pWeb Peer-to-Peer Web Hosting Communication
System and Dynamic Web Hosting

Alexander Pokluda¹, Abdalla Aartai², Md. Faizul Bari¹, Reaz Ahmed¹, and Raouf Boutaba¹

¹David R. Cheriton School of Computer Science
University of Waterloo
[apokluda | mfbari | r5ahmed | rboutaba]@uwaterloo.ca

²American University of Beirut
aha53@aub.edu.lb


*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 August 2013.*
Abstract

During the main phase of this project we identified and provided solution for the major challenges related to peer-to-peer (P2P) web hosting. More specifically, our solutions for naming, availability, ranking, indexing and dynamic web-hosting have been provided in previous reports. In this report we present a more concrete view of the pWeb infrastructure and address the issues related to a real-life deployment of the system.

First we provide an overview of the light-weight P2P communication protocol between pWeb clients for signaling and media streaming in Part I. In Part II we provide an architectural overview of pWeb system and show the mapping of the abstract components in pWeb architecture to real life devices and technologies. In order to make pWeb compatible with the existing Web technology we have developed a DNS gateway. In Part II Section 3, we provide the design and implementation of this DNS gateway. For efficient indexing and fast discovery of end-user devices and multimedia content, we have developed a cloud-based solution for crawling, indexing and searching. We present this component in Part II Section 4.

As the project evolved, we felt the necessity of developing a working application prototype to demonstrate the effectiveness of the proposed architecture. Hence, we developed a Android application to capture and host videos from end user devices. Although this was not an original goal of this extension, we are including the design and implementation of this application in Part III of this report. We also present the installation and usage guides for various components in the pWeb architecture in Appendix A and Appendix B.
## Contents

### Part I: Peer-to-Peer Web Hosting Communication System

1 **Peering Protocol for Mobile Devices**  
   1.1 HTTP Server with Server-to-Server Peering Protocol for Mobile Devices  
      1.1.1 Introduction  
      1.1.2 Comparison of HTTP Server Implementations for Mobile Devices  
      1.1.3 lighttpd  
      1.1.4 mongoose  
      1.1.5 boost.asio  
      1.1.6 Android SDK  
      1.1.7 i-jetty  
      1.1.8 Other Servers from Deliverable 3  
      1.1.9 Summary  
   1.2 Peer-to-Peer Communication Protocol  
      1.2.1 Alternate Design Choices  
      1.2.2 Design Decision  
      1.2.3 HTTP Interface Between Home Agents and the Lower Tiers  
   1.3 Peer-to-Peer Messages between Tier-II and I devices  
   1.4 Architecture of Mobile App Peer-to-Peer Communication using Web Sockets

### Part II: Dynamic Web Hosting

2 **pWeb System Architecture**  
   2.1 Core Architecture  
   2.2 Device Naming Services Overview  
   2.3 DNS Gateway  
   2.4 Device Indexer  

3 **DNS Gateway**  
   3.1 Domain Name System (DNS) Overview  
      3.1.1 DNS Address Resolution Mechanism  
   3.2 pWeb Device Naming Integration with Existing DNS Infrastructure  
   3.3 Scalability  
   3.4 Design and Implementation  
      3.4.1 DNS Gateway to Home Agent Protocol  
      3.4.2 Configuration  
      3.4.3 Logging  
      3.4.4 Instrumentation  
      3.4.5 Known Issues  
   3.5 Installing the DNS Gateway from Source  
      3.5.1 Dependencies
# 3.5.2 Building

- 3.6 Installation as a System Service
- 3.7 Test Tools
- 3.8 Performance Evaluation
- 3.9 Summary

# 4 Device Crawler and Indexer

- 4.1 Home Agents
- 4.2 Crawlers and Manager
- 4.3 Solr Cloud
- 4.4 Web Front-End
- 4.5 Summary

## Part III: pWeb Client for Android and Demo

## 5 pWeb Client Application

- 5.1 Installation and Launching
- 5.2 Within the Client Application
- 5.3 Video Web Application

## Conclusion

## 6 Conclusion

## Appendix A: pWeb Component Installation and Configuration

- 1 pWeb Component Installation and Configuration
- 2 Installing and Configuring the pWeb Home Agent
- 3 Installing and Configuring the pWeb Registration Portal
- 4 Installing and Configuring the pWeb DNS Gateway and Crawler
- 5 Installing and Configuring the pWeb Search Database
- 6 Installing and Configuring the pWeb Website

## Appendix B: DNS Gateway to Home Agent Protocol

- 1 Current DNS Gateway to Home Agent Protocol
- 2 Proposed DNS Gateway to Home Agent Protocol
3 Report Produced by ResPerf

References
Part I:
Peer-to-Peer Web Hosting Communication System
1 Peering Protocol for Mobile Devices

1.1 HTTP Server with Server-to-Server Peering Protocol for Mobile Devices

1.1.1 Introduction

In our previous report, we presented a pWeb software implementation for use on desktop computers. This implementation was tailored for streaming large media files, such as movie trailers and feature-length films, in a peer-to-peer configuration. This implementation used the Lighttpd HTTP server together with PHP and MySQL to serve dynamic content combined with the Red5 streaming media server.

The pWeb project aims to create a peer-to-peer based platform for developing a new generation of scalable web applications. The number of global smartphone service subscribers is expected to reach 1.7 billion by the end of this year and nearly 4 billion by the end of 2017\(^1\). In developing countries, devices are frequently the only means individuals have of accessing the internet. It has been the goal of the pWeb project since the beginning to incorporate these devices into the pWeb network, allowing them to both consume and provide content and services. All modern mobile devices will be able to consume multimedia content served from the above mentioned pWeb software using HTML5 video over HTTP. However, smartphone and mobile devices have a very specialized application development and runtime environments. It is for this reason that we need to have a separate implementation of the pWeb software for providing content from mobile devices.

Our implementation of the pWeb software for mobile devices will also use HTTP for peer-to-peer communication. Mobile devices often connect to restrictive network where HTTP is the only protocol allowed to exit the network, often passing through a visible or transparent proxy server. The next section contains a comparison of HTTP servers that we considered for use on mobile platforms.

1.1.2 Comparison of HTTP Server Implementations for Mobile Devices

When evaluating each server, we focused primarily on the Android mobile platform. Android is currently the most popular mobile computing platform among end-users and held 69.7% market share in the fourth quarter of 2012\(^2\). Android is also the only leading mobile platform that is also open source, which has also make it extremely popular in the software development community.

1.1.3 lighttpd

Lighttpd is written in portable C code and is available for Android devices\(^3\). It would seem natural to use Lighttpd for the pWeb implementation on mobile platforms since we are using it for our implementation of the pWeb software on desktop platforms; however, the installation process is far from trivial. Lighttpd can only be installed on rooted Android devices, and requires significant technical knowledge.

---

1 Credit Suisse Research and Analytics Report: https://plus.credit-suisse.com/researchplus/ravDocView?docid=TpQjQj
2 http://www.gartner.com/newsroom/id/2335616
3 http://hex.ro/wp/blog/php-and-lighttpd-for-android/
1.1.4 mongoose

Mongoose is an extremely lightweight but full-featured web server written in portable C. Mongoose can be either embedded in another program or deployed as a standalone application. The processes of installing mongoose suffers from the same limitations as Lighttpd.

1.1.5 boost.asio

The Boost.Asio library is an asynchronous networking library that has been proposed for inclusion in the next official C++ standard library version. The documentation on the Boost website contains a very basic HTTP server that could be extended and incorporated into larger applications. Several adaptations of the Boost libraries are available for Android. Boost.Asio has been proven to be extremely fast and efficient and extending the sample HTTP server implementation would give us the most flexibility. This has both positive and negative aspects: because we would be implementing a lot of the web server functionality ourselves, we could ensure that it integrates well with the Android platform, including integration with Java software. On the other hand, implementing a lot of the web server functionality ourselves would require a large development effort.

1.1.6 Android SDK

The Android SDK contains Java classes for performing low-level and mid-level network operations directly. For example, there are classes for performing socket operations directly as well as classes for basic HTTP protocol handling. Developing the pWeb software for mobile platforms using this API would be similar to developing using Boost, however, the implementation would be simpler because there would only be one language involved instead of two.

1.1.7 i-jetty

i-jetty is a port of the very popular open source Jetty HTTP server and servlet container. Jetty is written in Java and is efficient, flexible, has a small footprint and supports all modern protocols used on the web. Jetty is used by major companies such as Google, which uses Jetty to power the Google AppEngine.

The fact that Jetty can be used as an HTTP server and Java servlet container gives it a lot of flexibility, especially on Android. A Java servlet is a Java application that has a well-defined interface and is run within the web server. A Java servlet on Android has access to the Android SDK API, making it fairly easy to write a server-side web program that has access to the mobile device features.

i-jetty is distributed in several forms. i-jetty is distributed as an Android Application Package (APK file) that can be installed from the i-jetty website or through Google Play, the official software repository for Android devices. Once installed, i-jetty provides an interface that allows users to download and install webapps (another name for servlets) from anywhere on the internet. Alternatively, the i-jetty source code can also be downloaded from the i-jetty website, which can be repackaged and distributed with bundled servlets.
1.1.8 Other Servers from Deliverable 3

We considered using the servers discussed in Deliverable 3, Part III, Section 4 but found that none of these servers were suitable for use on mobile devices. Cherokee, Hiawatha and Nginx do not provide versions for mobile platforms. All three are programmed in C and would have to be ported manually to each mobile platform, which is both challenging and time consuming. HFS is not suitable for use on mobile platforms due to the limitations that are discussed in Deliverable 3.

1.1.9 Summary

Based on the above reasoning, we decided to proceed to develop the pWeb software for mobile devices using i-jetty. We also evaluated Tiny Java Web Server and Servlet Container (also known as Miniature JWS) and NanoHTTPD but decided not to use them due to their lack of maturity.

1.2 Peer-to-Peer Communication Protocol

In this section, we present the peer-to-peer communication protocol. The P2P communication has three faces. First, devices in the topmost tier (Tier-III), i.e., the home agents, communicate between themselves to provide name resolution service to the lower tier devices. Second, the lower tier (Tier-II and I) devices communicate with the home agents for the naming service. Finally, the lower tier devices communicate between themselves for content exchange. We first discuss the alternate design choices for implementing the protocol (Section 1.2.1). Then we present the HTTP based communication protocol between the home agents and the lower tier devices (Section 1.2.3) that we have implemented. Finally, we describe the peer-to-peer messages between the lower tier devices.

1.2.1 Alternate Design Choices

To implement the two types of P2P messages, we considered the possibility of taking the following three approaches:

**Clean Slate over TCP**  TCP provides a reliable end-to-end communication protocol between hosts. It is possible to build any application layer protocol on top of TCP from scratch. This gives us the flexibility to define our own messages and apply possible optimization in terms of reducing message size, message transfer etc. But the main overhead is we need to implement each of the messages from the scratch, and more effort is required to extensively test the application layer protocol.

**Communication over HTTP**  HTTP is a widely used request-reply communication protocol mainly used for file transfer. However, its usage is not only limited within file transfer domain. HTTP is being extensively used in RESTful applications as a message passing interface. HTTP already has a rich set of protocol messages and error codes. It can be easily exploited to transfer messages via HTTP. The drawback of using HTTP is that we cannot optimize message lengths, since the headers are already defined. Also, we shall need at least one webserver at one communication end point for HTTP communication. In that case, the devices need to run at least a minimal functionality HTTP web servers for P2P communication.
BitTorrent Protocol  BitTorrent is a protocol for distributing files in a peer-to-peer fashion. Essentially, it is a file transfer protocol like HTTP. But the major advantage of BitTorrent over HTTP is that when a chunk of a file is downloaded from a source, the downloaded chunk also becomes a source for other downloads in the network. Which increases the file download throughput, file availability and enables support for a large number of simultaneous download without imposing heavy load on the sources. BitTorrent has a number of peer-to-peer messages defined. A subset of these messages to manipulate user’s download rates, a subset of messages to express interest for contents, and a subset of message related to the actual transfer of files. However, BitTorrent does not have any messages for replication group formation, since it does not have the concept of replication groups. In our case, we have the concept of forming $\beta$-availability groups among the peers. So, we need to have some messages for group formation and content replication among peers. Adopting to BitTorrent’s protocol messages will allow us to implement peer-to-peer data transfer functionality, but we shall need to implement an additional set of messages for the replication protocol.

1.2.2 Design Decision

As described earlier, the P2P communication has three faces: (a) communication between the Tier-III devices, i.e., the home agents. (b) communication between the lower tier devices and the home agents (c) communication between the lower tier devices for group formation, content replication and content access. We have chosen HTTP as the communication protocol for (b) and (c). The rational behind using HTTP for (b) and (c) is that, in order to host web contents the devices will be already running some lightweight HTTP webserver. So, we can easily add some module in the lightweight HTTP servers to enable peer-to-peer communication between peers. Also, to keep things simple we used HTTP for (b) as well. However, we optimized the messages transferred between home agents by implementing that part of the communication as a clean slate protocol over TCP.

1.2.3 HTTP Interface Between Home Agents and the Lower Tiers

In this section, we present the HTTP interface of the home agents for the lower tier devices. The HTTP interface can be accessed by making a HTTP GET request with a specially formatted URL as will be described later. The HTTP GET request returns status (success or failure) and error text (in case of failure) indicating the state of the execution of that particular request. Table 1 gives a list of general error codes.

<table>
<thead>
<tr>
<th>HTTP Error Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>200</td>
<td>Action Successful</td>
</tr>
<tr>
<td>400</td>
<td>Malformed request</td>
</tr>
<tr>
<td>405</td>
<td>Invalid (or no) method name</td>
</tr>
<tr>
<td>451</td>
<td>At least one of the parameters are missing</td>
</tr>
</tbody>
</table>

Table 1: General HTTP Return Code and Description for HTTP Interface

REGISTER_USER
Purpose  Register a user for the first time with the home agent.

Example  Send a HTTP GET request to any home agent in the following format:
<home_agent_name>:20005/?method=registeruser&name=<user_name>&
password=<password>&email=<email>&full_name=<full_name>&country=<country>&affiliation=<affiliation>

It is to be noted that the query string is case sensitive.

Return Value  On success returns {"status":"success"} and on failure returns {"status":"error","message":"<error text>"}.

EXIST_USER

Purpose  To check whether a user name exists or not.

Example  Send a HTTP GET request to any home agent in the following format:
<home_agent_name>:20005/?method=existusername&name=<user_name>

It is to be noted that the query string is case sensitive.

Return Value  If the user name is available then returns {"status":"success","available":"yes"}. If the user name is not available then returns {"status":"success","available":"no"}. In case of a failure returns {"status":"error","message":"<error text>"}.

AUTHENTICATE_USER

Purpose  To validate a user’s credentials.

Example  Send a HTTP GET request to any home agent in the following format:
<home_agent_name>:20005/?method=authenticate&name=<user_name>&password=<password>

It is to be noted that the query string is case sensitive.

Return Value  If the user name and password pair is valid then returns {"status":"success","authneticated":"yes"}. If the user name and password pair is not valid then returns {"status":"success","authneticated":"no"}. In case of a failure returns {"status":"error","message":"<error text>"}.

GET_DEVICE_LIST

Purpose  Returns a list of devices registered by a particular user.

Example  Send a HTTP GET request to any home agent in the following format:
<home_agent_name>:20005/?method=getdevicelist&username=<user_name>

It is to be noted that the query string is case sensitive.
**Return Value**  Returns a list of devices registered by user with specified user name. In case of a failure returns \{"status":"error","message":"<error text>"\}.

**REGISTER_DEVICE**

**Purpose**  Register a device belonging to a specific user.

**Example**  Send a HTTP GET request to any home agent in the following format:
<home_agent_name>:20005/?method=register&name=<devicename.username>
&port=<port-number>&type=<device_type>&ip=<ip>&os=<OS>&description=<description>&is_publicly_indexed=is_publicly_indexed>

It is to be noted that the query string is case sensitive.

**Return Value**  On success returns \{"status":"success"\} and on failure returns \{"status":"error","message":"<error text>"\}.

**MODIFY_DEVICE**

**Purpose**  Modify information about a particular device.

**Example**  Send a HTTP GET request to any home agent in the following format:
<home_agent_name>:20005/?method=modifydevice&Oldname=<old_device_name.user_name>&newname=<new_device_name.user_name>&port=<port>&
public_folder=<public_folder>&private_folder=<private_folder>

It is to be noted that the query string is case sensitive.

**Return Value**  On success returns \{"status":"success"\} and on failure returns \{"status":"error","message":"<error text>"\}.

**EXIST_DEVICE**

**Purpose**  To check whether a device name exists or not.

**Example**  Send a HTTP GET request to any home agent in the following format:
<home_agent_name>:20005/?method=existdevice&device_name=<devicename.username>

It is to be noted that the query string is case sensitive.

**Return Value**  If the device name is available then returns \{"status":"success","available":"yes"\}. If the device name is not available then returns \{"status":"success","available":"no"\}. In case of a failure returns \{"status":"error","message":"<error text>"\}.

**UPDATE**

**Purpose**  Update the location, i.e., IP:port corresponding to a device name.
<table>
<thead>
<tr>
<th>HTTP Error Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>495</td>
<td>Invalid timestamp (zero or negative)</td>
</tr>
</tbody>
</table>

Table 2: HTTP Return Code and Description for GET_ALL Method

**Example**  Send a HTTP GET request to any home agent in the following format:

```
<home_agent_name>:20005/?method=update&name=<device_name>&port=<port_number>
```

It is to be noted that the query string is case sensitive.

**Return Value**  On success returns 

```
{"status":"success"}
```

and on failure returns 

```
{"status":"error","message":"<error text>"}
```

**GET_ALL**

**Purpose**  Get the name of neighbouring home agents as well as all the names saved in this home agent.

**Example**  Send a HTTP GET request to any home agent in the following format:

```
<home_agent_name>:20005/?method=getall&timestamp=<time_stamp>
```

**Return Value**  Error code and text are returned as HTTP error code and text, respectively. The details of the return code are given in Table 2.

**Return Data**  The Home Agent will return a list of its neighbours in the Plexus overlay network and a list of meta information for devices that have been created or modified since the timestamp passed as a parameter in the URL in a pseudo-XML format. Sample output is provided below. The `<neighbours>` contains zero or more `<home agent>` elements and the `<devices>` element must contain zero or more `<device>` elements. All of the other elements will always be present and listed in the order shown. If a particular element has no value, then the tags will still be present, eg. `<tag></tag>`). This is to simplify parsing.

The `<content updates>` section lists all the devices that have pushed content metadata to the Home Agent since the point in time represented by the timestamp parameter.

The first time that a crawler makes a getall request, it will pass a timestamp of zero and the home agent will return a list of all devices that it knows about. The `<content updates>` section will list all devices known to the Home Agent that have pushed content metadata to the home agent.

When the timestamp is zero, this element will contain a comma separated list of all the devices that the Home Agent is storing metadata for. The Crawler will iterate through this list and issue one HTTP GET request for each device in this list using the getcontent method described below.

```
<getall>
<name>mypc0</name>
<neighbours>
<home agent>
```

12
<hostname>localhost</hostname>
<port>20005</port>
</home agent>
</neighbours>
<devices>
<device>
<owner>Faizul Bari</owner>
<name>nexus2.faiz</name>
<port>12345</port>
<timestamp>1368462463</timestamp>
<location>Waterloo, ON, Canada</location>
<description>my first android phone...</description>
</device>
<device>
<owner>Faizul Bari</owner>
<name>laptop.faiz</name>
<port>12345</port>
<timestamp>1369087107</timestamp>
<location>Waterloo, ON, Canada</location>
<description>asd</description>
</device>
<device>
<owner>Faizul Bari</owner>
<name>desktop.faiz</name>
<port>12345</port>
<timestamp>1369142462</timestamp>
<location>Waterloo, ON, Canada</location>
<description>asdasd</description>
</device>
</devices>
<content updates>nexus2.faiz,laptop.faiz,desktop.faiz</content updates>
</getall>

1.3 Peer-to-Peer Messages between Tier-II and I devices

These messages will be implemented as a payload over HTTP protocol. The peers will run lightweight HTTP servers to host contents. This message passing module will be an addon with the lightweight HTTP web server in the end devices.

1.3.1 Availability and Group Formation

All peer-to-peer availability messages are relayed through the group management overlay and delivered asynchronously. Messages destined for off-line peers will be held by the group management overlay and delivered when the destination peer returns to the on-line state.

AVAILABILITY_ADVERTISEMENT
**Purpose**

This message is sent from a peer to the group management overlay when it first joins the network and on a periodic basis (with updated availability).

**Parameters**

- $S$ = The availability vector of the peer (an array of probabilities)
- $A$ = The binary encoded availability vector of the peer
- $ID$ = An ID uniquely identifying the peer sending the availability advertisement
- $Loc$ = Network location of the peer (IP address and port number)
- $IDg$ = A unique value representing the group the peer belongs to
- $Ig$ = The index location of the group that the peer belongs to or NULL if the peer does not belong to any group at the time of sending the message

**CREATE_GROUP**

**Purpose**

sent from a peer to the group management overlay in order to create a group.

**Parameters**

- $IDg$ = A unique value representing the group the peer belongs to

**GROUP_INVITATION**

**Purpose**

The group management overlay will forward the invitation to a peer with an availability vector similar to the requested availability vector.

**Parameters**

- $S$ = The availability vector of the peer (an array of probabilities)

**JOIN_GROUP**

**Purpose**

This messages causes the group management overlay to update its records to indicate that the new peer has joined the group and then it will notify the inviting peer that a new member has joined the group.

**Parameters**

- $IDg$ = A unique value representing the group the peer belongs to
- $ID$ = An ID uniquely identifying the peer sending the availability advertisement

**LEAVE_GROUP**
**Purpose**  This message causes the group management overlay to update its records to indicate that the peer has left the group and then it will notify the inviting peer that a member has left the group.

**Parameters**

IDg = A unique value representing the group the peer belongs to

ID = An ID uniquely identifying the peer sending the availability advertisement

### 1.4 Architecture of Mobile App Peer-to-Peer Communication using Web Sockets

pWeb is a platform for developing and providing next generation, scalable peer-to-peer web applications. As we have already demonstrated, this framework can be used for peer-to-peer streaming of multimedia content. To demonstrate the pWeb functionality on mobile devices, we have chosen to implement a less resource intensive service—a peer-to-peer instant messenger. The general architecture of the pWeb software implementation for mobile platforms as well as specific details about the instant messenger application are discussed below.

In the pWeb implementation for desktop computers, the primary way in which a user would use the pWeb software is through a web browser. On mobile computing platforms, such as Android, however, it is less common for users to use web browsers. Often users will expect to use an app for tasks that they would have used a web browser for on a desktop or laptop computer. As one example, Facebook does not provide a desktop application and must be used through a web browser on desktop computers, but when accessing the Facebook website on a mobile device, the user is encouraged to download and use the mobile app instead.

Many mobile apps are developed using web technologies such as HTML5 and most use HTTP for network communication. Thus, mobile apps can be developed using standard web development tools and function much like a web page. Apps can access pWeb services by sending HTTP requests to the loopback or localhost address just as a web browser would. Bidirectional communication can be established between two applications in pWeb by using two concurrent HTTP or XMLHttpRequests (one for sending and one for receiving), or by using websockets.

As shown in Figure 1, the pWeb software core functionality will be contained in a Java servlet running in i-jetty. It is possible to run any number of servlets in i-jetty and we may have additional or optional pWeb services running in a second servlet. All core operations will be handled by the pWeb Core servlet. For example, a user can request a web page from a remote peer by entering a URL of the form http://localhost/mobile.alice.uw/homepage.html where mobile.alice.uw is a device name that will be resolved to an IP address. The HTTP request generated by the web browser will be received by i-jetty and forwarded to the pWeb Core servlet. The pWeb Core servlet will query the Plexus overlay in Tier III in order to determine the IP address of the remote peer and forward the HTTP request to the i-jetty instance on the remote peer. The remote i-jetty instance will return the resource to the local i-jetty instance that will return the content to the user's browser.

We are building a chat application in order to demonstrate how the pWeb software for mobile devices will work. The chat application will be a mobile app that run outside of the web browser.

---

4http://developer.android.com/training/basics/network-ops/connecting.html
Figure 1: pWeb software architecture for mobile device implementation

Figure 2: Sequence diagram for sending a chat message from one device to another
but communicates with the local i-jetty server as described above and shown in Figure . The chat app on device 1 will send an HTTP post request containing the message to the local pWeb Core servlet using a URL of the form \texttt{http://localhost/mobile.alice.uw/chat/send_message} (the URL is shown as a function call in the sequence diagram for simplicity). The pWeb Core module will query the local device’s home agent to resolve the remote device name, \texttt{mobile.alice.uw}, to a valid IP address. The module will then forward the HTTP post request to the pWeb Core module which will return the message to the chat app on the remote device via an outstanding XMLHttpRequest.
Part II:
Dynamic Web Hosting
2  pWeb System Architecture

2.1  Core Architecture

The core of the pWeb system is a three tier architecture as shown in Figure 3. Nodes in Tier III perform distributed name resolution and indexing of devices in the pWeb network while devices in Tier I and Tier II both provide and consume application level content and services.

Nodes in Tier III are also known as pWeb Home Agents because they perform a role similar to that of the home agent in Mobile IP, an Internet Engineering Task Force standard communications protocol that is designed to allow mobile device users to move from one network to another while maintaining a permanent IP address [19, 20]; however, rather than provide devices with a permanent network address, pWeb Home Agents provide users with permanent device names that can be used to access services from the device regardless of the device’s physical location and network address, similar to dynamic updates in the domain name system [26].

The infrastructure for Tier III consists of stable nodes at well known network locations. It is expected that organizations, institutions and individuals will contribute heterogeneous cloud resources to this tier by hosting instances of the pWeb Home Agent software in order to enable members of the organization or public at large to participate in the pWeb network. This is the same principle that is used by other similar and successful services such as the Jabber Instant Messenger service [13] and other services based on XMPP [12].

In addition to providing device name registration and network address resolution to devices in the pWeb network, the Home Agents make certain information available via an HTTP interface to enable the development of external applications. The Device Indexer, discussed shortly, makes use of this feature.

Content and services in pWeb are provided by nodes in Tier I and Tier II that also participate in a peer-to-peer network. In general, these nodes are assumed to have shorter but regular uptimes and less bandwidth. Nodes in this tier may be servers, desktop computers, laptops or even mobile devices. A user publishes content using the client software running on one of these nodes. The client software will push the content meta-data, such as content name and location, to the cloud hosted indexing infrastructure in Tier III. The content itself will remain on the user’s machine and be made publicly accessible over HTTP.

To ensure that content remains accessible despite peer churn and the limited resources of individual peers, the nodes in Tier II will be organized into replication groups. Each member of a replication group will mirror the content of the original publisher and make it publicly available as well. Nodes in Tier I are primarily publishers and consumers of content and use the public services offered by Tier I and Tier II to publish and access it.

2.2  Device Naming Services Overview

The Home Agent software is based on Plexus [1, 3, 4]. Plexus is a peer-to-peer search protocol and distributed hash table (DHT) that provides an efficient mechanism for advertising a bit sequence and discovering it using any subset of its 1-bits. The partial matching capability of Plexus can also be extended for routing on names [2] in Information Centric Network [6]. When a new pWeb Home Agent is brought online and joins the Tier III overlay network, it inserts its own name and network addresses into the distributed hash table. For example, a Home Agent hosted by the University of Waterloo would insert a record that is conceptually equivalent to the record shown in Listing 1. The
Name field is a label, similar to a hostname, that uniquely identifies the Home Agent in the pWeb network. A label consists of a single word that may contain only the ASCII letters ‘a’ through ‘z’ in a case insensitive manner, the digits ‘0’ through ‘9’ and the hyphen ‘-’.

pWeb device names consist of three labels separated by periods, such as mobile.alice.uw. The leftmost label is assigned to a device and chosen by the device’s owner. The middle label is the is chosen by and identifies the user, and the rightmost label identifies the home agent as mentioned above.

Registered device names and IP addresses are stored locally at the Home Agent where the device name was registered. This reduces the amount of data that must be stored in the DHT and makes device name registration and update operations much faster.

Home Agents also have the responsibility of resolving device names to network addresses, such as an Internet Protocol (IP) address, for their users. When a Home Agent receives a request to resolve a device name to network address, such as desktop.bob.ut, that is not registered locally, it extracts the Home Agent label from the device name, ut in this case, and looks up the network address for the remote Home Agent in the DHT. The local Home Agent then contacts the remote Home Agent and asks for the device’s current network address, which it returns to the requester. Both the remote Home Agent’s network address and the remote device’s address will be cached.
at the local Home Agent for an appropriate period of time in order to serve future requests more quickly and reduce the load on the network.

This architecture for device name resolution and was chosen to support the explosive growth in smartphone services subscribers that is expected over the next few years. Recent forecasts by Credit Suisse, one of the world’s largest and most profitable banks, predict that the number of global smartphone subscribers will reach 1.7 billion by the end of 2013 and nearly four billion by the end of 2017 [25]. A full analysis of the scalability of the device naming and Home Agent architecture is out of scope for this report, however.

2.3 DNS Gateway

Group formation and content replication in Tier II are not included in the fundamental feature set implemented in the current implementation of the pWeb system. Instead, devices in Tier I and Tier II are assumed to be always online and have sufficient bandwidth to serve the published content. Tier III will allow users to register and resolve device names in order to share and access content. Users will be able to register names for Tier I and Tier II devices and update their network addresses using a name updater application that will function similar to a Dynamic DNS client [26] and will serve content using a standalone HTTP server. Tier III devices will use a standard, unmodified web browser to access content on the pWeb network. Names will be resolved by the web browser using a DNS compatible interface provided by a DNS Gateway that acts as an intermediary between the traditional domain name system (DNS) infrastructure and the Home Agents in Tier III of the pWeb architecture. This architecture is depicted in Figure 4.

Here content served by the device named nexus.alice.uw can be accessed by an unmodified web browser using the domain name nexus.alice.uw.dyngw.org, or the domain name nexus.alice.uw.dht.
if the local network has been configured to support the .dht pseudo top level domain that will be explained in Section 3. In the future, applications developed specifically for use with pWeb will have the ability to resolve device names to network addresses by contacting the device’s Home Agent directly, bypassing the domain name system infrastructure; however, it is expected that the DNS Gateway will remain a critical piece of the pWeb infrastructure because the DNS Gateway enables seamless integration between pWeb and traditionally hosted web content. The DNS Gateway also enables pWeb content to be linked to by traditional hosted websites and web search engines to index content on pWeb, providing a unified browsing experience for the end user.

2.4 Device Indexer

The as previously mentioned, Home Agents in Tier III will make certain information available publicly over HTTP using a representational state transfer (REST) architecture [28]. Currently all registered device names and services offered by each device and the Home Agent’s neighbours in the Plexus peer-to-peer network are published using this interface. This information is made available to enable the development of internal and external applications that contribute to the functionality of the pWeb system.

The first application to take advantage of this interface is a Device Indexer that is being developed as part of the pWeb project. The Device Indexer itself consists of three main components: crawler and manager processes, a searchable database, and a web service front-end. Multiple crawler processes, coordinated by a manager periodically poll each Home Agent’s REST interface to discover all device names registered in the pWeb network. The device names are inserted into a searchable database that is made accessible to users through a web interface and other applications through its own REST interface. The Device Indexer and searchable database are represented by the Central Search in the top right corner of Figure 3.

3 DNS Gateway

As mentioned in the introduction, the pWeb DNS Gateway in an important piece of the pWeb infrastructure. It provides a seamless integration of pWeb and traditionally hosted web content and is the primary method of resolving a device’s permanent name to its current network address in the current implementation of pWeb.

The pWeb device naming system has been designed to support billions of devices. Naturally, when using the domain name system as an interface to the pWeb network, we must consider the scalability of this interface. There are two aspects that must be taken into consideration: the load that is placed on the existing domain name system infrastructure before queries reach the DNS Gateway, and the scalability of the DNS Gateway itself. The domain name system infrastructure was not designed for host mobility, but previous studies have shown that the domain name system can be used as a location repository for mobile devices without placing undue stress on the global root nameservers [29, 30]. Thus, using the domain name system as an interface to the pWeb naming system should not place appreciable load on the root servers. The analysis of the scalability of the DNS Gateway itself is presented in Section 3.3, but first an overview of the domain name system is

---

5 We plan to add extensive security and privacy controls to the Home Agent software that will enable users to strictly control the information that is made publicly available [5, 7, 8]; however, the current version of the software makes all device names and last update times available publicly.
3.1 Domain Name System (DNS) Overview

The domain name system is a hierarchical distributed database that provides a naming system for computers and resources connected to the Internet. The domain name system associates human readable names with various record types. The predominant use of the domain name system—at least for human users—is to translate mnemonic names to resource Internet Protocol (IP) addresses. A domain name that is associated with at least one IP address is commonly referred to as a hostname.

The domain name system distributes the responsibility of assigning domain names and mapping domain names to network addresses to authoritative nameservers in each domain. The authoritative servers may in turn delegate responsibility for subdomains to other authoritative nameservers. The domain name system specifies the technical functionality of the database as well as the data structures and protocols for communication between domain nameservers and clients, and the algorithms that a nameserver will use to respond to queries against its database.

The domain namespace takes the form of a tree structure. Each node in the tree has zero or more resource records associated with it that associate specific information with the domain name. The tree structure subdivide domains into administrative zones, starting at the root zone. Each zone contains at least one domain and zero or more subdomains.

Domain names consist of a series of labels containing ASCII letters, digits and hyphen characters each terminated with periods, also known as dots. The last dot is frequently omitted. The exact rules for forming domain names are defined in [18, 9, 10]. An example of a domain name is www.pwebproject.net. The rightmost label, net, is known as the Top Level Domain (TLD). The domain pwebproject is a subdomain of net and www is a subdomain of pwebproject. A subset of the domain namespace that shows the administrative zones for the domain name www.pwebproject.net is shown in Figure 5.

Figure 5: A subset of the domain namespace showing administrative zones presented in Section 3.1 and a detailed description of pWeb device naming with the existing DNS infrastructure is given in Section 3.2.
3.1.1 DNS Address Resolution Mechanism

The operating system on and end host such as a laptop or desktop personal computer or Internet server typically contains a stub resolver that resolves hostnames to network addresses on behalf of applications on the machine. The stub resolver offloads most of the work of resolving a name to a recursive DNS resolver (also know as a nameserver) that is hosted in the local network or in the Internet Service Provider’s network. The recursive DNS resolver uses the following algorithm to resolve a domain name, which is depicted in Figure 6. Note that the recursive DNS resolver performs a recursive operation from the client’s perspective but actually queries other nameservers in an iterative manner.

1. When the recursive DNS resolver receives a query asking for the IP address corresponding to a domain name such as `www.pwebproject.net` it will look in its cache for the address of one of the root nameservers and forward the the query for `www.pwebproject.net` to the root nameserver. The cache is initially populated with a seed file containing the addresses of all known root nameservers and is periodically updated by system administrators from a reliable source.

2. The root nameserver will reply that it does not know the answer to the query but the authoritative nameserver for the `net` domain might. Thus, the root nameserver delegates responsibility to the subdomain `net` that is in a different administrative zone.

3. The recursive DNS resolver forwards the query for `www.pwebproject.net` to one of the authoritative nameservers for the `net` domain.

4. The authoritative nameserver for the `net` domain will reply that it does not know the answer to the query but the authoritative nameserver for the `pwebproject` domain might.
5. The recursive DNS resolver forwards the query for `www.pwebproject.net` to one of the authoritative nameservers for the `pwebproject.net` domain.

6. The authoritative nameserver for the `pwebproject.net` domain will respond with the answer to the query. Note that the responsibility for the `www` subdomain is not delegated in this case. The `www.pwebproject.net` domain is in the same administrative zone as the `pwebproject.net` domain. Whether or not responsibility for a subdomain is delegated to another administrative zone is decided by the administrator of the parent domain.

In reality the algorithm as presented above would overload the nameservers near the root of the DNS namespace. In order to reduce the load on these servers and improve the look-up time for subsequent requests, caching is employed at each stage in the name resolution process.

The delegation of responsibility from one administrative zone to another is achieved by an authoritative nameserver in the parent domain returning a nameserver record (NS) containing the addresses of authoritative nameservers in the subdomain.

### 3.2 pWeb Device Naming Integration with Existing DNS Infrastructure

The pWeb DNS Gateway supports deployment in two main configurations. It can be deployed using a gateway domain or using a pseudo-TLD. Both deployment configurations are described below. In each case, a DNS domain name is appended to the pWeb device name before the domain name system is used to resolve the device name to a network address. For example, when a gateway domain is used, the gateway domain is appended to the device name. Suppose an application needs to resolve the pWeb device name `mobile.alice.uw` to its current IP address and is using gateway domain `dht.pwebproject.net`. The application then performs a standard DNS query for the pseudo-domain name `mobile.alice.uw.dht.pwebproject.net`. The application then performs a standard DNS query for the pseudo-domain name in order to retrieve the device’s current IP address.

Alternatively, the application could be configured to use a pseudo-TLD such as `dht` in order to resolve device names their current IP addresses. In this case, an application proceeds in the same manner by appending the TLD to the device name and performs a DNS query for the result. The difference is how the query is handled by the recursive nameserver handling the application’s request. In either scenario, the query is eventually forwarded to a pWeb DNS Gateway server.

The method an application uses to determine whether a gateway domain or pseudo-TLD should be used for resolving pWeb device names is out of the scope of this report. However, one approach that could be adopted by applications designed specifically for use with pWeb is to perform a query using a pseudo-TLD, and if that fails, retry the query using a well-known gateway domain.

**Integration Using a Gateway Domain**  
In order to set up a pWeb gateway domain, the parent of the gateway domain must be configured to delegate queries for the domain to one or more DNS Gateway servers. This is achieved by configuring NS records in the parent domain.

The domain `dht.pwebproject.net` is currently set up as a DNS gateway to the pWeb naming system and is available for public use. It will will be used as an example in this section to explain how a gateway domain works. The domain `pwebproject` has been registered as a subdomain of

---

6. Here the domain name `mobile.alice.uw.dht.pwebproject.net` is referred to as a *pseudo-domain name* because it is not a domain name as defined by the domain name system.
the **net** top level domain\(^7\). As part of the registration process, NS records for the **pwebproject** subdomain were created in the **net** domain, delegating responsibility for queries to authoritative DNS servers in the **pwebproject.net** authoritative zone.

Each of the authoritative nameservers for the **pwebproject.net** domain are in turn configured with NS records that delegate responsibility for the **dht.pwebproject.net** subdomain to DNS Gateway servers\(^8\). DNS resource records, such as NS records, have a standard textual representation that is defined in [18]. The textual representation of the NS records for the **dht.pwebproject.net** domain are shown in Figure 2. Lines 1-4 Figure 2 show that there are two authoritative nameservers for the **dht.pwebproject.net** domain at the IP version 4 addresses 129.97.26.34 and 129.97.26.35, denoted by A records. When a DNS query arrives at a DNS Gateway server, the gateway and one or more Home Agents will find the device’s current IP address and the gateway will return the result as a DNS reply. The procedure that the DNS Gateway and Home Agents use to process a query is outlined below and depicted in Figure 7.

1. When the DNS Gateway receives a query, it checks that the domain being queried matches its configuration. If it does not match, the gateway returns an error. Next the device name is extracted and a query for the device’s current IP address is sent to a Home Agent in the pWeb network. The DNS Gateway to Home Agent protocol is discussed in Section 3.4.1. The DNS Gateway must be configured with the hostname of one or more Home Agents. The DNS Gateway configuration is discussed in Section 3.4.2.

2. When the Home Agent receives the query, it extracts the device’s Home Agent identifier from the device name. It then retrieves the Home Agent’s IP address from its cache or the DHT.

3. The Home Agent that received the query then contacts the device’s Home Agent and asks for the device’s current IP address. The device’s current IP address is returned to the DNS Gateway and it returns the current IP address to the requester using a DNS reply. The current version of the DNS Gateway software does not cache the results from Home Agents, but it does specify a configurable Time To Live (TTL) value in the DNS reply sent back to the recursive DNS server. This TTL value specifies the amount of time that the recursive DNS server should cache the result obtained from the DNS Gateway.

**Integration Using .dht Pseudo Top-Level Domain** Recall that in a typical scenario, stub resolvers on an end system such as a desktop or laptop machine, offload most of the work of resolving

---

\(^7\) The registration of domain names under top level domains is out of the scope of this report, but many resources explaining the registration process can be found on the web.

\(^8\) A subdomain of the **pwebproject.net** domain is used as the DNS Gateway domain rather that **pwebproject.net** domain itself so that other subdomains, such as **www.pwebproject.net** can be created as well.
domain names to a recursive DNS server in the local network or in the Internet Service Provider’s network. The recursive DNS server will then query the a root server and any other nameservers that it is referred to in order to resolve the domain name to an IP address. In order to support a pseudo-top level domain in the network serviced by the recursive resolver, and entry for the dht pseudo-top level domain is permanently added to the recursive DNS server’s cache. This entry contains NS records that delegate responsibility for the dht domain to pWeb DNS Gateway servers that are also hosted in the local network.

When using a gateway domain, no non-standard configuration is required at any point within the domain name system, and by default a DNS Gateway server set up using a gateway domain will be globally accessible. By contrast, when using the pseudo-top level domain, changes must be made to the nameservers in the network to be served and the pseudo-top level domain will only be usable within that network. However, this approach is less centralized than the gateway domain approach, provides more redundancy and is more scalable.

**Interception of a Gateway Domain in Local Network** For a well-known gateway domain, such as dht.pwebproject.net, the recursive DNS nameservers for a network could be configured in order to “intercept” queries against this domain and forward them to local pWeb DNS Gateways. This approach has the advantage that applications designed for use with the pWeb system could be configured to use a globally accessible DNS Gateway, but when the application is used within a network supporting this type of interception, the query is forwarded to a local DNS Gateway server, improving the response time and reducing the load on the globally accessible DNS Gateway servers.
3.3 Scalability
The pWeb DNS Gateway has been designed to be extremely scalable. In its basic form, the DNS Gateway translates messages between the DNS protocol and the Home Agent protocol. No communication takes place between DNS Gateway servers and any number can be deployed in parallel. Thus, the pWeb DNS Gateway is not an architectural bottleneck in the pWeb device name resolution system. Additionally, the implementation of the DNS Gateway software has been designed so that it can take full advantage of modern server hardware. More details about the efficiency and scalability of the implementation can be found in following section and Section 3.8.

3.4 Design and Implementation
The pWeb DNS Gateway has been programmed from scratch with scalability and performance in mind. The DNS Gateway exclusively uses asynchronous I/O for network communication through the Boost Asio library [16] that is a cross platform C++ library for network and low-level I/O programming and has been proposed for inclusion of the next version of the official C++ standard [15]. The Asio library uses the most efficient asynchronous I/O API offered by the operating system on each platform that it supports [27, 17]. For example, on Linux 2.6, the Asio library uses epoll as the socket event notification mechanism.

Network servers have traditionally been programmed using a process-per-connection or a thread-per-connection model. The overhead of these strategies limit the scalability of the software on modern hardware [14, 27]. In this approach one process or one thread (or even two threads in some cases) is created to service a single TCP connection. The creation of processes or threads is a relatively expensive procedure that consumes both processor time and memory. After the process or thread is created, the application issues a read or write system call whenever it is ready to read or write data from the connection that blocks until the operation completes, preventing the process or thread from doing any other useful work. A large number of processes and threads causes the operating system to spend a significant amount of time switching between them and scheduling times for them to run. Most network servers that use a process-per-connection or thread-per-connection strategy cannot handle more than ten thousand concurrent connections even on modern hardware. This scalability limit is known as The C10K Problem [14]. Blocking operations on UDP sockets cause similar scalability problems, although the effect is much less severe.

The asynchronous I/O approach, however, has been designed and implemented to ensure that the software is not a bottleneck to scalability and can take full advantage of modern hardware with multicore processors and fast network connections. In the asynchronous I/O approach, all network operations such as establishing a connection or performing a read or write operation, are non-blocking or asynchronous. This decouples the threading model from network operations, enabling the application designer to use as many or as few threads as they see fit. The pWeb DNS Gateway uses one application thread by default, but provides a configuration parameter to increase the number of threads, with an option to match the number of application threads to hardware cores. Matching the number of application threads to the number of hardware cores ensures that DNS Gateway can take advantage of all available processing power if necessary on the host machine.

The DNS Gateway uses both TCP and UDP connections. The DNS protocol defaults to UDP for simple queries in order to avoid paying the cost of establishing a TCP connection. TCP connections are used when a query or reply is too large for a UDP datagram and for more complex operations. The DNS Gateway to Home Agent protocol, discussed in the next section, uses TCP exclusively.
The DNS Gateway has flexible configuration and logging subsystems that are discussed in Section 3.4.2 and Section 3.4.3 respectfully. The DNS Gateway also has an instrumentation subsystem that is discussed in Section 3.4.4. A discussion of known implementation issues is discussed in Section 3.4.5.

3.4.1 DNS Gateway to Home Agent Protocol

The DNS Gateway to Home Agent protocol is a custom message-based application protocol over TCP. The DNS Gateway to Home Agent protocol uses two types of messages: PeerInitiateGET messages are sent from the DNS Gateway to Home Agents in order to request a device’s current IP address, and MessageGET_REPLY messages are sent from the Home Agent to DNS Gateway to answer device IP look-up requests. All messages are sent over TCP connections. For every message sent, the sender opens a TCP connection, writes the message to the socket and then closes the connection. The full specification of the protocol is included in Appendix B Section 1. The format for the protocol was chosen in order to simplify the connection handling code in the Home Agent software. Unfortunately, this format is not very appropriate for the DNS Gateway to Home Agent protocol and suffers from the following limitations.

- A new TCP connection must be established for each message sent and received. This introduces connection establishment overhead and latency to the communication protocol.
- Most of the fields in the messages exchanged are unused. This means that most of the data exchanged over the network between the DNS Gateway and Home Agents serves no useful purpose.
- Both the Home Agent and DNS Gateway must be able to accept incoming connections. This introduces administration overhead when the DNS Gateway is behind a firewall because additional firewall ports must be opened.
- The DNS Gateway does not attempt to establish a connection to the Home Agent until it has a query to send. This means that the DNS Gateway does not know in advance if the Home Agent is inaccessible because it has crashed or is permanently or temporarily offline.

A different protocol that does not suffer from these limitations was originally proposed for use between the DNS Gateway and Home Agents. This protocol has been implemented in the DNS Gateway but has not yet been adopted by the Home Agent software. The specification for this protocol is included in Appendix B Section 2.

3.4.2 Configuration

The DNS Gateway software uses the Boost Program Options library [21] for runtime configuration management. Most configuration options have long and short forms and can be specified on the command line or in a configuration file. Short options specified on the command line must be proceeded by a single dash, -, and long options must be proceeded by a double-dash, --. Some options require an argument. An argument is specified immediately after and separated from the option name by either a space or an equals character, ‘=’, on the command line. In a configuration file, an equals sign must be used. More information on the configuration file format is available in the Program Options library documentation at the following link: http://www.boost.org/doc/
When run with the option `--help`, the DNS Gateway will print a summary of the available configuration options and exit. This summary is shown in Listing 3. The `--version`, `--help` and `--config` options may be specified only on the command line while all remaining options may be specified on the command line or in a configuration file.

**pWeb DNS Gateway v1.0.0**

**Allowed options:**

**Generic options:**
- `-v` [ `--version` ] [`=arg(=1)`]  Print program version and exit
- `-h` [ `--help` ] [`=arg(=1)`]  Print summary of configuration options and exit
- `-c` [ `--config` ] `arg`  Path to configuration file

**Configuration:**
- `-ttl` `arg` (=3600)  Number of seconds DNS clients will be instructed cache name to IP mappings
- `-t` [ `--timeout` ] `arg` (=12)  Maximum number of seconds to wait for response from a Home Agent before returning an error to the DNS client
- `-l` [ `--log_file` ] `arg` (=dnsgw.log)  Log file path
- `-L` [ `--log_level` ] `arg` (=WARN)  Log level
  - Only log messages with a level less than or equal to the specified severity will be logged. The log levels are NOTSET < DEBUG < INFO < NOTICE < WARN < ERROR < CRIT < ALERT < FATAL = EMERG
- `-i` [ `--iface` ] `arg`  IP v4 or v6 address of interface to listen on for DNS queries
- `-p` [ `--port` ] `arg` (=53)  TCP and UDP port to listen on for DNS queries
- `-H` [ `--home_agent` ] `arg`  List of Home Agent addresses to connect to
  - Any number of Home Agent addresses may be specified, separated by commas. Each address should have the form '<hostname or IP address>:<port>'. The DNS Gateway will use all the Home Agent addresses specified in a round-robin manner.
- `-N` [ `--nshostname` ] `arg`  The hostname of the DNS gateway
  - This hostname is included in DNS replies and in messages sent to to Home Agents. Home Agents will send replies to this hostname using separate TCP connections.
Suffix to be removed from domain names in order to obtain pWeb device names

Number of application threads
Set to 0 to use one thread per hardware core

Address of instrumentation server
The address should have the form '<hostname or IP address>:<port>'

Listing 3: DNS Gateway configuration options

There is one additional undocumented option that can be specified on the command line or in a configuration file: the option --debug with short form -d enables debugging mode. In debug mode all log messages are printed to the console as well as the log file.

3.4.3 Logging

The DNS Gateway uses log4cpp [11] as its logging subsystem. Log4cpp is a very efficient and flexible logging library modelled after the popular log4j library. It can be easily configured to log to multiple destinations using multiple formats. The DNS Gateway currently supports logging to a file only, but support for other destinations, such a syslog daemon, may be added in the near future.

3.4.4 Instrumentation

The DNS Gateway contains a custom instrumentation service. The DNS Gateway collects information about every query it receives including the device name being queried, the time the query was received, the amount of time taken to process the query with sub-millisecond accuracy on most platforms, and a status code that identifies any errors encountered while processing the request. In the current implementation, the instrumentation data is serialized using the Boost Serialization library [22]. Instrumentation data is collected in a buffer that uses a Nagle timer before being sent over UDP. Serialized instrumentation objects are on the order of tens of bytes. The Nagle timer helps reduce the overhead of the UDP protocol by packing as many instrumentation datagrams as possible into a single UDP datagram during one Nagle period.

The purpose of the instrumentation library is to facilitate the real-time monitoring of one or more DNS Gateways. Instrumentation data could be fed into a network operations centre’s monitoring system and used to produce real-time charts of the Gateway’s performance and status indicators on a dashboard or heads-up display.

3.4.5 Known Issues

The current version of the DNS Gateway has a few known limitations. Each of these limitations are relatively easy to correct and may be fixed in future versions of the DNS Gateway.

- The device name to IP mappings retrieved from the Home Agents are not cached in the DNS Gateway itself although the results will be cached by the DNS servers querying the DNS Gateway.
• DNS queries contain a 16-bit serial number that is used to match queries and responses. The DNS Gateway copies the DNS query serial number to the PeetInitiateGET message sent to a Home Agent and treats all serial and sequence numbers globally. This has reliability and security implications. A more robust solution would be to use a larger namespace for sequence numbers in PeetInitiateGET messages that is formed by hashing the DNS client’s IP address and port with the DNS serial number.

• The DNS Gateway does not implement support for some standard DNS resource record types such as NS and SOA records.

**Standards Compliance** The DNS Gateway has been designed to be compliant with RFC 1035 [18]. This RFC defines the DNS resource record formats and algorithms used by DNS servers in the domain name system. This is still the canonical definition of the domain name system; however, the system was designed to be extensible and many new features, such as extensions for compression and encryption, are now in widespread use but not currently implemented in the DNS Gateway.

### 3.5 Installing the DNS Gateway from Source

The DNS Gateway is written in portable C++ and makes use of several portable, high quality C++ libraries. The following sections describe the DNS Gateway dependencies and build procedures. More complete instructions are given in the README file distributed with each of the pWeb components. The README files are presented in Appendix A for reference.

#### 3.5.1 Dependencies

The DNS Gateway depends on the Boost and log4cpp libraries. The DNS Gateway distribution also includes two additional tools for testing and debugging the gateway and it’s configuration. One of these tools requires the cURL library. The libraries may be downloaded and installed from their project websites or from your platforms package management system; however, the DNS Gateway requires Boost version 1.53 or newer, which is not available in the official Ubuntu repositories for the current versions of Ubuntu. Boost 1.53 for Ubuntu Precise Pangolin (12.04 LTS) and Quantal Quetzal (12.10) can be installed from this author’s PPA on Launchpad at [https://launchpad.net/~apokluda/+archive/boost1.53](https://launchpad.net/~apokluda/+archive/boost1.53).

The DNS Gateway also includes a few tools for testing and debugging the gateway.

#### 3.5.2 Building

The DNS Gateway uses the CMake build system. Listing 4 shows the commands to build and install the DNS Gateway from source on Unix-like platforms. The recommended way to build and install the DNS Gateway on Windows platforms is to use the GMake GUI. Instructions on how to use the CMake GUI are available on the CMake website at [http://www.cmake.org](http://www.cmake.org).

```bash
alex@alex-desktop:~$ tar -xzf pWeb-1.0.0-Source.tar.gz
alex@alex-desktop:~$ cd pWeb-1.0.0-Source/
```

---

3.6 Installation as a System Service

The recommended way to run the DNS Gateway is as a system service. On Unix-like platforms, the recommend way to configure the DNS Gateway to run as a system service is to use the daemon utility. The daemon utility can be installed on Ubuntu platforms by installing the system daemon package. Once the package is installed, you must configure your system to start and stop the DNS Gateway daemon when the system boots and shuts down. Ubuntu systems use Upstart for this purpose. Listing 5 shows an Upstart script that can be used to run the DNS Gateway as a system service using the daemon utility on Ubuntu. The script should be saved to /etc/init/dnsgw.conf.

```
description "pWeb DNS Gateway Server"
author "Alexander Pokluda <apokluda@uwaterloo.ca>"

start on runlevel [2345]
stop on runlevel [!2345]

exec daemon --user=daemon:daemon --name=dnsgw --respawn -- \
 /usr/local/bin/dnsgw -c /etc/dnsgw.conf
```

Listing 5: Upstart script to run the DNS Gateway as a system service on Ubuntu

3.7 Test Tools

The DNS Gateway distribution contains two test tools that can be used to aid in development and testing of the DNS Gateway. The program `dnsgw_test_client` is designed to check the status of a set of home agents and/or register device names at a set of home agents. The tool has been designed to be extremely scalable and uses the same design principles as the DNS Gateway itself in order to support potentially tens of thousands of Home Agents very efficiently.

The program `dnsgw_test_ha` is a mock home agent. This program includes a subset of the Home Agent code. In particular, it uses the same connection handling and message parsing and composing code as the Home Agents. The difference is that rather than query the DHT and/or a remote Home Agent for a device’s current IP address it will return either the IP address 6.6.6.6 or a Name Not Found error for each query. The default is to respond to each query immediately, but the tool can also be configured to wait a specified amount of time before sending a response in order to simulate the latency associated with performing a look-up operation in the DHT followed by contacting a remote Home Agent.

3.8 Performance Evaluation

The `dnsgw_test_ha` test tool from the previous section has been used to estimate a lower bound on the performance of the DNS Gateway using the ResPerf DNS performance measurement tool\textsuperscript{10} that
is designed to estimate the latency and throughput of DNS servers. The test was run with all three components—the DNS Gateway, measurement tool, and mock home agent—on a quad-core desktop PC with 16 GiB of RAM. Queries for pWeb device names were generated by the measurement tool and sent to the DNS Gateway. The DNS Gateway in turn issued a query for the device’s current IP address to the mock Home Agent. The mock Home Agent was configured to return the IP address 6.6.6.6 immediately. The report produced by the ResPerf measurement tool is included in Appendix B Section 3.

As can be seen in the report, the performance of the DNS Gateway and mock Home Agent starts to suffer at about 4 seconds into the test. This is because the mock Home Agent is unable to keep up with the increasing rate of queries generated by the measurement tool. When the DNS Gateway does not receive a response from the mock Home Agent after a short period of time, it responds to the measurement tool’s query with an error message. The statistics section of the report shows that the DNS Gateway responded to all queries issued by the measurement tool. The pWeb Home Agent uses a traditional thread-per-connection design as discussed in Section 3.4 that limits its scalability. This test shows the benefit of the asynchronous I/O approach used by the DNS Gateway and that the DNS Gateway can achieve a throughput of at least 7,084 queries per second using one application thread.

3.9 Summary
This section discussed the system and software architecture, design and implementation of the pWeb DNS Gateway. The DNS Gateway is provides a bridge between the domain name system infrastructure and pWeb device naming system and helps provide a seamless integration between pWeb and traditionally hosted web content. It is and will continue to be an important part of the pWeb infrastructure.

4 Device Crawler and Indexer
The pWeb Device Indexer consists of crawler and manager applications that are being programmed from scratch. The design and implementation of the crawler and manager parallels that of the DNS Gateway and test tools. They use efficient and scalable asynchronous I/O for network communication, the Boost Program Options library for runtime configuration, the log4cpp library for logging and the cURL library for accessing HTTP interfaces.

The Device Crawler and Indexer is an important part of pWeb. It allows users to discover other users, devices and content. The Web front-end constructs a URL in the search results that can be used to access content. From the user’s perspective, the URL works like a regular HTTP URL except that the request for the IP address of the host of the resource is received by a DNS Gateway and forwarded to a Home Agent to retrieve the device’s last know IP address. We expect that the search interface is going to be one of the main methods that users use locate content in pWeb.

The architecture of the Device Indexer system is shown in Figure 8. The following sections describe each of the main components of the pWeb Device Crawler and Indexer system.

---

\(^{10}\)The ResPerf DNS performance measurement tool is available from Nominum at [http://www.nominum.com/support/measurement-tools/](http://www.nominum.com/support/measurement-tools/).
Figure 8: The pWeb Device Indexer architecture
4.1 Home Agents

The Home Agent acquires information about devices through various means while it is providing services to user devices. For example, the Home Agent is told the name of the device’s owner and device’s physical location at the time of registration. The Home Agent is informed of the device’s current IP address by the IP updater module of the pWeb client software. The Home Agent may also learn and store new information as the pWeb system evolves. For example, we are currently developing a mechanism for the pWeb client software to push metadata about hosted content to the Home Agent. The Home Agent then makes some of this information available to external services through an HTTP RESTful interface in an XML-like format provided that the user’s privacy settings allow it. Listing 6 shows what the output from the current version of the Home Agent looks like. This version of the output lists the Home Agent’s name, its neighbours in the Plexus overlay network, and a list of new and updated devices and their metadata. A timestamp is included as an HTTP GET parameter in the request URL which is used by the Home Agent to determine which results to return.

4.2 Crawlers and Manager

The pWeb Crawler process is responsible for polling pWeb Home Agents in order to discover new and updated devices in the pWeb network. It uses the HTTP interface to retrieve Home Agent and device information as shown in Listing 6. The crawler uses the list of the Home Agent’s neighbours in order to discover all the Home Agents in the pWeb network.

The Crawler process uses efficient and scalable asynchronous I/O and has been designed to take advantage of multi-core processors. A single pWeb Crawler process can be run independently and, on powerful enough hardware, monitor hundreds to thousands of Home Agents with short polling periods. When run in this mode, the Crawler process immediately starts monitoring any newly discovered Home Agents.

Multiple Crawler processes, coordinated by a Manager process, can be run in parallel in order to scale additional orders of magnitude. When a Crawler process is run with a coordinating Manager process and it discovers a new Home Agent, it will first check in its local cache to see if it has seen a reference to that Home Agent before. If it has not, then it will inform the Manager that it has seen a new Home Agent. The Manager will then check in its local database if any Crawler process in the system is currently monitoring the new Home Agent\textsuperscript{11}. If not, then it will order the least loaded Crawler process to start monitoring the newly discovered Home Agent. Once the system has reached a stable state and all or nearly all Home Agents have been discovered, very little communication occurs between the Crawlers and Manager. In this case, updates will only ever be propagated from the Crawlers to the Manager if the structure of the Plexus overlay network changes significantly or a new Home Agent is brought online.

If and only if a polled Home Agent returns information about one or more updated devices, then the Crawler process will post an update to an Apache Solr server. Solr, to be discussed more in the next section, is an enterprise search server and database. In pWeb, an instance of Solr is used to store and search among user, device and content metadata.

\textsuperscript{11}Currently the Crawler and Manager processes keep their cache and database in memory only. If it is deemed necessary for disaster recovery, the Crawler and Manager processes could be modified to read and write their cache and database from and to persistent storage.
Listing 6: Sample XML-like output returned from Home Agent HTTP interface
4.3 Solr Cloud

Solr is an open source enterprise search database that is used by many large companies including AOL, eBay and Netflix, among others, to power the search features on their public Websites. Apache Solr shares a common ancestry with Apache Nutch, Lucene and Hadoop. Like the Crawlers, a single Apache Solr instance can be run in a standalone configuration, but Solr can also scale to handle really large databases using a distributed search configuration, known as a Solr Cloud.

There are two ways to set up Solr in a distributed fashion. The first is replication. When set up this way, multiple Solr instances hold exact replicas of the search database. This enables greater redundancy and linear scalability in the number of queries per second that the system can handle. The second way to configure Solr in a distributed manner is to use sharding. With sharding, multiple Solr instances each hold a distinct portion of the search database. This enables the size of the search database to scale beyond the storage and processing limits of a single server. Replication and sharding can be used together for maximum scalability and redundancy.

Solr provides an HTTP RESTful interface that is the primary way for external applications to interact with Solr. This interface is used for inserting data into the database, searching for data, and management functions. The Crawler uses this interface to insert user, device and content metadata into the database. Solr also provides a Web-based graphical user interface that can be used for management and testing. Solr also provides a simple templating mechanism that can be used to create basic search interfaces; however, this system is not intended to be used in production systems. The search interface provided at http://pWebproject.net/search is implemented using the Django Web framework, which will be discussed further in the next section.

The format of the data that is sent to Solr, as well as the type and behaviour of expected queries, is defined by a schema. The schema that is currently in use for pWeb is shown in Listing 7. This schema defines fields for the device metadata, such as the name of the device’s owner, the device name, its Home Agent, and other values. The values of the owner, name, home, location and description are copied to a text field that is not stored in the database but enables efficient searching on all of these fields at once. This special text field also employs splitting and steaming so that a query like “wi fi network” will match results that contain “WiFi Networks.”

4.4 Web Front-End

One or more Web servers running the Django Web framework provide a Web front-end that can be used by human users in order to search for devices in the pWeb network based on the published metadata. A Django Web application has three distinct layers known as the model, view and template layers. The model layer is responsible for structuring and manipulating persistent data; the view layer is responsible for encapsulating the logic for processing a user’s request; and the template layer provides a designer-friendly syntax for rendering the information to be presented to the user. All three layers are programmed in Python.

When a user performs issues a query from the search interface at http://pWebproject.net/search, the view layer of the Django Web application receives the request and issues a query to Solr to perform a search and retrieve the results. Solr returns the results as Python data structures that are passed to the Template layer that renders them as HTML to be displayed to the user. The model layer is not used in the pWeb device search Web application because all persistent data is stored in Solr’s search database. Because the Web interface does not store any context or state information, its scalability is not a concern. Any number of Web front-ends can be deployed in
<?xml version="1.0" encoding="UTF-8" ?>
<schema name="pWeb_device" version="1.5">
  <fields>
    <field name="owner" type="string" indexed="true" stored="true" required="true" multiValued="false" />
    <field name="name" type="string" indexed="true" stored="true" required="true" multiValued="false" />
    <field name="home" type="string" indexed="true" stored="true" required="true" multiValued="false" />
    <field name="port" type="int" indexed="false" stored="true" required="true" multiValued="false" />
    <field name="timestamp" type="long" indexed="false" stored="true" required="true" multivalued="false" />
    <field name="location" type="string" indexed="true" stored="true" required="true" multiValued="false" />
    <field name="description" type="string" indexed="true" stored="true" required="true" multiValued="false" />
    <field name=".version." type="long" indexed="true" stored="true" />
  </fields>
  <!-- Field to use to determine and enforce document uniqueness. -->
  <uniqueKey>name</uniqueKey>
  <!-- copyField commands copy one field to another at the time a document is added to the index. It’s used either to index the same field differently, or to add multiple fields to the same field for easier/faster searching. -->
  <copyField source="owner" dest="text"/>
  <copyField source="name" dest="text"/>
  <copyField source="home" dest="text"/>
  <copyField source="location" dest="text"/>
  <copyField source="description" dest="text"/>
  <types>
    <fieldType name="string" class="solr.StrField" sortMissingLast="true" />
    <fieldType name="int" class="solr.TrieIntField" precisionStep="0" positionIncrementGap="0" />
    <fieldType name="long" class="solr.TrieLongField" precisionStep="0" positionIncrementGap="0" />
    <fieldType name="text_en_splitting" class="solr.TextField" positionIncrementGap="100" autoGeneratePhraseQueries="true">
      <analyzer type="index">
        <tokenizer class="solr.WhitespaceTokenizerFactory"/>
        <filter class="solr.StopFilterFactory" ignoreCase="true" words="lang/stopwords_en.txt" enablePositionIncrements="true" />
        <filter class="solr.WordDelimiterFilterFactory" generateWordParts="1" generateNumberParts="1" catenateWords="0" catenateNumbers="0" catenateAll="0" splitOnCaseChange="1" />
        <filter class="solr.LowerCaseFilterFactory"/>
        <filter class="solr.KeywordMarkerFilterFactory" protected="protwords.txt"/>
        <filter class="solr.PorterStemFilterFactory"/>
      </analyzer>
      <analyzer type="query">
        <tokenizer class="solr.TokenizerFactory"/>
        <filter class="solr.StopFilterFactory" ignoreCase="true" words="lang/stopwords_en.txt" enablePositionIncrements="true" />
        <filter class="solr.WordDelimiterFilterFactory" generateWordParts="1" generateNumberParts="1" catenateWords="0" catenateNumbers="0" catenateAll="0" splitOnCaseChange="1" />
        <filter class="solr.LowerCaseFilterFactory"/>
        <filter class="solr.KeywordMarkerFilterFactory" protected="protwords.txt"/>
        <filter class="solr.PorterStemFilterFactory"/>
      </analyzer>
    </fieldType>
  </types>
</schema>

Listing 7: Solr database and query schema for pWeb user and device metadata
parallel and the domain name system, a load balancer, or both can be used to evenly share the load between them.

4.5 Summary

The pWeb Device Indexer consists of Crawler and Manager processes that pull user and device metadata from the Home Agents and push updates to a single Solr search server or a cluster of Solr search servers. The Home Agents and Crawler processes will be extended to include metadata about hosted content as well. A Web front-end implemented using the Django Web framework provides an interface for human users of pWeb to search for users, devices, and in the near future, content as well.

The Crawler and Managers have been implemented using the same scalable asynchronous I/O as the DNS Gateway and can take advantage of modern multicore processors for scalability on a single machine and multiple Crawler processes can be deployed in parallel with a coordinating Manager process for further scalability. Solr is a mature Apache project that shares ancestry with Nutch, Lucene, and Hadoop. The open source community has invested a lot of effort into ensuring that Solr also scales well. Multiple Solr servers can be deployed for scalability using both shards and replication. The Web front-end that implements the pWeb search interface at http://pWebproject.net/search is also scalable. The search interface itself is stateless and any number can be deployed in parallel.

The search feature described in this section is an important part of pWeb. Together with the DNS Gateway and Home Agents, it allows users to discover content hosted in pWeb and access it directly. We expect that the search interface is going to be one of the main methods that users use to locate content in pWeb.
Part III:
pWeb Client for Android
5 pWeb Client Application

In this section we explain the pWeb Client application for the Android platform. The application consists of an Android interface - used to launch, visualize, and control the execution of the application. It also has an Android and Jetty-based lightweight web server, and an Android and JavaScript-based web application that must be installed onto the web server that allows video browsing and playback from a connected device.

The console video web application scans the device’s storage for video files with an ‘mp4’ format. It then makes this content available via the I-Jetty web server. The Android application, named ‘pWeb Client’, uses the results found by the web application - stored in a json format - to determine the list of videos found on the device. This list of videos, along with its attributes is then stored in an embedded SQLite database. The user of the device can modify the video attributes and save these results to the database. The list is then sent to a home agent the device is registered at in a pseudo-xml format. Thus the video content is made available to users of the pwebproject website. Users connected to the same network that an I-Jetty server is connected can then stream the public video content progressively from their desktop, laptop, or mobile device.

5.1 Installation and Launching

This section will explain installing and launching the pWeb Client application.

Accepting the “Permissions”. When installing the application, you will be presented with a list of permissions that must be agreed upon for the usage of this application, as seen in Figure 9. The application needs full network access, accessing device storage, accessing device vibration, accessing the device’s camera, and accessing the device’s microphone.
Home Screen Icon. Tap on the icon, seen in Figure 10, created on your homescreen to launch the pWeb Client application. This icon can also be found in your app drawer.

Main Screen. When this application is launched, you will be presented with the main screen, as seen in Figure 11.
5.2 Within the Client Application

This section will explain the various functionalities of the application.

About. To learn more about this application, tap the (i) icon in the center of application’s menu bar, seen in Figure 11. A pop-up menu will appear, as seen in Figure 12.

Settings. Tap on the settings icon on the right of the application’s menu bar, seen in Figure 11, to open the settings screen, seen in figure 13. Here you can change some aspects of how this application behaves. You can turn sound and/or vibration on/off.
Install WebApp. Tap on the “Video WebApp” button on the main screen, seen in Figure 11, to open the video webapp installation screen, as shown in Figure 14. Installing this video web application will give your I-Jetty server the ability to allow users to view the video content on your device and to stream this content on their PCs, laptops, or mobile devices. Tap on the “Install WebApp” button to begin the installation process. A pop-up menu will appear, as shown in Figure 15a, indicating the progress of installation. Once the complete, a pop-up menu will appear, as shown in Figure 15b, indicating the successful installation of the video web application. If the video web application has already been installed, a pop-up menu will appear, as shown in Figure 15c, asking whether or not you wish to re-install the web app.
Figure 14: Install WebApp

(a) Installation Progress

(b) Installation Complete

(c) Re-install WebApp

Figure 15: Notifications

**IP Updater.** Tap on the “Network Configuration” button on the main screen, seen in Figure 11, to open the device and user information screen, shown in Figure 16. Here you must enter the name of your device as registered on pwebproject.net, as well as your home agent and username/password combination.
Once these have been entered, tap done, and once verified, you will be presented with the options screen, shown in Figure 17.

Choose “Port Forwarding” and you will be presented with the port number and IP screen,
shown in Figure 18. Enter the port number and choose the IP of your device to allow the home agents to interact with your device and this application. Choose “Nat Traversal” and you will be presented with a screen where you must enter the IP address of your directory server, as seen in Figure 19.

![Figure 18: Port Number and IP](image)

![Figure 19: Nat Traversal](image)

**I-Jetty Web-server.** I-Jetty is a light-weight mobile web-server, that will allow your mobile device to serve content to users who have access to your network. Tap on the “Run I-Jetty” button on the main screen, seen in Figure 11, to open the i-jetty screen, shown in Figure 20. If you wish to change the configuration of I-Jetty, tap on the “Configure” button, and you will be presented with a list of options to control the deployment of the web-server, as seen in Figure 21. Otherwise, tap on “Start Jetty” to run the web-server. Once you wish to stop the server, tap on “Stop Jetty”.

48
Figure 20: I-Jetty

Once running, a notification will appear indicating the server has been started, and the port on which you are running this server will be displayed in the Console, in addition to the network interface, as shown in Figure 20.

Figure 21: Configuring I-Jetty
With I-Jetty running, you can now access any installed web applications you have installed on the server. If a new web app is installed, I-Jetty must be stopped and restarted.

**Taking Videos.** In order to take a video and give the video your own descriptors, tap on the camera icon in the menu bar on the main screen, seen in Figure 11, to launch your phone’s built-in camera, shown in Figure 22. The interface will vary with your device. At this screen, you may change a few settings specific to your device’s camera, as shown in Figure 23, including color saturation and flash. The options available will once again be dependent on your device’s camera.

![Figure 22: Taking a Video](image1)

![Figure 23: Changing Video Settings](image2)

Once ready to take the video, tap the red button and begin recording your video. This is shown in Figure 24.

When your video is complete, tap the red stop button, and confirm the taken video by tapping the check mark, or cancel by tapping the x, shown in Figure 25.
You will then be taken back to the main screen shown in Figure 11. If you have chosen to accept the taken video, you will then be asked to choose a title for the video, as shown in Figure 26.
List of Videos. In order to view a list of the videos you have stored on your device, and change these videos’ attributes, tap on the “Video Configuration” button, seen in Figure 11, to open the video list screen, shown in Figure 27.
Tapping on any one of the videos in the list will take you to a Video attribute modification screen, shown in Figure 28.

![Video Attributes](image)

Figure 28: Video Attributes

You may change any one of these attributes. HOWEVER, changing the ID, Mime type, or Size of the video will lead to errors. Once you have changed the attributes, tap on the “Update DB and HA” button. A notification will appear, telling you the changes have been made successfully. Tapping the “View Database” button, seen in Figure 27 will allow you to see all the videos and their attributes in a table form, as seen in Figure 29 and Figure 30.
5.3 Video Web Application

This section will explain the use of the video web application.

**Video WebApp.** Using the video web application, shown in Figure 31, will allow users - who are connected to the same network that an I-Jetty server is connected - to view the list of videos stored on the device. The URL that needs to be entered into a web browser as follows:

http://<device-name>.<username>.<home-agent>.dht.pwebproject.net:<port-number>/console
Clicking on any one of the video titles will open a new window in your browser, where the video is streamed, as shown in Figure 32.
Conclusion
6 Conclusion

The pWeb project has addressed and provided solutions for the major challenges inherent in a peer-to-peer Web hosting system. Specific solutions to the challenges of distributed content naming, distributed searching [1, 3], ensuring content availability [23, 24], content indexing and search result ranking, and dynamic webpage generation were presented in previous deliverables. In this final report, we described the final pWeb software—how it works as well as how to install, configure and use it. After having read the report, someone unfamiliar with pWeb should be able to configure a standalone pWeb network for demonstration purposes.

In Part I we presented background research that influenced design decisions in our final software as well as the protocols that we developed for communication between various pWeb components. In Part II we provided an overview of the architecture of the system including all the major pWeb components, their external interfaces, and the compatibility of pWeb as a whole with existing Web technologies. Part III was dedicated to discussing the pWeb client application that we developed for Android and a demonstration of how to use it. Although it was not an original objective of this deliverable, the pWeb client application for Android clearly demonstrates the effectiveness of the pWeb system.
Appendix A:

pWeb Component Installation and Configuration
1 pWeb Component Installation and Configuration

A fully functioning pWeb system requires several components to be set up and configured. At a minimum, a fully functioning system requires at least one Home Agent, one registration Web server, one DNS Gateway, one Crawler Poller process, one instance of the Solr search database, one instance of the main pWeb website and an Android-based device running the pWeb client software. All of these components can be run on a single system running GNU/Linux for demonstration purposes. It is recommendend that you install the components in the order listed above.

2 Installing and Configuring the pWeb Home Agent

The pWeb Home Agent software is distributed in a compressed archive. The archive contains a README file that lists all of the steps necessary to install a Home Agent. The README file is included below for reference.

This document describes how to install the pWeb Home Agent on any POSIX platform. I has been tested on all versions on Ubuntu and custom OS on PlanetLab. It has a number of dependencies that can be easily installed using a few simple commands.

You should have received a source distribution of the Home Agent as well as a binary distribution. The binary distribution is the simplest way to install on Ubuntu 12.04.

Required libraries

1. gcc 4.4 and upward (tested on Ubuntu 10.04, 10.10, 11.04, 11.10, 12.04, 12.10, 13.04)
2. sqlite3

Installation

First install the dependencies:

```
sudo apt-get install build-essential sqlite3 libsqlite3-dev
```

Then build by runninh the `make_agent_client.sh` script

```
./make_agent_client.sh
```

Two files will be created

`agent`

`client`
Run Command

./agent to run the home agent
./client to run the client to bootstrap the home agents

Configuration file

1. config
node_file --> where to run the agents
monitors_file --> list of monitors for logging
log_server_name --> default logging server name
log_server_user --> log server user
check_point_row --> number of rows after which to send log entries to logging server
k --> number of random links
timeout --> connection timeout
retry --> number connection retries
cache_storage --> end_point | path
cache_type --> cache type
client_listen_port --> default listening port [will be replaced by port specified in node_file]
seq_file --> tracks the sequence of runs [used for experimental purpose]
alpha --> parameter used from publishing names [used for experimental purpose]
name_count --> number name to lookup [used for experimental purpose]
input --> name database for lookup [used for experimental purpose]

2. host list (name contained in config)
<number of agents>
<hostname> <port number> <ip address> <home agent alias>

3. monitor list (name contained in config)

3 Installing and Configuring the pWeb Registration Portal

This registration portal can be found in the home agent root folder under the folder “ui”. It has been tested on PHP 5 and above. Deploy any web server that supports PHP, install PHP, copy the content of “ui” folder in the http_root directory of the web server and this web portal should be up and running.

4 Installing and Configuring the pWeb DNS Gateway and Crawler

The pWeb DNS Gateway and Crawler software is distributed in a compressed archive both as a source and binary distribution. The source distribution contains a self-contained README file that lists all of the steps necessary to install these pWeb components from either the source or binary distribution. The README file is included below for reference.
This document describes how to install the pWeb DNS Gateway and Crawler on Ubuntu 13.04 (Raring Ringtail) and other versions of Ubuntu. The DNS Gateway and Crawler have been tested on Ubuntu 12.04 LTS, 12.10, and 13.04. The DNS Gateway and Crawler/Poller have a number of dependencies that can be easily installed using a few simple commands.

Earlier versions of the DNS Gateway and Poller/Crawler have also been tested on Windows Server 2008 (equivalent to Windows XP/Vista for desktops) and Windows Server 2008 R2 (equivalent to Windows 7 for desktops). However, installing the dependencies and compiling on Windows is much more complicated, and isn't described here. Additionally, the software has not been tested on Windows in quite some time and will likely need some source code modifications to compile successfully.

You should have received a source distribution of the DNS Gateway and Crawler well as a binary distribution. (You probably opened this file from the source distribution). The binary distribution is the simplest way to install the DNS Gateway and Crawler on Ubuntu 13.04.

Installation Instructions for Ubuntu 13.04 from BINARY

Note: These instructions are for Ubuntu 13.04 (raring) only. For installation on other versions of Ubuntu, see the following section.

First install the dependencies:

$ sudo apt-get install libboost-program-options1.53.0 
   libboost-thread1.53.0 
   libboost-date-time1.53.0 
   libboost-chrono1.53.0 
   libboost-serialization1.53.0 
   libcurl13 
   liblog4cpp5

Now install the DNS Gateway and poller binaries. The following commands assume that you have extracted the DNS Gateway/Crawler binary archive to <pweb>:

$ cd <pweb>
$ sudo install dnsgw.bin/dnsgw /usr/local/bin
$ sudo install crawler_poller.bin/poller /usr/local/bin
Now you are ready to proceed to the section "Configuring the DNS Gateway and Crawler."

Installation Instructions for Ubuntu 12.04, 12.10 or 13.04 from SOURCE

If you are installing the software on Ubuntu 12.04 or 12.10 you need to install the Boost libraries from Alexander Pokluda’s PPA, because the official repositories for Ubuntu 12.04 and 12.10 provide only older versions of Boost. (The software has been developed and tested with Boost 1.53. Newer versions of Boost should work, but haven’t been tested, because they don’t exist yet!). Follow the instructions on the PPA archive page to install the Boost 1.53 PPA: https://launchpad.net/~apokluda/+archive/boost1.53.

Now install the dependencies:

$ sudo apt-get install cmake libboost1.53-all-dev \
   libcurl4-openssl-dev liblog4cpp5-dev build-essential

We will use CMake to generate GNU Makefiles that are used to compile the DNS Gateway and Crawler/Poller. The recommended way to use CMake is to do "out of source builds". This means that we will leave the source directory untouched and use a separate directory to compile the software. The following commands assume that you have extracted the DNS Gateway/Crawler source archive to <src>:

$ mkdir pweb
$ cd pweb
$ cmake <src>

The compilation process uses a precompiled header to reduce the compile time. Due to a bug in the build system, we need to link the precompiled header source file to the build directory:

$ ln <src>/shared/stdhdr.hpp .

Now we can continue with the compilation:

$ make
$ make install
Now continue with the next section.

---------------------------------------------------------------------

Configuring the DNS Gateway and Crawler
---------------------------------------------------------------------

This section assumes that you have completed either the section about installing from BINARY or SOURCE above.

In order to run the DNS Gateway and Crawler as a daemon, first install the daemon package:

$ sudo apt-get install daemon

Now the DNS Gateway and Crawler can be launched on system startup and controlled as a system service using Upstart. Create the following two files in /etc/init:

--- /etc/init/dnsgw.conf ---

description "pWeb DNS Gateway Server"
author "Alexander Pokluda <apokluda@uwaterloo.ca>"

start on runlevel [2345]
stop on runlevel [!2345]

exec daemon --user=daemon:daemon --name=dnsgw --respawn -- /usr/local/bin/dnsgw -c /etc/dnsgw.conf

--- end of file---

--- /etc/init/crawler.conf ---

description "pWeb Device Crawler - Poller Process"
author "Alexander Pokluda <apokluda@uwaterloo.ca>"

start on runlevel [2345]
stop on runlevel [!2345]

exec daemon -c --user=daemon:daemon --name=poller --respawn -- /usr/local/bin/poller -c /etc/poller.conf

--- end of file ---

Note that the crawler init script starts only one poller without a manager process.
The above scripts use DNS Gateway and Crawler configuration files /etc. The following configuration files can be used as a starting point for your own. Note that dnsgw.conf tells the DNS Gateway to listen for DNS queries on 127.0.0.1:5533. The next section explains how to run all the pWeb components on a single machine with the DNS Gateway listening on the loopback interface. When running the DNS Gateway in a production environment, you must tell the DNS Gateway to listen on a public interface and on port 53 and point both the DNS Gateway and poller to your Home Agents.

--- /etc/dnsgw.conf ---
# pWeb DNS Gateway Configuration File
#
# The option names and values are the same on the command line and in
# this configuration file. More information about each option can
# be see by running
#
# $ dnsgw -h
#
# on the command line.
#
# The verbosity of messages printed to the log file
# Valid options are: DEBUG, INFO, NOTICE, WARN, ERROR, CRIT, ALERT, FATAL = EMERG
log_level=INFO

# Path to the log file
log_file=/var/log/dnsgw.log

# The hostname of the dnsgw
nshostname=dnsgw1.pwebproject.net.

# The interface to use for listening for DNS requests
iface=127.0.0.1

# The port to use for listing for DNS requests
port=5533

# The port to use to receive responses from the Home Agent
nsport=56142

# The suffix that must be removed from DNS queries to get
# a device name. (This device name will be sent to the Home Agent).
suffix=dht.pwebproject.net.

# The TTL value to insert into DNS responses
# Home Agents that can respond to queries for device name lookups.
# Any number of home agents can be specified by listing the 'home_agent'
# parameter multiple times. The value has the format '<host name>:<port>',
# where <port> is the port number of the TCP interface.
# If the Home Agents are not able to answer a query locally, they
# will lookup the device’s IP address in the Plexus network on behalf
# of the DNS Gateway.
#
# Example:
# home_agent=cn101.cs.uwaterloo.ca:20000
# home_agent=cn102.cs.uwaterloo.ca:20000
# home_agent=localhost:20000

--- end of file ---

--- /etc/poller.conf ---
# pWeb Crawler - Poller Process Configuration File
#
# The option names and values are the same on the command line and in
# this configuration file. More information about each option can
# be seen by running
#
# $ poller -h
#
# on the command line.

# The verbosity of messages printed to the log file
# Valid options are: DEBUG, INFO, NOTICE, WARN, ERROR, CRIT, ALERT, FATAL = EMERG
log_level=INFO

# Path to the log file
log_file=/var/log/poller.log

# The polling interval for each Home Agent. A value of x means
# that each Home Agent will be polled once every x seconds.
interval=5

# The URLs of the Apache Solr server to send updates to for the
# devices and content cores (databases)
soldevrurl=http://localhost:8081/solr/pweb_devices/update
solconrurl=http://localhost:8081/solr/pweb_content/update

# Home Agents to seed the crawling process. Any number of Home Agents
# can be specified. The poller process will use the default HTTP port
# for the home agents. If you are running your own Home Agent(s),
# you need to put there addresses here!
#
# Example:
#home_agent=cn101.cs.uwaterloo.ca
#home_agent=cn102.cs.uwaterloo.ca
home_agent=localhost

--- end of file ---

If you want the DNS Gateway to accept DNS queries on the default
port of 53 (eg, for a production server), you need to give it permission
to bind to privileged ports. This can be done with the command

$ sudo setcap 'cap_net_bind_service=+ep' /usr/local/bin/dnsgw

When run as a daemon, the DNS Gateway and poller will not
have permission to create their log files in the /var/log
directory. In order to allow the to write to this directory,
we have to create their log files manually:

$ sudo touch /var/log/dnsgw.log && sudo chown daemon:daemon /var/log/dnsgw.log
$ sudo touch /var/log/poller.log && sudo chown daemon:daemon /var/log/poller.log

Once the above configuration files are in place and you have run
the above commands, you should be able to start the DNS Gateway
and poller manually with the commands

$ sudo service dnsgw start
$ sudo service crawler start

You can monitor the DNS Gateway and poller log files in real-time
with the command

$ tail -f /var/log/dnsgw.log

or

$ tail -f /var/log/poller.log

If the DNS Gateway and Poller started successfully, you should
see the following output in each log file:

--- dnsgw.log ---
In order for the DNS Gateway to be useful, you need to configure some sort of delegation to the DNS Gateway. For example, the pWeb Project Team has registered pwebproject.net and have multiple BIND DNS Servers configured to handle queries for that domain. The bind servers are configured to delegate queries for *.dht.pwebproject.net to a pool of DNS Gateway instances. This setup is beyond the scope of this README, but the following shows how you can set up DNSmasq on your local machine to forward queries to *.dht.pwebproject.net to a DNS Gateway also running on your local machine. This section assumes that you have already followed the instructions above and that you are installing and running/demonstrating everything on a single machine that is running Ubuntu 13.04 with the Unity desktop environment installed and using the Network Manager GUI.

Use your favourite text editor to create the file

/etc/NetworkManager/dnsmasq.d/dnsgw.conf

Add the following two lines to that file:

server=/dht.pwebproject.net/127.0.0.1#5533
host-record=dnsgw1.pwebproject.net,127.0.0.1

You should now have a fully functioning DNS Gateway and Crawler setup. Next you will need to install the Solr and Django. Instructions and configuration files for setting up these services are provided in separate archives. You will also need to install and set up your Home Agent(s) and registration web server if you haven’t done so already.
If you need help, please feel free to contact me at apokluda@uwaterloo.ca.

Good luck!

5 Installing and Configuring the pWeb Search Database

The configuration for the pWeb search database is distributed as a compressed archive. The archive contains a README file that lists all of the steps necessary to install and configure the search database. The README file is included below for reference.

This folder contains the configuration for the pWeb content and device search database. No data is provided. The database can be populated using the pWeb Crawler.

Solr databases are called "cores." These instructions tell you how to set up the pweb_devices and pweb_content cores as a back-end for the search function on the pWeb website. The Solr search database has been set up and tested on Ubuntu 12.04 and 13.04. The instruction below describe how to set up the search database on Ubuntu 13.04 (Raring Ringtail), but the instructions should be similar for other versions of Ubuntu and GNU/Linux.

Configuring Solr for pWeb on Ubuntu 13.04

Step 1: Install a Java Servlet container

Apache Solr runs as a servlet in a Java servlet container such as Tomcat or Jetty. We have been using Tomcat 7 to host the Solr servlet. You can install Tomcat with the command

$ sudo apt-get install tomcat7

By default tomcat runs on port 8080. The server that hosts the public pWeb website pwebproject.net has other services running on port 8080, so the Tomcat server for Solr is run on port 8081. We recommend that you also run Tomcat on port 8081 so that your setup is the same to avoid any potential configuration problems.
To change Tomcat’s default port to 8081, edit the file

/var/lib/tomcat7/conf/server.xml

and change the port for the "Connector" like so:

```xml
<Connector port="8081" protocol="HTTP/1.1"
    connectionTimeout="20000"
    URIEncoding="UTF-8"
    redirectPort="8443" />
```

Tomcat has to be restarted in order for this change to take effect. Stop Tomcat now (we’ll restart it after installing Solr):

```
$ sudo service tomcat7 stop
```

Step 2: Install Solr

As of this writing, the version of Solr in the Ubuntu repositories is out of date. Download Solr 4.4 from the following URL:

http://www.apache.org/dyn/closer.cgi/lucene/solr/4.4.0

Extract the solr-4.4.x.zip archive and copy the Solr war archive to the Tomcat webapps directory and the supporting libraries to Tomcat’s lib directory:

```
$ unzip solr-4.4.0.zip
$ sudo cp solr-4.4.0/dist/solr-4.4.0.war /var/lib/tomcat7/webapps/solr.war
$ sudo cp solr-4.4.0/example/lib/ext/* /usr/share/tomcat7/lib/
```

Create a directory to hold the configuration and database files for the pWeb cores and extract the database configuration to that directory:

```
$ sudo mkdir /usr/share/solr
$ sudo tar -xjvf "pWeb solr config - July 30 2013.tar.bz2" -C /usr/share/solr
```

Now create a file called solr.xml in the directory

```
/var/lib/tomcat7/conf/Catalina/localhost
```

with the following content
Now start Tomcat again:

$ sudo service tomcat7 start

If you go to

http://localhost:8081/solr

in your web browser, you should see an administration interface for Solr without any errors. (Errors would show up in a big red box at the top of the page). You should see "pweb_content" and "pweb_devices" in the "Core Selector" drop-down box on the left side of the page.

Congratulations! You should now have a functioning Solr instance. Next you should set up an instance of the pWeb website if you have not done so already.

6 Installing and Configuring the pWeb Website

The all of the files necessary to run a copy of the main pWeb website are distributed as a compressed archive. The archive contains a README file that lists all of the steps necessary to configure and run a local copy of the website for demonstration purposes. The README file is included below for reference.

This folder contains the implementation of the pWeb public website at http://www.pwebproject.net, including the web interface to the device search feature. The following instructions tell you how to set up a local copy of the website on Ubuntu 13.04 (Raring Ringtail). The website has also been tested on Ubuntu 12.04 few, if any, modifications should be necessary to the following instructions when setting up the website on a different version of Ubuntu or GNU/Linux operating system.
Setting and Running up the pWeb Website Locally on Ubuntu 13.04

Step 1: Install the required software

The pWeb website uses the Django web development framework and django-cms. These can be installed using the following commands. If

$ sudo apt-get install python-setuptools
$ sudo easy_install pip
$ sudo pip install django==1.5.1 django-reversion==1.7.1 django-cms==2.4.2

Step 2: Extract website configuration

Create a directory for the website files and extract the contents of the archive to it:

$ sudo mkdir /var/lib/django
$ sudo tar -xjf "pWeb django website - July 30 2013.tar.bz2" -C /var/lib/django/

Django comes with a "development server" that is sufficient for running the pWeb website on a single machine. You can start the Django development server with the command

$ /var/lib/django/pweb/manage.py runserver

Now you should be able to go to

http://localhost:8000

in your web browser and see a copy of the pWeb website running locally. Test the device search page to make sure that it’s working. You may not have any entries in your database yet, but performing a search for, say "test," should not produce any errors.

Note: The device registration link in the top menu is hardcoded to the public registration page at http://cn109.cs.uwaterloo.ca/. It does not link to your local registration server, so make sure you know which pWeb network you’re using!
Congratulations! You now have a working local copy of the pWeb website. You should have already set up your DNS Gateway and Crawler, and search database. If you haven’t set up your Home Agent and registration server, do that next. Then you’re all done!
Appendix B:

DNS Gateway to Home Agent Protocol
1 Current DNS Gateway to Home Agent Protocol

The following pages contain the specification for the version of the DNS Gateway to Home Agent protocol currently in use. The protocol was designed to simplify connection handling in the implementation of the Home Agent software but suffers from a number of limitations that are discussed in Section 3.4.1.
All messages are sent over TCP connections. For every message sent, the sender opens a socket, writes the message to the socket, and then closes the connection.

The receiver listens for incoming connections. When a connection is established, the receiver passively reads a message and then waits for the connection to be closed by the sender.

There are two types of messages: PeerInitiateGET, and MessageGET_REPLY. Each of these two message types is preceded by an ABSMessage header.

**Field Types**
All integer values are transmitted in *little-endian* format. Integers may be signed (int) or unsigned (uint). The width of each integer type is denoted in bits (eg, 8, 32).

Double values are transmitted in IEEE 754 Double-Precision Floating Point Format. Each double is 8 octets long, transmitted in *little-endian* format.

Strings are transmitted as a 4 octet length field (int 32) followed by 'length' ASCII characters. Strings are not null terminated.

**Unused Fields**
Many fields are unused and must consist entirely of zero bits.

**ABSMessage**
The following fields are sent in order. The use and/or value of each field is explained for PeerInitiateGET (aka GET) and MessageGET_REPLY (aka REPLY) messages from the perspective of the DNS Gateway. GET messages are sent from the DNS Gateway to the Home Agent and REPLY messages are sent from the Home Agent to the DNS Gateway.

- **Message Type (uint 8)**
  - GET: The value of this field must be 5 for PeerInitiateGET
  - REPLY: The value of this field must be 43 for MessageGET_REPLY.
- **Sequence Number (int 32)**
  - GET: The DNS Gateway copies the DNS serial number received from DNS clients to GET messages sent to the home agent. (A more robust approach could be implemented if necessary by the DNS Gateway by hashing the DNS Client’s serial number with connection specific information such as source/destination IP addresses and port numbers.)
  - REPLY: Ignored.
- **Destination Hostname (string)**
  - GET: The hostname of the home agent that the DNS Gateway is sending the request to.
  - REPLY: Ignored.

- **Destination Port (int 32)**
  - GET: The port of the home agent that the DNS Gateway is sending this message to.
  - REPLY: Ignored.

- **Source Hostname (string)**
  - GET: The hostname of the DNS Gateway (replies from the home agent should be sent back to this hostname).
  - REPLY: Ignored.

- **Source Port (int 32)**
  - GET: The port number that DNS Gateway is listening on for replies sent over a separate TCP connection.
  - REPLY: Ignored

The following 7 fields are unused/ignored in both GET and REPLY and must be set to zero in messages sent from the DNS Gateway to the Home Agent:

- **Overlay Hops (uint 8)**
  - GET: Unused
  - REPLY: Ignored

- **Overlay TTL (uint 8)**
  - GET: Unused
  - REPLY: Ignored

- **IP Hops (int 32)**
  - GET: Unused
  - REPLY: Ignored

- **Latency (double)**
  - GET: Unused
  - REPLY: Ignored

- **Destination Overlay ID (int 32)**
  - GET: Unused
  - REPLY: Ignored

- **Destination Prefix Length (int 32)**
  - GET: Unused
  - REPLY: Ignored

- **Destination Max Length (int 32)**
  - GET: Unused
  - REPLY: Ignored

- **Source Overlay ID (int 32)**
  - GET: Unused
  - REPLY: Ignored
PeerInitiateGET
Messages of this type are sent from a DNS Gateway to a pWeb Home Agent in order to request that in resolve a device name to IP address. A PeerInitiateGET message consists of the ABSMessage Header and the following fields:
- Device Name (string)
  - The name of the device that the home agent is being asked to find the IP address for.

MessageGET_REPLY
Messages of this type are send from a Home Agent to a DNS Gateway in order to provide a response to a PeerInitiateGET message. A MessageGET_REPLY message consists of the ABSMessage Header and the following fields in order:
- Resolution Status (int 32)
  - Ignored
- Resolution Hops (int 32)
  - Ignored
- Resolution IP Hops (int 32)
  - Ignored
- Resolution Latency (double)
  - Ignored
- Origin Sequence Number (int 32)
  - Used as DNS sequence number
- Target Overlay ID (int 32)
  - Ignored
- Target Prefix Length (int 32)
  - Ignored
- Target Max Length (int 32)
  - Ignored
- Destination Hostname (string)
  - This field contains the IPv4 address for the device as a string in dotted-decimal notation if the lookup was successful.
- Destination Port (int 32)
  - Ignored. This field presumably contains the number of a port that can be used to access some sort of service on a device, but there is no standard mechanism in the DNS protocol to return this information to the DNS client.
- Device Name (string)
- Ignored.
2 Proposed DNS Gateway to Home Agent Protocol

The following pages contain the specification for a proposed DNS Gateway to Home Agent protocol.
Overview
The pWeb DNS Gateway connects to one or more home agents using TCP connections. In this document, the DNS Gateway may be referred to as the client and the Home Agent may be referred to as the server.

Unless specified otherwise, all integer values are transmitted as unsigned integers in network byte order (big endian). IP address values are also transmitted in network byte order.

This is a preliminary version of the protocol. It may evolve and change over time.

Connection Establishment
The DNS Gateway will be configured to connect to one or more home agents. When the DNS Gateway starts up, it will attempt to establish a TCP connection with each home agent at the hostname and port specified in its configuration. This TCP connection will be used for all name lookup requests sent from the DNS Gateway to the Home Agent. If the connection is closed unexpectedly, the DNS Gateway will attempt to reestablish the connection.

Handshake
The handshake consists of the client sending a version number to the server as a two octet integer. In this version of the protocol, the version number is 1.

After receiving the version number, the server should verify that it supports the version of the protocol specified and then send the same version number back to the client. The connection is now fully established and the client can begin sending queries to the server.

Messages
Each message starts with a two octet integer that identifies the type of the message (message ID). Currently two message types are defined:

- QUERY = 1
  - A request to resolve a device name to an IP address sent from the client to the server. This message is called QUERY.

- REPLY = 2
  - A response sent from the server containing a status code and optionally an IP address. This message is called REPLY.

QUERY Message
A query message starts with a two octet serial number. The serial number is used to match REPLY messages with QUERY messages. Reply messages can be sent in any order—they do
not need to be sent in order that the QUERY messages were received. The client must ensure that all unanswered queries have unique serial numbers.

After the serial number is a two octet integer specifying the length (in octets) of the device name being queried, followed by the device name encoded in ASCII format.

**REPLY Message**
A reply message starts with the same two octet serial number sent with the corresponding query. Next is one octet status code. Currently two status codes are defined:

- **SUCCESS = 0**
  - The device name was resolved to an IP address successfully and an IP address follows the status code (see below).
- **GENERAL FAILURE = 1**
  - The Home Agent was unable to resolve the device name to an IP address.

A status code of 0 must be followed by an IP address. An IP address is sent as a one octet version number (4 for IPv4 and 6 for IPv6) followed by either 4 octets for an IPv4 address or 16 octets for an IPv6 address.

**Miscellaneous**
When the connection between the DNS Gateway and Home Agent is terminated, all outstanding queries/requests must be cancelled.

The server must respond to all queries within 30 seconds. If it is unable to resolve a name within 30 seconds, then it must send a reply indicating failure.

**Examples**
The following are examples of messages that could be sent from the client to the server and server to client in a session. The response to the query with serial number 8547 may be sent before the response to the query with serial number 24513.

All integers represented in base 10.
One box (marked with "-----") indicates one octet.

```
CLIENT -> SERVER

<table>
<thead>
<tr>
<th>Msg ID</th>
<th>Ser #</th>
<th>Len</th>
<th>Device Name</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>1</td>
<td>14</td>
<td>nexus . alice . uw</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Msg ID</th>
<th>Ser #</th>
<th>Len</th>
<th>Device Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>8547</td>
<td>16</td>
<td>laptop . shihab . uw</td>
</tr>
</tbody>
</table>
```
## SERVER -> CLIENT

<table>
<thead>
<tr>
<th>Msg ID</th>
<th>Ser #</th>
<th>Stts IPv IP Address</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>2</td>
<td>8547 0 4 2170661478</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Msg ID</th>
<th>Ser #</th>
<th>Stts</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>1</td>
<td>24513</td>
</tr>
</tbody>
</table>

---
3 Report Produced by ResPerf

The following pages contain a report produced the ResPerf DNS performance testing tool. The test was run using the mock Home Agent, DNS Gateway and performance testing tool on the same machine.
Resperf report 20130414-2054

Resperf output

DNS Resolution Performance Testing Tool
Nomun Version 2.0.0.0

[Status] Command line: resperf -P 20130414-2054.gnuplot -s 127.0.0.1 -p 5354 -d 1000000-dht
[Status] Sending
[Status] Reached 65536 outstanding queries
[Status] Waiting for more responses
[Status] Testing complete

Statistics:

<table>
<thead>
<tr>
<th>Metric</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Queries sent:</td>
<td>94571</td>
</tr>
<tr>
<td>Queries completed:</td>
<td>94571</td>
</tr>
<tr>
<td>Queries lost:</td>
<td>0</td>
</tr>
<tr>
<td>Run time (s):</td>
<td>100.0000</td>
</tr>
<tr>
<td>Maximum throughput:</td>
<td>7084.00000 qps</td>
</tr>
<tr>
<td>Lost at that point:</td>
<td>10.53%</td>
</tr>
</tbody>
</table>

Plots

```
Query / response / failure rate

<table>
<thead>
<tr>
<th>Time (seconds)</th>
<th>Queries sent per second</th>
<th>Total responses received per second</th>
<th>Failure responses received per second</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>2000</td>
<td>2000</td>
<td>0</td>
</tr>
<tr>
<td>4</td>
<td>4000</td>
<td>4000</td>
<td>0</td>
</tr>
<tr>
<td>6</td>
<td>6000</td>
<td>6000</td>
<td>0</td>
</tr>
<tr>
<td>8</td>
<td>8000</td>
<td>8000</td>
<td>0</td>
</tr>
<tr>
<td>10</td>
<td>10000</td>
<td>10000</td>
<td>0</td>
</tr>
<tr>
<td>12</td>
<td>12000</td>
<td>12000</td>
<td>0</td>
</tr>
</tbody>
</table>
```
References


