Advanced pWeb Features

Md. Faizul Bari, Shihabur Rahman Chowdhury, Alexander Pokluda,
Reaz Ahmed, and Raouf Boutaba

David R. Cheriton School of Computer Science
University of Waterloo

[mfbari | sr2chowdhury | apokluda | r5ahmed | rboutaba]@uwaterloo.ca

University of Waterloo Technical Report: CS-2013-13*

*This report is a modified version of an internal technical report produced for Orange Labs, the project's sponsor. The original technical report was submitted to Orange Labs in June 2012.

Abstract

This technical report describes five crucial components of pWeb. First, we give a review of the pWeb system architecture in Part I; second, we explain the architecture and design components of our file system in Part II; third, we describe how peers host and access dynamic content using light-weight HTTP servers and client-side scripts in Part III; forth, we present XML data management and access capabilities in Part IV; and finally, we conclude by providing a demo of our client software implementation in Part V.

Contents

Part I: System Architecture **7**

1 System Architecture **8**

1.1 Advertisement Process 10

1.2 Peer Join and Group Maintenance Process 11

1.3 Query Process 13

Part II: File System **14**

1 Introduction **15**

2 Requirements for the File System **15**

3 An Overview of Existing P2P File Systems **16**

3.1 CFS 17

3.2 Freenet 17

3.3 OceanStore 18

3.4 IVY 18

3.5 BitTorrent 18

3.6 Discussion 18

4 File System for pWeb **19**

4.1 Overview 19

4.2 Naming and Directory Structure 19

4.3 Object Structure 21

4.4 File System Details 22

4.4.1 Disk Layout 22

4.4.2 File System APIs 23

5 Summary **24**

Part III: Server-side Scripting Enabling Dynamic Page Generation **25**

1 Introduction **26**

2 Requirements of a HTTP Server for P2P Web Hosting **26**

3 Alternative Deployment Choices **26**

3.1 Mozilla Firefox: LiveConnect . . . 27
3.2 NPAPI plugin . . . 27
3.3 Internet Explorer: ActiveX . . . 28
3.4 Internet Explorer: `BrowserHelperObjects` (BHOs) . . . 28
3.5 Internet Explorer: Plugins . . . 28
3.6 Standalone HTTP Servers . . . 29
3.7 Java-Based Ajax Servers . . . 29
3.8 Summary . . . 29

4 Light-Weight HTTP Servers **31**

4.1 Cherokee . . . 31
4.2 Hiawatha . . . 31
4.3 lighttpd . . . 31
4.4 nginx . . . 32
4.5 HFS (HTTP File Server) . . . 32
4.6 Comparison of Lightweight HTTP Servers . . . 32

5 PHP and Java: Integration Techniques **32**

5.1 Separate Java HTTP Server . . . 33
5.2 Java Webservice . . . 35
5.3 CGI . . . 36
5.4 Example: Java Function Call . . . 37
5.5 Conclusion . . . 38

6 Summary **38**

Part IV: XML Data Management and Access Capability **39**

1 Introduction **40**

2 Native XML Database **41**

3 Native XML Database versus Flat-file XML Database **43**

4 General XML Schema for pWeb Documents **44**

Part V: Client Software Implementation **45**

1 Introduction **46**

2 System Architecture **46**

2.1 Server Side PHP Components 46

2.2 Client Side Java Components 46

3 First Database Architecture **47**

4 Application Features **49**

4.1 User management system 50

4.2 Upload Content 51

4.3 Database Lookup or Video Search 51

4.4 Content Access and Video Streaming 52

4.5 Group Management System 52

5 Index Server Management **54**

5.1 Patterns and binary tree 54

5.2 Routing Table 56

5.3 Upload 57

5.4 Search 57

6 Java Component **57**

7 Choice of Technology **59**

8 Installation Manual **60**

9 Summary **61**

Part I:

System Architecture

1 System Architecture

In this section we present the overall architecture of pWeb system so the reader may have a better understanding of the subsequent parts of this report. The components of the pWeb hosting architecture and the interactions between them are shown in Figure 1 and Figure 2, respectively. At the core of this architecture resides Plexus routing [4] and all other components are built around it. Figure 1 shows the components that make it possible to realize a web hosting framework on dynamic P2P network and Figure 2 shows the interaction between these components. The arrows in Figure 2 depict the "uses" relationship between the components. From Figure 2 we can see that the Group Management component uses four other components, namely: P2P Naming system [5, 7], Plexus Routing [4], Dynamic Page Parser, and Data Storage. Similarly the Indexing & Publishing and Query Processing [1, 3] components use the Group Management component. A brief outline of the functionality of these components is presented below.

1. **P2P Naming System:** This component uses the Plexus routing mechanism to publish website names to the pWeb network and resolve website names to alive peers' IP:Port pairs in the network.

2. **Group Management:** The purpose of this component is to keep track of the uptime history for each peer in a distributed manner, which is used to construct replication groups. This component also keeps track of the existing peer groups and provides a mechanism for mapping peer groups to an alive peer in that group at any given time. This component uses the Plexus routing, P2P naming system and data store components for performing these tasks.

3. **Indexing & Publishing:** This component provides a user interface for publishing websites to the pWeb network. The task of publishing includes publishing the website name using the P2P naming system, advertising indices using the Plexus routing protocol, and replicating websites within a replication group using the group management component. This component collaborates with every other component within a peer and accesses the data storage component.

4. **Query Processing:** The query processing component converts query keywords to a list of codewords and then forwards the codewords to the peers responsible for them using the Plexus routing mechanism. This component uses the P2P naming system to get the names of the websites where any matching content is found. It

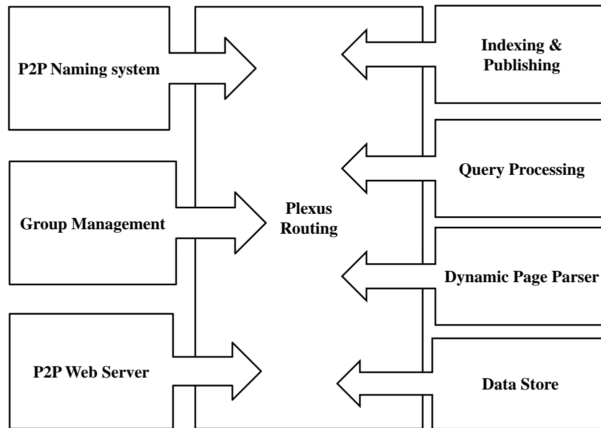

Figure 1: Components of pWeb hosting architecture

also interacts with the group management component and data storage component for finding an alive peer and searching local repository, respectively.

5. **Data Store:** This component allocates some space from a peer's secondary storage for storing indices, web pages, images, scripts, Flash content etc. For enabling light-weight database support, an XML-based file management API is also offered by this component.

6. **Dynamic Page Parser:** This plugable component enables hosting of dynamic web pages and conforms to the interface exported by the P2P web server. Developers can build various implementations of this component enabling different server-side scripting languages for pWeb hosting.

7. **P2P Web Server:** Each peer runs a local web server that will handle all P2P Web requests. This web server operates in three modes: a) browsing, b) hosting, and c) searching. In *browsing mode*, the web server resolves a web page name to target IP:port pair using the P2P naming system and downloads the document from the target peer's web server. If a web page contains dynamic content then the hosting peer's web server uses the dynamic page parser component to generate the page locally. In *hosting mode*, a web server responds to the download requests from other web servers by providing web pages. Finally, in *searching mode* the web server uses the query processing component to resolve a query.

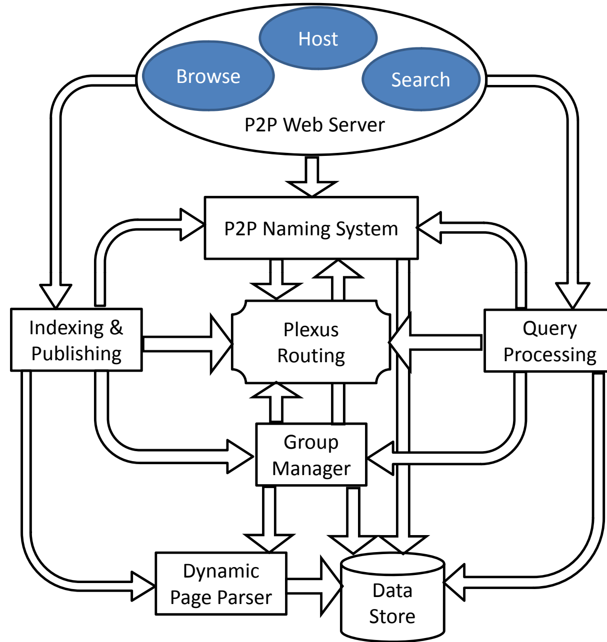

Figure 2: Interaction between the components of pWeb hosting architecture

Our P2P web hosting stack is built on top of a Plexus routing and indexing framework. Though in Plexus [4], we used Golay Codes for routing, here we use the Plexus variant using Reed-Muller Codes introduced in [23]. Using Reed-Muller codes, Plexus can scale to millions of peers without using subnets or super-peers. This way the overlay network can have a flat and symmetric structure. On top of this overlay network, replication groups are formed based on uptime history of peers. The functional dependencies between the architectural components have a direct impact on the naming scheme and vice versa. The three crucial processes within the naming component–Advertisement, Peer Join and Group Maintenance, and Query–are presented below.

1.1 Advertisement Process

To facilitate searching web sites efficiently, we use a distributed Hamming distance based indexing mechanism on top of Plexus routing. As depicted in Figure 3, the advertisement process consists of the following four steps:

- Step a: Each peer in the system will belong to a replication group. Suppose peer X belongs to group G and wants to advertise site S. The search keywords (or other meta information) related to site S are r, s and t. Peer X sends this information

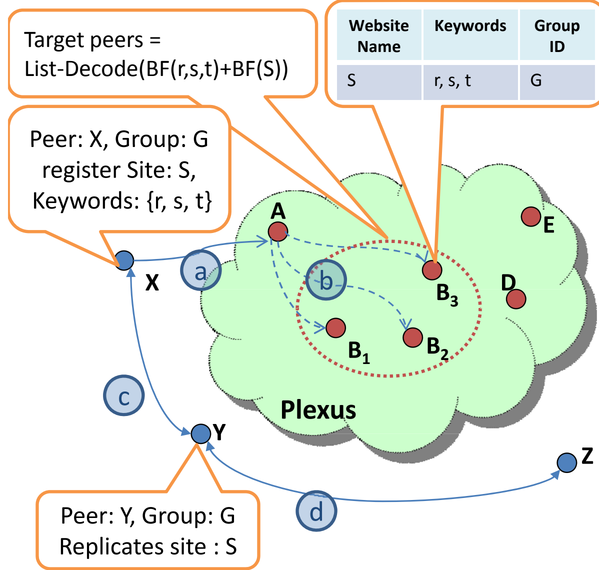

Figure 3: Advertisement of a Site

to a peer A in the Plexus indexing framework.

- Step b: In this step, peer A creates two advertisement patterns (basically Bloom Filters). One from the meta information (*i.e.*, , r, s and t) and the other from the website's name. Then peer A list decodes the patterns and computes the set of codewords within a pre-specified Hamming distance from the advertised patterns. Then it uses the Plexus routing mechanism to multicast the site index to the peers (B_1, B_2, B_3) responsible for the computed codewords. The indices stored in the indexing peers (*i.e.*, B_i's) are in the form of <Website Name, Keywords, Group ID> triplets.

- Step c and d: Newly hosted sites or updates to existing sites are propagated to all the members (*i.e.*, peers Y and Z) of the hosting peer's (*i.e.*, X's) replication group (*i.e.*, group G). This update propagation (if any) takes place whenever a group member rejoins the network. More detail on the peer join and replication process is given next.

1.2 Peer Join and Group Maintenance Process

In order to maintain diurnal availability, peers collaborate in small groups in such a way that at any given instance in time at least one peer from a group is online with very high

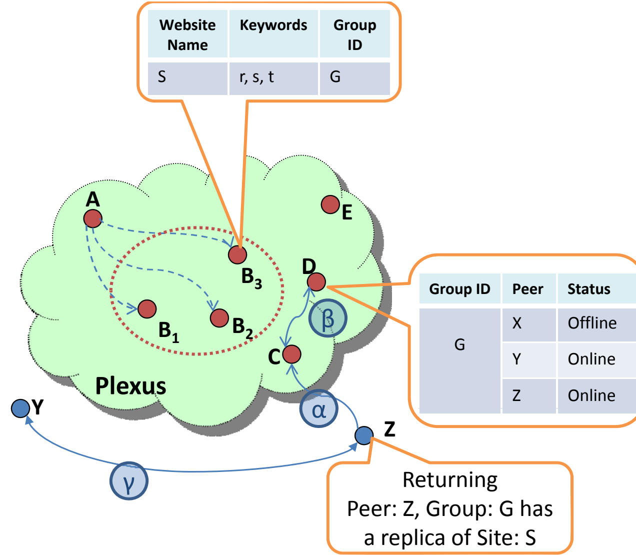

Figure 4: Accommodating new/returning peers

probability. Content in each peer within a group is fully replicated and synchronized. To ensure replica consistency, whenever a peer joins or returns to the system it finds its group's active members, updates its online status and synchronizes the replicated contents in the following manner:

- Step α: As depicted in Figure 4, peer Z, a member of group G, becomes online after being offline for a while. Once online, peer Z requests the address of a peer, say C, in the Plexus indexing framework to find the members of its own group G. If Z is joining the network for the first time, its replication group is determined based on its uptime history.

- Step β: Peer C, constructs a pattern (Bloom Filter) from the group ID G, decodes the pattern to find the closest codeword, and routes the query to the responsible peer (here D) using Plexus routing. Upon receiving the query, peer D updates the current status of peer Z to online, records its network address and returns the IP:port list of all of the online members of group G.

- Step γ: After learning about the current online members of group G, peer Z synchronizes website replicas with the online members.

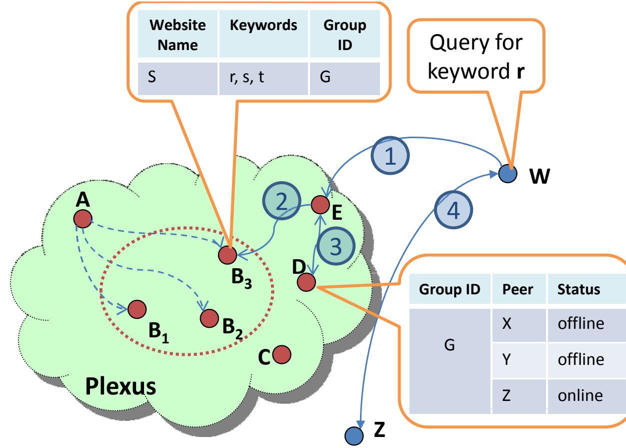

Figure 5: Keyword-based content searching

1.3 Query Process

There are two types of queries: a) keyword search and b) name search. The second type is more straight forward and a special case of the first type. Hence we explain the first type only, *i.e.*, query by keywords. As depicted in Figure 5, the keyword search process can be performed in the following four steps:

- Step 1: In the example scenario peer W is searching for the sites that have keyword r. It first sends the query to a random peer, say E in Plexus overlay.
- Step 2: Upon receiving the query, peer E constructs a query pattern (a Bloom filter) from the query keywords and uses list decoding to find the codewords within a pre-specified Hamming distance. Then it uses the Plexus routing to forward the query to the peers that are likely to have the meta-information on sites with the queried keywords. In the example in Figure 5, peer B_3 responds with the website name, keywords and the group ID (G) of the group hosting the website.
- Step 3: Once peer E receives the group ID G, it queries the Plexus network (similar to the rejoin process) to find the list of currently online members of group G. In this instance the network address of peer Z will be discovered and returned to the querying peer W.
- Step 4: Now peer W can directly browse the website from peer Z.

Part II:

File System

1 Introduction

A *file system* organizes persistent data by providing an interface to store, retrieve and modify data on persistent storage. It also organizes the free space in a storage device. The storage device can be a standalone physical device (*e.g.*, flash disk, magnetic disk), a network of such devices, or it can have a virtual existence and backed by another device. A *file system* usually provides the following functionalities:

- Space management on the storage device
- Organization of content (such as files in directories)
- Content Naming
- User access control to content
- Low-level interface to manipulate the storage
- High-level interface to end users that hides the storage level complexities

In this section, we describe a file system for pWeb. The file system defines a set of platform independent Application Programming Interface (API) for the pWeb dispatcher to store and retrieve data to and from peers. This protocol also defines mechanisms that enable a peer to request file system services from a remote peer.

We begin our description with a discussion about the requirements for such a file system protocol in Section 2. Then we present a short but necessary survey on the existing peer to peer file systems in Section 3. Section 4 discusses the architecture of the proposed file system for pWeb in more detail. Finally, we summarize the proposal in this part in Section 5.

2 Requirements for the File System

The requirements of our file system greatly depend on the pWeb architecture. pWeb has a decentralized architecture, where there are two types of peers: super peers for forming the DHT, and regular peers for storing content. Each peer is identified by a unique identifier called peer ID (`pID`). Peer groups are formed in such a way that at least $\beta (>= 1)$ group member is online at any given instance in time with a very high probability [36, 37]. These groups are called availability groups and they are identified by another unique identifier called group ID (`gID`). Each peer in an availability group stores the same contents as all other members in that group.

Based on the different properties of pWeb architecture, we have identified the following requirements for the file system:

- **Decentralized.** The file system must be decentralized. Individual peers should be able to work without any global knowledge of the system. The file system should work inline with the DHT based indexing scheme of pWeb.
- **Portable.** The file system should provide the peers with a platform independent interface for storing and retrieving content. This feature will enable content to be easily portable across different hardware and operating systems. The file system can be considered as a socket, and the content can be considered as a plug. The file system needs to be designed in a way that the hyper-linked content (*i.e.*, the plug) can be easily pulled out from one peer and plugged into another.
- **Read / Write Security.** The file system should be able to provide object level read and write capabilities to object owners, while making the storage secured and protected from any illegal access, including the peers replicating and hosting the objects.
- **Remote Invocation.** A peer will replicate all the content from all of its group members. Therefore, peers should be able to remotely invoke file services of any other peer for storing and retrieving content.
- **Scalable.** The file system should scale with the growth of stored content and group size, as well as updates to the existing content.

3 An Overview of Existing P2P File Systems

Distributed file systems have received a lot of attention in the past. Early works include the Network File System (NFS) originally developed by Sun Mircosystems, CODA, Andrew File System (AFS) *etc.* More recent works include The Google File System [28] (GFS), Hadoop File System [38] (HDFS) and Amazon's Dynamo [19]. These file systems are tailored to work optimally for specific needs *e.g.*, GFS is designed to work optimally with MapReduce programming model and perform well with the Google search engine. The content in these file systems is distributed across a number of servers but the operations on the file systems are coordinated by some centralized controller. Therefore, they do not meet the requirements of pWeb.

The inherent scalability property and decentralized nature of P2P systems along with their sucess over the last decade has generated many research efforts in designing distributed file systems. We have selected the following five projects as a representative set for discussion about P2P file systems: CFS [18], Freenet [11], OceanStore [25], IVY [31], and BitTorrent [14]. We will present a brief discussion on these research efforts in the following sections.

3.1 CFS

CFS [18] is a read-only peer-to-peer file system developed at MIT. It has four primary design goals: guaranteed efficiency, robustness, load balancing and scalability. It uses the Chord DHT for the storage of blocks. The set of blocks are distributed over the CFS servers. From the user's perspective CFS is a read-only file system.

CFS has a number of positive features. It divides files into chunks and distributes them across a number of storage nodes. This reduces the load on the storage nodes during file access. The Chord DHT enables logarithmic time lookup, and caching and replication are used to further reduce the lookup response time. However, maintaining files in small blocks imposes a large overhead on the system. All the blocks of a file are not guaranteed to be stored in a single storage node so a single file retrieval may involve routing along the longest path in the DHT.

3.2 Freenet

Freenet [11] is a read only P2P file system that enables the publication, replication and retrieval of content. Freenet operates as a location-independent distributed file system across many individual peers that allows files to be stored and requested anonymously. Freenet's design goals are anonymity, resistance to third party access, dynamic storage and routing, and decentralized policy.

Freenet identifies files by 160-bit keys obtained through the SHA-1 hash function. Search queries are also performed using hashed keys and can be either locally processed or forwarded to the lexicographically closest node in the routing table.

The main advantage of Freenet is that it provides anonymity for both publishers and consumers of information. It also provides fast lookup time for popular content by replicating it across the network. On the other hand, this system is vulnerable to security threats, such as Denial of Service (DoS) attacks and Dictionary Attacks. Furthermore the flat namespace requires globally unique identifiers and this affects scalability when the number of content items becomes large.

3.3 OceanStore

OceanStore [25] provides distributed access to persistent content on a global scale. It is designed using a cooperative utility model that has consumers pay a small fee to service provides to ensure they access to persistent storage.

The core of OcenStore's storage system is a set of untrusted servers provided by a number of service providers. OceanStore encrypts all data before storing it on the untrusted servers and aims to provide high availability. Data in the system is located by either a non-deterministic fast algorithm or a deterministic slow algorithm. OceanStore provides both read and write access but write access can be restricted using Access Control Lists (ACL). Lookup requests are routed between the servers using the Tapestry DHT.

3.4 IVY

IVY [31] is a read/write P2P file system. It uses journaling and a distributed hash table to provide users with an NFS [34] like interface. The journal is comprised of a number of logs maintained by each participant and enables recovery from failures. Although it provides an NFS like interface to end users, it performs two to three times slower than NFS.

3.5 BitTorrent

BitTorrent [14] is one of the most popular P2P networks. BitTorrent is a file sharing protocol that relies on other global components such as websites for indexing and searching torrent files. BitTorrent offers only read only access to files *i.e.*, original content cannot be modified. Files are divided into and distributed in chunks and once a peer has received a chunk it acts as a provider for that chunk. This enables content to be distributed quickly and efficiently throughout the network. Each peer is responsible for maximizing its download rate by contacting other peers. The Kademilla [29] DHT is used for index lookup in the BitTorrent system. BitTorrent can scale to a very large number of users but the dependency on global components restricts its use as a decentralized file system.

3.6 Discussion

Most of the P2P file systems discussed were designed as academic projects and do not have robust and efficient implementations. Only Freenet and BitTorrent have been deployed in a large scale real world environment, but neither of them allow content to be

updated once it has been published so they do not fulfill our requirement of a read/write file system. CFS is also a read only file system and is not suitable for pWeb. OceanStore and IVY are both read/write file systems but OcenaStore's utility model conflicts with our philosophy of voluntary participation and also depends on the untrusted servers of the service providers. Based on the observations in this section, we conclude that instead of using an existing P2P file system for pWeb, we will develop our own file system that is tailored to our requirements. The details of the pWeb file system are described in the following sections.

4 File System for pWeb

4.1 Overview

We propose a virtual disk based file system for pWeb. Each peer stores all of its content in a single file which appears to be a virtual disk to that peer containing data as objects. Every type of content that can be stored is treated as an object. The objects can be of different types. The major object types are (but not limited to):

- Web Pages
- Images
- Video
- Video Chunks (Since, video objects are large, they will be stored in chunks)
- XML files (for database)
- Style Sheets
- Client Side Scripts
- Plain Text Files (mainly for configuration purposes)

4.2 Naming and Directory Structure

The proposed naming scheme for pWeb [7, 8] identifies the individual objects using an ID called `pRL` that is comprised of a number of components. The `objectID` component of the `pRL` is used to identify the individual objects within a website while the remainder of the `pRL` is used to identify individual sites.

The grouping and naming scheme of the pWeb architecture is a two level hierarchy. Each peer stores the `objectID`s for individual `pRL`s (level 1), which in turn belong to a particular `pID` (level 2). Therefore, we propose the following naming scheme for the virtual file system:

Each object of a website will be identified by the `objectID` in the `pRL`. The objects will be stored in a two level hierarchical directory structure within the virtual disk. The levels will be organized as:

Level 0 **- The `root` directory:** The **root** directory will contain all other directories in the file system. The root directory will be identified by the "/" symbol.

Level 1 **- Directory for different peers:** There will be a directory for each peer having the same `gID`. These directories will be located one level down the `root` directory in the directory lever in the hierarchy. The directory for a peer will be identified by the `pID` of that peer.

Level 2 **- Directory for different sites:** The directory for a peer identified by `pID` will further contain a number of directories within it. Each of them will contain the objects of a site, and these directories will be identified by the site's `pRL` without the `objectID` field. For what follows next we refer to this as `site_prl`

The fully qualified name of an object will be `/pID/site_pRL/objectID`. Figure 6 gives a more clear picture of the hierarchical directory structure.

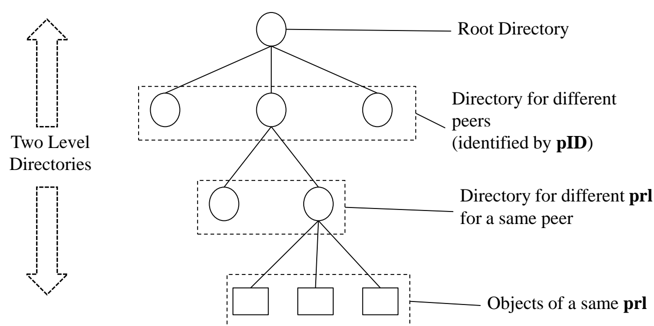

Figure 6: Two Level Directory Structure

4.3 Object Structure

An object will be a set of attribute-value pairs. We will use XML schema, a popular format for describing attribute-value pairs, to describe our objects. The minimal set of XML tags for describing an object are:

- **objectID:** The `objectID` part of the `pRL`.
- **publisherAssignedName:** Content publisher's assigned human friendly name.
- **type:** The type of the content (*e.g.*, webpage, images, videos *etc.*).
- **author:** Name of the content's publisher.
- **size (bytes):** Size of the content in bytes.
- **Last Access Time:** Time stamp indicating when the content was last accessed.
- **Last Modification Time:** Time stamp indicating the content's last modification time.
- **Version Number:** If the content exists in multiple version (*i.e.*, an image may have multiple versions for multiple resolutions), then this field is used to provide a version scheme.
- **iNodeNumber** This is similar to the I-node in UNIX like Operating System (OS) [27] except for the virtual disk.

This set of tags provides a generic interface for describing an object. For objects of specific types a number of extra attributes might be added if necessary. We provide a set of additional attributes for different object types described in Section 4.1.

- Webpage
  - **Page Type:** Type of language used to generate the page, *i.e.*, HTML, PHP *etc.*
  - **Technology Version:** HTML version, PHP version, *etc.*
- Image
  - **Encoding Type:** The type of compression the image is using, *e.g.*, jpeg, png, bmp, *etc.*
- Video

  - **Encoding Type:** The format of the video, *e.g.*, mpeg, mkv, avi, *etc.*
  - **Number of Chunks:** A video in this file system is divided into fixed size chunks. Each chunk is considered as an object. This field contains the number of chunks of a video content.
  - **Chunk Size:** The size of each video chunk
  - **Chunk Indices** An array where the i-th element contains the I-node number of the i-th chunk

- Video Chunk
  - **Video Object ID: objectID** of the constituent video file.
  - **Chunk ID:** A sequentially numbered identifier (starting from 0) to identify a video chunk of a video file.

- Client Side Script
  - **Script Language:** The language used for writing the script, *e.g.*, javascript, vbscript, actionscript *etc.*.

- Plain Text File
  - No extra attribute

4.4 File System Details

4.4.1 Disk Layout

The virtual disk will be flexible in size, that is, it will grow in size as needed. I/O will be performed on the disk in groups of bytes for performance since most physical file systems that will be used to store the virtual disk uses block level granularity instead of byte level granularity. This reduces the total number of disk operations (which are slow compared to main memory operations), and thus improves the system throughput.

We can adopt the I-node based UNIX file system with some simplification for implementing the virtual file system. An I-node is a data structure for each object and directory in the file system. An I-node keeps track of the disk blocks of an object or directory. Each I-node is identified by a unique I-node number starting from 0. The disk blocks of a directory will contain the name of the objects and their I-node numbers. The whole virtual disk will be divided into a number of regions:

Super block: This region will contain meta-data about the file system, *i.e.*, free I-node list, free disk block list, size of the file system *etc.*

I-node list: This region will contain all of the I-nodes in the system. In our context an I-node carries the same semantics as it carries in UNIX file system

Data blocks: This region will contain the object data.

4.4.2 File System APIs

The set of file system APIs will form two interfaces. One will provide lower level block I/O capabilities that will be used by the virtual file system itself to manipulate data on disk. The other interface will provide object level I/O capabilities to the pWeb dispatcher. The lower level API will depend on the OS specific file system calls to manipulate data on the virtual file system (which in turn is a regular file in the host operating system's file system). On the other hand the higher level API will provide a platform independent interface to the pWeb dispatcher. The pWeb file system is can be thought of as a socket with one end is attached to the host operating system while the other end provides an interface for putting in any plug (virtual disk) into it; therefore, all the contents of a peer can be easily ported to another peer.

Lower Level APIs The lower level APIs will encapsulate the operations on the virtual disk and they will provide block based I/O functionality to the higher level APIs. At minimum the following API function calls will be supported and available to the higher level APIs.

`readBlock(blockNo, buffer)`: This function reads the `blockNo` block from the disk and puts the data into the memory space pointed by `buffer`. This function returns the number of bytes read or a negative number on read failure.

`writeBlock(blockNo, buffer)`: This function writes the data contained in the memory space pointed by `buffer` to the **blockNo** block in the virtual disk. `buffer` points to a memory space which has a size equal to the number of bytes in a disk block.

`getNextFreeINode`: This function returns the next available free I-node from the I-node list.

`getNextFreeBlock`: This function returns the next available free block for storage from the virtual disk.

Higher Level APIs The higher level APIs will be available to the dispatcher to perform various operation on the file system. A list of API functions are as follows:

`createDirectory(pID, site_pRL)` This will create a `/pID/pRL` directory in the virtual file system of the invoked peer. **pID** identifies the invoking peer.

`createObject(pID, site_pRL, object)` This function call will create the passed object in the `/pID/site_prl` virtual directory of the invoked peer. `pID` identifies the invoking peer. First the $\langle attribute, value \rangle$ pairs will be serialized and then the serialized object will be written to the virtual disk using the lower level APIs.

`readObject(pID, site_pRL, objectID)` This function call will use the lower level APIs to read the byte stream corresponding to the `objectID` from the `/site_pID/pRL` directory on the virtual file system of the invoked peer. Then it will de-serialize the bit stream and return the object as a set of $\langle attribute, value \rangle$ pairs to the invoking peer.

`modifyObject(pID, site_pRL, objectID,` $\langle attribute, value \rangle$`list)` This function will read and de-serialize the whole object identified by `objectID` from the `/pID/site_prl` virtual directory of the invoked peer. Then it will update the values of the attributes in `attribute` with `value`. Then it will erase the old copy of the object from the virtual disk and will write the entire object to the virtual disk.

5 Summary

In this section, we have described a decentralized file system for pWeb. The file system provides an OS independent interface to the pWeb dispatcher. The file system's OS dependent part for performing low level disk I/O can be easily implemented in OS by mapping our low level disk access API to a number of system calls. The described file system API along with the file system protocol will provide a platform for developing easily portable websites.

Part III:

Server-side Scripting Enabling Dynamic Page Generation

1 Introduction

This module enables hosting of dynamic web pages in pWeb. Client and server side technology choices *i.e.*, scripting languages, data storage mechanisms, data transfer protocols, *etc.* greatly depend on the deployed webserver and its supported features. We introduce the requirements of a HTTP webserver in Section 2. Alternative deployment choices for web servers are presented in Section 3. Next, we present a survey of suitable webservers in Section 4. Section 5 describes how we can integrate server and client side scripts using CGI, PHP and Java.

2 Requirements of a HTTP Server for P2P Web Hosting

One of the primary objectives of pWeb is to be able to run seamlessly on a variety of platforms. For this reason, we have distilled the following requirements for the pWeb HTTP server:

- **Lightweight:** We envision our pWeb software to run on a wide range of hardware devices, including cell phones, smart phones, tablets, set-top boxes, laptops, desktops *etc.*A lightweight HTTP server with a small memory footprint will make it possible to deploy pWeb on nodes with restricted resources without affecting user experience.
- **Ease of deployment:** The HTTP server should be easy to deploy on a peer independent of its underlying hardware. A single installer should be able to install both the pWeb client and the HTTP server.
- **OS independent:** It should be installable or portable to all major OSs.
- **Suport for CGI:** The webserver should support CGI and FastCGI. CGI is required to access pWeb Java library from PHP scripts.
- **Site portability:** Websites deployed under the webserver should be easily portable. One should be able to run the website from a different peer just by copying the website directory from one peer to another.

3 Alternative Deployment Choices

In this section we evaluate different approaches for deploying an HTTP server on a peer and linking PHP and JavaScript running in the server to pWeb Java codebase.

3.1 Mozilla Firefox: LiveConnect

LiveConnect [22] is an API enabling JavaScript executing within Mozilla browsers to call Java code (and vice versa). However, these privileges only apply to the JavaScript code written as a Firefox extension, not the JavaScript running within HTML pages. The Java code called runs as a separate process, has full networking capabilities, access to the local filesystem, and can (in particular) start a local HTTP server. LiveConnect requires the Java plugin to be installed.

An advantage of LiveConnect is that Plexus is implemented in Java and can therefore be called within the same process, without additional complexity, delay and multi-platform compatibility and reliability issues. The disadvantage is that LiveConnect works on Firefox only. We would need to find a separate implementation for every other browser. Moreover, LiveConnect is unstable from Firefox version 4 and above.

3.2 NPAPI plugin

The NPAPI (Netscape Plugin Application Programming Interface) is an interface for web browser plugin development. NPAPI was originally created for the Netscape Navigator web browser and is still supported by most web browsers except Internet Explorer. The Mozilla Foundation has developed an ActiveX Control for hosting Netscape plugins in Internet Explorer [21]. However, that project is outdated and has a number of deficiencies in terms of the supported features. Despite the incompatibility with Internet Explorer, NPAPI has a number of advantages that are presented below:

- NPAPI is a well established standard; NPAPI plugins are supported by most browsers except Internet Explorer. In Google Chrome, NPAPI plugins can be included as browser extensions [10].
- No need to install Java plugin for the browser.

This approach has the following disadvantages:

- NPAPI plugins need to be recompiled for every operating system, so the deployment effort increases.
- NPAPI plugins are not subject to any sandbox security measures. Therefore, security exploits, *e.g.*, malicious JavaScript in HTML can have severe consequences for the machine running a browser with this plugin.

- The source code for P2P web server *etc.* would have to be incorporated into the NPAPI plugin, increasing the implementation's complexity and reducing manageability. Web browser code typically comes in libraries yet the interaction with such libraries still requires web server development knowledge, *i.e.*, how the web server operates internally. Therefore, the required knowledge goes beyond web server usage knowledge, *i.e.*, how to set up the web server (perhaps automatically) and integrate it with other applications.

	Browser Extensions	**NPAPI Plugins**
One solution for all browsers	No	Yes
One solution for all OSes	Yes	No
JVM required	Yes	No

Table 1: NPAPI Plugins vs. Browser Extensions

3.3 Internet Explorer: ActiveX

ActiveX allows the developer to use a number of functionalities from within a webpage, including access to the local file system. However, *ActiveX* works only on Internet Explorer, which in turn works only on *Microsoft Windows*. Therefore, we would have to create separate solutions for other browsers and operating systems. As a result, using ActiveX for starting and managing web servers would increase the development effort.

3.4 Internet Explorer: `BrowserHelperObjects` (BHOs)

BHOs extend the web browser [15], providing functionality similar to Firefox browser extensions. For example, BHOs can intercept page loads and modify the contents of HTML pages loaded within the browser. BHOs implement the Component Object Model (COM) interface and can be implemented in C# or C++ [16], [13], [12].

3.5 Internet Explorer: Plugins

It is possible to develop plugins for Internet Explorer, e.g. using the tool available at [17]. However, these tools and tutorials are outdated and are not compatible with current version of Internet Explorer.

3.6 Standalone HTTP Servers

A standalone HTTP server can be installed along with the pWeb client under the same installation package. The advantage of this approach is that the server can run in the background and data can be exchanged over the HTTP protocol. However, we need a mechanism for communicating between the PHP script and the Java code running on the same peer. This can be done using CGI. A CGI script written in scripting languages like Perl or Python can facilitate data transfer between PHP and Java. This approach has several other advantages such as ease of deployment, configuration and interoperability– any light-weight HTTP server with similar features can co-exist on the pWeb network so different HTTP servers can be run on devices with different capabilities.

3.7 Java-Based Ajax Servers

A local HTTP server could be implemented using indirect Ajax frameworks. Rich Internet Application (RIA) frameworks have opened a new era of developing complex user interfaces using regular web-browsers. RIA frameworks depend heavily on Ajax for dynamic and partial content loading. Ajax frameworks based on JavaScript are executed on the client and allow components to be created using JavaScript functions, generating the required HTML elements automatically.

Indirect Ajax frameworks based on Java require a central server, which can be deployed on each peer. This framework provides the following features:

- Establishes the HTTP server for serving both static and dynamic content
- Gives the user a convenient way to create his/her own web content, *e.g.*, with a WYSIWYG tool on top of the Ajax framework

However, indirect Ajax frameworks do not provide support for CGI. Direct communication between server side PHP scripts and client side pWeb Java code is not possible.

3.8 Summary

Based on the advantages and disadvantages of the deployment choices discussed in this section, we have made the following decisions:

- A standalone lightweight HTTP server will be deployed on each peer
- Java-based Ajax frameworks can be used for content editing and uploading

- Browser plugins help in handling pWeb Resource Locators (pRLs) by translating between pRLs and peer IP addresses

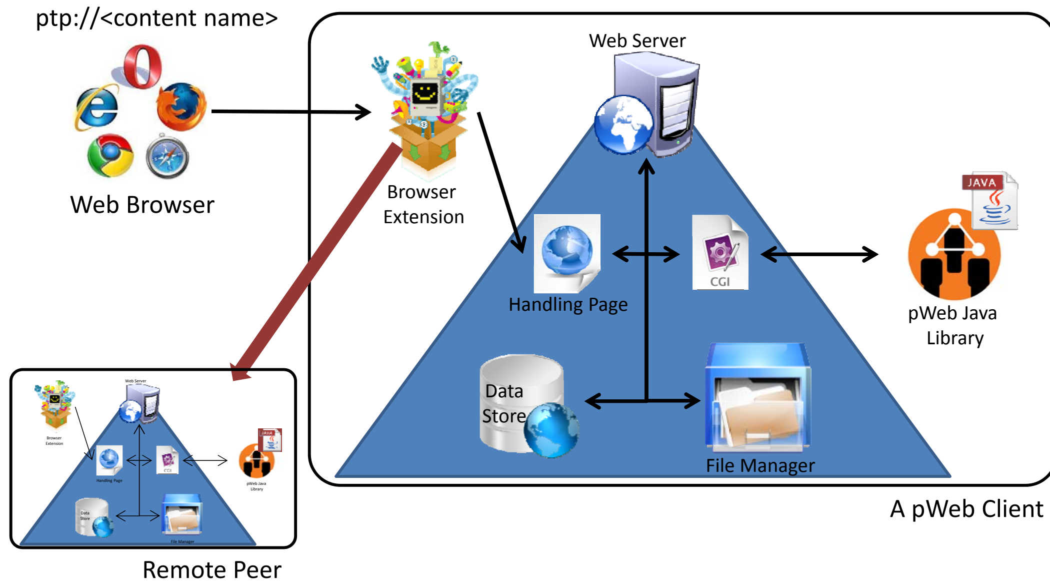

Figure 7: Interaction between server and client side components

Interaction between these components are shown in Figure 7. After a user enters a pRL in the browser address bar in the form of `ptp://<content name>` (here `ptp` stands for pWeb Transport Protocol), the browser invokes the pWeb plugin to handle the address. The plugin forwards the content name to a special handling page, hosted in local web server, for processing. The handling page forwards the content name to the pWeb Java client using CGI. The Java client invokes the name resolution module and resolves the supplied pRL to a currently alive peer's IP:port pair in the network hosting the corresponding content. The IP:port pair is returned to the browser extension through CGI, which then retrieves the web content from that IP. Each peer has a webserver which servers both static and dynamic content. XML files are used for storing dynamic data in the Data Store and the File Manager manages other static files like HTML, images, video, scripts, *etc.*

In the next section, we provide a brief survey of several lightweight HTTP servers along with their provided features and discuss the suitability of each server in the context of pWeb.

4 Light-Weight HTTP Servers

We intend to avoid running a fully-fledged HTTP server (e.g. the Apache HTTP Server) due to resource issues (particularly on mobile devices) and automatability of deployment. Instead, we require a minimal-weight, yet PHP-enabled web server, which requires no or minimal configuration from the user. In this subsection, several light-weight servers are compared with regards to whether they exhibit these properties.

4.1 Cherokee

The *Cherokee* webserver runs on all major platforms, including Windows, Unix-based systems and Mac OS, and supports a large number of server-side scripting languages but support for them must be set up manually. In particular, the PHP interpreter needs to be set up separately and Cherokee needs to be configured to use this particular interpreter. Cherokee is very well documented, but a user without background in computer science will not be able to set up the local HTTP server manually. As a result, Cherokee (like most other webservers) does not qualify in terms of ease of use.

4.2 Hiawatha

Hiawatha is a cross-platform, light-weight webserver with a comprehensive set of security features [26]. Hiawatha is similar to Cherokee in many ways, including the fact that the PHP interpreter needs to be downloaded and set up manually. While Hiawatha can be run as a Windows Service, it was "never fully tested" under Windows [26] and the author even recommends to use Hiawatha under Windows only for websites for testing purposes. While there are no high-availability requirements towards the peers of pWeb, it would still be desirable to have the server running reliably when the peer machine is running.

4.3 lighttpd

lighttpd is highly optimized for speed–in particular, it is intended to solve the "C10k problem", which is the problem that most web servers cannot handle more than 10,000 simultaneous connections [40]. As a result it "is used by a number of high-traffic websites" [24], including YouTube and Wikipedia. The intended purpose of lighttpd is the quick distribution of many small files ("Ad-Server Front-Ends") and "load-balancing the php-request over multiple PHP-servers". These goals differ from the typical scenario of a web server on a pWeb peer, which will likely not have to handle more than 10,000

simultaneous requests. The intended users of lighttpd are individuals who are very proficient in computers. Yet, lighttpd can be deployed within an installation package with preconfigured options, despite the focus on speed and efficiency. lighttpd comes in binaries, without having to do any compilation, which makes automated deployment and integration into a pWeb installation package easier, particularly under Windows.

4.4 nginx

nginx [39] is a cross-platform lightweight webserver that is similar to the lightweight webservers presented previously. In contrast to many others, including Apache and lighttpd, nginx does not automatically spawn FastCGI processes. As a result, providing PHP support via CGI or FastCGI is more complex to set up, reducing the automatability of deployment.

4.5 HFS (HTTP File Server)

HTTP File Server (HFS) [30] is a web server designed for easy-to-use file sharing. HFS is only available for Windows and does not support server-side scripting and dynamic content, *e.g.*, PHP. These drawbacks are the main reasons why HFS does not fulfil the requirements the pWeb project has for the local HTTP server.

4.6 Comparison of Lightweight HTTP Servers

In this context "automatability" refers to automatability of deployment, setup and operation of the HTTP server.

From our survey, we can conclude that most lightweight web servers are either (i) hard to set up automatically and have poor support for Windows or (ii) provide no support for dynamic content. lighttpd, however, is more automatable than most other lightweight HTTP servers, which makes lighttpd our choice.

5 PHP and Java: Integration Techniques

pWeb peers serve content to other peers via a lightweight local HTTP server, which needs to support PHP in order to support dynamic content. In this section, we refer to this server as the Content HTTP Server (CHS).

At the same time, major pWeb components, including name resolution and replication as well as content advertising and searching, are written in Java and need to be accessed by

	Dynamic Content	**Automatability**	**Recompile for each Platform**
Cherokee	Via CGI (Manual Setup)	No	Yes
Hiawatha	Via CGI (Manual Setup)	No	Yes
lighttpd	Via CGI (Manual Setup)	Yes	No
nginx	Via CGI (Manual Setup)	No	Yes
HFS	Not supported	Yes	Windows only

Table 2: Comparison of Light-Weight HTTP Servers

web browser extensions. As these run in a separate process, the web browser extension needs to communicate with the Java components via `XMLHTTPRequest`. This section presents and analyzes approaches on how to integrate PHP and Java components for collaboration on a peer.

5.1 Separate Java HTTP Server

JRE 1.6 includes a simple Java webserver, which can be used to expose the Java functionality to other processes, *e.g.*, a web browser extension. The local Java HTTP Server runs independently from the CHS. The server publishes the pWeb functionality within several resources, with which the browser extensions and provided webpages interact. The following list shows some examples:

- The resource `/nameresolve` is the interface for name resolution, which is performed by Java components. The URL to access this resource is `http://localhost:10001/nameresolve?contentname=www.google.ca`.
- The resource `/getsetting` is used to read settings. Only one setting can be retrieved at a time. The URL to access this resource is `http://localhost:10001/getsetting?settingname=<settingname>`.
- The resource `/updatesetting` is called to update settings. An example URL to update the two settings `maxdown` and `maxup` is `http://localhost:10001/updatesetting?maxdown=200&maxup=202`.

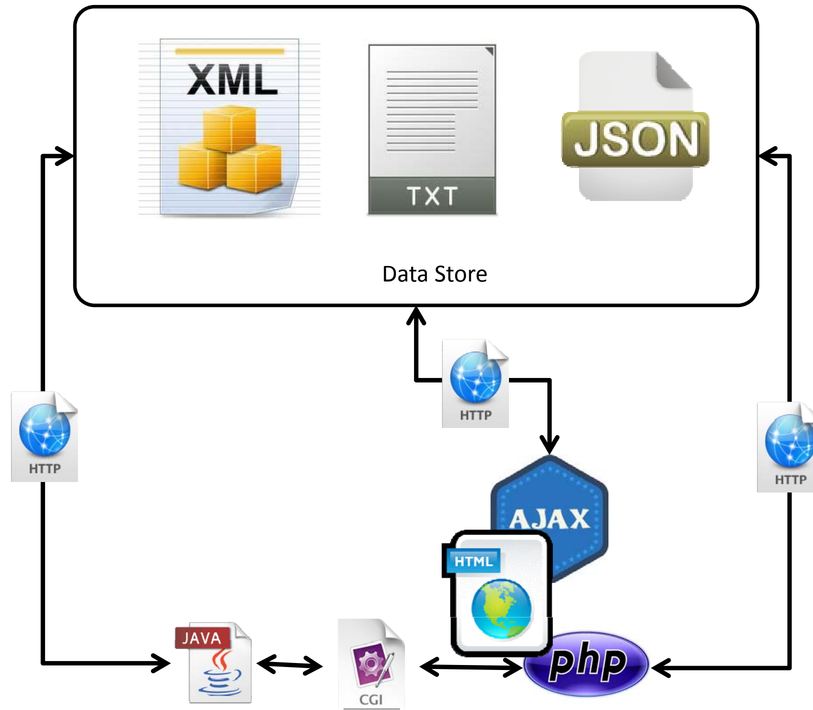

Figure 8: Interaction between different technologies

> Several settings can be changed with one call. Settings are read and written using a special Java component which connects the Java server to the other components of the pWeb installation. For example, when editing the settings of the CHS, the Java server will accept the request for setting update and then edit the configuration file of the CHS.

The Java Server can be called by any other process on the local machine. For example, the extJS framework (which is used to implement the UI) and the browser extensions can use `XMLHTTPRequest` to connect to the local Java HTTP Server. The Java server is not intended to receive connections from other machines, and will deny any requests which do not originate from `localhost`.

This solution takes advantage of the fact that the Java server is simple to implement. A major drawback of this solution is that two servers need to run on each peer. We intend to avoid having to run several servers and having to use several ports on the machine, due to:

- performance, and

- port management: Assume pWeb uses *e.g.*, port 80 for a web content server and another port for a server exposing the Java component's functionality to web browser extensions. Then, that other port can be either static, causing collision problems, or the Java server can be dynamically bound to a free port upon each startup. In this case, however, the port needs to be communicated to the web browser extension at each startup. Also, when the functionality implemented in Java is exposed under another port than the script where the UI pages are hosted, the same-origin policy prevents those webpages from consuming the Java functionality directly.

Note: Due to the reasons stated above, the Java webserver solution has been replaced with the CGI solution presented in Section 5.3. The resources presented above have been replaced with a single resource named `update` on the light-weight CHS, which calls the appropriate Java components via CGI.

5.2 Java Webservice

This solution exposes the functionality of the Java components as a webservice, which is then accessed by a PHP wrapper function. In Java, it is possible to expose methods to a webservice with very little coding effort:

```
@WebService
@SOAPBinding(style=Style.RPC)
public class Concatenator {
  public long concat(String val1, String val2) {
    return val1 + val2;
  }
}
```

Listing: The class whose functionality is exposed to webservice. In this example, the class performs a simple string concatenation.

```
public class ConcatServer {
  public static void main (String args[]) {
    Concatenator c = new Concatenator();
    Endpoint endpoint =
      Endpoint.publish(
        "http://localhost:8080/concatenator",
```

```
        c
        );
    }
}
```

Listing: Creation of webservice.

```
try {
        $x = new SoapClient(
                "http://localhost:8080/concatenator?wsdl"
        );
        $a = "abc";
        $b = "def";
        $y = $x->__soapCall("concat", array($a, $b));
        echo $y;
} catch (Exception $e) {
        echo $e->getMessage();
}
```

Listing: PHP wrapper function.

The URL of WSDL file is: `http://localhost:8080/concatenator?wsdl`.

However, this simple interface hides the complexity and resource consumption behind offering a webservice. A separate server is started, requiring a second port to be reserved besides the CHS's port. Therefore, this solution also requires two ports.

5.3 CGI

Using CGI, the webserver–lighttpd in our case–redirects requests for resources of specified types to a predefined executable. The following listing shows the part of the lighttpd configuration file where `.php` files are sent to the PHP executable and `.jav` files are sent to a batch file named `CGIConsole.bat`. The batch file then processes the request by running a JAR file with the parameters of the request, and then returns the values returned by the Java component.

```
cgi.assign=(
        ".php" => "C:/php/php-cgi.exe",
```

```
        ".jav" => "C:/CGIConsole.bat"
)
static-file.exclude-extensions = ( ".php", ".jav" )
```

Listing: CGI configuration of lighttpd

To interact with functionality implemented in Java, the pWeb HTTP server (lighttpd, as mentioned earlier) provides one single resource. The resource is named `update` and is called using POST parameters. The `fname` parameter specifies the function name, *e.g.*, `getSetting` for retrieval of a setting. The `param` parameter specifies the arguments by which the function is called, encoded in JSON format.

5.4 Example: Java Function Call

The following listing shows an example for a function call of a Java component. Assume the Java function is named `updatesetting` and requires the parameters `settingname` and `settingvalue`, so the setting "maxup" is updated to "100". The function definition of the Java source file would be `void updatesetting(String settingname, String settingvalue)`.

A process outside the Java component calls the function `updatesetting` by invoking the URL `http://<ip of peer>/update`, which is hosted by lighttpd, with the POST parameters `fname=updatesetting` and `param=<parameters>`, where `<parameters>` is a placeholder for the listing below.

```
[
                {
                    "paramname": "settingname",
                    "paramvalue": "maxup"
                },
                {
                    "paramname": "settingvalue",
                    "paramvalue": "100"
                }
]
```

Listing: JSON encoded parameters when updating a setting of the pWeb software. For every parameter of the function to be called, the JSON array contains another element.

5.5 Conclusion

There are extensions to CGI, namely FastCGI and SCGI (Simple CGI), which provide better scalability and an easier interface. However, both of these technologies rely on a separate server, so the required number of ports would be two for FastCGI and SCGI. We choose CGI to couple PHP and Java, since CGI does not require a second server or port.

6 Summary

In this section we presented the mechanisms for supporting dynamic web content in pWeb. A typical user will be able to create his or her dynamic content using a mix of PHP, XML, JavaScript, and Ajax, which have become popular technology choices for creating dynamic content. Here, XML files are used as data store, which are accessed from PHP or Ajax for generating dynamic pages. Moreover, these scripts can be ported to any peer just by copying them to that peer and the new peer can start serving those pages right away. The PHP scripts access the pWeb Java code base by using CGI, which is provide by lighttpd web server. Our selected mechanism for supporting dynamic pages provides a coherent solution, where each individual component is self-contained and connected to other components through well defined interfaces.

Part IV:

XML Data Management and Access Capability

1 Introduction

XML (EXtensible Markup Language) is widely used to transport and store data. It provides a way to represent data in an object-oriented fashion. It is very similar to HTML, but the tags are not predefined. Tags in XML are defined by the users based on the data that needs to be stored or transported. Figure 9 shows an example of XML file representing employee information.

```
<Employee>
    <FirstName>Rakibul</FirstName>
    <LastName>Haque</LastName>
    <Address>Waterloo</Address>
     <Position>Research Assistant</Position>
</Employee>
<Employee>
    <FirstName>Golam</FirstName>
    <LastName>Rabbani</LastName>
    <Address>Waterloo</Address>
     <Position>Teaching Assistant</Position>
</Employee>
```

Figure 9: Example of XML

The following rules are enforced when defining an XML document:

- All XML elements must have an opening and a closing tag
- XML tags are case sensitive
- XML elements must be properly nested
- XML documents must have a Root element
- XML attribute values must be quoted

A webserver uses storage for the websites that generate dynamic webpages as shown in Figure 10. We chose XML as the storage for the following reasons:

- Objects/data can be represented in a structured self-descriptive manner
- Transferring objects/data will be easier
- XML is platform-independent

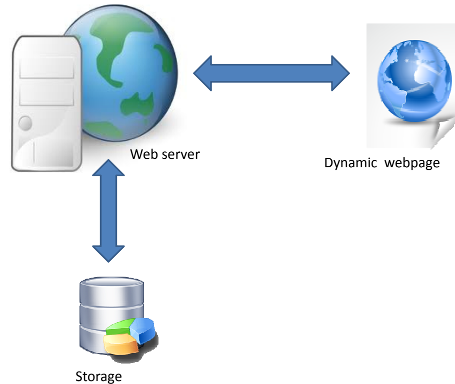

Figure 10: Dynamic webpage

- The whole website needs to be portable; the use of XML as storage will increase portability as opposed to using a dedicated database management system.
- XML processing and query tools are widely available and supported in many platforms.

2 Native XML Database

A native XML database is one that treats XML documents and elements as the fundamental structures rather than tables, records, and fields. Such a database enables developers to use tools and languages that more naturally fit the structure of the documents they're working with, thereby enhancing productivity.

In most XML databases, the fundamental unit is the XML document, which roughly corresponds to a record in a relational database. One big advantage of a native XML database is that it can run queries that combine (or join, in SQL parlance) information contained in multiple XML documents. The need to query multiple documents explains the design of XQuery, the developing query language for native XML documents, which is in turn based on XPath 2. In fact, the ability to query multiple documents is probably the single most fundamental difference between XPath 1 and XPath 2/XQuery. What SQL is to relational databases, XQuery is to native XML databases.

XQuery does not provide as much functionality as SQL does. Whereas SQL has four DML (Data Manipulation Language) operations: SELECT, INSERT, UPDATE, and

DELETE as well as some DDL (Data Definition Language) commands for creating and dropping tables and users. XQuery only supports SELECT to retrieve information from an XML database. It can't add documents to the database, delete documents from the database, or modify existing documents. Most native XML databases implement an XQuery extension for data manipulation. For example, XUpdate is implemented by eXist [20].

There are a few commercial and open source native XML databases. Some of them are listed below:

- ***Qizx*** **[33]** is an XML database engine which allows the users to store and index XML documents, of any type and size, and to perform searches and transformations on stored documents. It can be integrated into a standalone Java application, or it can be the core of a server. It fully supports the XQuery language and its extensions XQuery Full-Text and XQuery Update.

- ***BaseX*** **[9]** is a platform independent, open source, light-weight and scalable XML Database engine and XPath/XQuery 3.0 Processor. It includes full support for the W3C Update and Full Text extensions. It also provides an interactive user-friendly GUI to give the users an insight into their XML documents.

- ***eXist-db*** **[20]** is an open source database management system built using XML technology. It stores XML data according to the XML data model and features index-based XQuery processing. It is highly compliant with the XQuery standard. The query engine is extensible and features a large collection of XQuery Function Modules. eXist-db provides a powerful environment for the development of web applications based on XQuery and related standards. Entire web applications can be written in XQuery, using XSLT, XHTML, CSS, and Javascript (for Ajax functionality). XQuery server pages can be executed from the filesystem or stored in the database.

- ***Sedna*** **[35]** is a free native XML database which provides a full range of core database services, *i.e.*, persistent storage, ACID transactions, security, indices, and hot backup. Flexible XML processing facilities include W3C XQuery implementation, tight integration of XQuery with full-text search facilities and a node-level update language. It also provides user role and privileges for database security purposes.

- ***OrientX*** **[32]** is a native XML database system, developed at the Renmin University of China. The OrientX system stores XML data and preserves its tree

structure. It also allows users to retrieve XML data in the form of the XPath/XQuery language and has an API for C/C++ development. OrientX stores XML data in its native tree structure, according to the XML data model, and supports element-based and subtree-based granularity.

- ***Xindice*** [**42**] is a native XML database designed specifically for storing XML data. It uses XPath for its query language and XML:DB XUpdate for its update language. Xindice provides an implementation of the XML:DB API for Java development and it is possible to access Xindice from other languages using a built-in XML RPC API.

3 Native XML Database versus Flat-file XML Database

Native XML databases can process and manage large volumes of XML documents efficiently. These native databases will be useless for fewer numbers of documents. If the number of XML documents is small, then a simple XML parser is sufficient to process them. In pWeb, each website will be stored as a stand-alone file and replicated irrespective of the underlying hardware and software systems as shown in Figure 11. If a native XML database is used, compatibility among the systems will be an obstacle for the portability of websites. As the number of XML documents is small for each website, it would be wise to use flat-file XML documents for describing websites' files and data.

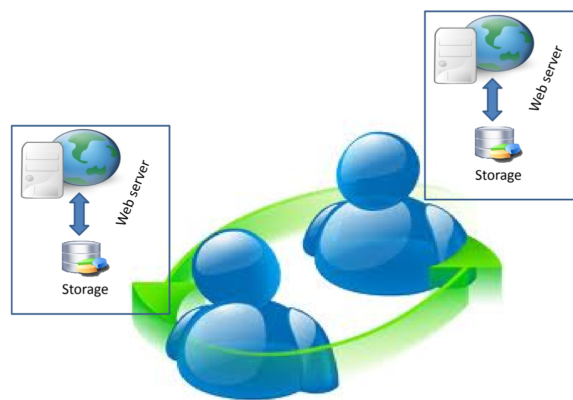

Figure 11: Portable website

4 General XML Schema for pWeb Documents

A minimal XML schema for pWeb documents is:

- **objectID:** The `objectID` part of the `pRL`.
- **publisherAssignedName:** Content publisher's assigned human friendly name.
- **type:** The type of the content (*e.g.*, webpage, images, videos *etc.*).
- **author:** Name of the content's publisher.
- **Version Number:** If the content exists in multiple version (*i.e.*, an image may have multiple version for multiple resolutions), then this field is used to provide a version scheme.

Part V:

Client Software Implementation

1 Introduction

Our implementation of pWeb provides a video streaming service to the end users. The end users can register video stored in their local machines to a set of servers that index meta information about the videos. Any user can search for a video in their local server, which is responsible for routing the search query to other servers to resolve it. After a query is resolved the user can view the video, which is stored in another user's local machine.

We discuss the system architecture in Section 2 and the database architecture in Section 3. Section 4 describes the features available in the prototype implementation and briefly shows the message exchanges between server(s) and client(s). The implementation specific choice of technology is described in Section 7.

2 System Architecture

The components in the system can be classified as: (a) Server side PHP components and (b) Client side Java code. In the subsequent sections we give a brief description of the functionality of these components. A high level view of the system is given in Figure 12.

2.1 Server Side PHP Components

The server side PHP components are responsible for:

- **Client to index server communication:** Allow client-side uploading for meta-information and searching the system, peer-device to IP mapping, and content replica/location update
- **Server to server communication:** Server to server peering for database information exchange, server locations exchange
- **SQL Database**: Stores the meta-information, users and peer information

2.2 Client Side Java Components

The client side Java components are responsible for:

- **Client to server communication**: Upload meta-information of advertised or replicated contents, name-to-IP binding, and server information download

- **Client to client communication**: Streaming video contents, and exchange known server locations

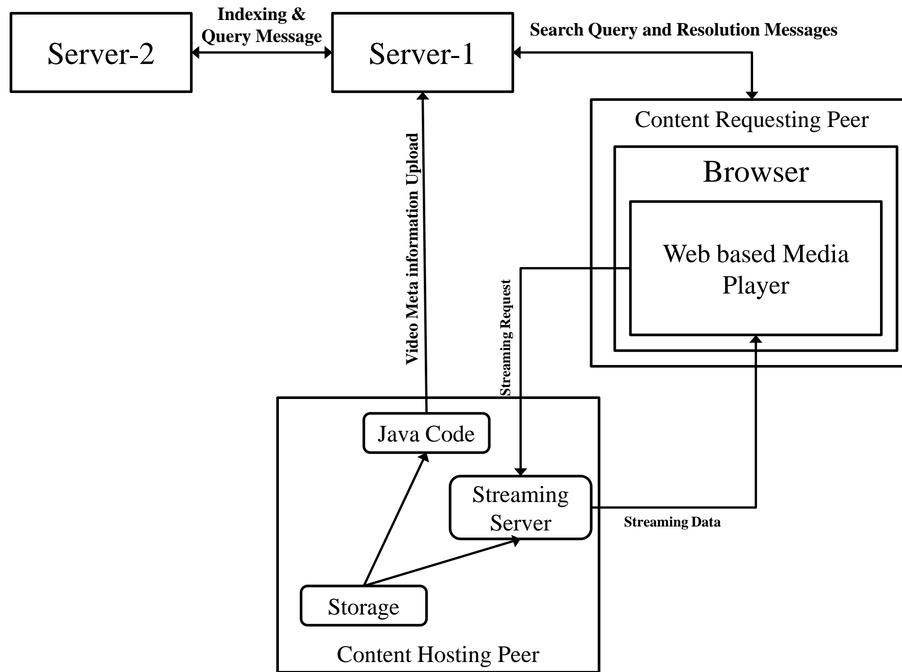

Figure 12: Architecture of Prototype Implementation

3 First Database Architecture

The diagram on Figure 13 represents the first database prototype:

We introduce the concept of User and Peer entities in this architecture. They are both defined by a UUID (`uID` and `pID`) and as the schema shows, a user can have multiple peers. Indeed, a user can own one account but we can assume that he will connect to the pWeb application through many devices. But each device has its own IP address, so the Peer entity is directly linked to a device (device_name, device_type). A user has access to the group features (linked to the uID) while a peer can upload meta-information (linked to the pID, especially because in order to stream a content, the application needs the IP adress of the device). This database is described by the following tables:

- **Users:**

- **uID:** the user ID is an unique identifier provided by the user's e-mail
- **Username:** the username is chosen by the user but is also unique in order to avoid duplicates in group features or video comments
- **Password:** the password is encrypted with the MD5 function provided by PHP
- **last_seen:** is updated every time a user connects to the application

- **Peers:**
  - **pID:** the peer ID is an unique identifier that is automatically provided by the database (automatically incremented integer)
  - **uID:** the user ID
  - **pIP:** the peer IP is the IP adress of the device used at the connection. It is automatically updated at each connection
  - **pPort:** the peer port is the port used by Red5 to stream content. It is set by default to 8088 but it can be changed by the user
  - **device_name:** a user can create a device every time he connects to the application. He will have to give it a name and choose the type of the device
  - **device_type:** can be Personal computer, Public computer, Laptop, Mobile phone, ...

- **Group_Members:**
  - **gID:** a group ID is a unique identifier provided by the group's creator
  - **uID:** uID of the user who have joined the group

- **Groups:**
  - **gID:** a group ID is a unique identifier provided by the group's creator
  - **isPrivate:** boolean that defines if a group is Public or Private
  - **Description:** a full description of the group provides by the group's creator

- **Uploads:**
  - **pID:** pID of the peer who uploads the video
  - **vID:** vID of the uploaded video

– **gID:** a video can be uploaded only in a group. If not, this case is blank
– **date:** date of the upload

- **Metas:**

  – **vID:** a video ID is a unique identifier that is an automatically incremented integer
  – **vName, Keywords, Description, Image:** video information provided by the uploader
  – **Views, Likes:** set to 0 when the video is uploaded
  – **peer serverIP:** server IP adress from which the peer has uploaded the metas

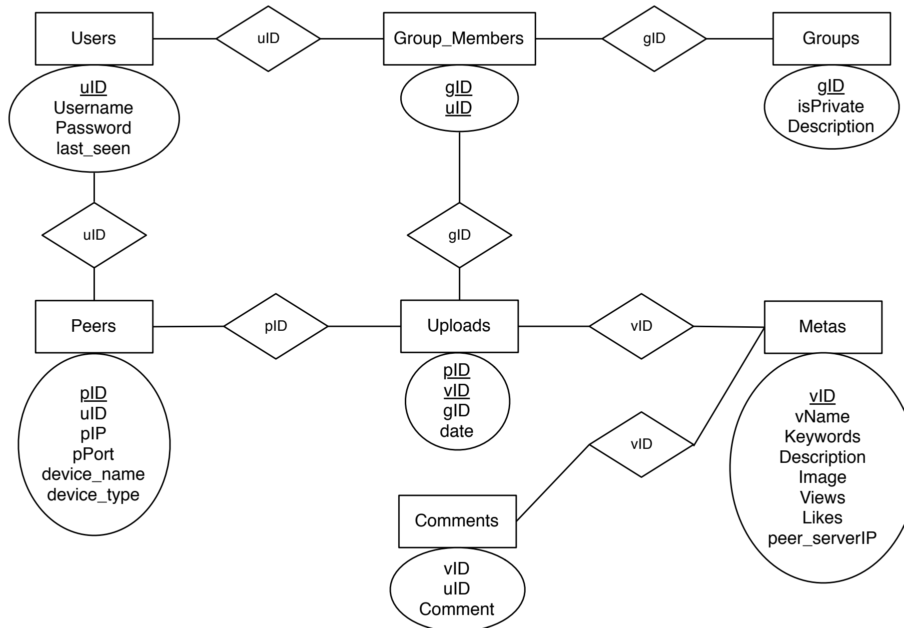

Figure 13: First database Architecture

4 Application Features

The prototype implementation of pWeb provides the following set of features:

4.1 User management system

Users can login into a web based interface in order to create a session. Information requested on the login page is the user's Username and Password. In the prototype implementation, the server applies the MD5 hash function to the password and stores user information in the database. A user can also fill out a form for a new User ID (his e-mail), a Username and a Password. Then, his login information is uploaded to the database and he is redirected to the index page. After login, the application asks the user which device he is using (device stored in the database) or offers the possibility of creating a new one. After registration, the application will automatically ask the user to create a new device. Then, the server checks the IP of the device and stores it into the database.

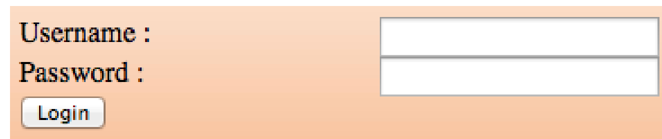

Figure 14: Login form

Figure 15: Choose or create a device after login

Figure 16: Registration form

Registration completed! Please, choose your device:
Type of your device : Public computer
Name of your device :
Add a new device

Figure 17: Create a new device after registration

4.2 Upload Content

Users can choose a file on their local storage and upload it (from a practical point of view, the file has to be on the specified Red5 home directory). The server wont store the file but only its meta information in the Metas table of the database. The upload function performs some check on the video file, *i.e.*, the extension must be .flv, it must not exceed a maximum video size and all the accented characters are replaced. The upload form contains the following fields:

- File: uploading HTTP data form
- Keywords (optional): the user can enter some keywords, if not, the application will create automatically keywords by parsing the video's name
- Description of the file
- A cover Image (URL)

File : Parcourir...
Keywords (optional) :
Description of the file :
Image (URL) :
Send the file informations

Figure 18: Upload form

4.3 Database Lookup or Video Search

Users can enter keywords on the form. Then, the server performs a local lookup on his database to find the related videos and returns all the matches to the browser. The user can choose one or more groups on the search form and the system will return the videos that have been uploaded into these groups.

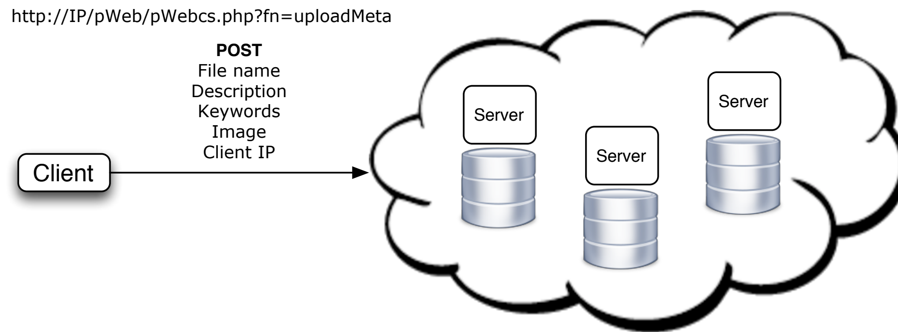

Figure 19: Upload content message

Figure 20: Search form

4.4 Content Access and Video Streaming

There is a link on each result for the search query that allows the user to watch the video. The JwPlayer makes a request to the video owner's Red5 server. Upon receiving the request, the Red5 server on the owners machine sends the video using the RTMPT protocol.

4.5 Group Management System

The grouping system allows users to upload and stream videos in small groups of users. The first prototype of this system aims to create interest groups but some extra features can be added like device syncing, sharing data, *etc*. Each group is identified by an unique ID called gID (`gID`). A group contains a list of Peers and can be Private or Public. A user can create a group by filling out a form with the group's name (gID), a description, and by setting the group's privacy. Then, the user will atomatically join this group.

A user can also join one or more groups (Figure 28). Then, he will be able to have some information about the group in the menu "My Groups" which lists all groups joined.

Figure 21: Results

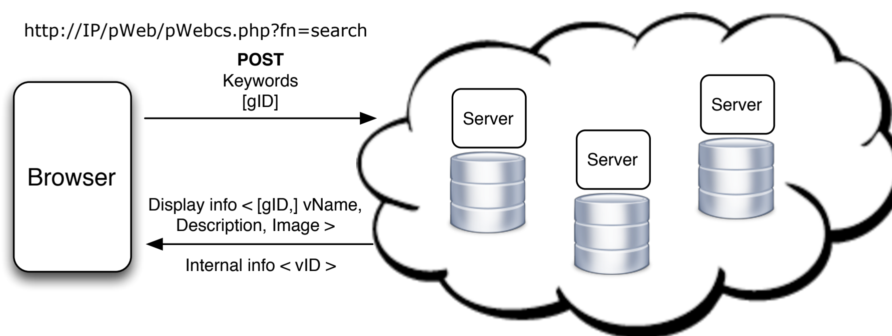

Figure 22: Look-up database message

Some features are available like "See group members", "Change privacy of the group", "Upload a video in this group" and "See the group videos".

After loging in, the button "My groups" appears. In this menu, the user can check all his groups, see the videos uploaded in those groups and the members who have joined

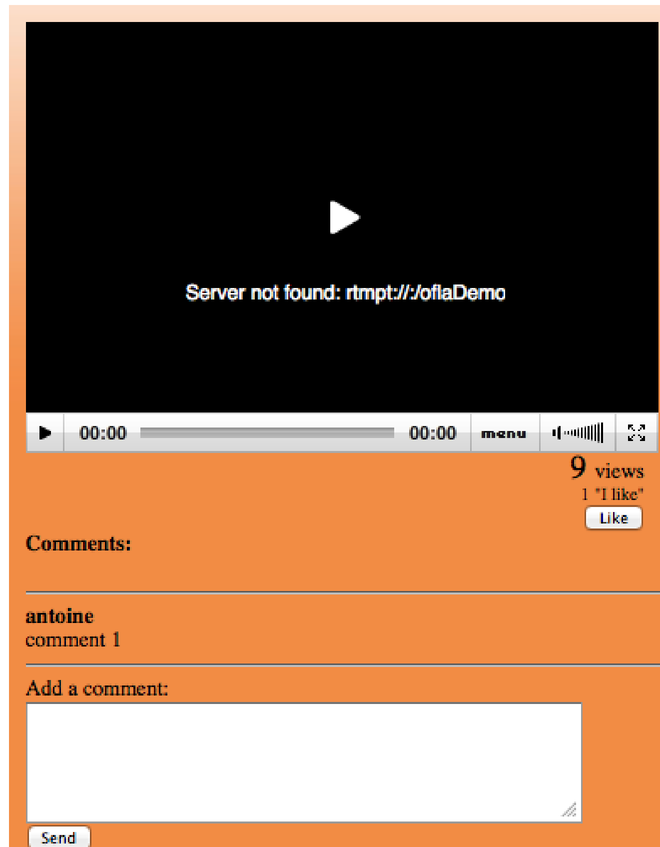

Figure 23: Streaming video player and comments

them. He can also change the group privacy.

5 Index Server Management

In order to keep persistent data, the metas have to be uploaded on different servers called index servers. The details of the index server system are described in the following sections.

5.1 Patterns and binary tree

Each index server stores one pattern. A pattern is a string of 36 characters (abcdefghijklmnopqrstuvwxyz0123456789) that represents a keyword. For example, if the metas has the following keyword: "Avatar", the pattern will be: "100000000000000000101010000000000000". Then, we can assign one different pattern to each server that will store

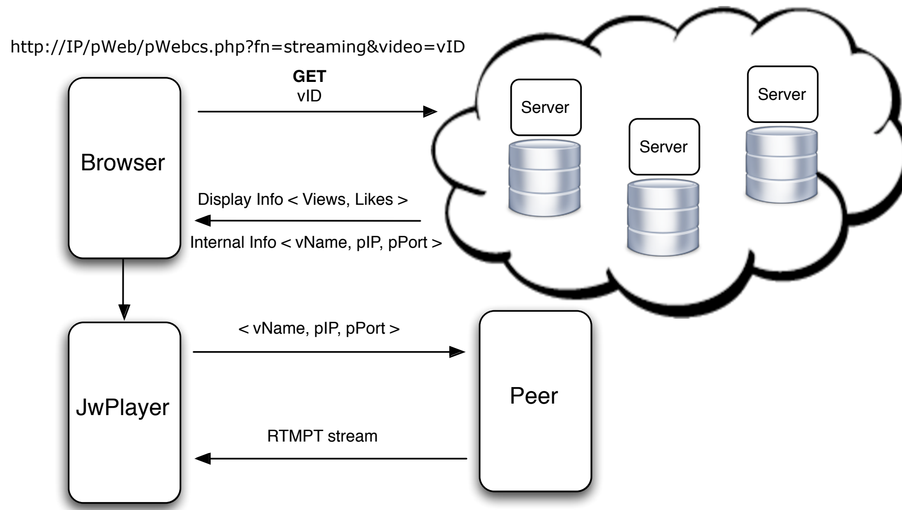

Figure 24: Video streaming message

Groupe name :
Description of the group :
Group privacy :
Private
Create a group

Figure 25: Create a group form

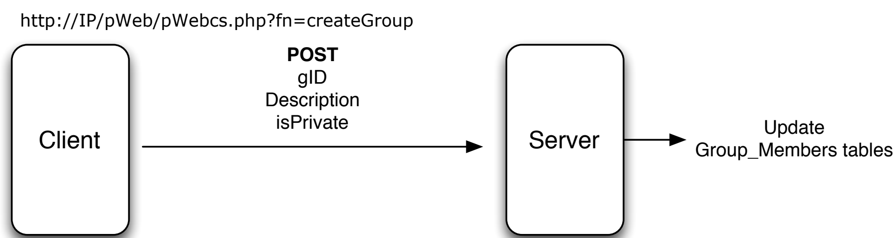

Figure 26: Group creation message

the corresponding metas. The servers are arranged as an ordered binary tree: In this example, the application uses five servers (full circles) and the binary tree has three levels. When a server is added, the binary tree becomes one level higher. The added server goes on the left (0) and the current one on the right (1). Then, a pattern is constructed by using the combination of the binary tree.

Figure 27: Join a group

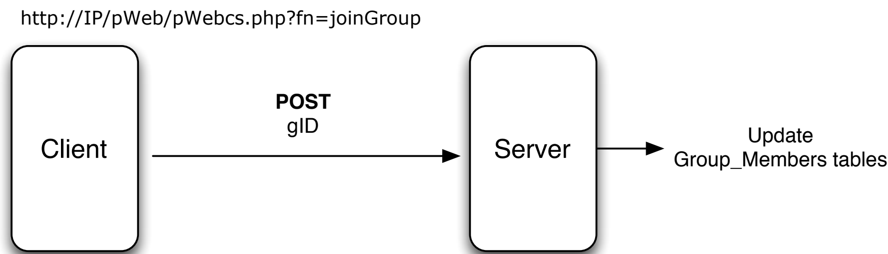

Figure 28: Group joining message

Figure 29: My groups menu

5.2 Routing Table

Each server has a routing table in its database. When a server is added, the Routing tables of the current server and the new one are updated. Each server has a direct link to the nearest ones. This table is described by the attributes below:

- **Level:** Level number of the server
- **Keywords:** Pattern stored in the server

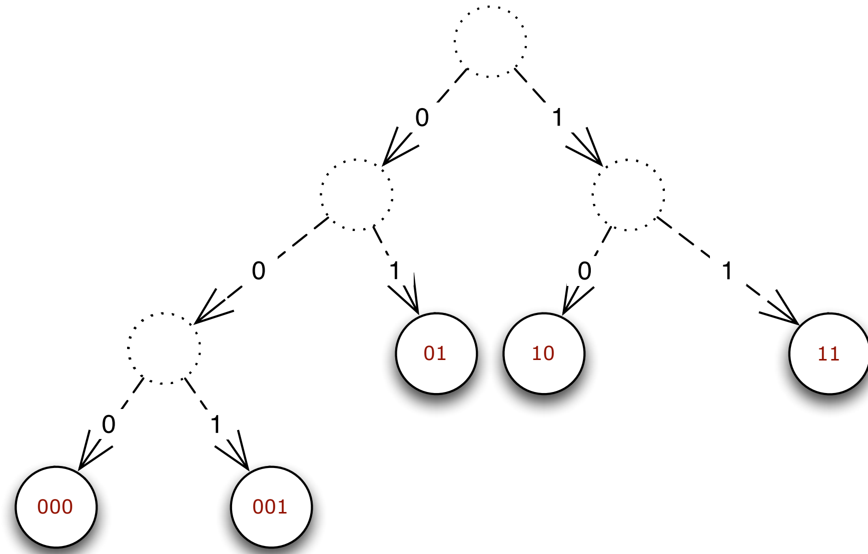

Figure 30: Example of an arrangement of indexing servers

- **Server IP:** IP adress of the linked server

5.3 Upload

The upload feature is now working differently. Indeed, the keyword of the uploaded metas is converted to a pattern. Then, by searching into the local server routing table, the query will be redirected to the index server where the pattern has the highest matching score. Then, the metas are inserted into the new server's Metas table. Also, the Uploads table of the local server is updated.

5.4 Search

This feature is also working differently. As explained in the previous section, the entered keyword is converted to a pattern. Then, the highest matching score algorithm finds the corresponding server. In order to provide the streaming of the video, the application is using the peer server IP attribute stored into the Metas table. By redirecting to this server, the application will find the IP and the Port of the Peer.

6 Java Component

Figure 31: Routing system with an upload

```
//creation of an URL
URL url = new URL("http://server_IP/pWeb/pWebcs.php?fn=check_login");
//creation of a connection using HTTPRequest
myConnection con = new myConnection(url);
//sending the logging informations
con.sendRequest(Username, Password);
con.sendRequest(deviceName);

//after login, go to Upload page
url = new URL("http://server_IP/pWeb/pWebcs.php?fn=uploadMeta");
//change the current url
con.setURL(url);

//creation of a DirectoryReader object
DirectoryReader dir = new DirectoryReader();
```

```
//list all the files in the directory
File[] files = dir.listFiles("/pWebVideos");

//creation of a myFile object
myFile fileList = new myFile();
//read the text file and store the checked files in an array
String[] read = fileList.read("/fileList.txt")
if(fileList.isNotEmpty()){
        foreach(file in files){
                foreach(read in fileList){
                        if(file.equals(read)) cpt++;
                }
                if(cpt==0){
                        fileList.write("/files.txt", file);
                        con.sendRequest(fileName, Description, Image);
                }
}else{
        foreach(file in files){
                fileList.write("/files.txt", file);
                con.sendRequest(fileName, Description, Image);
        }
}
```

Listing: Java component

The Java code checks the specified folder if new files have been added since the last check. If this is the case, then an HTTP request is sent to the server to upload the meta-information for the file. The name of the uploaded file is then stored in a text file that will be used during the next check.

7 Choice of Technology

A popular choice for video streaming server is Adobe Flash Media Server. But it comes with a exorbitant price. And its streaming protocol RTMPT (Real Time Messaging Protocol Tunneling) is a closed protocol. So, we have decided to choose Red5 streaming server, which is distributed under LGPL license. It is written in Java and the streaming protocol is open source. Its advantage is that is allows continuous diffusion to other

machines. In the front end we have used JwPlayer for playing streamed content.

This standalone server is going to encode the requested video for streaming and send it to the HTML page. Then, the user will be able to watch the video after a very short startup time. The video can be seeked to any time on the timeline. Here is the algorithm for the HTML page that invokes the JwPlayer and the video by the RTMPT protocol:

```
<script type='text/javascript' src='jwplayer.js'></script>

<div id='mediaspace'>Video</div>

<script type='text/javascript'>
  jwplayer('mediaspace').setup({
    'flashplayer': 'player.swf',
    'file': video_name.flv',
    'streamer': 'rtmpt://client_IP/video_directory,
    'autostart': 'true',
    'controlbar': 'bottom',
    'width': '470',
    'height': '320'
  });
</script>
```

Listing: Javascript code for invoking JwPlayer from a HTML page

8 Installation Manual

- Download and install XAMPP [41].
- After installation open XAMPP control panel and run MySQL and Apache.
- Extract the `pWeb.zip` file to the XAMPP's `htdocs` directory
- Run the `install.bat` file from the unzipped files
- Open a web browser and go to `http://localhost/phpmyadmin`
- Select the `pweb` database and then select the `routing` table
- Replace `127.0.0.1` with the IP address of your machine

- Open the `$XAMPP_HOME`[1]`/htdocs/pWeb/pWebcs.php` file and go to line 665.
- Replace `127.0.0.1` with the IP address of your machine, save and close the file.
- Open a web browser and go to `http://129.97.171.103/pWeb/pWebcs.php?fn=index` or `http://129.97.170.85/pWeb/pWebcs.php?fn=index`
- Select “Register” to create an account.
- Now Go to “Add Server” and fill the form with the IP address of your machine.

9 Summary

In this section we described the implementation of a pWeb prototype. The prototype functions as a decentralized video streaming system that supports uploading, searching and streaming videos. Users can host streaming content from their local machine by uploading the content's meta information to a set of indexing servers. Users can also search content by keyword. A search query is routed between the indexing servers using a simplified implementation of Plexus [2] targeted towards routing on names in Information Centric Networks [6]. For the prototype implementation each user runs Red5 video streaming server to stream their hosted content to other users.

References

[1] R. Ahmed and R. Boutaba. Distributed pattern matching for p2p systems. In *Network Operations and Management Symposium, 2006. NOMS 2006. 10th IEEE/IFIP*, pages 198 –208, 2006.

[2] Reaz Ahmed, Md. Faizul Bari, Shihabur Rahman Chowdhury, Md. Golam Rabbani, Raouf Boutaba, and Bertrand Mathieu. Route: A name based routing scheme for information centric networks. In *INFOCOM*, pages 90–94. IEEE, 2013.

[3] Reaz Ahmed and Raouf Boutaba. Distributed pattern matching: a key to flexible and efficient p2p search. *IEEE Journal on Selected Areas in Communications*, 25(1):73–83, 2007.

[4] Reaz Ahmed and Raouf Boutaba. Plexus: A scalable peer-to-peer protocol enabling efficient subset search. *IEEE/ACM Transactions on Networking (TON)*, 17(1):130–143, February 2009.

[1] $XAMPP_HOME is the XAMPP's installation directory

[5] Reaz Ahmed, Raouf Boutaba, Fernando Cuervo, Youssef Iraqi, Tianshu Li, Noura Limam, Jin Xiao, and Joanna Ziembicki. Service naming in large-scale and multi-domain networks. *IEEE Communications Surveys and Tutorials*, 7(1-4):38–54, 2005.

[6] Md. Faizul Bari, Shihabur Rahman Chowdhury, Reaz Ahmed, Raouf Boutaba, and Bertrand Mathieu. A survey of naming and routing in information-centric networks. *IEEE Communications Magazine*, 50(12):44–53, 2012.

[7] Md. Faizul Bari, Md. Rakibul Haque, Reaz Ahmed, Raouf Boutaba, and Bertrand Mathieu. Persistent naming for p2p web hosting. In Tohru Asami and Teruo Higashino, editors, *Peer-to-Peer Computing*, pages 270–279. IEEE, 2011.

[8] Md.Faizul Bari, Md.Rakibul Haque, Reaz Ahmed, Raouf Boutaba, and Bertrand Mathieu. A naming scheme for p2p web hosting. *Selected Areas in Communications, IEEE Journal on*, 31(9):299–309, 2013.

[9] Basex's website: http://basex.org/home/.

[10] Google Chrome. Npapi plugins.

[11] Ian Clarke, Oskar Sandberg, Brandon Wiley, and Theodore Hong. Freenet: A distributed anonymous information storage and retrieval system. In Hannes Federrath, editor, *Designing Privacy Enhancing Technologies*, volume 2009 of *Lecture Notes in Computer Science*, pages 46–66. Springer Berlin / Heidelberg, 2001. 10.1007/3-540-44702-4_4.

[12] Codeproject. How to attach to browser helper object (bho) with c# in two minutes.

[13] Codeproject. Mouse gestures for internet explorer.

[14] Bram Cohen. Incentives Build Robustness in BitTorrent, 2003.

[15] Microsoft Corporation. Browser helper objects: The browser the way you want it.

[16] Microsoft Corporation. Building browser helper objects with visual studio 2005.

[17] Microsoft Corporation. Spicie - simple plug-in creator for internet explorer.

[18] Frank Dabek, M. Frans Kaashoek, David Karger, Robert Morris, and Ion Stoica. Wide-area cooperative storage with cfs. In *Proceedings of the eighteenth ACM symposium on Operating systems principles*, SOSP '01, pages 202–215, New York, NY, USA, 2001. ACM.

[19] Giuseppe DeCandia, Deniz Hastorun, Madan Jampani, Gunavardhan Kakulapati, Avinash Lakshman, Alex Pilchin, Swaminathan Sivasubramanian, Peter Vosshall, and Werner Vogels. Dynamo: amazon's highly available key-value store. In Thomas C. Bressoud and M. Frans Kaashoek, editors, *SOSP*, pages 205–220. ACM, 2007.

[20] exist's website: http://exist-db.org/exist/index.xml.

[21] Mozilla Foundation. Activex control for hosting netscape plug-ins in ie.

[22] Mozilla Foundation. Liveconnect.

[23] M.R. Haque, R. Ahmed, and R. Boutaba. Qpm: Phonetic aware p2p search. In *Peer-to-Peer Computing, 2009. P2P '09. IEEE Ninth International Conference on*, pages 131 –134, sept. 2009.

[24] Jan Kneschke. Lighttpd.

[25] John Kubiatowicz, David Bindel, Yan Chen, Steven Czerwinski, Patrick Eaton, Dennis Geels, Ramakrishan Gummadi, Sean Rhea, Hakim Weatherspoon, Westley Weimer, Chris Wells, and Ben Zhao. Oceanstore: an architecture for global-scale persistent storage. *SIGPLAN Not.*, 35(11):190–201, November 2000.

[26] Hugo Leisink. Hiawatha.

[27] Linux inodes: http://www.linfo.org/inode.html.

[28] Petros Maniatis, David S. H. Rosenthal, Mema Roussopoulos, Mary Baker, Thomas J. Giuli, and Yanto Muliadi. Preserving peer replicas by rate-limited sampled voting. In Michael L. Scott and Larry L. Peterson, editors, *SOSP*, pages 44–59. ACM, 2003.

[29] Petar Maymounkov and David Mazires. Kademlia: A peer-to-peer information system based on the xor metric. In Peter Druschel, Frans Kaashoek, and Antony Rowstron, editors, *Peer-to-Peer Systems*, volume 2429 of *Lecture Notes in Computer Science*, pages 53–65. Springer Berlin / Heidelberg, 2002.

[30] Massimo Melina. Http file server (hfs).

[31] Athicha Muthitacharoen, Robert Morris, Thomer M. Gil, and Benjie Chen. Ivy: a read/write peer-to-peer file system. *SIGOPS Oper. Syst. Rev.*, 36(SI):31–44, December 2002.

[32] Orientx's website: http://idke.ruc.edu.cn/orientx/index.html.

[33] Qizx's website: http://www.xmlmind.com/qizx/.

[34] Russel Sandberg, David Goldberg, Steve Kleiman, Dan Walsh, and Bob Lyon. Design and implementation or the sun network filesystem, 1985.

[35] Sedna's website: http://www.sedna.org/.

[36] Nashid Shahriar, Shihabur Rahman Chowdhury, , Mahfuza Sharmin Reaz Ahmed, Raouf Boutaba, and Bertrand Mathieu. Ensuring β-availability in p2p social networks. In *5th International Workshop on Peer-to-peer Computing and Online Social Networks*, HotPOST 2013, 2013.

[37] Nashid Shahriar, Mahfuza Sharmin, Reaz Ahmed, Md. Mustafizur Rahman, Raouf Boutaba, and Bertrand Mathieu. Diurnal availability for peer-to-peer systems. In *CCNC*, pages 619–623. IEEE, 2012.

[38] K. Shvachko, Hairong Kuang, S. Radia, and R. Chansler. The hadoop distributed file system. In *Mass Storage Systems and Technologies (MSST), 2010 IEEE 26th Symposium on*, pages 1 –10, may 2010.

[39] Igor Sysoev. nginx.

[40] Wikipedia. C10k problem.

[41] Xampp's website: http://www.apachefriends.org/en/xampp.html.

[42] Xindice's website: http://xml.apache.org/xindice/index.html.