Managing your Content with CIMPLE - A Content-Centric Storage Interface

Thomas Delaet and Wouter Joosen
DistriNet, Department of Computer Science
K.U.Leuven, Belgium
{thomas,wouter}@cs.kuleuven.be

Abstract—A lot of computer usage involves dealing with files like videos, documents, photos. Unfortunately, for most users sharing (large) files, finding files, synchronizing files between devices and backing up files is far from trivial. To address these issues, we present an alternative for traditional filesystems: CIMPLE. CIMPLE provides a content-centric storage interface with integrated search and sharing capabilities. It also relieves the user of backup and synchronization problems. CIMPLE gives each content item a secure ID and understands content metadata.

Index Terms—metadata, content-centric, storage-interface, network file system

I. INTRODUCTION

Most of the actions users perform on a computer involve content: reading mail, watching a movie and working on a document. The amount of content each user owns increases rapidly [1], [2]. Tasks like sharing, finding, synchronizing and backing up content become more difficult when dealing with vast amounts of content. [3] conforms that many users struggle with these content management tasks.

Let us look at each of the content management tasks in more detail.

1) Sharing: Most content you share with other users is for reading. If you want to share a document, you mail it. An alternative is that you make the document available on your personal webspace. In both cases you need to duplicate your document yourself and notify your readers. Once duplicated, there is no relationship between documents anymore.

2) Finding: The search possibilities of operating systems have become more and more advanced. Nevertheless, they still do not satisfy all the user’s needs. How do you find content on other devices like your GPS, mobile phone, laptop, NAS and your children’s computer? How does the operating system know what content is in custom file formats applications use?

3) Synchronizing: Imagine a scenario where you have a desktop and a laptop. In many cases, your laptop hard drive is too small to fit all your content on, so you frequently erase content and copy content from your desktop that you need for offline usage. If you go on a holiday, you want a few movies available. If you go on a business trip, you copy your latest office documents to read on the plane. Your laptop will not know some of its files are duplicates of your desktops’ files. From the moment you copy a file - be it on the same computer or to a different computer - you create separate content items.

4) Backing up: If you use CDs, you quickly end up with a huge pile of discs with your content scattered over them. A USB or network attached hard drive typically fills up within a year. You will buy another one for backing up new content and end up in the same scenario as with CDs: Where did I put this file? Another alternative is to use an online backup service, which solves the above problems but then you need to encrypt all your content and make sure that you do not loose the encryption key.

The core problem is the discrepancy between how a filesystem deals with content and how a user deals with content. A filesystem thinks in terms of files and directories. A user thinks in terms of metadata, e.g. “I’m looking for the document I shared with my family two weeks ago”. Nowadays, synchronization and backup are not part of the filesystem, but are achieved with separate tools. Sharing requires the use of network filesystems or again, separate tools. Finally finding content is difficult since applications implement custom file formats to represent metadata.

Our solution is CIMPLE\textsuperscript{1}: Content Integration Makes your Personal Life Easy. The main contribution of CIMPLE is that it provides a holistic solution to manage a user’s content. CIMPLE hides synchronization and backup tasks from the user and provides an interface to share and find content. Each content item has an ID and is immutable. This ID is used to refer to content in a metadata model. Metadata is stored separately from content and is represented uniformly.

We describe the design and implementation of CIMPLE in sections II and III. Section IV evaluates CIMPLE. In section V, we describe how our content-centric storage interface touches on existing work. We conclude the paper in section VI.

II. DESIGN

CIMPLE is designed as a distributed system with two components: a Storage and Distribution component. The Storage component provides local storage for content and the Distribution component takes care of network communication.

\textsuperscript{1}Hence the name
We start with a description of how content is represented in CIMPLE. Next, we explain CIMPLE’s security properties. Finally, we describe the design of the Storage and Distribution components.

A. Content Representation

CIMPLE does not think in terms of files and directories, instead it represents each content item with an identifier calculated from the content itself. The identifier is the signature of the content item. We use the content creator’s private key to create the signature. Every instance of CIMPLE on a node needs such a private (and public) key.

Such an identifier has two advantages: integrated security and identical content identification between devices. First, because the content ID is the signature of the content it allows to check integrity and authenticity. Second, a unique ID allows metadata to be created and stored separately from content. In a filesystem model, metadata is embedded. Think of ID3 tags for MP3, EXIF data for JPEG, . . . . Identical content identification becomes more difficult with embedded metadata since some of the metadata might be changed. Say you share a photo with your colleague and your colleague associates different labels or a different rating with this photo. If metadata is embedded, it becomes difficult to recognize both photos as identical.

1) Content Attributes: In addition to the Content ID, we define a limited set of attributes that are always associated with a content item. Table I lists these attributes. These attributes are used in CIMPLE to deal with content intelligently. The Protocol Version allows future extensions. Author ID contains a reference to the public key of the content creator. Together with the content ID, the creator’s public key allows integrity and authenticity to be checked. Length is the content length in bytes and Content Type contains an identifier for the content data format. Creation Timestamp denotes when the content was created.

In CIMPLE, content and its attributes are immutable. It is not possible to modify or delete content. Modifying content would invalidate the Content ID since it is a signature generated from the content itself.

2) Content Metadata: In addition to the attributes defined in Table I, CIMPLE allows to define other metadata attributes for each content item, but these are not part of the representation of a content item. We use RDF [5]–[7] to represent other attributes. RDF is a metadata model defined by the World Wide Web Consortium. The idea of RDF is to make statements about resources. An example of an RDF statement is “The title of the song with ID xyz is Suspicious Minds”. RDF statements are represented as subject-predicate-object triplets. In our example, the Content ID “xyz” is the subject. “Title” is the predicate and “Suspicious Minds” is the object. A set of RDF triplets is represented as a content item itself. CIMPLE defines Content Type fields (see Table I) for different RDF syntaxes: N-Triples [8], RDF/XML [7] and Turtle [9].

B. Content Security

CIMPLE enforces integrity, authenticity, confidentiality and trust. CIMPLE verifies integrity and authenticity using the Content ID which is the signature of the content using the private key of its creator. This leaves us with confidentiality and trust.

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>$C$</td>
<td>Unencrypted Content Item</td>
</tr>
<tr>
<td>$S$</td>
<td>Symmetric Key</td>
</tr>
<tr>
<td>$E$</td>
<td>Entity. An Entity can be a user, program, . . . . In general: anything that can create content.</td>
</tr>
<tr>
<td>PUB($E$)</td>
<td>Public key of entity $E$</td>
</tr>
<tr>
<td>PRIV($E$)</td>
<td>Private key of entity $E$</td>
</tr>
<tr>
<td>ENC($C$, K)</td>
<td>Encrypted version of $C$. $C$ is encrypted with Key $K$ which can be a public key or symmetric key.</td>
</tr>
<tr>
<td>ID($C$, PRIV($E$))</td>
<td>Content ID of $C$ using private key of $E$.</td>
</tr>
</tbody>
</table>

Table II
SYMBOLS USED BY CIMPLE TO ACHIEVE CONFIDENTIALITY. SYMBOLS WITH MULTIPLE INSTANCES HAVE A NUMBER APPENDED IN SUBSCRIPT.

1) Confidentiality: We use symmetric keys to encrypt content items. The symmetric keys are content items. They are encrypted with the public keys of the identities allowed to see the original content items. The link between a content item and its encryption key is represented with RDF metadata.

Let us describe the method more rigorously. We first introduce a set of symbols in Table II. Say $E_1$ wants to encrypt $C$ and wants to make $C$ available to his friend $E_2$.

- $E_1$ generates a symmetric key $S$.
- $E_1$ encrypts $C$ with $S$. Therefore, he generates $\text{ENC}(C, S)$ and $\text{ID}($ENC($C$, S), PRIV($E_1$)). Respectively the encrypted version of $C$ and the ID of the encrypted version of $C$.
- $E_1$ generates $\text{ENC}(S, $PUB($E_2$))$ and $\text{ID}($ENC($S$, $PUB($E_2$)), PRIV($E_1$)). Respectively $S$ encrypted with the public key of his friend and an ID for the encrypted version of $S$.
- $E_1$ generates an RDF triplet with $\text{ID}($ENC($C$, S), PRIV($E_1$))$ as subject. $\text{cimple : encryptedWith}$ as predicate and $\text{ID}($ENC($S$, $PUB($E_2$)), PRIV($E_1$))$ as object. This triplet states the relationship between the encrypted version of $C$ and the key $S$ used to encrypt $C$.
- $E_1$ generates an RDF triplet with $\text{ID}($ENC($S$, $PUB($E_2$)), PRIV($E_1$))$ as subject. $\text{cimple : encryptedWith}$ as predicate and $\text{ID}($PUB($E_2$)), PRIV($E_2$))$ as object. This triplet states the relationship between $S$ and the public key of $E_2$ used to encrypt $S$.

The last three steps are repeated for every entity that is allowed to decrypt $C$.

2) Trust: Every entity in our system needs a public/private key pair to create content. To make other entities aware of its existence, an entity distributes its public key as a content item. The Content ID of an entity’s public key is the value of the Author ID attribute for content items created by that entity.
Table I

<table>
<thead>
<tr>
<th>Name</th>
<th>Length</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Protocol Version</td>
<td>1 Byte</td>
<td>Unsigned Integer</td>
<td>Currently always “T”. This attribute allows future changes like other schemes to generate the content ID.</td>
</tr>
<tr>
<td>Content ID</td>
<td>64 Bytes</td>
<td>Byte array</td>
<td>Signature of the content’s data using the creator’s private key. Public keys are also content items and are needed by recipients to check authenticity of content. The Author field contains the Content ID of the creator’s public key.</td>
</tr>
<tr>
<td>Author ID</td>
<td>64 Bytes</td>
<td>Byte array</td>
<td>The number of bytes in the content item. A number indicating the data format of the content’s body. The last bit of the content type denotes whether it is encrypted or not.</td>
</tr>
<tr>
<td>Content Type</td>
<td>4 Bytes</td>
<td>Unsigned Integer</td>
<td>The actual content.</td>
</tr>
<tr>
<td>Creation Timestamp</td>
<td>8 Bytes</td>
<td>“Length” Bytes</td>
<td>The actual content.</td>
</tr>
</tbody>
</table>

Table I
CONTENT ATTRIBUTES. OTHER ATTRIBUTES ARE DEFINED IN RDF AND ARE REPRESENTED AS CONTENT THEMSELVES.

Entities store public keys and use them to check the integrity and authenticity of content items.

When an entity \( E_1 \) wants to express its trust in another entity \( E_2 \), \( E_1 \) distributes the public key of \( E_2 \) but generates the content ID with its own private key. Trust relationships are inferred using transitivity [10]. Expressing trust itself is something that is left to applications using CIMPLE.

C. Content Storage

The Storage component is responsible for storing content and metadata on a node. It also deals with encryption and decryption of content. Listing 1 describes the interface of the Storage component in pseudo code. The interface contains methods to create content, create metadata, search metadata, share content and request content by its ID.

```plaintext
class StorageInterface:
    def create_content(path, type, encrypted):
        # add content
        # path contains current on-disk location
        # encrypted is a boolean
        # returns content ID

    def create_metadata(s, p, o, encrypted):
        # insert new metadata
        # encrypted is a boolean

    def search(query):
        # searches metadata
        # returns results

    def request(content_id):
        # requests content by ID
        # returns on-disk location

    def share(content_id, author):
        # share content with another identity
```

Listing 1. Interface for Storage Component.

- **create content**: The `path` is the on-disk location where the application that uses the Storage interface temporarily stores a content item before it is imported in CIMPLE. The `type` argument contains the content type and the `encrypted` argument is a boolean that expresses whether this content item is public or private.
- **create metadata**: Remember that metadata is represented using RDF subject-predicate-object triplets. The subject `s`, predicate `p` and object `o` are the first three arguments of this method. The `encrypted` argument is a boolean that expresses whether this metadata triplet is public or private.
- **search**: To search content, we (re)use existing RDF query languages. The `query` argument contains a metadata query. SPARQL [11] is one example of a language for querying RDF metadata. Listing 2 contains an example SPARQL query. In human language, the query expresses: “Give me the documents I shared with my family two weeks ago”. A user of CIMPLE will never be directly exposed to SPARQL. It is part of the API that applications of filesystems built on top of CIMPLE use.
- **request**: To request content, an application needs a content ID. Queries are used to find Content IDs. For example, the query in listing 2 returns a list of content IDs.
- **share**: To share content, your application needs to know the content IDs of the public keys owned by the users with whom you want to share content. Again, these content IDs can be obtained with a query.

D. Content Distribution

The Distribution component takes care of network communication. We limit our description to its interface. The Storage component uses the interface of the Distribution component. Listing 3 contains the methods of the Distribution component. The Distribution component allows to request content, query content and distribute content.

- **request**: Request content from the network by its ID. The difference with the request method of the Storage component is that the request method of the Distribution component requests content from the network and does not deal with decryption. In contrast, an implementation of the Storage component will first check if the content
is not available locally. If this is not the case, the Storage component will request the required content using the request method in the Distribution component. The Storage component also looks up the encryption key and decrypts the content item before providing content to the application.

- **query**: Query the network for content. The expressivity of network queries is limited because allowing arbitrary complex queries would impose too much load on other nodes. A content query can contain up to four attributes: author, type, creation and key. Each of these attributes can be left blank to indicate a wildcard query. If none of the attributes are left blank, a network query expresses something like “Give me all content items of type x, created by author y after creation timestamp z that are encrypted with key k.” Key denotes the content ID of symmetric encryption key or public key.

- **distribute**: Make content available on the network. The attributes of the distribute method conform to the attributes of table I. The path contains the temporary storage location of the content itself. The Length attribute from table I is not needed since it can be derived from the path attribute.

- **subscribe**: subscribe to new content on the network using the same format as the query method and provide a callback method.

### III. IMPLEMENTATION

Our implementation of CIMPLE’s Storage and Distribution components is about 4000 lines of Python code. In this section, we first describe the implementation characteristics of these two components. Next, we describe how we implemented CIMPLE’s security properties and the implementation of interfaces exposed by the Storage and Distribution components. We end with a discussion of how users interact with CIMPLE.

#### A. Content Storage

We distinguish three types of content: metadata content, trust content and all other content.

All metadata, including the attributes from table I is stored in an RDF database. We use the Redland libraries [12] with a Berkeley DB backend for RDF storage. The Storage component tries to aggregate all metadata that is available to it. Therefore, it uses the subscribe method and periodically executes the query method from the Distribution component.

Remember that public keys are used to verify trust, integrity and authenticity of content items. Trust content and public keys are aggregated with the subscribe and query methods from the Distribution component. The Storage component also tries to retrieve trust content if a content item contains an unknown or untrusted author.

All other content is stored as a blob in the local filesystem. Since many devices (laptops, mobile phones, ...) have not enough disk space to store every content item that arrives on a node over time, we implemented a Least Recently Used (LRU) policy to manage local storage. As a consequence, local storage of content is not persistent but a cache. By default we assume that all content is managed with CIMPLE and use all available space for storage before discarding content. The amount of disk space reserved for the CIMPLE cache is a parameter that can be configured. Persistent storage in the prototype with a storage software component.

The storage software is enabled on storage nodes and allows storage nodes to cooperatively realize the default policy “Store all my content in two different locations”, effectively integrating backup and persistent storage. When a storage node fails, other nodes cooperatively rebuild it by copying its content from storage nodes that have stored the same content.

Our prototype implements the Storage interface from listing 1 as follows.

- **create content**: Calculate Content ID, add it to the local cache and distribute it through the Distribution component.

- **create metadata**: Add metadata to RDF database. Since there is an overhead associated with creating content - the attribute fields of table I - it is not efficient to create a new content item for every RDF triplet that is created. Instead, we choose to accumulate metadata over a fixed time interval (two minutes). The accumulated metadata...
is then represented as one content item. The implicit assumption is that metadata is created in bulk. This is not unreasonable: importing videos or photos, or a user that starts tagging photos are examples where metadata is created in a short time interval.

- **search**: Query the RDF database. We support SPARQL [11] as a query language.
- **request**: First the local cache is checked. If content it not stored in the cache, it is fetched from the Distribution component. If it is encrypted, its key is also fetched from the Distribution component. The Storage component decrypts the content, stores it in the cache and makes it available to the application requesting the content.
- **share**: Encrypt symmetric key with public key of recipient and create metadata as described in section II-B1. Subsequently, distribute encrypted symmetric key through the Distribution component.

**B. Content Distribution**

The protocol we use to transport content to other nodes contains one message type: a content item. The header fields of this message type are the attributes we described in table I plus three extra attributes: **Start Byte**, **Body length** and **Time To Live**. Start Byte and Body length allow to fragment content items and Time To Live is used for content routing. The body of a content message is the actual content. We use TCP to do the actual content transfer. Nodes are discovered using DNS-SD/mDNS [13].

Our prototype implements the Distribution interface from listing 3 as follows.

- **request**: A request is modeled as a content item itself. Its body contains the ID of the requested content item. Once created, a request content item is distributed (broadcasted) in the network.
- **query**: A query is also modeled as a content item itself. Its body contains the query. It is distributed and answered using the same mechanism as a request.
- **distribute**: A content item that needs to be distributed to subscribers in the network.
- **subscribe**: Content that arrives on one of a node’s transport mediums is temporarily stored by the Distribution component and reassembled if necessary. Subscribers are notified of new content using the callback from listing 3.

**C. Security Mechanisms**

To keep content ID size small, we choose to use Elliptic Curve Cryptography (ECC) [14], [15] for public/private keys. ECC keys of 256 bit generate content IDs of 64 bytes. An ECC key length of 256 bits is used by the NSA [16]. The equivalent 3072 bit RSA keys [17] generate content IDs of 384 bytes. ECC operations are handled by the secure toolset [18].

AES-256 bit keys are used for symmetric key encryption and decryption. The OpenSSL [19] library handles symmetric key operations in our prototype.

**D. Interfaces**

Both the Storage and Distribution components expose their interface over D-BUS [20]. D-BUS is a bus system used in desktop environments like Gnome and KDE. Since they make use of a user’s private key, both the Storage and Distribution components run as a per-user service on each node. The logic for transferring content on the network (transport mediums) is modeled as a system-wide process.

**E. User Interface**

Although applications can use the CIMPLE API directly, they need to be redesigned to do this. To deal with existing applications, we implemented CIMPLE as a FUSE [21] filesystem that automatically takes care of synchronizing and backing up your content. The sharing capabilities are statically configured: every FUSE mountpoint needs an entry in the CIMPLE configuration file that states with whom the mount point is shared.

**IV. Evaluation**

We evaluate two aspects of CIMPLE. First, we measure two elements of our prototype: network and storage overhead. Second, we discuss CIMPLE’s features and limitations.

**A. Quantitative Evaluation**

In this quantitative evaluation, we answer two questions empirically. First: What is CIMPLE’s network overhead? Second: What is CIMPLE’s local storage overhead? Our test network consists of four identical machines connected to a Cisco Catalyst 2960 switch. The machines have an Intel Pentium III 500Mhz processor and 256M RAM. All machines have one 3Com 3c905C network card. Network connections are 100Mbps full-duplex. The machines run OpenBSD 4.4 [22].

To measure network and local storage overhead, we use three sets of files: a small set, a mixed set and a large set. The small set contains only small files. The large set contains only large files and the mixed set contains a representative mix of small, medium and large files. The total size of all sets is the same: 1GB. Small files are files with size $[0, 10K]$, medium files are files with size $[10KB, 10MB]$ and large files with size $[10MB, \infty]$. The small set contains only files of 4KB. The large set contains 2 files of 500MB and the mixed set contains 35% files of 5KB, 60% files of 5MB and 5% files of 50MB. The distribution of the mixed set is based on file size distribution measurements of 17 home directories of non-technical computer users. We assume all content in the set to be private. Remember that private content is encrypted with a symmetric key in CIMPLE which is a content item on its own.

1) **Network Overhead**: We compare the overhead of CIMPLE’s TCP network transport with TCP + SSL and SSH. Remember that CIMPLE uses 256 bit AES keys for symmetric cryptography and 256 bit Elliptic Curve Cryptography (ECC) keys. SSL and SSH also use 256 bit AES keys. Since SSL and SSH do not support ECC, we use the equivalent RSA keys (3072 bits). We also disable compression for all protocols.
Table III compares CIMPLE’s TCP network transport and TCP transport mediums with TCP + SSL and SSH for each of the three content sets.

One trend is that smaller content items cause more overhead. This overhead is influenced by two factors: connection setup and per-item overhead in CIMPLE.

1) **connection setup**: All TCP-based protocols have a connection setup phase. In the SSL and SSH protocols, the negotiation of security parameters is part of the connection setup phase. If the actual data transfer is small, the cost of connection setup is relatively high which causes the high overhead as a percentage of the actual content size.

2) **per-item overhead**: Every content item results in the creation of two additional content items in CIMPLE: a symmetric key used for encryption and a metadata content item that links a content item with its encryption key. Again, since these two extra content items are constant, they have a larger influence on the overhead for small content items.

The main difference between CIMPLE and the two secure protocols (SSL and SSH) on the other side is what they secure: CIMPLE secures the content, SSL and SSH secure the channel and then send the content through the secure channel.

For small files, the CIMPLE’s TCP transport medium performs worse than one of its competitors: SSL. The explanation is that CIMPLE’s TCP needs to deal with a double overhead for small files: the content header and TCP connection setup. For larger file sizes, the relative overhead of the content header quickly diminishes. This causes CIMPLE to outperform SSL and SSH for larger content sizes.

2) **Storage Overhead**: A stored (private) content item has four components: the content item itself, its header data, the symmetric key and the header data of the symmetric key. Our prototype stores all header data in an RDF database based on Berkeley DB, a hash database. Content items and symmetric keys are stored as files. Table IV contains the storage overhead for each of our three content sets.

One thing that table IV learns us is that storage overhead depends more on the number of content items than on their size. Every stored content item has a fixed header and a fixed symmetric key which is also a content item. As a result, storing a lot of small items yields a larger overhead. However, the results from table IV are excessive: almost 300% overhead for storing small content items. In theory, the overhead is much smaller. The small set contains content items of 4kb and has 362 bytes overhead for each content item: 64 bytes for the symmetric key, 149 bytes for the header of the symmetric key and 149 bytes for the header of the content item. As a result, the small set should have an overhead of 8.84%.

The main source of storage overhead is implementation related. The RDF database we use (Redland) is backed by Berkeley DB, a hash-based persistent database. The RDF database needs to access RDF triplets by each of the components (subject, predicate, object). Therefore, indexes are created. The storage cost of these indexes is what causes the high overhead for storing small content items.

### B. Limitations

CIMPLE takes care of backup and synchronization and provides an interface for content sharing and search. One limitation of CIMPLE is that metadata queries can only be as powerful as the metadata inserted in CIMPLE by applications.

The implementation of the Storage component needs to use a different database backend to deal more efficiently with small files. In the implementation of the Distribution component, we need to have a more intelligent routing algorithm. Broadcasting content works for a small home network but does not scale well.

### V. RELATED WORK

CIMPLE’s related work consists of projects and products that provide smarter content management to users. We distinguish three types of projects in this space: Content-Centric Networking, Vendor Solutions and Networked Storage.

#### A. Content-Centric Networking

The main inspiration of CIMPLE is the Content-Centric Networking (CCN) project from Parc Laboratories [23], headed by Van Jacobson. The goal of CCN is to design a replacement for the ubiquitous IPv4 protocol. While IPv4 deals with packets exchanged between a source and destination, CCN deals with content items. CCN is designed to be an overlay on IPv4, but is not tied to this protocol.

The CIMPLE and CCN projects are complementary. CIMPLE provides an alternative for a traditional filesystem with more powerful semantics and built-in network capabilities. CCN on the other hand is meant as an Internet-scale protocol. Since both CIMPLE and CCN use content as their building block, it should be possible to adapt the network backend of CIMPLE to use CCN so that it scales to Internet-size systems. We are not able to demonstrate this integration since, to the best of our knowledge, there are no concrete results from CCN available.
B. Vendor Solutions

Large companies have the resources and critical mass to deal with content management tasks by integrating applications. However, in many cases their primary goal is often vendor lock-in instead of a more pleasant user experience. Two companies that take the approach of integrating applications are Apple and Google.

Apple is historically known for bundling software and hardware. In recent years, Apple added services like .Mac and MobileMe to this bundle. Together with the addition of mobile devices like the iPod and iPhone, Apple gives customers a complete solution for their digital live. Sharing of content is possible because applications are linked with its online services. For example, users can put photo albums online from the iPhoto. To find content, Apple developed Spotlight as a search feature. Spotlight is not able to search inside file formats like the iTunes and iPhoto library, missing most available metadata. The most recent version of Spotlight does allow to search content on other devices running the same software. Synchronizing content is possible through the MobileMe service. MobileMe allows to sync content between your desktops, laptops and mobile devices. It has excellent support for contact information, mails and calendars but is not developed to share larger files like movies. Time Machine is Apple’s backup solution. It is tightly integrated with the Operating system and its applications. One disadvantage is that it is limited to one network disk that can be used as backup medium.

Google takes a different approach. Its focus is on online applications. Applications like Google Calendar, Google Docs and Google Notebook provide integrated sharing facilities. Sharing is as simple as entering a recipients mail address. Shared items are marked in the user interface. Every application provides the well-known search box and Google Desktop Search integrates searching of your local hard disk. Synchronization and backup is not necessary since all content is available online. The disadvantage of an online approach is that offline operation is not well supported. Google Gears offers support for offline operation, but not all applications use it.

C. Networked Storage

There exists a huge body of research on networked filesystems, cooperative content distribution, replica maintenance and using inexpensive NAS devices. We will not discuss this research as part of our related work since it provides little added value for the content management tasks as discussed in this paper (sharing, finding, syncing and backing up of content). Instead, we discuss projects that provide added-value over traditional network storage of files by dealing with one or more content management task(s). First we discuss content-addressable storage projects. Second we look at a high-quality peer to peer secure content sharing system. Finally, we discuss mobile storage systems.

1) Content-addressable storage: Content Addressable Storage (CAS) allows to retrieve content based on an ID, not its storage location. As is the case in CIMPLE, the ID is based on the content itself. In CAS systems, the ID is typically a cryptographic checksum of the content itself which allows to check content integrity. Since CIMPLE uses a signature as ID, it allows not only to check integrity but also authenticity.

EMC’s Centera [24] and Caringo’s CASTor [25] are both examples of commercial CAS systems. Venti [26] and Twisted Storage [27] are open-source CAS systems. All of these systems focus on backing up content for long term storage. They have little or no support for other content management tasks like sharing, synchronizing and searching content. Another common theme is their focus on enterprises.

Foundation [28] is a research system that can best be described as “Venti for consumers”. Foundation snapshots your local hard drive every night. Based on these snapshots, you can recover your data from any point in time together with the applications necessary to view or modify that data. To achieve this, the snapshots are virtual machine images and the software running on your computer is a virtual machine monitor on top of the base operating system that can switch images. Foundation uses a single USB drive for storage and the possibility to replicate content on remote storage through FTP. Content is encrypted before it is sent to the USB disk or FTP server. Foundation provides an original solution to the problem of not only recovering old files, but also being able to read them. As the authors themselves discuss in [28] its usage of a single USB disk is a limitation for large content collections. Foundation does not deal with content search, sharing and synchronization.

2) P2P Secure content-sharing: Wuala [29] is an online service and desktop application to securely store and share files online. A user can choose to buy online storage space or trade local disk space for online storage, effectively creating a distributed filesystem. Content stored in Wuala is encrypted with CryptTree [30] using AES-128 and RSA-2048 bit cryptography. Content is encrypted locally and a user can choose with whom he shares content. Wuala’s API is not public, making it difficult to integrate with other systems.

3) Mobile storage: Mobile storage systems deal with a user’s personal storage across a number of weak-connected devices like phones and laptops together with well-connected devices like desktops and media centers. Three of such state-of-the-art systems are Segank [31], PersonalRAID [32] and Perspective [3].

Both Segank and PersonalRAID use a small device that contains a log of filesystem operations. A user needs to carry this device with him to replay filesystem operations. Both systems provide a consistent namespace to the user. The CIMPLE equivalent of a consistent namespace are content items addressed by their identifier. PersonalRAID is designed for the completely disconnected operations scenario while Segank is designed for weakly connected devices. As a consequence, PersonalRAID replicates more data on each device. Since these systems are designed as filesystems, they do not deal with search. They do allow to synchronize content and offer limited sharing capabilities. Backup is something that can be
built on top of these systems. None of these systems secure content and they do not deal with the fact that most devices provide smaller drives than the total amount of your personal content. CIMPLE deals with this by using an LRU cache. Another major difference with CIMPLE is that both systems allow data modification and deletion. CIMPLE models data modification and deletion as data creation operations (see section II-C). The data modification and deletion features are responsible for most of the network overhead and data structures in Segank and PersonalRAID.

Perspective uses the concept of views. A view can best be compared to a metadata query in CIMPLE. Every device expresses what content it wants to store. Views are exchanged across devices as part of a publish/subscribe system. A device that contains content checks the views of all devices and sends content if they expressed interest. Perspective provides an interface for metadata search, deals with multiple (mobile) devices, allows to synchronize and share content and provides backup. The analysis of Perspective’s authors about personal content management is identical to ours and their solution is comparable. However, it is difficult to assess Perspective since there is very little information available about its operation. One major difference is the security of both systems: Perspective does not provide any of the security properties CIMPLE provides (see section II-B), making content management tasks like sharing easier to implement.

Our tour of related work shows that solutions exist for each of the content management tasks we mentioned in the Introduction (backup, synchronization, sharing and search). However, none of the existing solutions provides an integrated interface to deal with these tasks in a secure way. We believe this is CIMPLE’s main contribution.

VI. CONCLUSION

CIMPLE provides a content-centric storage interface as an alternative for traditional filesystems. It provides an interface to search content and share content and it takes care of backup and synchronization. CIMPLE represents content uniformly, regardless of whether it is on the network or on your local disk. It associates an ID and a number of other attributes with each content item.

The potential goes even further. A uniform content model and network communication model allows to integrate all your devices. Imagine a world where your wake-up device is programmed automatically because it can read your calendar, where the movies of your in-car computer system, media center(s) and computers are in sync, where your digital photo frames can access your entire photo collection, where your phone system uses the same contacts as your mail program, chat program and mobile phone, …

ACKNOWLEDGEMENTS

This research has been supported by a grant from the Institute for the Promotion of Innovation by Science and Technology in Flanders (http://www.iwt.be). We would like to thank Bart Vanbrabant, Sara Vermeulen and Frans Sanen for proofreading this paper.

REFERENCES