PEACHES and Peers

Massimiliano Pala and Sean W. Smith

Computer Science Department, Dartmouth College
6211 Sudikoff Laboratory, Hanover, NH 03755, US
{pala,sws}@cs.dartmouth.edu

Abstract. How to distribute resource locators is a fundamental problem in PKI. Our *PKI Resource Query Protocol (PRQP)*, recently presented at IETF, provides a standard method to *query* for PKI resources locators. However the *distribution* of locators across PKIs is still an unsolved problem. In this paper, we propose an extension to PRQP in order to distribute PRQP messages over a Peer-to-Peer (P2P) network. In this work, we combine PRQP with *Distributed Hash Tables (DHTs)* to efficiently distribute contents over a dynamic P2P overlay network. In particular we present the *PEACH* protocol and a *PEACH Enabled System (PEACHES)* which are specifically targeted toward solving the PKI resources discovery problem. Our work enhances interoperability between existing PKIs and allows for easy configuration of applications, thus augmenting usability of PKI technology.

1 Introduction and Motivation

Public key cryptography has become, in many environments, a fundamental building block for authentication. Many applications already support the usage of *Public Key Certificates (PKCs)*—e.g. browsers, mail user agents, web-based applications, etc. PKCs can be requested and obtained from different *Certification Authorities (CAs)* which may live within a large infrastructure or may be deployed as isolated entities. However, locating services and data repositories related to a CA is still an open problem. The lack of a standard way to distribute resource locators to applications heavily (and negatively) impacts interoperability between PKIs and usability of applications.

In order to manage and extend trust among CAs and PKIs, different trust models have been studied and deployed.

Regardless of what the adopted trust model is, achieving interoperability between deployed infrastructure is very difficult and the management of the authentication infrastructure can be frustrating. One of the main reasons for these interoperability issues is the difficulty in discovering resources related to a Certification Authority.

An example of an interoperability nightmare is certificate validation. In order to grant access to its service, an application needs to verify that a certificate is still valid by, at the very least, retrieving the revocation information provided by the issuing CA.

This information is usually provided by the issuing CA by means of a *Certificate Revocation List (CRL)* or by an *OCSP* server. If no previous configuration exists at the application level, finding where the revocation information is available can be very difficult. As discussed in our earlier work [1], current solutions are not capable of solving the problem even in controlled environments like Virtual Organizations and Computational Grids.

Motivated by how this lack of resource pointers impacts many usability aspects in PKIs, this paper extends the PKI Resource Query Protocol (PRQP) in a peer-to-peer network in order to provide an efficient Discovery service for PKI resources—thus enhancing practical interoperability and usability of currently deployed PKIs.

The core contribution of this work is PEACH, a scalable system for PKI resources lookup.

2 Background

In order to provide a distributed and interoperable method to provide applications with pointers to PKI resources, we combine two existing technologies: the PKI Resource Query Protocol (PRQP) and a distributed hash table routing protocol, based on a modified version of Chord [2]. In this section, we provide the reader with a description of the background knowledge needed to understand our work.

2.1 The PKI Resource Query Protocol

PRQP is a simple query-response protocol designed to provide applications with the ability to query an entity for locators of specific services associated with a CA. In PKIs, the CA can grant other entities the authority to provide specific information to clients. An example of this is OCSP, where the CA delegates to the OCSP server the authority to provide information about the revocation status of its certificates. Similarly, in PRQP the CA identifies an entity to be authoritative for PRQP answers. The identified entity is called the *Resource Query Authority (RQA)*. By querying the RQA, an application can discover the location (URL) of a service or of a repository associated with a specific CA.

PRQP identifies two different trust model for the RQA. In the first model, the RQA is directly designated by the CA by having the CA issue a certificate to the RQA with a specific value set in the `extendedKeyUsage` extension.

In the second model, the RQA acts as a *PRQP Trusted Authority (PTA)* for a set of users—e.g., users in an enterprise environment. When operating as a PTA, the RQA may provide responses about multiple CAs, without the need to have been directly certified by them. To operate as such, PRQP requires that a specific extension, i.e. `prqpTrustedAuthority`, should be present in RQA's certificate and its value should be set to `TRUE`.

A full description of the protocol and its details is provided in our RFC [3].

2.2 Distributed Systems: the Peer-2-Peer Revolution

One open problem in PRQP is how to find information about CAs without prior knowledge of the associated RQA. To solve this problem, we now provide a distributed discovery system for PKI resources.

However, designing, implementing and debugging a large distributed system is a notoriously difficult task. Consequently much of the current effort has been spent on easing the construction of such systems.

The simplest organizational model for distributed systems is the *Client/Server* model. This model is well-known, powerful and reliable. In this configuration, the server is a data source while the client is a data consumer. Simple examples that make use of this model are Web Services and FTP. The Client/Server model, however, presents some limitations. For one thing, scalability is hard to achieve. For another, the model presents a single point of failure. The model also leaves unused resources at the network edges.

Peer-to-Peer (P2P) systems try to address these limitations by leveraging collaborations among all the participating parties (peers). Within this paradigm, all nodes are both clients and server: any node can provide and consume data. Several characteristics make P2P systems very interesting for data or services distribution over the Internet. They can implement an efficient use of resources by equally distributing usage of bandwidth, storage and processing power at every node. Most importantly, P2P systems are scalable. Napster, Gnutella and Kazaa are popular examples of first-generation P2P applications known to be efficient in data lookup and distribution. These systems use different approaches to provide a lookup service about the data provided by the participating peers. Napster implements a centralized search service where a single server keeps track of the location of the shared contents. On the opposite side is Gnutella; in this type of network, search is implemented by recursively asking the neighbors for files of interest. The search goes on till a *Time To Live (TTL)* limit is reached. Other systems like Kazaa or Skype use a hybrid model where super-peers act as local search hubs. Super-nodes are automatically chosen by the system based on their capacities in terms of storage, bandwith and availability.

2.3 Distributed Hash Tables

The *second generation of P2P overlay networks* provide more advanced features: they are self-organizing, load-balanced and fault tolerant. A major difference over an unstructured P2P system is that these second-generation systems also guarantee that the number of hops needed to answer a query is predictable. These systems are based on *Distributed Hash Tables (DHTs)*. DHTs are a distributed version of a hash table data structure, which stores (*key*, *value*) pairs, where the *key* is the identifier of the resource while the *value* can be the file contents. DHTs provide an efficient way to look up data and objects in a P2P network. Each node in the network is assigned a *node-id* and is responsible for only a subset of (*key*, *value*) pairs. By using the DHT interface, these systems are capable of finding a node responsible for a given *key* and efficiently routing the specific request (e.g., *insert*, *lookup* or *delete*) to this node. There are several DHT routing protocols, often referred to as *P2P routing substrates* or *P2P overlay networks.* Chord [4, 5] implements a concept of a circular address space and provides simple operations—$insert(key, value)$, $lookup(key)$, $update(key, value)$, $join(n)$, and $leave()$—to maintain and update the network routing tables. Pastry [6] implements a similar interface to Chord; however it considers network locality in order to minimize hops that the messages travel. CAN [7] is based on a "d-dimensional" Cartesian coordinate space on a d-torus. In CAN, each node owns a distinct one in the space and each key hashes to a point in the space. Other examples of DHTs include Tapestry [8], Kademlia [9], and P-Grid [10].

3 The PEACH Protocol

This section describes our *PKI Easy Auto-discovery Collaborative Hash-table (PEACH)* protocol. Our protocol specifies how to find the location of a specific RQA, how RQAs join the system, and how to update the system in case a node leaves.

The PEACH protocol is derived from the Chord protocol. Although there are many similarities with Chord, we designed PEACH in order to leverage unique features of PRQP to provide support for a P2P overlay network specifically for RQAs.

In particular, PEACH differs from the Chord protocol in two main aspects. First, we implement a completely new method to assign node identifiers to peers. As a consequence, the lookup protocol directly provides the requesting entity with the address of the authoritative RQA for a requested CA. Moreover, because of this new idea, PEACH is simpler than Chord in that it does not require any $insert()$ or $get()$ operations to locate the needed data—that is, to recover the network addresses of RQAs.

The second change is related to the way nodes join the PEACH network. In fact, the new $join()$ operation allows only authenticated RQAs to join the network.

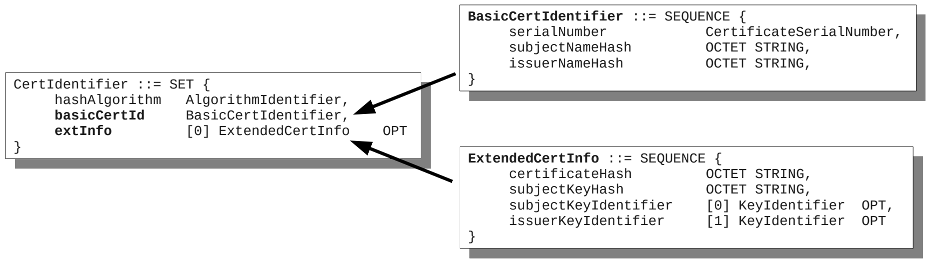
3.1 Overview

Similarly to Chord, PEACH uses a standard hash function to assign node identifiers. Differently from Chord, we use a PKI-specific method instead of network addresses to assign node identifiers.

In PEACH, the participating peers are RQAs. We assume that each RQA has a (cryptographic) key pair that has already being certified by its CA. Specific certificate contents—that is, the `extendedKeyUsage` field—allow the RQA to provide responses about resources related to that particular CA.

The basic idea is to leverage the direct link between a CA and its RA by building the node identifiers by using the CA's certificate fingerprint[1]. Since the

[1] The *fingerprint* of a certificate is calculated by computing a cryptographic hash over the DER- encoded certificate.



```
CertIdentifier ::= SET {
     hashAlgorithm   AlgorithmIdentifier,
     basicCertId     BasicCertIdentifier,
     extInfo         [0] ExtendedCertInfo    OPT
}

BasicCertIdentifier ::= SEQUENCE {
     serialNumber            CertificateSerialNumber,
     subjectNameHash         OCTET STRING,
     issuerNameHash          OCTET STRING,
}

ExtendedCertInfo ::= SEQUENCE {
     certificateHash         OCTET STRING,
     subjectKeyHash          OCTET STRING,
     subjectKeyIdentifier    [0] KeyIdentifier  OPT,
     issuerKeyIdentifier     [1] KeyIdentifier  OPT
}
```

Fig. 1. PRQP CertificateIdentifier data structure.

CA's fingerprint is often used as part of a PRQP request, it is a perfect candidate for a node identifier. Moreover, this allows a node to provide authentication information that may be used by the *successor* node to verify that the joining RQA is authoritative for a specific CA.

This method of building node identifiers frees us from the requirement of having to store any value on the participating peers. Therefore, when a peer joins or leaves the network no data need be moved (or copied) among nodes and there is no need to implement operations for data storing/retireving to/from the network (e.g., *insert*() and *get*()). This increases the network reliability and lowers the number and load of operations needed in order to manage *join*() and *leave*() operations.

In order to guarantee that the lookup of nodes takes place within $O(\log N)$ steps, a list of m pointers is maintained at each node. This list of pointers is the equivalent of the fingers table in Chord and has the same purpose.

3.2 Certificate-Based Node Identifiers

In PEACH, we use a cryptographic hash function to generate node identifiers. In particular, to maintain compatibility with current PRQP specification, we use the same hash function that is implemented in PRQP. The current PRQP Internet draft uses SHA-1 [11] to provide CAs certificates identifiers. PEACH does not depend on the hash function itself, therefore it can be easily updated to future functions, like SHA-256, to build identifiers.

Although it is possible to use different hash algorithms to build identifiers, is must be noted that the chosen algorithm has to be shared across the whole PEACH network.

Because PEACH is specifically designed to be used in conjunction with PRQP, we leverage some data structures already present in PRQP to build PEACH node identifiers. In particular, PRQP allows a client to request pointers related to a CA by providing the `CertIdentifier` within a request. This data structure allows for several ways to identify the certificate of the CA of interest. Figure 1 provides the ASN.1 description of the CertIdentifier data structure. The `BasicCertIdentifier` carries the information needed to identify a certificate: that is, the serial number, the hash of the subject name and the hash of the issuer name. The optional `ExtendedCertIdentifier` field bears more detailed information about the CA's certificate. In PRQP, this field is optional as the client could ask for pointers related to a CA without having the possibility to compute the fingerprint of the CA's certificate.

In PEACH, each peer that joins the network is assigned with a node identifier which corresponds to the hash of its CA's certificate. In other words, when an RQA joins the network, it uses the fingerprint of the certificate of the CA that issued the RQA's certificate.

In PRQP, an RQA can be issued certificates from more than one CA. This enables the RQA to be authoritative for each of those CAs. To accommodate for this possibility, the RQA is assigned multiple network identifiers as described in Section 3.5.

The hash function generates values of m bits (which for SHA-1 is 160, while for SHA-256 is 256). Therefore, the node identifiers can be seen as laid on an ordered circular structure modulo 2^m. Hence, the possible values for node identifiers range from 0 to $2^m - 1$.

3.3 The Lookup Operation

The lookup operation in PEACH is derived from the one implemented in Chord. To limit the number of hops needed to find if a node is present on the network, each node keeps a table of entries where data about nodes present on the network are stored. This table is called the *pointers* table. It is important to notice that PEACH does not need the *pointers* table in order to function properly. However the *pointers* table reduces the complexity of the lookup operation from $O(n)$ to $O(\log N)$.

The number of entries in the *pointers* table is equal to the number of bits (m) in the node identifiers. If the SHA-1 hash function is used, the *pointers* table has $m = 160$ entries. The contents of each entry in the table is provided in Figure 2. The table is ordered on the *Entry ID* field. The Entry ID values for node n range $(id_n + 2^0) \mod 2^m$ (for entry 0) to $(id_n + 2^{m-1}) \mod 2^m$ value (for entry i). As in Chord, the concept behind the *pointers* table is to divide the PEACH network space into progressively growing slices.

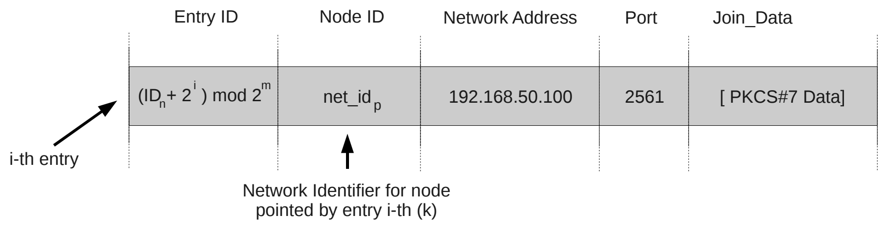


Fig. 2. i-th entry in the pointers table of node n.

For example, given the node n and the ith entry in its pointers table, be x_i^n such as:

$$x_i^n = (id_n + 2^i) \mod 2^m$$

then the node-identifier space related to this entry is:

$$\gamma_i^n = [x_i^n, x_{i+1}^n)$$

Figure 3 shows two consecutive slices on the PEACH network for node n. To perform a lookup of a node address, the application would execute a *find_node*(). At first, the node n will perform a *lookup_table* (i) operation. This would look at the local pointers table and return the closest node in the network that *is equal* or *precedes* the node we are searching for. Be node k the closes match. If a perfect match is not found in the local pointers table, then a *find_node_ex*(k, i) operation is performed which asks node k to perform a *find_node*(i). In other words, the query recursively traverses the PEACH network and returns: (a) the address of the matching node (if it exists on the network) or (b) the closest preceeding match that is present on the network. The pseudocodes for the *lookup_table*() and *find_node*() operations are reported in Figure 4.

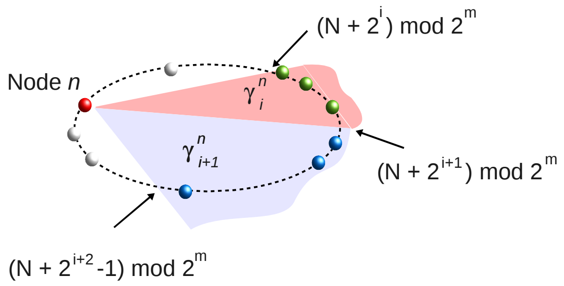


Fig. 3. γ_i^n and γ_{i+1}^n network identifier spaces. The size of the space doubles at each step.

The lookup function is different from the Chord protocol in that it looks for node identifiers, not for a key space assigned to a node.

The lookup function in a Chord-based network is generally used by *insert*() or *get*() operations to retrieve or store content to/from the network. Therefore, when searching for a particular key, a valid result (i.e., the node identifier that is responsible for the key space in which the searched key is in) is always returned on a lookup operation. In fact, the node whose identifier is the next on the Chord ring is the one responsible for storing all the values related to the specific key space.

In PEACH, instead, if no exact match is found on the network, the preceding closest node (RQA) is returned. When this happens, the lookup operation fails,

```
function lookup_table (id, $\gamma$):
        for $j \Leftarrow m-1$ downto 0
            if ( pointers[ j ].id $\in \gamma$ ) and
                        ( pointers[ j ].id $<$ id )
                // We found the closest match on the pointers table
                return pointers[ j ]
            end if
        done
        // No suitable pointer found in the table
        return ( null )


function find_node(id, $\gamma$):
        if ( id == self.id )
            return ( self )
        end if
        ret $\Leftarrow$ lookup_table ( id, $\gamma$ )
        if ( ret == null )
            // Current node is the closest on the network
            return ( self )
        end if
        // Propagate the search with reduced space
        $\gamma \Leftarrow \gamma$ - $\|self - ret\|$
        return ( find_node_ex ( ret, id, $\gamma$ ) )
```

Fig. 4. Pseudocode for finding a node with the *id* indentifier. The closest match on the local table is found by using the **lookup_table**() function. The peer will then ask closest matching node (*ret*) to perform a lookup by using the **find_node_ex** () function. All the search operations are constrained on a seach space γ.

meaning that the RQA that is authoritative for the particular CA is not present on the network.

3.4 The Join Operation

One of the most important operations in PEACH is *join*(). In order to be able to find an authoritative RQA, the network has to be made aware of its availability. Therefore when an RQA wants to provide services to the network via PEACH it has to *join*() the network.

In PEACH , each node maintains, at minimum, the information about its *successor* and its *predecessor* on the network. For the network to operate properly, this information must be up to date. This is achieved via the *join*() and *leave*() operations. When a node joins the network it performs the following operations:

a. connects to the one of the available nodes
b. finds its *successor* on the PEACH network

c. informs its *predecessor* and its *successor* of its presence on the network
d. (optionally) builds the list of pointers to other nodes

In order to perform step (a.), applications need to be instructed on how to reach one node that participates in the network. PEACH does not specify how to provide applications with a list of active nodes to contact at startup. However, it is possible to provide a pointer to active nodes by using different methods, e.g. DNS SRV records [12], DHCP extensions [13], or by scanning the local network for active nodes. Another very popular method is to simply embed a list of URLs that applications can use and update upon startup (e.g., root DNS server addresses are usually embedded into DNS server software like Bind [14]). Currently, in our test environment we use the latter approach. Future plans are to set up some stable nodes that RQAs can use as "entry points" to the PEACH network.

To find its successor on the network (step b.), the joining RQA performs a *find_node_ex*(). Be n the joining RQA, and k the "entry point".

The *find_node_ex*() function asks node k to perform a lookup. The *successor* of node n is found by simply using the *find_node_ex*() on node k with $id_{n+1} = (id_n + 1) \mod 2^m$. Be p the returned node. If the returned value is a perfect match, the successor of node n is found. If the returned value is not a perfect match, the returned value is the closest preceding node. In this case, to find its successor, the node n contacts *ret* and asks it for his successor. Ultimately, the node p will return the first entry in its pointers table to node n.

In order to find its *predecessor* on the network (step c.), the node n simply performs a lookup for node whose id is id_{n-1}. More details on format of the data packet that are exchanged over the PEACH network is discussed in Section 4.

Now, the joining RQA asks p to be considered as the new *predecessor*. To do so, the n node sends to p a PKCS#7 [15] signed object. This object carries the details about the n's network address as the data payload. The object type used for the exchanged message is `SignedData`. The RQA's certificate and the certificate of the issuing CA (the one for whom the RQA is authoritative) are embedded in the `certificates` field.

The receiving node, before updating the details about its *predecessor* but after joining the PEACH network, verifies that the RQA's certificate is valid and that it has been issued by the CA for which the RQA will be registered to be authoritative, In order to do so, it verifies that the signature on the PKCS#7 object is valid and that is has been signed with the private key that corresponds to the public key present in the RQA's certificate. Secondly it verifies that the RQA's certificate is issued by the presented CA's certificate (by verifying its signature against the public key of the CA's certificate). As a last step, the receiving node calculates the CA's certificate fingerprint and assigns it as the identifier for node n as its predecessor. The new id, together with the network address data, is stored on the receiving node. Differently from Chord where only the successor of a node is informed when a new node joins the network, in PEACH the joining peer pushes its information to its predecessor as well. Also differently from Chord, as a result of our *join*() operation, participating nodes do not need to execute any *update*() function to discover new nodes.

Once an RQA joins the PEACH network, it is required to maintain open one socket to its *predecessor* and one socket to its *successor*. This enables nodes to immediately be aware of a node leaving the network without the need to run any *update*() operation.

3.5 Multiple Joins

PRQP allows a Resource Query Authority to be authoritative for multiple CAs. To be able to provide PRQP responses for multiple CA, an RQA needs to be registered with multiple network identifiers.

To be assigned multiple node identifiers, the joining RQA performs multiple *join*() on the network. Let n be the number of certificate the RQA possesses. The set of its certificates (ϕ) can be expressed as:

$$\phi = \{x_1, x_2, \dots, x_n\}$$

Let θ be the set of network identifiers related to the joining RQA:

$$\theta = \{y_1, y_2, \dots, y_n\}$$

where:

$$\forall i \in [1, 2, \ \dots \ , n], \quad \exists x_i, y_i \ : \ x_i \in \phi \quad \vee \quad y_i \in \theta \Rightarrow y_i = H(x_i)$$

For each x_i the peer is authoritative for, the RQA has a different network identifier y_i which is based on the CA's certificate fingerprint. For each of these identifiers, the joining peers performs *find_node*() to find successor in the network ring and proceeds to register itself in the right position.

3.6 Other Operations

In this section we provide some considerations on the efficiency of PEACH. We especially focus our attention on the network maintainance operations.

Leaving the Network. PEACH does not require participating peers to execute any *leave*() operation. Because each peer is required to maintain open two TCP connections toward its *successor* and its *predecessor*, when the node leaves the network, the required TCP connections are dropped. Hence both its *predecessor* and its *successor* would be aware of the leaving operation right away.

Creating the list of pointers After an RQA has successfully joined the network, it needs to populate its *pointers* table. To do so, the joining peer queries the network for the entries in the *pointers* table. The algorithm is reported in

```
function **update_table ():**
        for ( $i = 1$; $i < m$; $i = i + 1$ )
            $ret$ = **find_node** ( $pointers[i].id$ )
            **if** $ret.id \in [pointers[i].id, pointers[i+1].id)$
                $pointers[i] \Leftarrow ret$
            **end if**
        **done**
```

Fig. 5. Pseudocode for building the table of pointers.

Figure 5. Ultimately, the list of pointers is built by performing lookups for each entry id, that is:

$$entry_{id} = (id_n + 2^i) \mod 2^m$$

where the table size is m, and the peer network identifier is id_n. The returned value is stored in the entry only if it is in the space identified by the ith entry (γ_i^n). For instance, let the space of identifiers be 3 bits, let x be the joining node and let 1 be its *id*. The nodes present in the network have identifiers 0, 2, and 3. The last entry in the x *pointers* table is:

$$\begin{aligned} x_{id} &= (1 + 2^{3-1}) \mod 2^3 \\ &= 5 \end{aligned}$$

Now, let k be the closest previous match on the network, and let 3 be its network identifier. The possible results of the performed search are:

$$res = \begin{cases} 5 & \text{if } x_{id} \ \exists \ \text{ on the network,} \\ 3 & \text{if } x_{id} \ \nexists \ \text{ on the network,} \end{cases}$$

Because no node with 5 as its identifier is present on the network, node k is returned instead. As the result should be stored only if it falls into the ith key space, the returned value is discarded.

Maintaining the list of pointers. When performing a *find_node*() operation, it may happen that an entry in the *pointers* table points to a node that is no longer available on the network. In this case, if the connection to the entry fails, the entry is simply removed from the table and the lookup operation is resumed. Occasionally a node may update its list of pointers in order to leverage the presence of new nodes in the network.

4 PEACHES Details

In this section we provide considerations about the implementation of the algorithm and an evaluation based on a PEACH simulator.

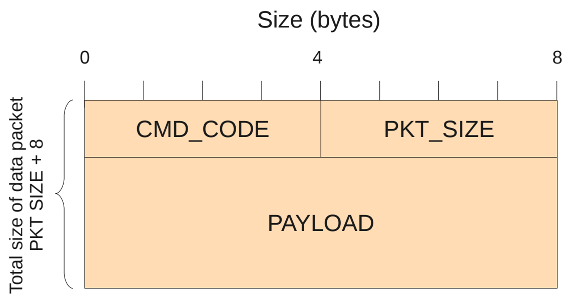


Fig. 6. Basic structure of data packets exchanged on the PEACH network. The payload depends on the specific command.

4.1 Network Communication

While building a PEACH enabled system, we also defined the format of exchanged messages over the network. To minimize the impact of the message format, we opted for a simple binary format.

The PEACH data packets are depicted in Figure 6. The format is consistent across the different commands that nodes exchange. In particular, each data packet has the following fields:

– command code
– packet length
– payload

The *packet length* is used to identify the lenght of the payload and it is 4 bytes long (type uint32_t). The *command code* is 4 bytes long as well and it specifies the action to be performed on the target node or the return code. The identified opt codes and their description are reported in Table 4.1.

Command Name	Code	Description
CMD_ERROR	0x200 + 0	General Error
CMD_SUCCESS	0x200 + 1	Cmd Successful
CMD_GET_NODE_INFO	0x500 + 0	Get node information
CMD_GET_NODE_SUCCESSOR	0x500 + 1	Get node successor
CMD_GET_NODE_PREDECESSOR	0x500 + 2	Get node predecessor
CMD_UPDATE_PREDECESSOR	0x600 + 1	Update predecessor info
CMD_UPDATE_SUCCESSOR	0x600 + 2	Update successor info
CMD_LOOKUP_NODE	0x600 + 2	Perform a lookup

Table 1. PEACH opt codes values and description.

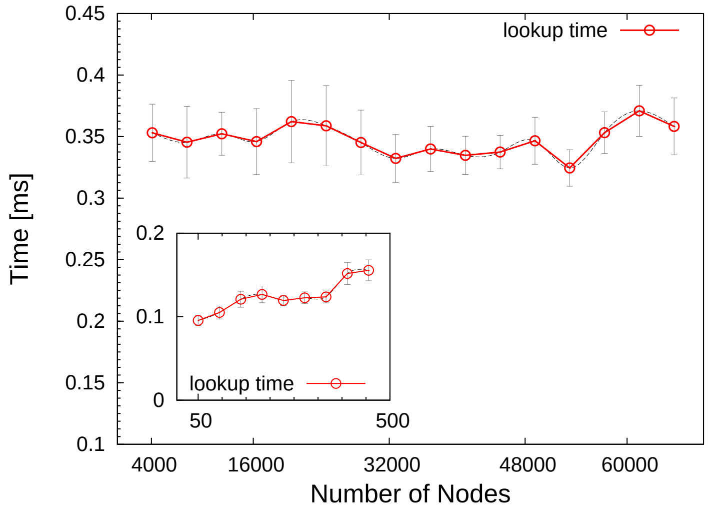


Fig. 7. Lookup times in the PEACH network simulator.

4.2 The PEACH Simulator

To analyze the feasibility of building a PEACH system, we developed a simulator that implements the protocol's operations. The current version of the simulator is written in PERL. To implement the cryptographic functionalities, we use a command-line based tool that is built on top of LibPKI [16]. This tool loads keys and certificates and provides the simulator with an easy-to-use PKCS#7 object generator.

Because the focus of the simulation is to test the feasibility of the PEACH deployment, we concentrated our attention on the routing of the data in PEACH. Because our work targets PKIs, we reasonably think that the number of participating RQAs for a PEACH based system could reasonably set between few hundreds to few thousands of peers. We have been able to successfully simulate systems with more than 65000 nodes on a 2.4Ghz Core-Duo laptop equipped with 2GB of memory. Moreover we measured the performance of the *lookup* operation in PEACH. The results are reported in Figure 7. We simulated PEACH networks with an increasing number of nodes, starting from 50 to 65535. As expected, because PEACH makes use of DHTs, after an initial trend (reported in the small box in Figure 7), the *lookup* time grows very slowly with the the number of nodes present in the network. It is to be noted that our simulator does not take in consideration network-related delays, therefore in a real deployment lookup times may be sensibly larger because of communication constrains between nodes.

During our simulations, we also noticed that the overhead introduced by the need to provide signed messages when joining the network was relatively high. The main reason for this is that we use an external command-line based tool[2] to generate the signatures. In PEACH, a peer is require to generate two different signatures in order to perform a *join*(). Therefore it is not surprising that, to simulate a *join*() operation, it can take up to a second, although the signing time per each signature should take only a few milliseconds. A fully C-based PEACH implementation would not suffer from this problem. We envision that the total time for a *join*() operation to occur (without considering time spent on network operations) can be less than 100ms without requiring special cryptographic hardware.

5 Integrating PEACH and PRQP

In PRQP each client could use one of the configured RQAs to query for resources related to a CA. When the contacted RQA has no information about the requested service, the client has no alternative way to discover where to forward the request to. By integrating PEACH into an RQA server, the RQA would be enabled to forward the requests into the P2P network and retrieve the missing information.

The RQA can leverage the PEACH in two different ways. The first one is to have the RQA act as a PRQP client. When a request cannot be answered, such as the case where there is a lack of information about the queried CA, the RQA searches for the authoritative RQA on PEACH and, if found, issues a request to the RQA. The returned response is then parsed, and the signature (and the server data) is substituted before sending the response back to the client. The problem with this solution is that the original signature on the response is lost, requiring the client to configure the RQA as a trusted authority (i.e., the RQA is allowed to provide responses for different CAs without having to be certified by them).

The second option is to have the RQA act as a proxy for the client. In this case, the RQA forwards the requests that it cannot answer to the authoritative RQA, but only when it is present on the PEACH network. The response is then forwarded back to the client.

When PRQP and PEACH are integrated, the P2P network maps network addresses to PKI services similar to the way DNS maps logical names to IP addresses. The main difference between the DNS and the RQA network is the absence of a hierarchical system approach.

6 Conclusions and Future Work

In our work, we describe the PEACH algorithm and its application to PKIs. In particular, the presented approach introduces—for the first time—the idea of providing interoperable and collaborative peer-to-peer-based services in X509 PKIs.

[2] openca-sv, available as part of the openca-tools package from `https://www.openca.org`.

We extend the PRQP protocol, which provides a PKI-specific protocol for resource discovery, by providing the starting point for the development of a PKI Resource Discovery Architecture. Under this novel system, different RQAs cooperate to access data that is not locally available.

In a more general sense, PEACH allows for the building of an authenticated peer-to-peer network for client-server-based PKI services. Furthermore, our work can be easily extended to provide other collaborative services. For example, the proposed PEACH protocol would allow for OCSP or TimeStamping services to be integrated among different CAs and PKIs.

In the future, we plan to investigate the applicability of PEACH to the real world. In particular, we plan to release an open-source version of the PEACH enabled server based on the OpenCA-PRQP daemon and to establish collaboration with other authorities to setup a public PEACH network. Overall, this new approach tries to enhance interoperability across PKIs and will be actively promoted within the IETF PKIX working group as a standardized protocol.

Ultimately, we believe that this work will have a signficant impact over the interoperability and usability of PKIs and that it will open up new X509 PKI models based on collaborative services.

Acknowledgments

This work was supported in part by the NSF (under grant CNS-0448499) and the U.S. Department of Homeland Security (under Grant Award Number 2006-CS-001-000001). The views and conclusions contained in this document are those of the authors and should not be interpreted as necessarily representing the official policies, either expressed or implied, of any of the sponsors.

References

1. M. Pala and S. W. Smith, "AutoPKI: A PKI Resources Discovery System," in *EuroPKI*, ser. Lecture Notes in Computer Science, J. Lopez, P. Samarati, and J. L. Ferrer, Eds., vol. 4582. Springer, 2007, pp. 154–169. [Online]. Available: http://dblp.uni-trier.de/db/conf/europki/europki2007.html#PalaS07
2. I. Stoica, R. Morris, D. R. Karger, M. F. Kaashoek, and H. Balakrishnan, "Chord: A Scalable Peer-to-Peer Lookup Service for Internet Applications," in *SIGCOMM*, 2001, pp. 149–160.
3. M. Pala, "The PKI Resource Query Protocol (PRQP)," Internet Draft, June 2007. [Online]. Available: http://www.ietf.org/internet-drafts/draft-pala-prqp-01.txt
4. I. Stoica, R. Morris, D. Karger, F. F. Kaashoek, and H. Balakrishnan, "Chord: A Scalable Peer-to-Peer Lookup Service for Internet Applications," *SIGCOMM Comput. Commun. Rev.*, vol. 31, no. 4, pp. 149–160, October 2001. [Online]. Available: http://portal.acm.org/citation.cfm?id=964723.383071
5. "Chord." [Online]. Available: http://www.pdos.lcs.mit.edu/chord/
6. "Pastry." [Online]. Available: http://freepastry.rice.edu
7. S. Ratnasamy, P. Francis, M. Handley, R. Karp, and S. Schenker, "A Scalable Content-Addressable Network," in *SIGCOMM '01: Proceedings of the 2001 conference on Applications, technologies, architectures, and protocols for computer*

communications, vol. 31, no. 4. ACM Press, October 2001, pp. 161–172. [Online]. Available: http://portal.acm.org/citation.cfm?id=383072

8. B. Y. Zhao, J. D. Kubiatowicz, and A. D. Joseph, "Tapestry: An Infrastructure for Fault-Tolerant Wide-Area Location and Routing," UC Berkeley, Tech. Rep. UCB/CSD-01-1141, # apr # 2001. [Online]. Available: http://citeseer.ist.psu.edu/zhao01tapestry.html
9. P. Maymounkov and D. Mazieres, "Kademlia: A Peer-to-Peer Information System Based on the XOR Metric," 2002. [Online]. Available: http://citeseer.ist.psu.edu/maymounkov02kademlia.html
10. K. Aberer, P. Cudr-Mauroux, A. Datta, Z. Despotovic, M. Hauswirth, M. Punceva, and R. Schmidt, "P-Grid: A Self-organizing Structured P2P System," *SIGMOD Record*, vol. 32, no. 3, September 2003, http://lsirpeople.epfl.ch/rschmidt/papers/Aberer03P-GridSelfOrganizing.pdf.
11. NIST, "FIPS PUB 180-2 — Secure Hash Standard," Processing Standards Publication 180-2, August 2002. [Online]. Available: http://csrc.nist.gov/publications/fips/fips180-2/fips180-2.pdf
12. J. Postel, "Domain Name System Structure and Delegation," RFC 1591, March 1994. [Online]. Available: http://www.ietf.org/rfc/rfc1591.txt
13. R. Droms, "Dynamic Host Configuration Protocol," RFC 2131, March 1997. [Online]. Available: http://www.faqs.org/rfcs/rfc2131.html
14. "ISC Bind Server," Homepage. [Online]. Available: http://www.isc.org/index.pl?/sw/bind/index.php
15. B. Kaliski, "PKCS #7: Cryptographic Message Syntax," RFC 2315, March 1998. [Online]. Available: http://www.ietf.org/rfc/rfc2315.txt
16. M. Pala, "The LibPKI project," Project Homepage. [Online]. Available: https://www.openca.org/projects/libpki/