pWeb : A Personal Interface to the World Wide Web

Reaz Ahmed*, Shihabur Rahman Chowdhury*, Alexander Pokluda*, Md. Faizul Bari*,
Raouf Boutaba* and Bertrand Mathieu†
†David R. Cheriton School of Computer Science, University of Waterloo, Canada
{r5ahmed | sr2chowdhury | apokluda | rboutaba}@uwaterloo.ca
†Orange Labs, Lannion, France
bertrand2.mathieu@orange.com

***Abstract*—Centralized social networking and media sharing portals provide inadequate support for preserving user privacy, content ownership and control. These problems can be mitigated through distributed Web services as demonstrated by a number of academic projects and industrial deployments. In general, these distributed services do not assign globally recognized, persistent names to the user devices. As a result, these solutions work in isolation and also cannot inter-operate with traditional Web technology. In this work, we present a decentralized and scalable platform, named pWeb, for distributing web services, like online social networks and media streaming, across end user devices. pWeb assigns Internet compatible names to end user devices, and provides name resolution and directory services. A user can retain ownership, and make the services and contents in his devices searchable and accessible at different privacy levels, e.g., friends, family and public. New services can be easily developed and deployed over the pWeb platform. We have developed a working prototype of the platform, and to demonstrate its effectiveness we have implemented a video streaming application for Android and Windows platforms. We also present performance results from our prototype implementation.**

I. Introduction

The World Wide Web has become the primary means for sharing personal multimedia contents among friends, family and the public. Media sharing portals, like YouTube, Metacafe and Flicker allow us to share videos and photos for free. Online social networks, like Facebook, Twitter and Google+, offer personal blogging and multimedia content sharing for free. However, these free Web services are just one side of the coin. The other side comes with a number of issues involving privacy, ownership and control.

In many cases, our real-life privacy is compromised through our online activities. We trust the Web service providers with our uploaded contents. But in the background, our online contents and activities are being analyzed and monitored by different agencies. The recent PRISM scandal illustrates the reality of large-scale digital surveillance. Apart from the privacy issue, we also run into the risk of loosing the ownership and control of our personal digital contents. Many service providers hold the right to reproduce, publish and distribute the uploaded contents. Moreover, in most cases, the end user license agreement can change without notice.

Decentralized solutions have been proposed to mitigate privacy, ownership and control problems in contemporary centralized Web services. For example, PeerSon, Persona and SafeBook propose to distribute the contents in an online social network over the end-user devices. Peer-to-peer video streaming solutions, like PPLive and Tribler, focus on distributing contents over their users' machines instead of uploading them to a centralized media streaming portal. However, all of these solutions treat user devices as second-class citizens in the Internet. User devices are not assigned globally recognized and persistent names. As a result, each of these solutions works in isolation and also cannot inter-operate with traditional Web technology.

Now imagine what can be achieved if each device had a unique, DNS-compatible name. We could use a regular Web browser to remotely control, configure and access the services and contents in our devices. We could share contents with our friends and family directly from our devices. New applications would emerge to improve our privacy through better control on our contents. Moreover, a search engine could index public shares in the end devices, giving them a global visibility. With this vision in mind, we have developed a simple but effective framework, named pWeb (www.pwebproject.net), for device naming and searching. We envision pWeb as a global platform for distributed Web services. pWeb is compatible with DNS and contemporary Web search engines. Each component within pWeb can be deployed independently without any central control. pWeb provides a global platform for developing distributed Web services. We have also developed a distributed video-streaming application to demonstrate the effectiveness of pWeb. The purpose of this paper is to present the pWeb architecture, along with our experience and lessons learned while developing and deploying pWeb.

The rest of this paper is organized as follows. In §II we present the related research works. We present the design goals and an overview of the pWeb architecture in §III followed by the functional specification in §IV. Design, implementation and deployment details for each component in the pWeb architecture are presented in §V. In §VI, we present the performance results from our live pWeb deployment in the Internet. Finally, we conclude in §VII.

II. Related Works

In this section, we discuss a representative set of research works on decentralized online social networks, device to device communication for decentralized Web service hosting and alternative name resolution architectures to handle the explosion of the number of mobile devices in use.

A. Decentralized Social Networks

A number of recent research efforts strive to address the privacy, ownership and control problems in centralized Online

Social Networks (OSNs). These research works propose a decentralized architecture for OSNs to overcome the limitations posed by the cloud-based centralized architecture as described in [1]. PeerSon [2], SafeBook [3] and SuperNova [4] are some of the prominent proposals for decentralized OSNs. These systems offer a variety of services ranging from location-based social networks to social music sharing services and also differ in the degree of decentralization they offer. However, they focus more on social network-like use cases, rather than providing a generic platform for developing decentralized Web services. A comprehensive survey of these decentralized OSNs can be found in [5].

B. Device to Device Communication

Recently, FreeDOM [6] proposed a Web browser-based platform for decentralized Web service hosting. Current Web browsers support a number of technologies like WebRTC [7] for direct interaction between Web-browsers, WebSQL [8] for key-value store, and IndexDB [9] for trusted and untrusted data storage. FreeDOM proposes to leverage these browser technologies to build a decentralized platform for Web hosting. The authors also demonstrate the design of a decentralized Wikipedia-like service using the proposed FreeDOM model. However, the FreeDOM model does not focus on the identification of the end devices. More recently, authors of [10] proposed *Clone2Clone*, a cloud-based communication and computation offloading mechanism for device-to-device communication. The authors propose to host a smartphone's clone image in a public or private cloud platform and perform all the computation along with the device-to-device communication between these clones. An end device needs to communicate with the cloud to synchronize the state with its clone. This type of clone-based peer-to-peer network also allows users to directly host content from their device. However, creating clone images of the smartphones and hosting them in cloud platforms require formidable storage and computation resources. Hence, their proposed method is more suitable for the device or firmware provider rather than the smartphone users. Clone2Clone's dependency on cloud services also inherits all the privacy and ownership concerns that we aim to resolve using pWeb.

C. Alternate Name Resolution Architectures

A number of research works have been carried out in the context of Information Centric Networking (ICN) for alternative name resolution architectures. The idea underlying ICN is to perform routing based on content names. The target is to make the network aware of the content it is carrying. New naming schemes and name resolution architectures have been proposed to facilitate name-based routing. A comprehensive survey of the ICN architectures can be found in [11]. More recently, DMap [12] has proposed a global name resolution service for supporting mobility and efficient content delivery in the Internet. DMap assigns a globally unique flat identifier (GUID) to each content. A content's GUID is hashed to obtain a list of IP addresses and the contents original location is indexed at those addresses. These indices are updated whenever the content is relocated. However, every network entity in DMap is required to have a global knowledge of the IP prefix advertisements from all the autonomous systems to successfully locate the index locations of a content.

Dynamic DNS (DDNS) provides a way to access and assign names to user devices residing even in private networks. This requires the users to configure their home gateways to enable port forwarding. However, the DDNS service providers do not collaborate. They work in isolation. Their client software and communication protocols are incompatible. On the contrary, pWeb provides a global platform for the organizations to collaborate. pWeb enables organizations to provide naming service to their clients. It assigns global names to user devices that can work with or without the legacy DNS. And finally, pWeb supports device mobility similar to DMap. However unlike DMap, pWeb does not require the participants to know about global IP prefix advertisements.

III. pWeb Architecture

A. Design Goals

In this section we define the goals that we want to achieve with the pWeb architecture.

First and foremost, we want to assign each end user device in the Internet a globally unique name that is compatible with DNS. DNS compatibility is essential for seamless integration with existing Web technology. Globally unique names will allow device identification, and it will be easier to access a device for various Web services.

Second, the name resolution time should be comparable to that of DNS. This is a challenging goal for two reasons: a) the sheer volume of end-user devices and b) the high frequency of change in name to IP address binding.

Third, we want to make the devices, and the services hosted in them searchable. The search process should have a low response time and low resource usage. The search interface should use standard Web technology.

Fourth, there should be a clear separation between the components in pWeb. This is essential to allow independent evolution and control. We envision pWeb as a global platform for distributed Web services. Hence, we expect that the components in pWeb will be administered and deployed by independent entities.

B. Overview

In this section we present a simple architecture that can achieve the goals explained in Section III-A. Figure 1 presents the functional components in this architecture. There are three layers in the architecture: service layer, resolution layer and index layer. Below we explain the components in each layer and their functionalities:

1) Service layer: This layer consists of the users and their devices. Users are virtual entities with authentication credentials. A user may register one or more devices under his credential. Web services, like content sharing, streaming, synchronization *etc.*, are distributed in the end-user devices. Each client device runs a pWeb client software that makes the services in the device available over the HTTP protocol.

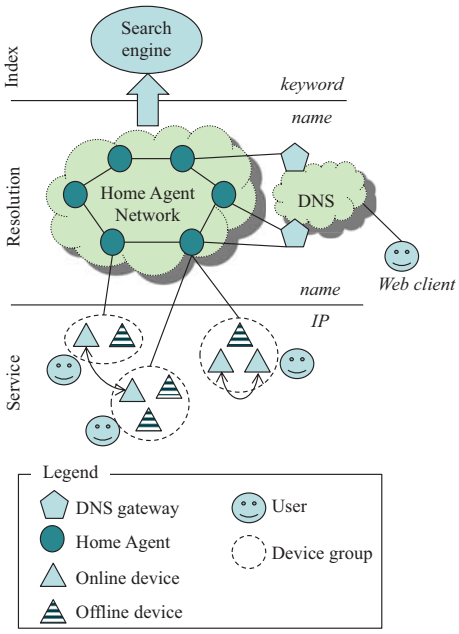


Fig. 1. Functional components in pWeb

2) Resolution layer: This layer consists of a network of stable servers, called Home Agents (HA). HAs will be deployed by different administrative entities. A user has to register with a HA. For each registered user, a HA maintains some meta-information, like the user's credentials, the devices registered by that user and the services running in each device. HAs collaborate with each other using a Distributed Hash Table (DHT). This DHT indexes only the HA names. In addition to the DHT links, each HA maintains shortcut links to other HAs. The HA DHT ensures that a HA can find any other HA in two to three overlay hops. This layer also contains DNS gateways that translate DNS requests to DHT lookup queries and sends the results back. DNS gateways ensure compatibility and integration with legacy DNS.

3) Index layer: Searching devices and services using meta-information (*e.g.*, user name, descriptive keywords, location *etc*.) is an essential functionality in pWeb. Due to the shear volume of devices and services it will not be feasible to index the device and service names within the HA network in a distributed fashion. Rather, the HA network indexes only the HA names using a Distributed Hash Table (DHT). Each HA keeps track of the users, devices and services registered in that HA. We use a dedicated search engine for enabling meta-information search in pWeb. Similar to the Web search engines, our search engine can be deployed in compute clusters (dataceters). The search engine crawls the HA network for available devices and services. It generates a global index to respond to user queries.

IV. pWeb Functional Specification

A. Naming

In the pWeb platform, we have to name users, their devices and the services running in those devices. The naming system in pWeb should have the following properties: 1) *human readable*: the names should be human friendly so that users can easily remember the names and search by names; 2) *DNS compatible*: to allow seamless integration with Web technology, pWeb names should be inter-operable with DNS; 3) *performance*: the name resolution system should be efficient in network usage, low in response time, scalable in the volume of names and name resolution requests, and effective in handling the update frequency of name to IP address binding.

We have used a hierarchical naming scheme to achieve the aforementioned requirements. An abstract name in pWeb and an example are give below:

Abstract: `device.user.ha-name.dns-gw/service`
Example: `nexus.bob.uw.pwebproject.net/camera`

This example identifies the `camera` service in a device named `nexus` that is registered under user `bob`. In addition, user `bob` and his devices are registered to the HA named `uw`. Here, `pwebproject.net` refers to a DNS gateway and can be replaced with any other DNS gateway URL. We have developed a portal for user and device registration. This portal can be accessed through pWeb project's homepage at `http://pwebproject.net`.

It is worth noting that the name uniqueness is ensured at each level of the naming hierarchy. The HA overlay ensures that each HA gets a unique name. At the next level, each HA ensures that the registered users get unique names. Finally, it is the responsibility of a user to assign unique names to his/her devices. A user cannot assign the same name to multiple devices.

B. Interfaces

The HA nodes provide a RESTful interface for interacting with external entities. The RESTful API has a common format of `http://ha-name:port/?method=method-name&parameter-list`. We have defined standard interfaces for communication between HA and other components in pWeb. In Table I we highlight the interfaces that will be mostly used by pWeb app developers.

1) Interface between HA and client: This interface provides users with registration and authentication services. New users can register with the system through our registration portal, which in turn uses this interface to update a HA. This interface also allows users to register their devices. The IP Updater Module in the client software uses this interface to update the HA whenever the IP address of the client device changes. This feature is necessary to support device mobility and allows access to devices behind NAT boxes.

2) Interface between HA and crawler: The device crawler periodically polls the HAs using this interface to retrieve the registered devices and content meta-data. The HA also provides the crawler with a list of neighbour HAs through the REST API calls. This list enables the crawler to start a set of seed HAs and proceed in a breadth-first manner.

C. Indexing and Discovery

The sequence diagram in Figure 2 presents the main processes in pWeb, namely, registration, crawling, searching and name resolution. In order to use pWeb, a user, say *Bob*, must register himself and his device, say *nexus.bob*, to a HA

REST API Method Name	Description
HA Interface with Client s/w and Registration portal	
`register_user`	Register a new user at an HA
`authenticate_user`	Authenticate a user at an HA against his provided credentials
`get_all_user_device`	Return all the registered devices of a given user
`update_user_device_ip`	Update the IP address and listening port number of a given device for a given user
`update_service_info`	Update content meta data stored in a given device of a given user
HA Interface with Crawler	
`get_all_device`	Return the list of all devices along with their owner information stored at an HA
`get_service_meta`	Return the list of service meta-data stored at an HA

TABLE I. SUMMARY OF HOME AGENT INTERFACES

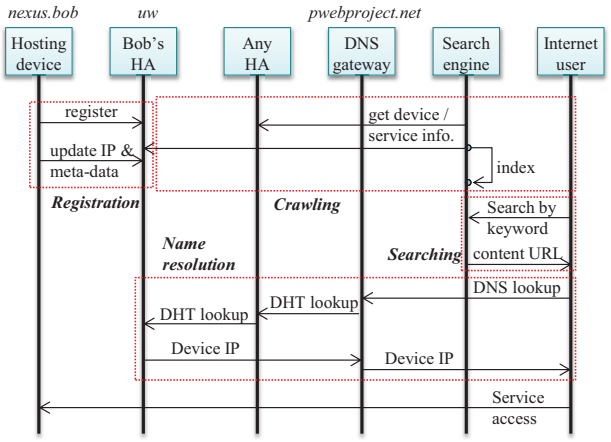


Fig. 2. Registration, crawling and discovery processes in pWeb

of his choice,and install the client software to his device. The client software updates the device's IP and information on the hosted services to Bob's HA, *i.e.*, *uw*. The search engine, on the other hand, crawls the HA-network for available devices and services, and indexes them. A user can discover the device and service URLs (*e.g.*, *nexus.bob.uw.pwebproject.net*) by searching the crawler's index. The user's browser will resolve this URL transparently as follows: the DNS request will be forwarded to the DNS gateway (*i.e.*, *pwebproject.net*); the DNS gateway will forward the request to Bob's HA (*i.e.*, *uw*) using the HA overlay; Bob's HA will respond with the most recent IP address of the hosting device (*i.e.*, *nexus.bob*), and; finally, the IP address will be forwarded to the requesting browser or application, and it will directly access the desired service in Bob's device.

V. DETAILED DESIGN AND IMPLEMENTATION

A. Home Agents

1) Design: Figure 3 shows the architecture of a single Home Agent (HA) node. The HA nodes work together to form a Distributed Hash Table (DHT). Any DHT scheme can be used to realize the HA network. We have used Plexus [13] for our implementation. As in all DHTs, each HA node maintains a list of HAs (logical neighbours) for forwarding any name look up or registration request. The DHT formed by the HA nodes index the HA names in the overlay network. A HA has the following functional components:

a) Messaging Interface: A HA provides a messaging interface for communicating with external entities, *i.e.*, the

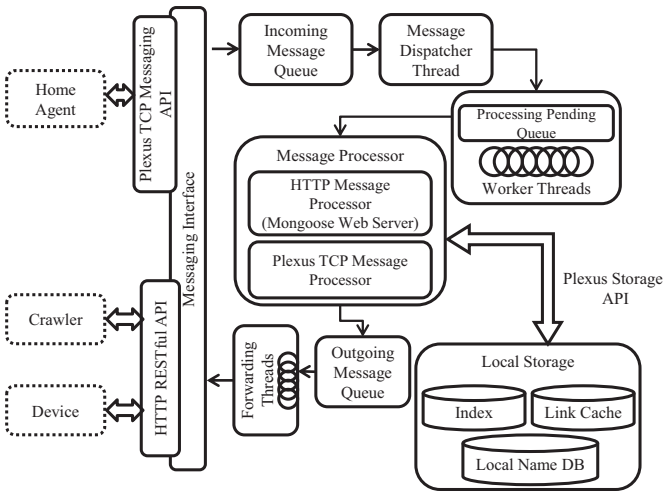


Fig. 3. Home agent architecture

crawlers in search engine, DNS gateway, end device and other HAs. The HA messaging interface is comprised of two parts: (i) a TCP messaging API for communicating with other HAs and the DNS gateway, and (ii) a HTTP RESTful messaging API for communicating with the end devices and the crawlers. We preferred raw TCP messaging over HTTP for HA-DNS gateway and HA-HA communication in order to improve name lookup performance.

b) Message Queues: A HA node also maintains a number of message queues delegated to different purposes. All incoming messages received through the messaging interface are placed into an *incoming message queue* and wait for dispatching. After a message is dispatched from the incoming message queue and processed, it is placed into an *outgoing message queue* if it requires further forwarding. In our implementation, we also have an optional queue for the log entries from different threads (not shown in the figure, since it is not part of the HA design). Log entries in this queue can be remotely monitored for tracking multiple HAs from a central location.

c) Dispatcher, Forwarding and Worker Threads: A *message dispatcher thread* acts as a switch and assigns a *worker thread* to an incoming message for further processing. A worker thread contains a message processing logic, which takes decisions for the assigned messages. There are separate types of worker threads based on the type of the received message (HTTP message and TCP message). The worker thread decides whether the message needs to be further forwarded or the request can be locally served. Messages requiring further forwarding are placed in an outgoing message queue and a number of *forwarding threads* dispatch these messages through the HA's messaging interface.

d) Local Storage: The local storage system contains (i) the DHT routing table, i.e., a list of HAs (logical neighbours) for forwarding a query (ii) a cache for storing non-neighbour HAs. This cache is used in addition to the regular routing table for making forwarding decisions; (ii) a local index where HA name-to-IP address information is indexed using Plexus; and (iii) a name database, which contains metadata on the registered users, devices and services.

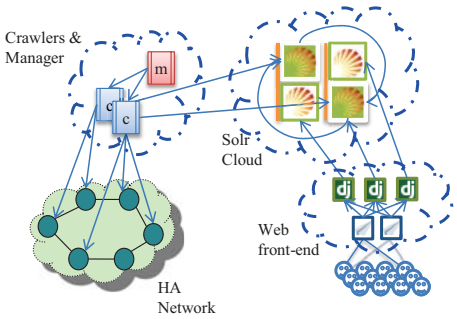


Fig. 4. Search engine architecture

2) Implementation: One of our implementation goals was to make the HA software as lightweight and platform-independent as possible. With this goal in mind we have developed the HA software in C/C++. The POSIX thread library (`pthread`) has been used for multi-threading and synchronization purposes. For storage purposes we used SQLite 3, a lightweight database engine. The remainder of this section contains more details regarding the implementation of different HA components.

As described earlier in Section III, a HA's messaging interface comprises a TCP messaging API and a RESTful HTTP messaging API. We have used mongoose [14], a lightweight open source web server written in C, to implement the RESTful API. For the TCP messaging interface we have used the POSIX socket interface. We have defined custom messages for the communication between the HAs and with the DNS gateway. These messages are converted to binary data streams before sending through the TCP socket as payload. These custom messages have the necessary serialization and de-serialization methods for converting to and from binary data.

In our implementation, the incoming and outgoing message queues have been implemented using STL's `queue` data structure. The message queues are shared between different threads. We have ensured synchronization by using `pthread` library's `read_write_lock` primitives. `pthread` library has also been used for implementing the threads used in HAs.

The routing table and link caches have been implemented using the STL `map` data structure with necessary synchronization primitives. We need a relational database for efficient management of user, device and service information. However, the popular relational database engines like MySQL or Oracle use a lot of resources, which contradicts our goal of keeping the HAs lightweight. We therefore chose SQLite, a self-contained, serverless, zero-configuration, transactional SQL database engine [15]. SQLite has been interfaced with our HA process through the `libsqlite3-dev` library.

B. Search Engine

1) Design: The search engine allows Internet users to discover pWeb users, devices, and services by building a searchable database containing metadata for all devices and services in the pWeb network. The Web front-end constructs a URL in the search results that can be used to access content directly from end devices. 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 the HA overlay to retrieve the device's last known IP address. The search interface is one of the main methods that users locate services in pWeb.

The search engine architecture is shown in Figure 4. We describe the major components in the following:

a) Crawlers and Manager: The pWeb Crawler process is responsible for polling HAs in order to discover new and updated devices, services, and content in the pWeb network. It uses the HA's RESTful interface to retrieve information about the HA itself, its neighbours, registered devices, and new services and content. The crawler uses the list of the HA's neighbours in order to discover all the HAs in the HA overlay.

A single Crawler process can be run in a standalone mode or any number of Crawler processes, coordinated by a Manager process, or can run in parallel in order to scale additional orders of magnitude. If, and only *if*, a polled HA returns information about one or more updated devices, the Crawler process will post an update to an Apache Solr server. Solr, to be discussed further in the next section, is an enterprise-search server and database. In pWeb, Solr is used to store and search among user, device, and service metadata.

b) Solr Cloud: Solr is an open-source, enterprise-search database that is used by many large companies including AOL, eBay and Netflix, to power the search features on their public websites. Like the Crawlers, a single Apache Solr instance can be run in a standalone configuration, but Solr can also be scaled to handle large databases using replication and sharding. A distributed configuration of Solr is known as a Solr Cloud.

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 service metadata into the database. The public search interface is implemented as a Django Web application that also uses this interface. The search interface will be discussed further in the next section.

c) 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.

When a user issues a query from the search interface, 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 which are passed to the Template layer that renders them as HTML to be displayed to the user. The Web interface does not store any context or state information, so its scalability is not a concern. Any number of Web front-ends can be deployed in parallel and the domain name system, a load balancer, or both can be used to evenly share the load between them.

2) Implementation: The pWeb Crawler and Manager processes have been programmed from scratch with scalability and performance in mind. They exclusively use asynchronous I/O for network communication through the Boost Asio library [16], which is a cross platform C++ library for network and low-level I/O programming and has been proposed for

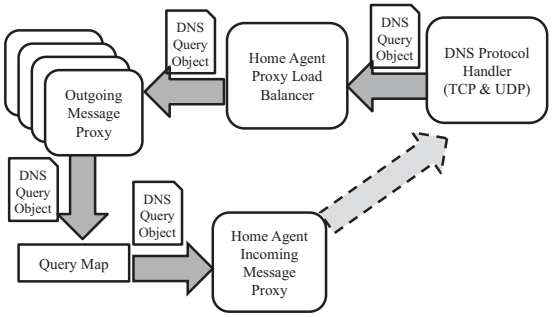


Fig. 5. Components in the DNS Gateway

inclusion in the next version of the official C++ standard. The Asio library uses the most efficient asynchronous I/O API offered by the operating system.

The asynchronous I/O approach ensures the software does not limit scalability and takes 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. This decouples the threading model from network operations, enabling the application designer to use as many or as few threads as they see fit.

C. DNS Gateway

1) Design: The primary purpose of the DNS Gateway is to translate DNS query messages to name lookup messages in the HA protocol and back again. The main conceptual components of the DNS Gateway are shown in Figure 5. A HA Outgoing Message Proxy object is created in the DNS Gateway upon startup for each configured HA. A single HA Incoming Message proxy is responsible for receiving messages from all HAs. The main components of the DNS Gateway are explained below in terms of a query flowing through the system. The gateway also contains logging and instrumentation subsystems, which are omitted for clarity.

When a DNS message arrives at the DNS Gateway, the DNS Protocol Handler receives the message and verifies that the gateway can process the message. If the query is valid, the DNS Protocol handler constructs a query object to represent the query and passes the query object to the HA Proxy Load Balancer, which passes it to one of the HA Outgoing Message Proxy objects. Once the outgoing message proxy receives the DNS query object, it constructs an HA protocol message for the query. This message will contain the device name to be looked up as well as the DNS query sequence number. It then instructs the DNS query object to start an internal timeout timer, stores the query object in a shared map using the sequence number as a key, and sends the HA message.

When a reply message is received from the HA, the HA Incoming Message Proxy extracts the name lookup result and DNS query sequence number from the HA protocol message and retrieves the DNS query object from the shared map. It stores the name lookup result in the DNS query object and instructs it to send a reply back to the client. If the DNS query timeout timer expires before a reply message is received from the HA, the DNS query object will then remove itself from the map and send an error code back to the client.

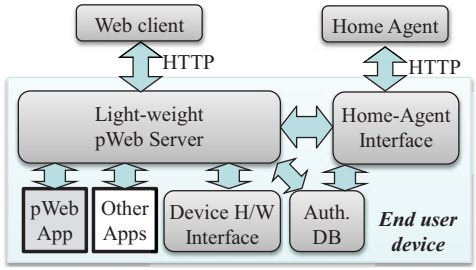


Fig. 6. Components in the pWeb client software

2) Implementation: The implementation of the DNS Gateway parallels that of the Crawler and Manager. All network operations are asynchronous and implemented using the Boost Asio library. The 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 Crawler and Manager processes can take advantage of all available processing power if necessary on the host machine. A single pWeb Crawler process can monitor tens of thousands of HAs with short polling periods on modest hardware, as shown in Section VI.

D. Client Software

1) Design: The client software is an integral part of the pWeb platform. It runs at an end user device and facilitates seamless integration of the existing third-party apps (*e.g.*, remote access to IP cameras, NAS, *etc*.) and new apps to be developed for the pWeb platform. It uses the HA's RESTful API to publish the pWeb apps (*i.e.*, services) running in end-devices. Major components of the client software are depicted in Figure 6 and described as follows: (a) The *pWeb server* is a lightweight HTTP server that works as a translator between HTTP messages and native function calls. It works as a relay between the hosted apps and Web clients (*e.g.*, a web browser or another pWeb app). Third-party apps hook into the pWeb server as plugins. The pWeb server also provides API for the apps to access device hardware resources and HA interface; (b) The *HA interface* encapsulates the REST API between HA and pWeb client; (c) The *authentication database* (Auth. DB) stores user credentials. The pWeb server uses this module to check the authenticity of an incoming request, while the HA interface retrieves user credentials from this module for device registration and IP update processes; (d) The *Device H/W Interface* is an extendable library that abstracts the hardware resources from app developers, and; (e) The client software contains a built-in *pWeb App* that provides a GUI for configuring different components in the client software.

2) Implementation: Instead of developing the client software from scratch, we extended the Jetty webserver for Windows and its Android counterpart, iJetty. We developed the pWeb app as a plugin to the extended iJetty server. The pWeb app provides the essential user interface needed to configure the client software and register devices and services. We developed a video app on the client software, which allows a user to record video using his device's camera. The user can then choose to share the videos as public or private. Another

user can locate the shared videos using the pWeb search portal and stream the videos directly from the end user device.

VI. Experimental Results

In this section, we present experimental results showing the performance of the proposed name resolution infrastructure and device crawler. The name resolution infrastructure is the basis of all future pWeb services, while the search engine builds the database for discovering content and services in pWeb. Therefore, we focus on the performance of these two components rather than showing the performance of any service deployed over pWeb. We begin by describing the experimental setup in Section VI-A, followed by the evaluation scenarios in Section VI-B. The results are discussed in Section VI-C.

A. Experimental Setup

We experimented separately with the name resolution infrastructure and the device crawler. We had two experimental setups when experimenting with the name resolution infrastructure. In the first setup, we deployed the HA nodes and DNS Gateways in 25 PlanetLab nodes [17] spread across the globe. Figure 7 shows the geographical placement of the HAs and DNS Gateways. We setup a local BIND DNS server in one of our local server machines in Canada, which is responsible for resolving names with the suffix `pwebproject.net`. It is configured to be geo-aware, *i.e.*, it forwards a DNS query to its geographically nearest DNS gateway. If a name is searched multiple times, the DNS query will reach the DNS server in Canada only once, since the geographically nearest DNS gateway will be cached at the client. Therefore, the local DNS server does not become a bottleneck node in the system. The name-resolving clients were placed in 30 PlanetLab nodes spread across the world. They use the *Domain Information Groper* (`dig`) command line tool to query for names and measure the name look-up latencies.

In the second setup we performed micro-benchmarking of a HA node to test its scalability. To perform this experiment, we ran one instance of an HA process on a machine with a quad-core Intel Xeon CPU running at 2.13 GHz and 12 GB of RAM. In another machine, we ran a micro-benchmarking tool that can generate a specified load on the HA. The HA process used for this experiment is multi-threaded, using different threads for listening to the incoming requests, processing messages, and sending out replies. We then tested the HA's scalability in this setup with various loads.

For the name data set, we collected anonymized DNS traces from our campus network at the University of Waterloo. We used this trace to generate both the set of names and the queries. In our experiment we used 2.5×10^5 unique names and generated a Zipf distributed query sequence from this name data set. For the Zipf distribution we used the parameters from [12] since it represents a realistic model of content request in the Internet. The query sequence was divided among the 30 PlanetLab clients and each client performed 5×10^5 queries in parallel.

We then performed a micro-benchmark of the Crawler run in standalone mode on a server with the same specifications as the HA micro-benchmark. The Crawler ran on a large simulated HA network that consisted of simulated HA output which was statically generated offline and then served from a RAM disk by an Nginx server. Delays were introduced by a modified Nginx echo module to simulate the HAs being geographically distributed. Nginx is an open-source, high-performance HTTP server[1] and the echo module enabled a number of features that are useful for software testing, including the server performing an asynchronous sleep for a specified duration before returning the HTTP response.

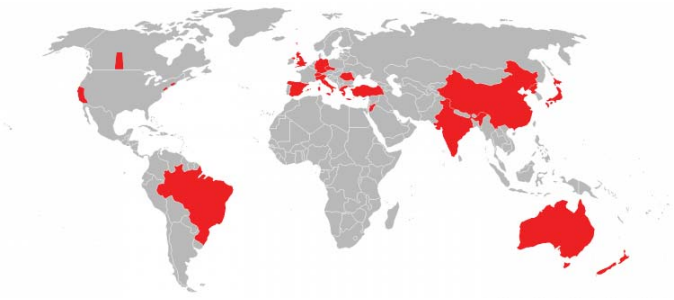


Fig. 7. Geographic Placement of Home Agent and DNS Gateway

The simulated HA network was constructed with the assumption that the Crawler would be located in Canada and have access to HAs distributed around the globe. To construct the simulated HA output, we first obtained the P2PSim King dataset[2] which contains estimated round-trip times between pairs of hosts on the Internet. Since this data was from June 2004, which was the most recend that we could find, we obtained the Maxmind GeoIP database also from June 2004 and translated all IP address pairs to countries. We were able to obtain estimated average latencies from Canada to 116 other countries.

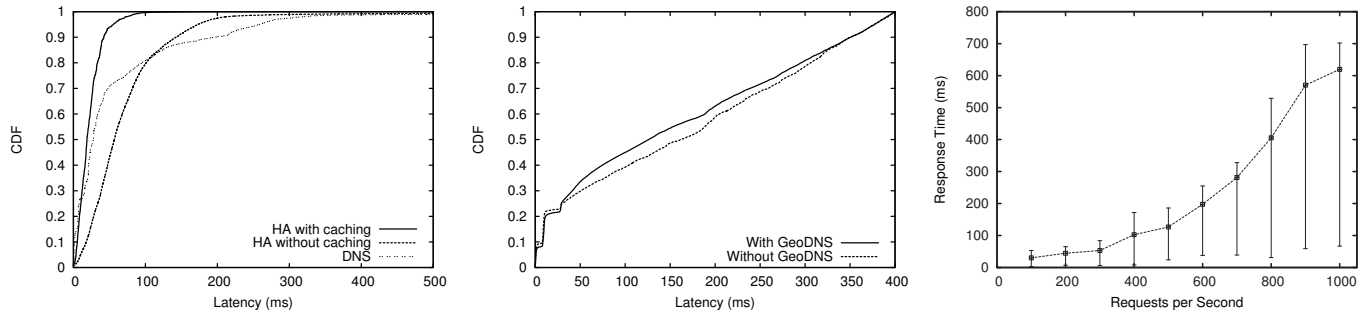
Next, we modified the third-party Nginx echo module to sleep for durations based on the distribution of these latencies. Each request to a simulated Home Agent includes that HA's country code in the HTTP GET query string. The modified echo module uses this country code to index into a table to retrieve the estimated latency from Canada to that country, then performs an asynchronous sleep for that duration, effectively delaying the HTTP response.

We wrote a Python script to generate the simulated ouput for each HA. The script contains the number of mobile phone subscribers per country in 2012, which we obtained from the World Bank Open Data website[3]. The script takes a parameter the percentage of all mobile phone subscribers assumed to be using pWeb. It generates the simulated HA output such that each country has at least one HA and there are at most 50,000 users per HA. The number of simulated HAs generated for each country is proportional to the number of mobile phone subscribers in that country.

B. Experiment Scenarios

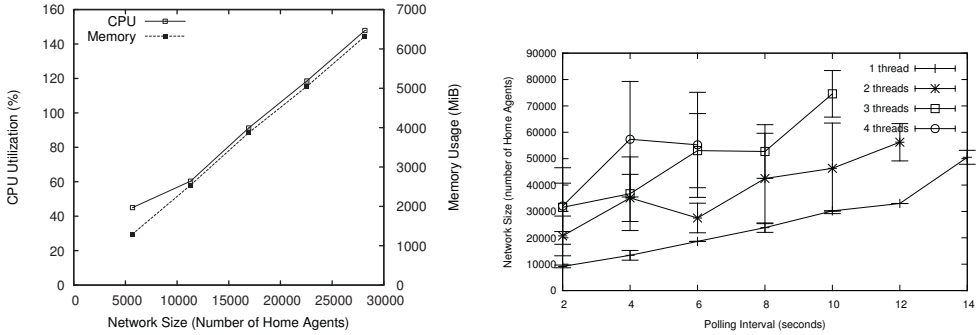
We evaluate the performance of the name resolution infrastructure in the following scenarios:

[1] http://nginx.org
[2] http://pdos.csail.mit.edu/p2psim/kingdata/
[3] http://data.worldbank.org/indicator/IT.CEL.SETS



(a) Name Look up Efficiency of Home Agent (b) Impact of GeoDNS on Name Resolution (c) Home Agent Scalability

Fig. 8. Performance of Name Resolution Infrastructure



(a) Impact of Network Size on Crawler Resource Requirement (b) Impact of Polling Interval on Supported Network Size

Fig. 9. Performance of the Crawler

Home Agent performance: In this scenario, we experimented with the HA network only, *i.e.*, without the DNS gateway and local DNS servers. We published the name dataset within the HA network and the HA nodes performed name lookup amongst themselves. The goal of this experiment was to show: (a) the name lookup performance of Plexus protocol running in the HA network; and (b) the impact of link caching within the HA network.

Effect of geo-awareness: In this scenario, we showed the impact of having geo-awareness in name resolution. To illustrate this, we performed two separate experiments with geo-awareness enabled and disabled, respectively. When geo-awareness is disabled, the local DNS server is configured to forward a name resolution query to a random DNS gateway in PlanetLab.

Home Agent scalability: In the final scenario, we performed micro-benchmarking evaluation of the HA's scalability. We developed a micro-benchmarking tool that can generate different loads on the HA (in terms of requests per second), which measures the HA's response time under different load conditions.

The Crawler's performance is evaluated in the following scenarios:

Crawler scalability: In this scenario, we measure the resources consumed by a single Crawler process to crawl a HA network of a given size in order to gain insight into how well the design and implementation scales.

Effect of polling interval: In this scenario, we measure the effect of increasing the polling interval by measuring the maximum size of the Home Agent network a Crawler can crawl without falling behind.

C. Results

1) Home Agent Network Performance: Figure 8(a) compares the performance of the HA network with that of DNS. We collected DNS traces at our university for two months and computed the latency values for each query leaving our network to the Internet. As shown in figure 8(a), we plotted the CDFs for lookup latency in case of DNS, HA network with caching, and without caching. As can be seen from the figure, the DNS performs better within the 80th-percentile than the HA network without caching, but after that the DNS's performance degrades and the HA network performs significantly better, even without caching. With caching, the HA network always performs better than DNS, except for the very low latency lookups within 10ms. This phenomena is due to the presence of cached DNS records in the client cache as well as the the presence of closeby DNS servers (and caches) provided by the ISP serving our university.

2) Effect of Geo-Awareness: Figure 8(b) shows the impact of geo-awareness on the end-to-end name resolution latency of our system. It is evident from the figure that there is about 50ms reduction (about 28%) in the median name lookup latency with the geo-awareness in place, although the lines converge near the tail of the distribution. The high latency values are due to queue build up in the HA process and the geo-awareness does not have much impact on the name lookup latency during this queue build up. The effects of queue build up are discussed more in detail in the following section.

3) Home Agent Scalability: Figure 8(c) shows the HA's response time under different load conditions. We varied the load on the HA from 100 requests per second up to 1000 requests per second in 100 requests per second intervals. The plot reports the average response time of the HA along with the 5th and 95th percentile values. As the results indicate, the HA can deliver a $\leq 100ms$ response time for up to 400 requests per second. Response times increase rapidly beyond this threshold and can be as slow as $600ms$ on average for the largest load in our experiment. We investigated the cause for this performance decrease, and discovered that beyond the 400 requests per second threshold, the internal queues of the HA process build up very quickly and drain very slowly. One solution to this problem is to increase the number of threads handling the message processing and forwarding. However, in our experiment the HA process had a number of threads equal to the number of cores in the machine, and the performance improvement is quite negligible as the number of threads exceeds the number of processor cores on the machine.

4) Crawler Scalability: Figure 9(a) illustrates the resources required to run the Crawler in standalone mode with two threads on the simulated HA network with 5% to 25% of users assumed to be using pWeb, 5,684 to 28,172 HAs respectively, for a fixed polling frequency of 10 seconds. When 5% of mobile phone subscribers are assumed to be using pWeb, a single Crawler process can poll all 5,684 HAs every 10 seconds, and requires approximately 45% of 1 CPU core and 1,529 MiB of memory. Alternatively, when 25% of users are assumed to be using pWeb, a single Crawler process can poll all 28,172 HAs every 10 seconds and requires approximately 148% of 1 CPU core and 6,320 MiB of memory. As can be seen from the figure, both the CPU time and memory required scale linearly with the HA network size.

5) Effect of Polling Interval: Figure 9(b) shows the maximum size of the HA network a single Crawler process can crawl before it starts to fall behind for polling intervals of 2 to 14 seconds. For this experiment, falling behind is defined as the first instance that at least two HAs were not polled within the polling interval. The figure shows the maximum network size that a single poller can support increases linearly with an increasing polling interval and increasing number of threads. On average, increasing the polling interval by 2 seconds increases the maximum HA network size that the Crawler can support by 30%. Increasing the number of threads from 1 to 2 increases the network size that the Crawler can support by 90%, from 2 to 3 by 50%, and from 3 to 4 by 20%. This experiment shows that a single Crawler process can monitor up to approximately 70,000 HAs when no device or content updates are being processed by the system. When device updates are present, this value is halved due to the second HTTP request that the Crawler must make in order to submit the device updates to the search database. When device and content updates are present in the system, this value will be substantially less. We plan to evaluate the scalability of the Crawler with content updates in the near future.

VII. Conclusion

In this paper, we addressed a basic question "How to seamlessly access end devices using the Web technology?. This question has become more significant with users having multiple and more powerful devices than before. Our investigation revealed that a standalone app or software is not sufficient to address this question. A collaborative open platform is essential for naming and indexing the end devices for seamless integration with the Web. We have developed pWeb to provide this functionality. So far we have deployed the pWeb platform in PlanetLab and in the University of Waterloo compute cluster. We have also developed client software for Windows and Android platforms. Experimental results from our current deployment have proved the effectiveness of the proposed architecture. Our current focus is to open pWeb for public use. Specifically, we are improving the HA software for better security and broader hardware support. We are generalizing the pWeb client software for supporting wider range of third party apps and more operating systems. Any organization can become a part of the pWeb naming system by deploying our HA software. App developers will be able to deliver more innovative apps by building on the pWeb client software.

References

[1] C. man Au Yeung, I. Liccardi, K. Lu, O. Seneviratne, and T. Berners-lee, "Decentralization: The future of online social networking," in *W3C Workshop on the Future of Social Networking Position Papers*, 2009.

[2] S. Buchegger, D. Schiöberg, L.-H. Vu, and A. Datta, "Peerson: P2P social networking: early experiences and insights," in *SNS*, 2009.

[3] L. Cutillo, R. Molva, and T. Strufe, "Safebook: A privacy-preserving online social network leveraging on real-life trust," *Communications Magazine, IEEE*, vol. 47, no. 12, 2009.

[4] R. Sharma and A. Datta, "Supernova: Super-peers based architecture for decentralized online social networks," in *COMSNETS*, 2012.

[5] A. Datta, S. Buchegger, L.-H. Vu, K. Rzadca, and T. Strufe, *Handbook of Social Network Technologies and Applications*. Springer, 2010, ch. Decentralized Online Social Networks.

[6] R. Cheng, W. Scott, A. Krishnamurthy, and T. E. Anderson, "FreeDOM: a new baseline for the web," in *HotNets*, 2012.

[7] A. Bergkvist, D. C. Burnett, C. Jennings, and A. Narayanan, *WebRTC 1.0: Real-time Communication Between Browsers*. Working Draft, W3C, September 2013.

[8] I. Hickson, *Web SQL Database*. Working Group Note, W3C, Nov. '13.

[9] N. Mehta, J. Sicking, E. Graff, A. Popescu, J. Orlow, and J. Bell, *Indexed Database API*. Candidate Recommendation, W3C, July 2013.

[10] S. Kosta, V. C. Perta, J. Stefa, P. Hui, and A. Mei, "Clone2Clone (C2C): Peer-to-peer networking of smartphones on the cloud," in *HotCloud*, 2013.

[11] M. F. Bari, S. R. Chowdhury, R. Ahmed, R. Boutaba, and B. Mathieu, "A survey of naming and routing in information-centric networks," *IEEE Communications Magazine*, vol. 50, no. 12, 2012.

[12] T. Vu, A. Baid, Y. Zhang, T. D. Nguyen, J. Fukuyama, R. P. Martin, and D. Raychaudhuri, "Dmap: A shared hosting scheme for dynamic identifier to locator mappings in the global internet," in *ICDCS*, 2012.

[13] R. Ahmed and R. Boutaba, "Plexus: A scalable peer-to-peer protocol enabling efficient subset search," in *IEEE/ACM TON*, vol. 17, 2009.

[14] "Mongoose." [Online]. Available: https://code.google.com/p/mongoose

[15] "SQLite." [Online]. Available: http://www.sqlite.org

[16] C. Kohlhoff, "Boost.asio - 1.53," http://www.boost.org/doc/libs/1_53_0/doc/html/boost_asio.html, 2012.

[17] A. C. Bavier, M. Bowman, B. N. Chun, D. E. Culler, S. Karlin, S. Muir, L. L. Peterson, T. Roscoe, T. Spalink, and M. Wawrzoniak, "Operating systems support for planetary-scale network services." in *NSDI*, 2004.