Windows Communication Foundation (WCF)

Windows Communication Foundation (WCF) was introduced with .NET Framework 3.0. In the following sections, a brief introduction to its characteristics that are of interest for Ubidas is given.

3.2.1 Introduction

WCF is a set of application programming interfaces (APIs) providing an advanced networking stack. A developer can choose from a variety of transport mechanisms, security configuration options, orchestration, etc.

3.2.2 Service-Oriented Architecture (SOA)

A major benefit is its tools and libraries to build consistent service-oriented architectures. Although SOA is a buzzword and often overused, it is a ground-breaking revolution of how application components interact with each other. WCF takes SOA to the level where even within the same machine, processes can communicate with each other in a SOA-like manner with named

[9] `http://www.ietf.org/rfc/rfc3056.txt`

pipes, called *net.pipe* in WCF.[10] This might sound like an over-design but the consistent separation of function (code) from configuration makes an application extension from two communicating processes, with a change of configuration only, to communication with a distant computer easy.

3.2.3 Endpoints: Address, Contract, and Binding

The underlying principle of the above mentioned separation is implemented in the concept of endpoints in WCF. As Figure 3.1 shows, an endpoint consists of an address, a contract, and a binding. This allows a single application to have various endpoints for different consumers of the same service, without the need for a single change in the service code (function) itself. A service client can then choose whatever endpoint suits its needs best. More information about the concept of endpoints can be found on the Microsoft MSDN site.[11]

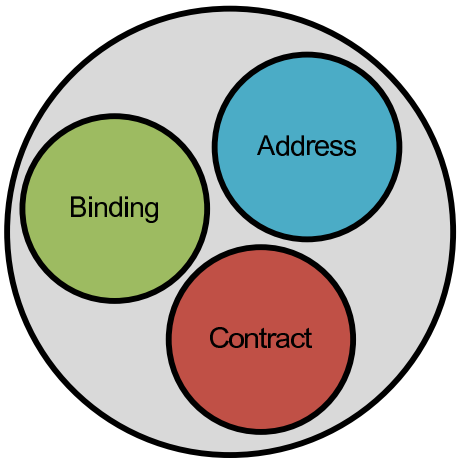

Figure 3.1: WCF Endpoints consisting of Address, Contract, and Binding

3.2.4 Evaluation and Restrictions

While WCF is not a p2p library it was a good starting point as a network stack to build Ubidas on top. The first Ubidas communication stack layer that sits directly on top of the WCF APIs is Ubidas End2End, as explained in Section 2.4.

A major advantage of using WCF is that in future work, Ubidas can be easily extended to serve low-level system read-only clients such as web browser over an exposed RESTful API. To

[10] `http://msdn2.microsoft.com/library/ms788992.aspx`
[11] `http://msdn2.microsoft.com/library/aa480210.aspx`

implement RESTful web services, WCF holds a variety of standards-compliant configurations ready—without any need for code changes.

From an implementation perspective, it meant more work to re-build all abstraction layers outlined in Figure 2.1. But having a solid foundation of distributed, secure reliable, and real end-to-end communication will help for future work to rapidly build systems on top of Ubidas.

3.3 System Architecture

This section presents selected system architecture elements. First, two central class hierarchy diagrams are introduced to show the use of pure object orientation in a p2p system, then issues such as instance handling and performance are explained in the second section to outline limitations and optimizations made to overcome them.

3.3.1 Selected Class Diagrams

Two central class diagrams were selected and presented in the following sub sections. It is interesting to note that object-orientation made the design of the central TOC component very straight-forward, enabling generic serialization for persistent DHT records including base and derived types for detailed characteristics implementation.

Ubidas Message Payload Class Structure

The message payload class hierarchy is displayed in Figure 3.2. The basic concept is that for an Ubidas message, there is one base type, `UbidasMessagePayload` to represent a the generic payload of a message in Ubidas. It can be as simple as a string message (`UbidasStringMessagePayload`), more complex for DHT records (`DHTRecordBase`, `DHTRequestRecord`, and `DHTResponseRecord`), or for node characteristics updates broadcast messages between nodes (`NodeCharacteristicsUpdateRecord`).

WCF supports derived types for services. This is an important feature because it allows services to handle messages transparently without needing to know about a derived type's inner workings! Ubidas is designed to bubble-up message events to subscribers based on the received message payload type. This allows for a plug-in-like infrastructure and extensibility on the implementation level—without any changes needed to the base Ubidas End2End layer.

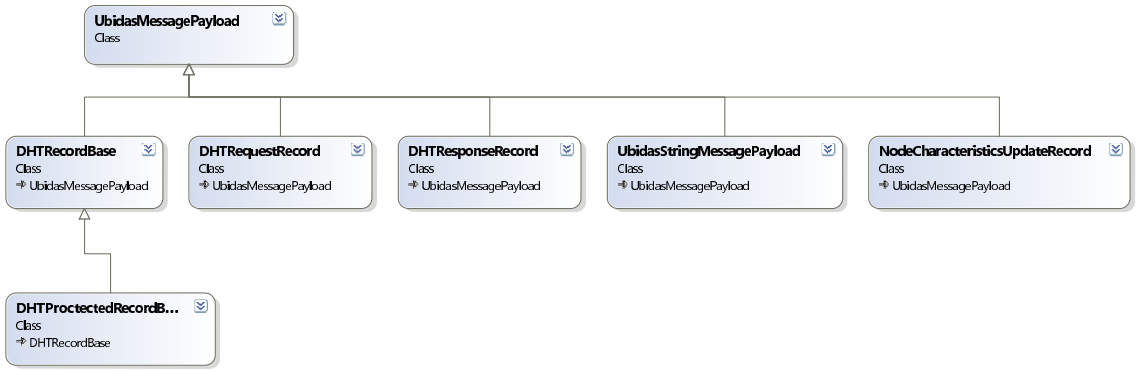

Figure 3.2: Ubidas Message Payload Class Structure

Ubidas TOC Class Structure

As mentioned in Table 2.8 Ubidas' TOC records are hierarchically structured to represent the real-world hierarchy structure in a user's file system. In the actual implementation, the TOC record types all derive from the same parent class, `DHTRecordPayloadBase` which is a member of the aforementioned `DHTRecordBase` class. `HierarchicalRecord` is a derived DHT record type to handle the nature of the hierarchical file system structure. Its deriving child classes are, in logical order, `UserRecord`, `MachineRecord`, the generic `FileSystemRecord` that handles drives and directories, and `FileRecord`.

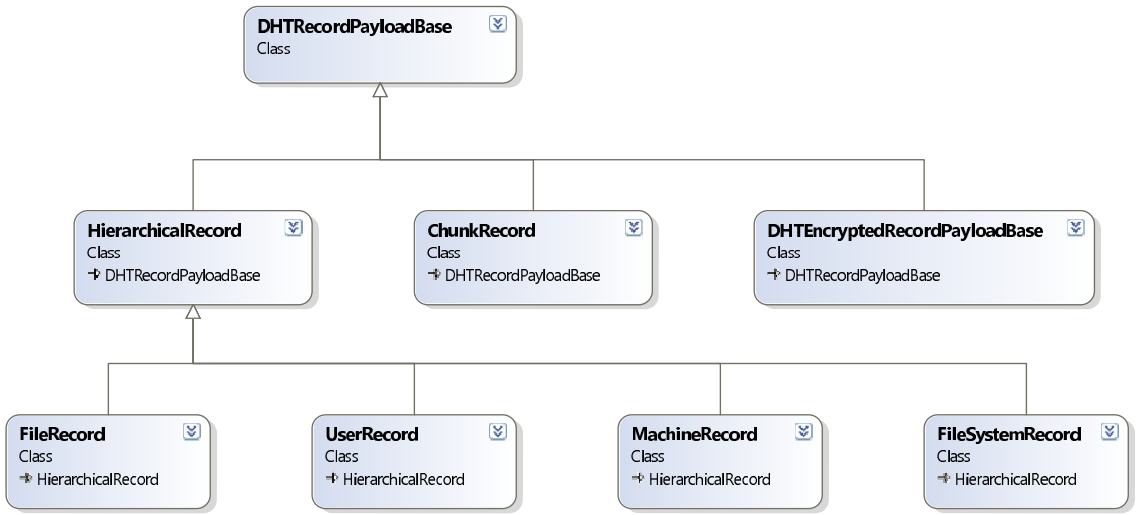

Figure 3.3: Ubidas TOC Hierarchical Class Structure

3.3.2 Instance Handling and Performance

There are three known meanings of *instance handling* in service-oriented applications: application or service instance handling or service instance hosting. In an effective distributed application, instance handling is central to the uptime of a system. Ubidas can be configured to launch at startup as one background running process. Alternatively, suitable for server nodes, Ubidas could be hosted in a Windows Service without the need for a user to sign-in first.

On the other hand, service instance handling concentrates on how to handle individual service instances. There are two basic modes: singleton and multiple (i.e., per call or per session instance). While the first has memory footprint advantages, the later can perform much faster under heavy request load. A central issue with singleton handling is that every call has to be queued and the next call will only be handled, when the earlier has completed. If the service communication pattern is asynchronous or heavy load is a characteristic of the system, this becomes unfeasible.

Ubidas uses long running (i.e., 20 seconds) poll calls that hang before the service returns a result. This pattern is used for private clients that need to subscribe to a transit node. Once the transit node has an incoming message for the requesting node, it returns the polling call immediately with $true$, indicating that transit items are pending, otherwise it returns $false$ after the pre-specified timeout.

This makes it unfeasible to run Ubidas services in singleton service mode and multiple instances (i.e., threads) must be enabled. Multiple threads draw the consequence of proper multithreading which is hard, especially in distributed applications. During the development of Ubidas, numerous tests and optimizations were conducted to find the most suitable configuration of the parameters: *maximal service instances*, *thread pool size*, and *asynchronity of service calls*. The findings in the case of Ubidas is to throttle service instances to less than 20 and to leave the thread pool size for fire-and-forget-style *send* operations at 25 threads. Asynchronity can still become an issue to resource locking, blocking other waiting calls in the individual handling threads behind the service facade. Handling operation service calls synchronously and as fast as possible is key to fast performance. In a service application this can be measured by effective successful method calls per second. In Ubidas this was measured on an application level—up to 40 messages per second were measured as sustained operations throughput.

The third aspect of instance handling is service instance hosting. Services must be hosted by a process that continuously runs on the operating system and waits for incoming calls. In WCF service hosting is completely detached from the actual code and is only a question of the respective configuration settings. WCF offers the following service hosting scenarios:

- **Internet Information Services (IIS):** IIS is the web server suite for Windows Server operating systems. Hosting services is only feasible for servers as most client operating systems do not run a web server.
- **Windows Activation Services (WAS):** To host services independent of a parent application, WAS was introduced in Windows Vista. WAS is a container process, installed as a Windows service that hosts WCF services, only. This is suitable for light-weight WCF applications that have a thin service layer (e.g., a service orchestration service).
- **In-process hosting:** The last option is to use in-process service hosting. This simply means that a WCF service is hosted from within a client application. This can be a simple console application or a more complex WinForms application that has a GUI.

Ubidas services are hosted with the in-process hosting method. Allowing the Ubidas Controller application to host the WCF service layers.

3.4 Implementation Highlights

The following sections are code-centric and show the usage of single Ubidas components in code. They are simplified for better readability. The fully functional implementation of Ubidas can be found on the accompanying CD-ROM.

3.4.1 Sentinel

Ubidas Sentinel can be configured to:

- include certain directories and whether to include their subdirectories,
- exclude certain sub-directories within included directories, and
- exclude certain file types from included directories.

Listing 3.1 shows such a configuration section of Ubidas Sentinel. The code is fairly self-explanatory; worth noting is line 6 where all `temp` directories are excluded from sub-directories of `c:\users`. An example for exclusion is `c:\users\john\temp`. On line 9, several file types are excluded from Sentinel. This makes sense for temporary and large, frequent-changing files (e.g., `*.ost` for Microsoft's Outlook offline cache) that should be saved with an application-level

```
1  ..
2  SentinelCoreConfig config = new SentinelCoreConfig();
3  config.FileWatcherConfig.AddDirectories(new string[] { @"c:\users" },true);
4  config.FileWatcherConfig.AddDirectories(new string[] { @"d:\pictures" },true);
5  ..
6  config.FileWatcherConfig.AddExcludeDirectory(@"c:\users\*\temp");
7  config.FileWatcherConfig.AddExcludeDirectory(@"c:\pictures\old");
8  ..
9  string[] excludeExt=new string[] { "dat", "log", "tmp", "db", "ost", "pst" };
10 config.FileWatcherConfig.ExcludeExtensions.AddRange(excludeExt);
11
12 SentinelCore core = new SentinelCore(config);
13 core.StartCore();
14 core.FileSystemItemUpdated
15         += new SentinelCore.FileSystemItemUpdateEventHandler(core_FileSystemItemUpdated);
16 ..
```

Listing 3.1: Sentinel Configuration, Event Subscription and Launch

backup plug-in that understands the file's semantics. Finally on line 14/15 a client—in this case Ubidas Core—subscribes to the Sentinel file system events. An event is fired for every changed file matching the specified rules.

3.4.2 Basic Messaging

Ubidas End2End presented in Section 2.4 enables real end-to-end communication. This level of abstraction allows for simple operation invocations to send basic, unencrypted messages from node to node.

Listing 3.2 shows how a basic string message is constructed and sent to another node in the Ubidas system. Lines 1 and 2 instantiate and start the Ubidas Core. Line 6 specifies the unicast recipient of the message with the target node's machine guid. As the message payload, an instance of the message payload type `UbidasStringMessagePayload` is created on line 7. Finally the message is submitted for delivery to the Ubidas Core sub-system on line 10.

3.4.3 DHT Operations

The following two sections show how basic DHT operations are performed. The first section writes simple unencrypted DHT records to, whereas in the second section they are being retrieved from the Ubidas DHT sub-system.

```
1  UbidasCore ubidasCore = new UbidasCore();
2  ubidasCore.Start();
3  ..
4  UbidasMessage stringMessage = new UbidasMessage();
5  stringMessage.FromMachineGuid = ownGuid;
6  stringMessage.ToMachineGuid = new Guid("3e27d121-645e-4f22-b2e6-ce85ba48eed4");
7  UbidasStringMessagePayload stringPayload = new UbidasStringMessagePayload();
8  stringPayload.StringMessage = "hi there!";
9  stringMessage.MessagePayload = stringPayload;
10 ubidasCore.Send(stringMessage);
11 ..
```

Listing 3.2: Basic Messaging In Ubidas

```
1  UbidasCore ubidasCore = new UbidasCore();
2  ubidasCore.Start();
3  ..
4  ubidasCore.DHT.Put(ubidasCore.DHT.CreateOpenRecord("New York"));
5  ubidasCore.DHT.Put(ubidasCore.DHT.CreateOpenRecord("Paris"));
6  ubidasCore.DHT.Put(ubidasCore.DHT.CreateOpenRecord("London"));
7  ..
```

Listing 3.3: DHT Put Operation in Ubidas

DHT Put

Commonly write operations to a DHT are named *put*. Lines 4 through 6 in Listing 3.3 submit new DHT records to the Ubidas DHT sub-system. Note that these calls are non-blocking as a delivery queue is filled in the sub-system and delivery is performed by a background thread pool.

DHT Get

To retrieve DHT records, the commonly named *get* operation is invoked. Listing 3.4 retrieves the written DHT records from Listing 3.3. Note that lines 6 and 7 are blocking lines with a maximum failure timeout of currently five seconds. The DHT sub-system sends out a *DHTRequestRecord* asking for the specified hash value. If the returned record is not null, it is displayed in the console on line 10.

Note that for simplification and illustration purposes only, on line 7, an open record is created with the message string as the payload argument. This would of course make no sense in reality and a hash from a TOC store would be supplied to retrieve the corresponding DHT record value.

```
UbidasCore ubidasCore = new UbidasCore();
ubidasCore.Start();
..
foreach (string messageString in new string[] { "New York", "Paris", "London" })
{
    DHTResponseRecord responseRecord
        = ubidasCore.DHT.Get(UbidasTools.CreateOpenRecord(messageString).HashKey);
    if (responseRecord != null)
    {
        Console.WriteLine("Response: " + responseRecord.DHTAnswerRecordPayload.RecordName);
    }
    else
    {
        Console.WriteLine("Response was null");
    }
}
..
```

Listing 3.4: DHT Get Operation in Ubidas

This construct though is very useful to test successful DHT writing and retrieving.

3.5 Running the Ubidas System

The Ubidas system does not need to be installed to run. It is enough to simply copy-paste the binaries to a target node and execute the Ubidas Controller. As a prerequisite .NET Framework 3.0 is the only required component other than a Windows operating system. Note that the code can also be compiled under Mono to run on a variety of operating systems (untested). The Ubidas Index Server (UIS) is a simple console application that can be integrated in a service wrapper to run at startup. As a cache database, UIS uses Microsoft SQL Server Express 2008.[12] The database file on the accompanying CD-ROM can be attached to the SQL Server and no further linking from within the application is needed.

3.6 Implementation Conclusions

This chapter outlined the technology evaluation path taken, introduced WCF, explained selected class diagrams of hierarchical Ubidas data structures, and discussed some implementation high-

[12] Microsoft SQL Server Express 2008 is free and can be downloaded from: `http://www.microsoft.com/express/`

lights of Ubidas. After the implementation Ubidas, a clear conclusion can be drawn: p2p overlay libraries for the real world must be designed specifically to meet the unique application needs in terms of:

- reachability in terms of real end-to-end connectivity,
- robustness and reliability,
- security, and
- flexibilty.

WCF proved to be a solid base layer for a service-oriented p2p system such as Ubidas aspires to be. It offers a wide variety for future complementary implementations such as a RESTful interface for low-tech clients (i.e., web browsers), easy portability to other platforms, including feature and resource-restricted mobile devices, etc. The effort to bring this level of abstraction (Ubidas End2End, DHT, and TOC) to Ubidas was higher than applying an off-the-shelf p2p library, but the learn-effects and pinpointed improvements were more than worth the extra effort.

4

Evaluation

The previous Chapters 2 and 3 explained the concepts and implementation of Ubidas. Because Ubidas introduces several new concepts, some aspects of the system were already evaluated qualitatively and compared against similar systems. These include:

1. *End2End* in Section 2.4 presenting real end-to-end connectivity,
2. *IPTree* in Section 2.6 introducing concept of prioritization of local resources, and
3. *DHT* in Section 2.7 with its two new redundancy principles.

The goal of this chapter is to evaluate certain selected highlights of Ubidas qualitatively in Section 4.1 and quantitatively in Section 4.2. A central idea in the design of Ubidas is its *holistic approach*. Most systems concentrate on a certain aspect such as DHT record distribution effectiveness in Chord [Stoica et al., 2001] and evaluate its performance against the same parameters in similar systems. Ubidas is more like a mash-up in the sense that it combines known concepts, improves certain aspects of them, and chooses parameters such that the design goals outlined in Section 1.3 are met. This trade-off dilemma of conflicting goals was explained in detail in Section 1.3.1. The Section 4.2 *Numeric Analysis* and its results thus have to be read with these remarks in mind. Finally Section 4.3 discusses the limitations of Ubidas.

4.1 Selected Evaluation Highlights

Additionally to the discussed concepts in Sections 2.4 through 2.7, selected highlights of Ubidas are evaluated qualitatively in this section.

4.1.1 Persistent DHT Store

Ubidas DHT has a key space capacity of 2^{512}. Because DHT records must be retained beyond runtime, they are stored persistently on the node's hard disk. 2^{512} is a large number and modern operating systems cannot deal with such large numbers in terms of built-in types and modern file system such as the *New Technology File System* (NTFS) cannot handle so many objects (i.e., directories and files). The first issue is addressed with the help of an open source implementation, titled *BigInteger*.[1] A `BigInteger` object can hold any size of an integer number. The second issue is harder to deal with; NTFS has a limit of $2^{32} - 1$ addressable objects (i.e., directories and files) per volume.[2] This makes it impossible to create one file per DHT record as it would not be able to scale to the full key space limit of 2^{512}. On the oder hand, NTFS has a minimal file size allocation of 4 kilobytes. An average unencrypted DHT record in Ubidas consumes only about 1 kilobyte: storing every DHT record in its single file would, again, be suboptimal.

Under these two constraints, Ubidas established an optimal concept for a persistent DHT store with a key space of 2^{512}. Its design is evaluated in this section. The optimal solution was found in a constraint-oriented process: an Excel file was created, containing all constraints and system parameters that influence the persistent DHT store—the distribution, size and number of records files. The Excel file can be found on the accompanying CD-ROM. One can change single system parameters and see how other dependent values change. For example, one might want to know the effect of a doubled *Cycle Redundancy* factor on the average number of hash files to store per node.

A formatted version of the Excel file is rendered in Table 4.1; it consists of the constraints such as maximal object in the NTFS file system and system parameters such as *Derive Fraction Redundancy*, *Cycle Redundancy*, and average number of files per user including usage distribution with two basic types. The table's result show an average of 16 million hash records per user in a fully synchronized system. Assuming an average serialized TOC record's storage requirements is 1 kilobyte, a user would need to donate about 16 gigabytes for TOC storage, on average. This seems like a high number at first, but as stated in the introductory Section 1.3.1 titled *Conflicting Goals: Choosing a Design Philosophy* designing a system is always a trade-off between several goals. The presented Excel sheet is therefore a handy tool to evaluate chosen system parameters on their feasibility. With the rising storage space on user's computers, storage requirements is not a primary concern, rather it is to achieve safe and reliable backups and restores. The chosen Ubidas

[1] `http://www.codeproject.com/KB/cs/biginteger.aspx`
[2] `http://technet2.microsoft.com/windowsserver/`

system parameters need to be seen in this perspective.

4.1.2 Local Transit Nodes Emergence

This section qualitatively evaluates the automatic emergence of local transit nodes in Ubidas. It is common that p2p overlay networks establish real end-to-end connectivity, fault-tolerance, and robustness but forget about the underlying network structure. In a traditional DHT implementation such as Chord [Stoica et al., 2001] or Pastry [Rowstron, 2001] messages are routed to their target based on the content-address-based shortest path with $O(log\ n)$ performance. The semantics of the underlying network structure is completely erased through overlay abstraction. For certain p2p applications such as backups considering the underlay network can improve system performance. Ubidas introduced IPTree in Section 2.6 which re-establishes the underlay network semantics. Private nodes—nodes that are not directly accessible from outside nodes—need to subscribe to a transit node. In a traditional p2p system this node would be chosen randomly; in Ubidas, through the use of IPTree, the optimal transit node is found and subscribed to. Because the IPTree of a node with transit capabilities in the same subnet has a high conformity with the IPTree of the private node it is ranked top in the transit node evaluation process and chosen with highest priority. This results in the emergence of local transit nodes where available. As briefly described in Table 2.4, the local subnet is over-provisioned, therefore communication within the same subnet is currently superior by about two to three orders of magnitude[3] compared to communication with nodes on the internet. This results in a substantial increase in transit speed in Ubidas if at least one local transit node is available. Figure 4.1 illustrates this.

4.1.3 IPTree

The previous Section 4.1.2 evaluated the emergence of local transit nodes due to local resource prioritization. This results from the re-introduction of underlying network semantics through the use of IPTree. IPTree is explained in Section 2.6. This section will evaluate qualitatively its characteristics over classic DHTs without network underlay network semantics.

The phenomenon of the emergence of local transit nodes is one example for the use of IPTree. To evaluate its net performance, one has to analyze the overhead efforts needed to re-create the underlying network semantics. There were other works with the goal to achieve prioritization of local resources. One is compared to Ubidas, below.

[3] Assuming internet bandwidth is 1MBit/s and local bandwidth 100MBit/s to 1GBit/s

System designed size	1′000′000 users
Files per user	10′000 files
Folders per user	1′000 folders
Max chunk size	10 MB
Data size distribution per user (power user)	
Type I (Images,Docs,etc)	90%
Avg Type I size	2 MB
Data Type I	18′000 MB
Chunks for Type I	9′000
Type II (Movies)	10%
Avg Type II size	300 MB
Data Type II	300′000 MB
Chunks for Type II	30′000
Originary hash records per user	
Files	39′000
Folders	1′000
Total originary hashes	40′000
Redundancy factors	
Derive Fraction Redundancy factor	10
Cycle Redundancy factor	40
Average TOC store requirements	
Hash records per user	16′000′000
System capabilities	
Bits per hash	512
in bytes	64
TOC hash space	2^{512}
System usage at designed size	
Total data	318′000′000′000 MB (318′000 TB)
Total chunks	40′000′000′000
Total different hash records (w/o cycle redundancy)	1′600′000′000′000
Is equivalent to	2^{47} bits (140′737′488′355′328)
TOC store organization	
NTFS limit	$4^{16} = 4'294'967'296$
Byte-groups of	16 bytes
Total TOC files	65′536
Storage need per record	1′000 bytes
Hashes per file	244.140625
Hash file size	0.244 MB

Table 4.1: The formatted optimal persistent DHT store computation. The original Excel file with adjustable parameters can be found on the accompanying CD-ROM

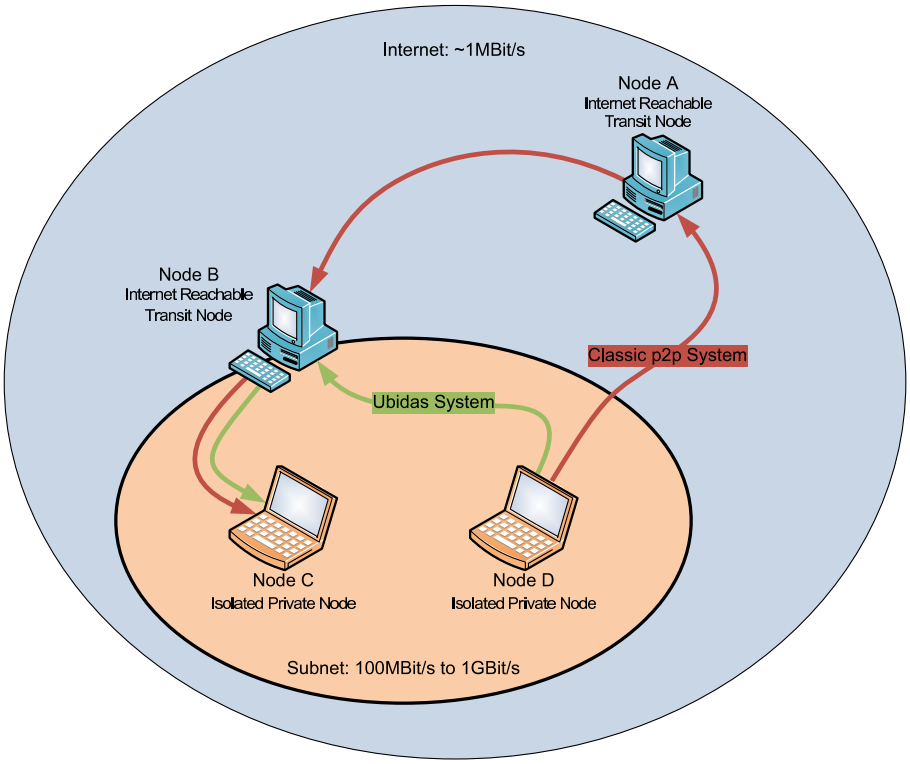

Figure 4.1: Local Transit Node Emergence: Comparison between a classic p2p system, routing traffic over node in the internet with low bandwidth (e.g., 1MBit/s) and Ubidas, routing traffic over available local transit node with considerably higher bandwidth (e.g., 100MBit/s).

[Castro et al., 2002] propose to establish topology-awareness for DHT record routing in p2p overlay systems by exchanging two messages between every pair of communicating nodes. This out-of-band approach imposes a communication overhead to gain topology-awareness. Another issue is scaling, because for every node to communicate with, the topology-awareness has to be established first. This leads to a scaling performance of $O(log\ n^2)$. Also, the knowledge has to be kept in sync as a node's network location might change over time. Again, $O(log\ n^2)$ effort is required to keep the topology-awareness up-to-date.

On the other hand an Ubidas node performs one traceroute, generate its IPTree, and globally identify its network location. To select available local resources (e.g., a local transit node) it then prioritizes nodes with a similar reported IPTree. This approach is in-band and scales with $O(log\ n)$ performance due to its global comparison instead of local negotiation style of [Castro et al., 2002]. Figure 4.2 shows a graph for each approach and how its communication overhead for topology-awareness generation scales with an increasing number of nodes On can see that the blue graph

that represents [Castro et al., 2002]'s approach clearly performes worse than the red graph, representing Ubidas' scaling performance.

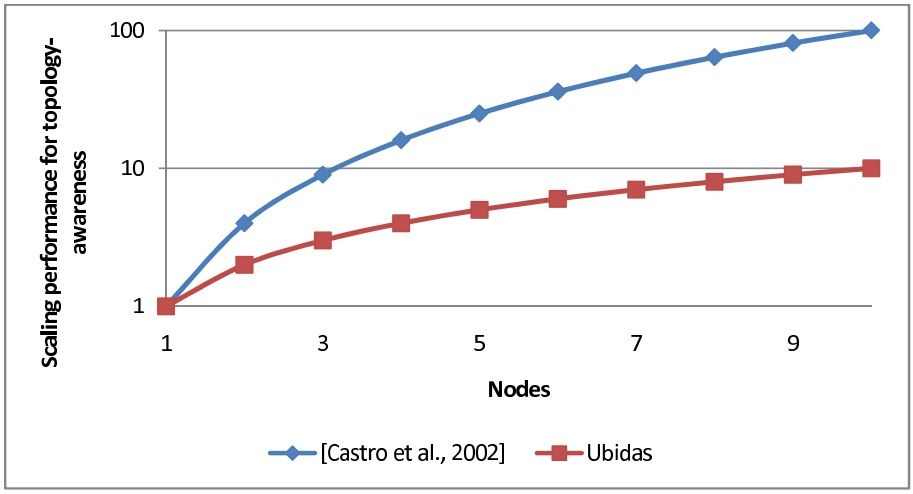

Figure 4.2: Scaling performance of [Castro et al., 2002] and Ubidas to generate topology-awareness. Note the logarithmic scale on the vertical axis.

4.2 Numeric Analysis

After evaluating selected aspects of Ubidas qualitatively in previous the Section 4.1, this section analyzes highlights of Ubidas quantitatively. As stated in the introduction Chapter 1 it is difficult to compare related work with Ubidas. Each of the discussed related work in Section 1.2 introduces a new concept (e.g., DHTs in Chord [Stoica et al., 2001]) or concentrates on improving a certain aspect of p2p systems (e.g., symmetric traffic trades in PeerStore [Zhang et al., 2004]). Ubidas, at its core, brings together the highlights of the discussed related work an introduces the new IPTree concept for local resource prioritization. This numeric evaluation section therefore analyzes the impact of this combination of combining p2p system concepts and prioritization of local resources on selected system properties. It is shown that if the usage pattern of individual users involves several nodes within close network proximity (e.g., in the same subnet), Ubidas can improve the data distribution process and the amount of time needed for complete restores, drastically.

4.2.1 Data Distribution Process Evaluation

If local nodes are present, Ubidas automatically detects them with IPTree and prioritizes distribution of data to them up to a specified number of nodes (r_{local}). This results in data redundancy in the local subnet and enables high-speed restores. The defined total redundancy r_{total} minus r_{local} results in the needed remote redundancy r_{remote} to guarantee safer, but slower restorable replicas.

In this evaluation the parameters were chosen as follows:

$$\text{total redundancy: } r_{total} = 10,$$
$$\text{local redundancy: } r_{local} = 3, \text{ and}$$
$$\text{data: } data = 1GB$$

and simulated in Excel. All spreadsheets can be found on the accompanying CD-ROM. If local resources are prioritized, the amount of data to distribute outside the local subnet for a given r_{total} decreases with an increasing number of available local nodes up to the local redundancy limit r_{local}. Figure 4.3 illustrates this correlation and compares with a classic DHT where every node is treated equally resulting in a higher amount of data to be replicated outside the local subnet.

Because local subnets are over-provisioned network zones compared to internet access, as shown in Table 2.4, the resulting time needed for a node to replicate its data to reach r_{total} is considerably shorter in Ubidas. Especially, users that move around with laptop computers, benefit from faster backup times.

4.2.2 Data Restore Time Evaluation

To restore data from a p2p backup system, a node requests replicas from other nodes in the system. In a classic DHT, data is distributed evenly over all participating nodes, because the system is content-addressable and abstracts the underlying network topology into a homogeneous overlay network. Ubidas introduced the prioritization of local resources where available with IPTree. Data replicas in Ubidas are distributed up to a specified number, r_{local}, to local nodes. This enables faster distribution time up to a r_{local} replicas and, therefore, allows laptop computer nodes to move to another network, earlier. On the other hand restores are faster if local replicas are present.

Again, an Excel spreadsheet was used to simulate restore times in both a classic DHT-based p2p system and Ubidas. The file can be found on the accompanying CD-ROM. The restore time

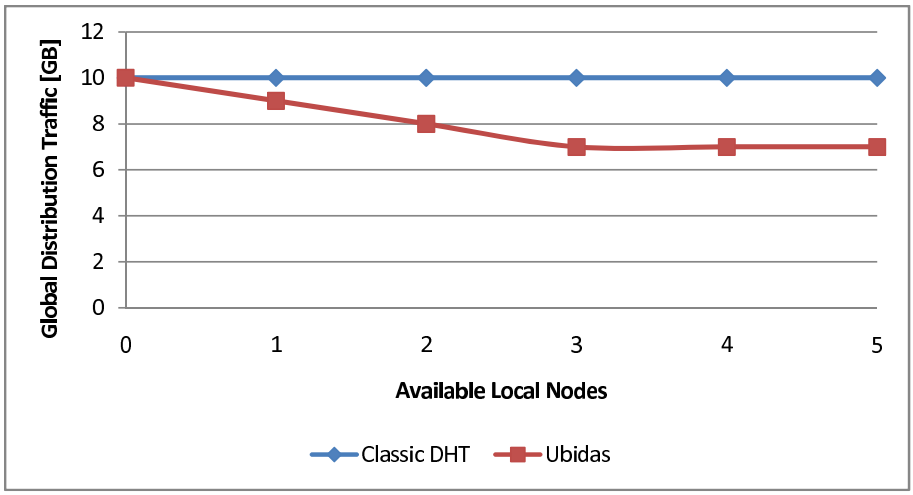

Figure 4.3: Global Data Distribution Evaluation with system parameters $r_{total} = 10$, $r_{local} = 3$, and $data = 1GB$

from nodes on the internet depends on the upload speed these nodes can provide. Traditionally, home and small office nodes are equipped with an asymmetric bandwidth distribution—the upload bandwidth is considerably smaller than the download bandwidth. If data is restored from several nodes at the same time, this constraint is relaxed because individual node upload speeds add up until the download speed of the restoring node is reached.

Figure 4.4 compares restore times in classic DHT-based systems with Ubidas for increasing upload speeds of internet nodes. The chosen simulation parameters were:

$$\text{total redundancy: } r_{total} = 10,$$
$$\text{local redundancy: } r_{local} = 3,$$
$$\text{data: } data = 1GB, \text{ and}$$
$$\text{local speed: } speed_{local} = 100Mbps$$

and yield a faster restore time in Ubidas by one to two orders of magnitude, when local replicas are available.

4.3 Limitations and Evaluation Conclusion

The discussed evaluations in this chapter show that prioritization of available local resources in Ubidas result in substantially faster data replication and restores. On the other hand, a limitation is the lack of related work that perform similar concepts: Ubidas' local resource prioritization

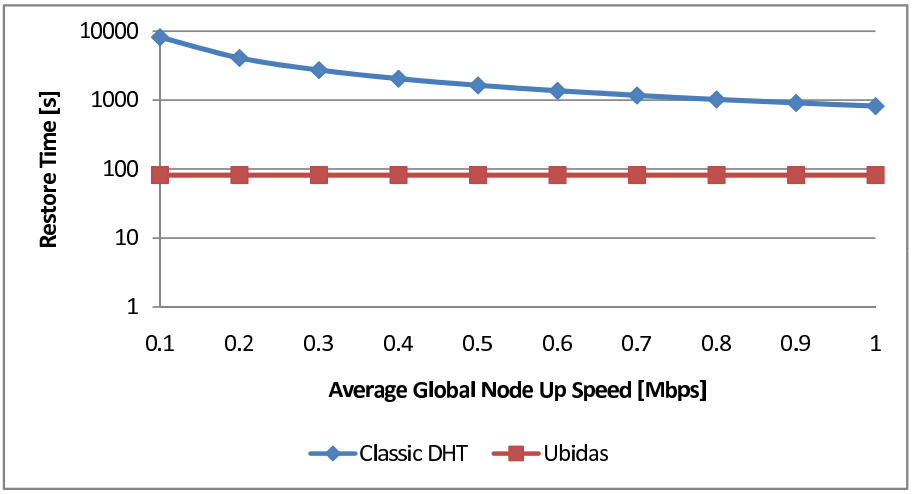

Figure 4.4: Data Restore Time Evaluation with system parameters $r_{total} = 10$, $r_{local} = 3$, $data = 1GB$, and $speed_{local} = 100MBit/s$ Note the logarithmic scale on the vertical axis.

should be introduced to other p2p systems to isolate its huge effect on performance and therefore enable comparison of finer-grained details (e.g., communication overhead).

5

Future Work

The Ubidas design goals were set out to implement a solid communications stack enabling real end-to-end communication, reliable, distributed, hierarchical, and encrypted storage. There were several feature cuts that had to be made in respect to the time available. They are described in detail in the following section. This chapter will close with final remarks and a conceptual outlook.

5.1 Application-Level Transfer Layer

The Transfer Layer was only prototyped and the final version of Ubidas includes the communication stack up to and including the TOC layer. For a complete p2p backup system, a fully functional transfer layer implementation is needed. Ubidas serves all tools to implement this as a future work:

1. **Sentinel:** Ubidas Sentinel delivers real-time events about file system changes for subscribed directories. This can be used to trigger transfers based on user activity on files. Special file types (e.g., for databases) would have to be handled adequately to prevent too frequent update propagation to the Ubidas system. On the other hand a priority queue could be implemented to prioritize most recent changed files to enable quick resource sharing with other participants. Beyond basic file sharing, participating users would also get updates for files they have access to, automatically.

2. **TOC:** The Ubidas TOC allows an abstracted storage layer for any type of hierarchical data. It supports encryption and hierarchical resource sharing.

To complete the implementation of a Transfer layer, intensive testing and performance analysis would have to be conducted to yield optimal overall system performance. An example of such an analysis would be to analyze average file update propagation rate to determine the need and capabilities for real-time backups versus a throttled approach.

5.2 Resource Sharing

Ubidas TOC already includes support for hierarchical resource sharing. A single key can be shared with another participant to allow dedicated read access to a directory and all its children subdirectories and files. Finer-grained support for write access could be introduced in a future version of Ubidas. Also a good program interface should be provided to support a user in using resource sharing in Ubidas.

5.3 Distributed Ubidas Index Server (DUIS)

Currently Ubidas Index Server (UIS) is deployed as a single server in the Ubidas system. It is not a mission-critical component as restores are always possible for a certain amount of time, depending on system size and overall churn rate. To improve performance for a large Ubidas system, multiple DUISs could be introduced. They could operate on a higher trust level to synchronize states and would enable simple system scaling.

5.4 Distributed Public/Private Key Infrastructure

In its current implementation the authentication mechanism in Ubidas can be attacked by collecting large amounts of originary DHT record *put* operations and then replaying the same request later when the original content has updated. This authenticity issue of change requests can be solved by introducing a public/private key infrastructure. Change requests could be digitally signed to prove its origination. A responsible node then verifies the signature by encrypting the signed message hash with the requesting node's public key and compares it against the supplied claim.

To enable a true distributed nature of the system, the public/private key infrastructure should be either based on trusts between nodes and/or a recommendation system.

5.5 Conceptual Outlook

Ubidas is a foundation for future work in the field of p2p backups. With the current trend of growing network bandwidth on the edge of the internet—individual home and office computers—commercial systems will probably arise, further improving the concepts of Ubidas.

Key issues and subjects of future work are:

1. Implemented, tested, and optimized Transfer layer.
2. Application Programming Interface (API) to support third parties in making Ubidas an application platform rather than a standalone backup application.
3. Creation of economic benefit models to trigger voluntary storage behavior of individuals and enterprises.

6

Conclusions and Final Remarks

6.1 Summary

This diploma thesis introduced Ubidas, a reliable, secure, and efficient p2p communication stack for backups. An extensive prototype was developed and tested on a small number of nodes, located in different network zones. Real end-to-end connectivity was achieved with Ubidas End2End. Ubidas DHT introduced IPTree to bring underlying network topology semantics back to an overlay p2p system. Beyond basic messaging in Ubidas DHT, encrypted records can be enabled to store private data on untrusted nodes. Finally, the TOC layer concludes the current implementation of Ubidas, offering hierarchical data structure organization for distributed, safe, and reliable storage.

6.2 Contribution

Ubidas' main contribution is the combination of known p2p concepts and the introduction of IP-Tree, an algorithm to prioritize local resources with $O(log\ n)$ system scaling performance. Further, the Ubidas implementation is an open framework and its usage for systems that build on top of it are strongly encouraged. The complete source code can be found on the accompanying CD-ROM.

6.3 Outlook

While Ubidas offers the whole communication stack for distributed, real end-to-end, reliable, and safe data storage in its implementation, the development of applications on top of Ubidas is left

open for future work. With the current development of ever-growing internet access speeds, bringing power to the edge of the internet will become more and more important. From an economical point of view, parts of expensive central data centers can be substituted for considerably cheaper resources by pushing data from within the cloud to the edge, to the blue-sky, and bright future.

List of Figures

List of Tables

List of Listings

Bibliography

[Batten et al., 2002] Batten, C., Barr, K., Saraf, A., and Trepetin, S. (2002). pStore: A Secure Peer-to-Peer Backup System.

[Bolosky et al., 2000] Bolosky, W. J., Douceur, J. R., Ely, D., and Theimer, M. (2000). Feasibility of a serverless distributed file system deployed on an existing set of desktop PCs. *Measurement and modeling of computer systems*.

[Castro et al., 2002] Castro, M., Druschel, P., Hu, Y. C., and Rowstron, A. (2002). Topology-Aware Routing in Structured Peer-to-Peer Overlay Networks.

[Comito et al., 2006] Comito, C., Patarin, S., and Talia, D. (2006). A Semantic Overlay Network for P2P Schema-Based Data Integration. *Computers and Communications, 2006. ISCC '06. Proceedings. 11th IEEE Symposium on*.

[Correia and Abreu, 2004] Correia, L. and Abreu, A. (2004). Forgetting and Fatigue in Mobile Robot Navigation. *Advances in Artificial Intelligence SBIA 2004*, Volume 3171/2004:434–443.

[Druschel and Rowstron, 2001] Druschel, P. and Rowstron, A. (2001). PAST: A large-scale, persistent peer-to-peer storage utility.

[Landau, 1909] Landau, E. (1909). *Handbuch der Lehre von der Verteilung der Primzahlen*. Teubner, Leipzig.

[Lyman and Varian, 2003] Lyman, P. and Varian, H. R. (2003). How much information? 2003.

[Rowstron, 2001] Rowstron, A. (2001). Pastry: Scalable, decentralized object location and routing for large-scale peer-to-peer systems.

[Stoica et al., 2001] Stoica, I., Morris, R., Liben-Nowell, D., Karger, D. R., Kaashoek, M. F., Dabek, F., and Balakrishnan, H. (2001). Chord: a scalable peer-to-peer lookup protocol for internet applications. *IEEE/ACM Transactions on Networking (TON)*, 11.

[Tridgell and Mackerras, 1996] Tridgell, A. and Mackerras, P. (1996). The rsync algorithm.

[Zhang et al., 2004] Zhang, H., Tan, K.-L., and Landers, M. (2004). PeerStore: better performance by relaxing in peer-to-peer backup. *Peer-to-Peer Computing, 2004. Proceedings. Proceedings. Fourth International Conference on*.