Hochschule für Angewandte Wissenschaften Hamburg
Hamburg University of Applied Sciences

Ausarbeitung Projekt 2 - SoSe 2011

Alexander Knauf

Implementation and Evaluation of the Extension for the MP2PSIP RELOAD Stack

Fakultät Technik und Informatik
Department Informatik

Faculty of Engineering and Computer Science
Department of Computer Science

Contents

1 Introduction

1.1 Motivation

Multimedia conferences are traditionally build on a centralized model in which a single entity maintains the signaling relations to all participants [1]. In a multimedia conference using the Session Initiation Protocol (SIP) [2] for signaling, the central entity is called *focus*. A conference focus must be uniquely identified by a SIP URI and SIP requests to this URI are exclusively routed to that focus [1]. A common deployment of this scenario meeting the latter requirement is a dedicated server infrastructure that provides a reliable conferencing service to its clients. A conference URI can be requested from a dedicated focus that may further invite user agent clients to a new ad-hoc conference [3]. The distribution of the conference media is often handled by the provider of the conference service using a dedicated multimedia server that is designed to guarantee an appropriate media quality. The maintenance of a dedicated conferencing infrastructure is expensive and typically provided by a third party hosting its participants. A conferencing solution for P2P scenarios that scales up to the size of conferences supported by a dedicated infrastructure is missing.

An alternative approach for multimedia conferencing is introduced by the RELOAD Usage for Distributed Conferencing (DisCO) [4]. A single multiparty conference is hosted by multiple independent entities called *focus peers* that use a common SIP URI as their identifier. The conference URI is registered at a REsource Location And Discovery (RELOAD) [5] overlay. RELOAD is designed as a distributed alternative to the SIP proxy/registrar infrastructure [2], in which SIP records can be stored in a trustworthy P2P overlay using a central authority only for peer authentication.

To implement a conferencing environment using the DisCo protocol scheme, it is a precondition to have an advanced RELOAD implementation. In previous works we [6] presented an analysis and the design to enhance the *MP2PSIP* RELOAD stack. This document is dedicated to the implementation of the concept and evaluation of the RELOAD stack by measuring the temporal behavior of the main overlay functionalities.

1.2 Objectives

The original implementation of the MP2PSIP RELOAD stack was designed as a demonstrator for a RELOAD overlay that serves as a mobile VoIP application. The result was a RELOAD implementation that was limited to the SIP Usage for RELOAD [7] to enable storage and retrieval of SIP records. The data storage protocol of RELOAD, however, specifies a meta-model providing all necessary headers and attributes to process any message body. The rest of the PDU is reserved for the application specific data called RELOAD Kinds. The initial implementation

of the data storage of the MP2PSIP RELOAD stack has tied the meta-model of RELOAD to the data model of the SIP registration Kind. Consequentially, the MP2PSIP implementation was not enabled to serve any other application data as a storage service, since the required meta-data structures were inseparable bound to the SIP registration. The concept for extending the MP2PSIP projects proposed the re-implementation of data storage compliant to the RELOAD specifications. Additionally, the architecture of the stack will be extended by a new interface to the upper applications allowing seamless implementation of new RELOAD Usages. A further re-implementation of the RELOAD access control model will enable us to enforce standard access control policies, as well as access policies specified by other Usages. All extensions are improving the compatibility with the RELOAD protocol and create an advanced basis for implementing the DisCo approach [4] and its associated Usage for Shared Resources (ShaRe) [8].

The reminder of this document is structured as follows. Section 2 constitutes the principle part of this document by starting with an overview 2.1 and the new proposed architecture in section 2.2. Section 2.3 presents a detailed description of the implementation of the new components and section 3 evaluates the achieved results. This document concludes and gives an outlook into future work in 4.

2 Implementation of MP2PSIP Extensions

2.1 Overview

The previous analysis [6] of the MP2PSIP project has shown that the stack implementation lacks components that are specified in the RELOAD base draft. This analysis further presented a concept for enhancing the MP2PSIP project and proposed the re-implementation of several components that are indispensable for a implementation of the RELOAD Usage for DisCo [4]. The following components were identified as insufficient or missing and require re-implementation meet the MP2PSIP RELOAD stack to meet the requirements of the RELOAD Usage for DisCo:

Message Transport: The previous implementation of the message transport component was limited to operations for storage and lookup of SIP registrations [7]. The meta-model of the RELOAD storage protocol is bound to the data structures of the SIP registration Kind.

Application Layer: The previous MP2PSIP RELOAD stack did not provide any kind of interface to implement new RELOAD Usages. The entire stack design is exclusively confined to the on a SIP usage.

Storage: The previous implementation of the MP2PSIP RELOAD stack just allows the data storage of SIP records. There exists no generic mechanism for other data types nor an implementation of RELOAD Resource/Kind model.

Access Control: The previous implementation of the MP2PSIP RELOAD stack did not implement the access control model of RELOAD. Peers were allowed to storage values at every overlay locations.

DHT Routing: The previous RELOAD stack does not implement a proper Chord routing table based on a *finger table*. Even though a finger table structure exists, its fingers are never initialized with addresses of the corresponding overlay peers and thus not used for routing.

Even though the MP2PSIP project is extended by several new functionalities as presented along this document, it is not a full RELOAD implementation. The following protocol specifications of RELOAD are out of scope of this implementation:

ICE Implementation: RELOAD specifies the usage of the Interactive Connectivity Establishment (ICE) protocol for NAT traversal. A full implementation of ICE would be well beyond scope.

TURN and Certificate Usage: RELOAD specifies the implementation of the two default Usages for TURN and certificate storage. Both are not required to implement the DisCO or ShaRe Usages and hence left out of scope.

Config Update Message: RELOAD specifies a *ConfigUpdate* message used to reconfigure the RELOAD overlay configuration document if a peer detects an out dated configuration of another peer. In the deployment scenarios for this RELOAD stack, there will be probably no case in which the overlay peers have different versions of the configuration document and hence an implementation of the *ConfigUpdate* request is not necessary.

2.2 Extended Architecture

A major challenge while extending the MP2PSIP project pose the integration of the new functionalities in the stack architecture without exchanging the entire business logic. This is achieved by the *Usage Manager* component added on top of the RELOAD stack and below the applications calling it. The resulting stack architecture is shown in figure 1. Applications using the RELOAD stack are on the top layer of the software architecture. Currently, there are application invoking the stack. The RELOAD MDI management console that enables the emulation of multiple RELOAD peers through graphical user interface and the RELOAD console client. The latter is an extension explicitly designed for an automated execution by a script language like bash-script or python. It can be used for measurement and evaluation processes

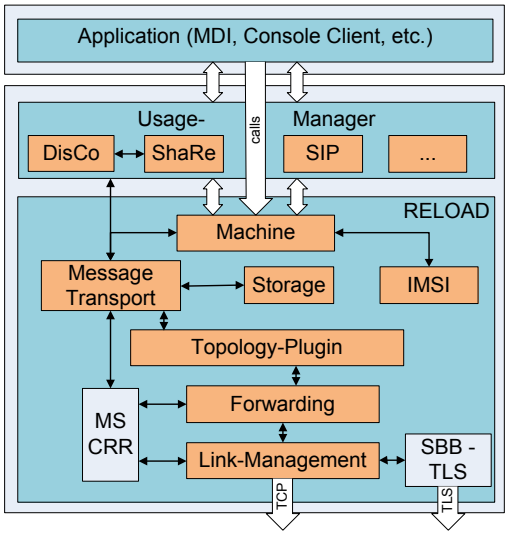


Figure 1: Extended architecture of the RELOAD stack

that need to be repeated for multiple iterations. The RELOAD stack can be initialized from the console with several arguments to instantiate RELOAD peers or clients with different execution behaviors.

Applications can inquire on the supported RELOAD Usages through the lower Usage manager component. For example, a graphical user interface could render several menu items to provide services belonging to a specific RELOAD Usage. To use the RELOAD messaging and storage service, each application must have a reference to the *Machine* component which is the background worker waiting for user commands. Commands are passed to the stack as a list of string arguments with a preceding Usage identifier. The identifier is not proposed in the RELOAD specification, but facilitates the interpretation of the user commands. On incoming commands, the machine then utilizes the Usage manager to instantiate a concrete Usage object with the given arguments. Another reference exists between the message transport and the Usage manager. On receiving a store request, the message transport passes the received byte stream to the Usage manager that will unmarshal the stream to a specific Usage object representing the application data.

Furthermore a new Storage module is added to the RELOAD stack. In the previous stack design, the storage component was integrated into the topology plugin and was just enabled to store SIPRegistration objects. The Storage module is now separated and is invoked by the message transport. It maintains StoredData objects which are container for Usage objects including their meta data.

2.3 Re-implementation of MP2PSIP Components

2.3.1 Reload.Machine

The *Reload.Machine* class is the central component to interconnect the applications with the RELOAD stack logic. The Machine class got adapted while implementing the MP2PSIP extensions to provide the new functionalities to the upper application.

In the previous stack design, user commands were passed to the RELOAD stack via a *void SendCommand(cmd: String)* method of the *Machine* class. The string argument contains the desired RELOAD operation (e.g., "Fetch", "Store") in plain text and a whitespace separated argument thus the resource name of a SIP registration being stored. The commands were added into a command queue which was continuously read out by the *Machine*. This design has two deficits, first, it just allows transport of single Kind data per Store or Fetch request and, secondly, just allows a single argument for the Kind to be created.

The enhanced *Machine* component separates the enqueue of user commands from their invocation by the RELOAD stack. To remain compatible to the previous stack design, the *SendCommand()* method changed to another functionality. Instead to enqueue a single user command, it pushes previously gathered user commands into the command queue. User commands are now gathered through a new *GatherCommand()* method whose header is shown in listing 1.

```
1 // Gathers user commands
2 public void GatherCommands(string cmd, UsageCodePoint ucp, int type, ↩
      params
3 object[] args)
```

Listing 1: GatherCommand() method header

Its command string identifies the desired RELOAD operation, a unique code point selects the desired Usage, an integer that indicates a specific type of a Usage. The last argument is a *var arg* object whose elements carry the Usage specific data. To send a RELOAD operation, a developer passes the RELOAD Kind data by calling the *GatherCommand()* method for each Kind needed by a Usage and calls the *SendCommand()* to enqueue these commands.

2.3.2 Message Transport

A major task while extending the MP2PSIP project was the re-implementation of the *Message Transport* component. The focus was to re-implemented the message struct definitions of RELOAD. Simple data definitions like *DataValue* or *ArrayEntry* are implemented as public

C# structs. Meta data structures like *StoredData*, *StoredDataSpecifier*, *StoreKindData* and *StoredDataValue* are implemented as classes with several public constructors and member properties. For example, the *StoredData* class has a property for each data field defined in the RELOAD base document (*cf.* [5]-15 p. 81) and two constructors. One used by the creator of a *StoredData* object and another used by the receiver of a byte stream containing a StoredData.

As proposed in the extension concept [6], the re-implementation of the *Store* and *Fetch* operations split each message into a request and an answer message. Furthermore, the message content does now follow the definitions in the RELOAD base protocol. This redesign caused a full re-implementation of the message transport component. However, to keep the new StoreReq/Ans and FetchReq/Ans classes complaint to the remaining stack implementation, both messages still implement the abstract *RELOAD_MessageBody* class that obliges to realize the *Dump()* and *FromBytes()* methods. As an example, the implementation of the *StoreReq.Dump()* method is given in listing 2. It shows how a store request is serialized to a byte stream in network byte order.

```
1  public override UInt32 Dump(BinaryWriter writer) {
2    UInt32 length = 0;
3    writer.Write(HostToNetworkOrder((short)Store_Request));
4    writer.Write(HostToNetworkOrder((int)length));
5    length += WriteOpaqueValue(writer, resourceId.Data, 0XFF);
6    writer.Write((Byte)replica_number);
7    length += 1;
8    foreach (StoreKindData kind in store_kind_data) {
9      writer.Write(HostToNetworkOrder((int)kind.kind));
10     length += 4;
11     writer.Write(HostToNetworkOrder((long)kind.generation_counter));
12     length += 8;
13     foreach (StoredData storedData in kind.values) {
14         writer.Write(HostToNetworkOrder((int)storedData.Length));
15         length += 4;
16         writer.Write(HostToNetworkOrder((long)storedData.StoreageTime));
17         writer.Write(HostToNetworkOrder((int)storedData.LifeTime));
18         storedData.Value.Dump(writer);
19         storedData.Value.GetUsageValue.dump(writer);
20         length += storedData.Length;
21       }
22   }
23   return length;
24 }
```

Listing 2: Example: Implementation of StoreReq.Dump()

The argument of Dump() is a *BinaryWriter* that contains the entire byte stream of the serialized RELOAD message. Lines 3, 5 and 6 show how the message type, the Resource Id and the replica number are set. The instruction in line 4 just reserves space to further set the length of the entire message body after its serialization. The serialization of all *StoreKindData* objects commences at line 8. Each *StoreKindData* is representing a single RELOAD Kind that will be sent within this request. A *StoreKindData* can contain several data values to be stored. Those are represented by *StoredData* objects whose serialization is shown beginning at line 13. Line 18 and 19 show the generic part of the Dump() implementation. A store request does not care about the data model nor the type of Usage been stored. The value contained in a *StoredData* has its own *storedData.Value.Dump()* method (line 18) that is aware how to serialize itself. Likely in line 19, the Usage object contained within a *StoredDataValue* is aware how to serialize the concrete Usage object to bytes. Finally, the Dump() method returns its own length to the instance that invoked it and the serialization of the store request message within the *BinaryWriter* object.

The transaction handling of RELOAD messages is realized within the *Reload.MessageTransport* class. To implement the abstract transport concepts, several members of this class were modified or re-implemented. Accordingly, the methods for store, inbound store, fetch, inbound fetch and the procedure for key handover got adapted. Figure 2 shows the transition from the previous to current method headers. The *Store*, *Fetch* and *OnFetchedData* are taking the new classes *StoreKindData*, *StoredDataSpecifier* and *FetchKindResponse* as arguments. The boolean *remove* flag of the previous *Store* method is replaced by an *exists* flag. This semantical change enables to store non-existent values whose Kind data is empty but explicitly signed by its owner. The *OnFetchData* method of the previous stack version had performed the application procedure of the SIP Usage to obtain the contact of the called party. The re-implemented *OnFetchData* is just a single instruction calling the *UsageManager* class and passes the *FetchKindResponse* argument to the manager. The latter is enabled to select the corresponding application procedure depending on the received Kind data. A more detailed description of the *UsageManager* component is presented in following section 2.3.3.

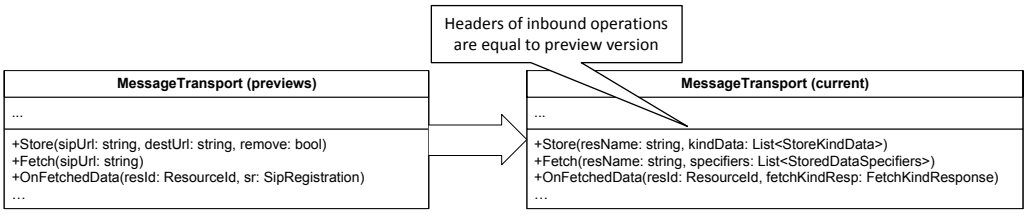


Figure 2: Adaption of Store and Fetch transactions

2.3.3 Usages

The extension of the MP2PSIP RELOAD stack allows a variety of different applications to use this stack as a messaging and storage service. A developer intending to use RELOAD just needs to implement the abstract Usage layer on top of the stack architecture. As proposed in the concept document [6], the Usage layer consists of the *UsageManager* class and the implementation of a Usage interface. The implementation of the Usage manager is kept mostly very simple since the application specific procedures must be known by classes realizing a RELOAD Usage. Two code snippets 3 of the Usage manager show the simplicity of the Usage manager. It is designed as a factory pattern where new Usages can registered and afterwards created.

```
1  public void RegisterUsage(Usage usage) {
2    if (usage == null)
3       throw new ArgumentNullException("Usage reference is null");
4    usages.Add(usage.CodePoint, usage);
5  }
6  public Usage CreateUsage(UInt32 usageCode, int? type,
7                           params string[] arguments) {
8    if (type == null && arguments == null) {
9      return usages[usageCode];
10   }
11   return usages[usageCode].Create(type, arguments);
12 }
```

Listing 3: Examples: Implementation of UsageManager

Lines 1 to 5 show the registration functionality. First, a simple check if Usage argument is not a null reference and then the new Usage got added into the dictionary of registered Usages. The registered Usage object is an empty instantiation of a concrete Usage implementation and serves as prototype for the manager component.

Lines 6 to 12 show how an already registered Usage can be instantiated using the *CreateUsage()* method. The if statement in line 8 checks if there are enough arguments and, if not, returns a empty instance of the concrete Usage demanded. Otherwise, the manager returns a new instance containing the passed arguments by using the concrete Usages' *Create()* method.

The objective using a factory pattern design is to enable a seamless implementation of further RELOAD Usages without the need to adapt the RELOAD core implementation. Hence, the Usage interface that must be implemented by further applications using RELOAD demands a lot of functionalities as presented in the concept document [6]. A condition that is not covered by the Usage interface is the demand that a Usage class must have at least a public constructor

taking a reference to the usage manager as argument. It is used only for registration of a concrete Usage at the manager component. Currently, the RELOAD stack has three classes implementing the Usage interface, the *SipRegistration*, the Usage for Shared Resources [8] and the *DisCoRegistration* class.

2.3.4 Storage Module

The implementation of the new *Storage Module* is designed to handle *StoreKindData* object from inbound store requests and *StoreDataSpecifier* on inbound fetch requests. This simplifies its invocation from the message transport component since it just passes the retrieved message payload to the storage component. Internally, for each retrieved data that have a common overlay address a new *Resource* object will be created and stored. Each *StoreKindData* is disassembled into a RELOAD Kind-Id, the data to be stored itself and the generation counter for this RELOAD Kind. In this way, the storage module provides all necessary meta data about a stored data value for maintenance, e.g., a periodical check whether the life tome of a stored data is expired.

2.3.5 Access Control Policies

A RELOAD concept called access control policies (ACP) was out of scope in the previous MP2PSIP project. ACPs specify a set of rules that control whether retrieved data values from an incoming store request are permitted to be stored at an overlay location. As proposed in the concept paper [6], a *factory* pattern implements the access control policies in the extended MP2PSIP project. The implementation access control policies slightly differs from the proposed design. The verification of origin of an entire inbound message is implemented as part of the *AccessController* class and not within a concrete instance of an *AccessControlPolicy* interface. The operation to verify an entire RELOAD message is independent of the internal object it carries. A recipient of a RELOAD message must verify a signature and the authorization certificates contained within the message. The signature over the message is computed over the overlay name, transaction ID of the message, the message payload and the *SignerIdentity* structure. Each certificate retrieved from inbound messages is stored as opaque string in a dictionary using hash of the signer identify value as key.

The four default access control policies specified in the RELOAD base document [5] which are NODE-MATCH, USER-MATCH, USER-NODE-MATCH and NODE-MULTIPLE are realized as a native C# implementation of an *AccessControlPolicy* interface. As an entire store request is parsed and verified, the internal *StoreKindData* objects are forwarded the *AccessController* class. Depended on the carried RELOAD Kind and its defined ACP, the controller chooses the corresponding *AccessControlPolicy* class to verify whether the requester is allowed to store

data at this peer. The verification is performed by using a *Signature* class that represents the corresponding *Signature* structure defined in RELOAD ([5], p.53). A Signature indicates the algorithm used to sign a stored data object, contains the *SignerIdentity* object used to identify the certificate of the creator of a stored data object and, finally, a *Signature* object that contains the value of signature. The latter is computed using the resource ID, the Kind ID, the storage time, the *StoredDataValue* object and the SignerIdentify object both represented a opaque string. To enforce an ACP, e.g., the USER-MATCH policy, the representing ACP class requests the AccessController class for the certificate that corresponds to the signer identity value of the stored data object. The certificate is returned as *TElX509Certificate* object that is the representation of the Secure BlackBox library [9] for a X.509 certificate [10]. If the rfc822Name attribute of the certificate carrying the username of a RELOAD participant hashes to the requested resource ID, the verification of the USER-MATCH ACP returns true and the data object will be stored.

Access control policies that are defined by further RELOAD Usages, e.g., USER-CHAIN-ACL of ShaRe [8] or NODE-ID-MATCH of ReDir [11], can be loaded at runtime into *AccessController* class while parsing the XML overlay configuration document. This approach follows the draft on *Access Control Codes* by M. Petit-Huguenin [12] to announce new access control policies as ECMAScript [13] code. ECMAScript embedded into configuration document contains the source codes for non-standard ACPs and shall be executed if RELOAD implementations are not aware of the ACP of a incoming RELOAD Kind.

```
1  public void Square(string arg) {
2    JintEngine engine = new JintEngine();
3    string script = (
4  @"function square(x) {"+  \\
5  "   return x * x;"+       \\ JavaScript
6  " }");"                   \\
7    engine.Run(script);"
8    engine.SetParameter("x", arg);
9    Console.Write(engine.Run("square(x)"));
10 }
```

Listing 4: Example: Using Jint lib

The implementation of *Access Control Codes* in the current MP2PSIP RELOAD stack uses the *Jint* library [14]. It provides a light-wight script engine that executes JavaScript source code on a .Net runtime environment. The sample code snippet in listing 4 shows the general mode of processing JavaScript in C#. A *JintEngine* object instantiated in line 2 will be used to interpret the JavaScript that is defined in lines 3 to 6 as a '+'-concatenated string. The script just calculates the square and is referred to the engine as shown in line 7. By using the *SetParameter()* method, the method argument *arg* is passed as value *x* to the script interpreter. Finally, the result will be printed onto the console as shown in line 9.

To process *Access Control Codes*, a small EcmaScript library is implemented in an external JavaScript file. The library realizes those EcmaScripts objects that are specified in [12] and needed to process the access control codes.

2.3.6 Finger Table

Former implementation:

RELOAD is designed to be independent of the overlay routing algorithm but specifies a Chord [15] overlay as default routing algorithm. The head developer of the MP2PSIP project carried out a new RELOAD stack revision that included a Chord *finger table* for efficient overlay routing.

The evaluation of the RELOAD stack figured out that the Chord routing algorithm was not well implemented. Overlay messages did not reach their destination if the number of peers joining a emulated RELOAD overlay was larger than 30 peers. Like in SIP, transmission errors were caused by exceeding the maximum size of via header entries. RELOAD uses via headers to enable a recursive routing for response messages following the entires in the via header. The maximal size of the via header list was exceeded because overlay routing was implemented as simple ring routing. Hence, if the hop distance between two nodes was greater than the maximal size of the via header the request messages were dropped.

The primary reason for the inefficient routing was an incorrect use of the Chord *finger table* for routing. The stack implements a finger table as a list of *FTableEntry* objects containing the following attributes:

Finger: The resource ID of a Chord *finger* of a peer

Successor: The successor to the *finger*

dtLastSuccFinger: The time of the last known activity of the successor

pinging: A boolean indicated whether this peer received a ping message from the successor

valid: A boolean indicating whether this finger is still reliable

At instantiation of the RELOAD stack, the finger table is initialized correctly by setting the *FTableEntry.Finger* i to $localNode.Id + 2^{128-i} mod 2^{128}$. Further, the valid finger table entries were consulted if a peer needs to send/forward a RELOAD message. However, the finger table entries were never assigned a corresponding successor and thus never used to route a message. Another problem was the implementation of *findSuccessor()* method of Chord. The corresponding *FindNextHopTo(key: NodeId)* method of the MP2PSIP RELOAD stack always returned the immediate predecessor if the destination peer is not in a peers successor or predecessor list.

Improved implementation:

The extended MP2PSIP RELOAD stack is now using the correct Chord finger table for routing. Therefore, the extension add several new functions to the stack:

IsFinger(): This method determines whether the originator of an inbound message should be listed in the local finger table.

AddFinger(): This method assigns a successor to the corresponding fingers of a peer.

AttachFinger() This procedure is added to the joining procedure of a RELOAD peer. As a peer is attach to all its succeeding peers, it tries to attach to series of fingers to enrich its finger table with corresponding successors. This procedure is further called to add new peers into the routing table if it detects new fingers.

GetClosetPrecedingPeer() This method determines the closest preceding peer to a given overlay ID. That peers successor will be the next hop to destination. The implementation of *GetClosestPrecedingPeer* is shown in listing 5.

```
1  private Node GetClosestPrecedingNode(NodeId key) {
2    List<NodeId> nextHopList = new List<NodeId>();
3    /* insert all successors into list */
4    nextHopList.AddRange(m_successors);
5    /* predecessor is appropriate only if it precedes the given id */
6    if(m_predecessors.Count > 0)
7      foreach(NodeId pre in m_predecessors)
8        if(key.ElementOfInterval(pre, m_local_node.Id, false))
9          nextHopList.AddRange(m_predecessors);
10   /* determine closest preceding finger of finger table */
11   Node closetPrecedingFinger = GetClosestPrecedingFinger(key);
12   nextHopList.Add(closetPrecedingFinger.Id);
13   Node closestNode = null;
14   nextHopList.Add(key);
15   nextHopList = removeDuplicates(nextHopList);
16   int sizeOfList = nextHopList.Count;
17   if(sizeOfList > 1)
18     nextHopList.Sort();
19 /* Index < index of key is id of the closest predecessor or key. */
20   int keyIndex = nextHopList.IndexOf(key);
21 /* Returns index of preceding peer */
22   int index = (sizeOfList + (keyIndex - 1)) % sizeOfList;
23   return GetNode(nextHopList[index]);
24 }
```

Listing 5: Chord: Closest preceding peer algorithm

3 Measurement and Evaluation

3.1 Measurement Setup

To evaluate the implementation of the extended MP2PSIP RELOAD stack, this document presents several measurements taken from emulated RELOAD overlays. Two devices were used to emulate small RELOAD overlays. First, an enrollment server running on an Ubuntu (10.04) Linux OS, with 8GB RAM at 2 * 2.4Ghz. Second, a Windows 7 OS device with 8GB RAM at 4 * 2.66Ghz processing power emulating the RELOAD peers. Both devices are connected via 802.3u Ethernet in the same IP domain.

The test application is a thin console client of RELOAD that is automatically executed by a Python script. The logging functions of the stack were reduced to a minimal amount of data, thus to minimize its influence to measured delay times. The test application uses the same Concurrency and Coordination Runtime (CCR) [16] library for logging as the RELOAD stack for asynchronous messaging. Hence, the logging results were written asynchronously without blocking the main thread of the RELOAD stack.

Event though the test application is reduced to the minimal functions to emulate a RELOAD overlay, each initialized RELOAD peer consumes an amount of approx. 35 MB RAM. The large size due to the dynamical load of several .Net framework modules at runtime. Consequentially, the maximal number of peers that can be evaluated on the test device is limited. For the following measurement result the maximal number of peers is reduced to 100 emulated peers including the bootstrap peer. This setup guarantees that the measurement results are not affected by exhausted memory capacities.

The following measurement results for joining the overlay and for storing/fetching data values are the average result of 50 independent runs.

3.2 Joining a RELOAD overlay

The joining procedure that was measured combines the following steps:

- Contacting the enrollment server for configuration and authentication
- Attaching the bootstrap peer including onwards forwarding to the admitting peer
- Attaching to a set of RELOAD peer to enrich a peers routing table
- Joining the admitting peer to finally participate the overlay

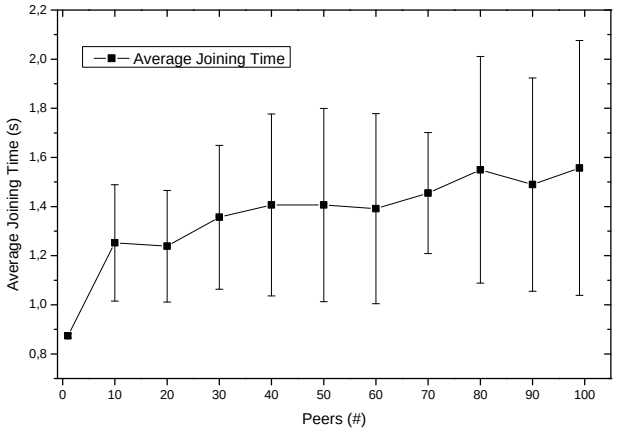
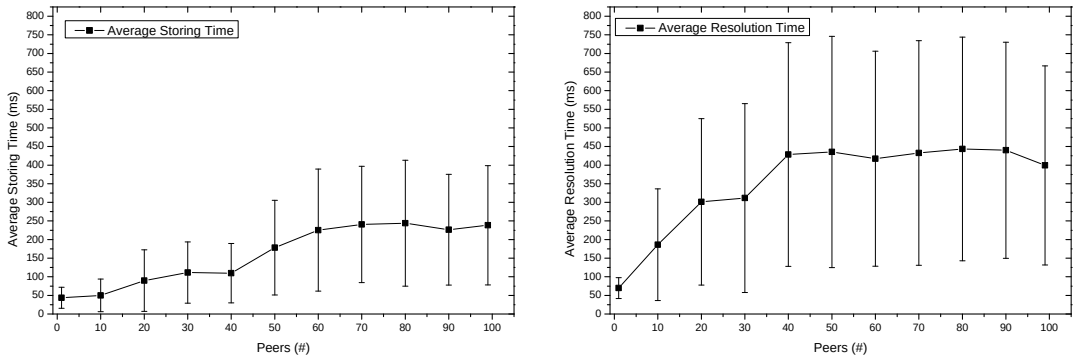


Figure 3: Measurement result: Average delay to join a RELOAD overlay

After the successful Join request to the admitting peer, the joining peer is ready to take its place in the DHT. The routing table will contain up to three successors and predecessor and additional 8 fingers. It becomes part of the overlay routing and is responsible to store values in its address range. Hence, the key-handover from the admitting peer to the joining peer and the RELOAD procedure to publish its arrival in the overlay (sending Update messages to peer in routing table) was not included in the measurements.

The measurement results in figure 3 present the average delay times to join a RELOAD overlay. The ordinate shows the average delay in time in total seconds to depending on the total number of peers shown in the abscissa. Additionally, at each 10th peer the standard deviation in total seconds is indicated. The resulting graph shows a rapid linearly growth until an amount around 10 peers, followed by a more moderate increasing joining times afterwards. The joining procedure takes $\sim 0.87s$ for the first peer and increases to $\sim 1.5s$ at 99th peer joining. A small overlay can not use the advantage of a logarithmic scaling finger table. In an overlay participated by around 7 peers each peer has a direct connection to all other peers because its knows its next three preceding and succeeding peers. Hence the time to join simply increases because of the additional hops. As the number of peers growths, the more imported becomes the finger table for the routing decision. A routing decision via the via finger table may causes (one or two) additional hops to reach the destination. This behavior is represented by the measurement results. The delay times to join a RELOAD overlay is growing, but remains in an interval of approx. $1.2s$ to $1.5s$. A further measurement setup with a superior amount of peers should validate that joining delays will increase logarithmically.



(a) Average storing delay

(b) Average fetching and appattach delay

Figure 4: Measurement results: Average delay to store (a) and resolve (b) a SIP record

3.3 Storing and Resolving Records

The second measurement in this document presents the average delay times to store a SIP record in the overlay and to resolve the latter to a contact address. The registration of a SIP record is achieved by a single Store request routed throughout the overlay to storing peer. The procedure to resolve a SIP record to a contact address is more complex. If a SIP registration is of type *sip_registration_uri*, the initial fetch request on a SIP URI returns another SIP URI that need to be resolved again until a SIP record resolved to a *sip_registration_route* type. The latter contains a list of RELOAD *Destinations* and corresponding contact preferences. In the simplest case, the list contains a single node ID of the RELOAD peer to be called. A calling peer then performs an *AppAttach* request routed to that node id. The corresponding answer message contains the contact address (ICE candidate) to called peer. For the measurement results presented in figure 4, the SIP record is directly stored as *sip_registration_route* type, thus having just two operations to resolve a SIP URI.

The measurement results in figure 4a present the average delay in milliseconds to store a SIP registration while increasing number of peers. Additionally, the graph shows standard variation at each 10th peer. The storing times are slightly increasing to an amount of 50 peers. At an amount of 60 peers, the storing delay is flattening and remains constant until the 99th peer storing a SIP record. This behavior due to slowly adapting routing tables of the peers compared to the frequency of new appearing peers. The Chord maintenance routines are a factor 10 slower than the frequency of new peers joining the overlay. Established peers might have out of date routing tables causing the longer routing of messages. This effect decreases if the overlay has a greater amount of peers. The attach request send to gather new fingers while joining the overlay, already return viable set of successors. Explained with a counter example,

if a peer is joining an overlay of 20 peers, the pre-joining attach requests to enrich its finger table might return just two different successors due to large gaps in the address space. Those already established peers will update their finger tables through the fix fingers procedure and will obtained a more suitable set of fingers.

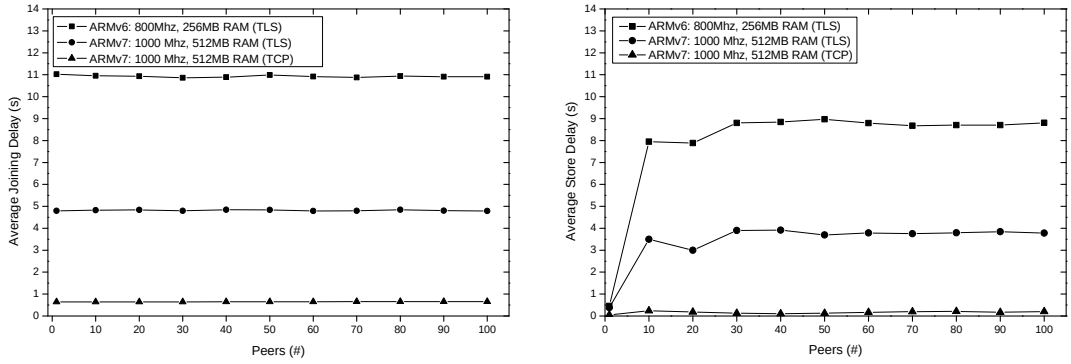
The relatively high standard variation reflects the randomized assignment of node IDs. In an iteration of measurement a store by peer N might routed to a peer with a node ID close to N, in another iteration, the same store request might be routed to the almost furthermost node ID compared to N.

Analogously, the measurement results in figure 4b presents the average delay time for a *Fetch* request with subsequent *AppAttach* procedure. As expected, the delay to resolve a SIP record to a contact address increases linearly to an amount of 40 peers and further remains constant. This due to the same effects as explained for the storage. Further measurements with an amount of thousands of peers will show that the storing and resolution delay will continue logarithmically.

3.4 Joining RELOAD from a mobile device

The third measurements presented in this work shows the average delay to join a RELOAD overlay from a mobile device with a subsequent registration of a SIP record. The measurement setup is slightly different compared to setup of the previous results. The mobile device will join the overlay as a RELOAD *client*. The mobile clients joining the overlay over TLS perform the same enrollment procedure as the RELOAD peers. Peers joining the overlay over TCP just skip contacting the enrollment server and use a local configuration file. After enrollment, the mobile client sends a RELOAD *Attach* request to the bootstrap peer which will forward the request to the admitting peer of the mobile client. The corresponding answer message attaches the mobile client to the overlay and completes its joining procedure. The mobile devices were connected via a 802.11 interface. A measurement of the delay times through a 3G network was out of scope of this evaluation.

The following measurement results shown in figure 5 presents the average delay time in total seconds to join a RELOAD overlay and to store a SIP record over TLS and TCP. Thereby both graphs compare two different mobile devices in dependency to the size of an emulated overlay. The overlay is emulated as described in the measurement setup 3.1 for the previous measurements. However, the measurement results represent the average delay times of 10 independent iterations. The two mobile devices have different hardware capabilities. The first device has 256MB RAM with a Samsung S3C6410 core at 800Mhz (ARMv6). The second device has 512MB RAM with a QSD8250 SnapDragon at 1 Ghz (ARMv7).



(a) Average joining delay

(b) Average storing delay

Figure 5: Measurement results: Joining and storing delay from a mobile device via 802.11

The average joining delays over TLS shown in figure 5a of both devices are relatively large compared to the joining delays over TCP. While the 1Ghz devices needs approx. $4.8s$ to join, does the 800Mhz devices take an amount of $\sim 11s$ to connect to the overlay. The large joining times due from the effort to establish a secure TLS connection with the overlay peers. The 1Ghz Snapdragon architecture benefits from the ARMv7 *NEON* [17] engine providing a more efficient floating point unit. The same measurement performed over TCP remains constantly around $\sim 0.625s$ and supports the assertion of the complexity to establish a secure connection from a mobile device. It is suspected that the TLS library is not optimized for usage on devices with limited computing capacities and hence cause those large delays for connection establishment. Overall, the average joining times seem to a constant complexity. Actually, the time to setup a secure connection is a factor of 10 to 30 times greater than the subsequent routing of overlay messages - the RELOAD *Attach* request.

The average delay for storing a SIP record is shown in figure 5b. It also compares the average delay needed by both mobile devices in dependency to the overlay size over TLS and TCP. In contrast to the joining procedure a store request takes mush less time if the overlay consists of just a single peer, thus the bootstrap peer. The mobile device with 800 Mhz needs a average delay of $\sim 0.48s$ and the test device with the 1Ghz Snapdragon processor $\sim 0.38s$ to store a value over TLS and $\sim 0.25s$ over TCP. In this test setup, a mobile client has already established a TLS connection to the bootstrap peer that takes the role of the admitting peer of the client. Hence, it can use an exiting connection to send the store request that will be processed by the bootstrap in its additional role as the storing peer. If the number of peers increases the average storing times are increasing as well if TLS is used as transport protocol. If the admitting peer is any other peer than the bootstrap, the mobile client has to establish another TLS connection to its admitting peer. This procedure is not yet finished on sending the store request containing

the SIP registration. This causes the large delay times shown in figure 5b commencing at 2th peer in the overlay with delays about $\sim 3.5s$ up to $3.9s$ for the 1GhZ SnapDragon and $\sim 7.9s$ to $\sim 8.3s$ for the Samsung device. The storage of values over TCP is unaffected of burden to establish a second connection and remains constant with a delay around $0.25s$. If the size of the overlay is greater than 30 peers, the probability that the bootstrap peer is also the admitting peer decreases linearly and forces the mobile client to establish a second connection.

4 Conclusion and Outlook

This work was dedicated to the implementation of extension to the MP2PSIP RELOAD stack. The RELOAD stack is now enabled to seamlessly integrate further RELOAD Usages and Kind definitions. The stack interface to upper layer applications provides a generic mechanism to process any kind of application data supported by the stack. A renewed message transport module is now following the protocol standards. A on top Usage layer allows a seamless implementation of further RELOAD Usages and its defined Kind data structures. A new storage module is separated from the RELOAD topology-plugin and is enabled to store any Kind of data. A policy module is now controlling the RELOAD access control policies and is enabled to execute ECMAScricpt codes used for new policies not implemented by the RELOAD core classes. The Chord *Finger Table* is now actively used and maintained by the stack and enables a scalable overlay routing.

The evaluation of the MP2PSIP RELOAD stack showed that the advantages of a Chord finger table are noticeable if the number of joining peers exceeds than 40-50. As all peers have an adequate number of routing table entries, the delay of routing of overlay messages was constant until the maximum amount of 100 peers evaluated in this work. Further measurement performed by mobile devices revealed that the joining and storing procedures are 10 to 30 times slower if TLS is used as the underlying transport protocol. This is reasoned by less computing power of mobile devices and a may inefficient implemented TLS stack.

It is future work to emulate RELOAD overlay of much larger numbers of peers that will validate the logarithmic scaling of this Chord based overlay. Therefore, the RELOAD stack should be adapted to be executable on a Linux OS upon the .Net cross-platform Mono[18]. Further evaluation could be performed using the Planet-Lab [19] environment to emulate thousands of peer at various geographical locations. Finally, an implementation of the DisCo specification will demonstrate that distributed conferences in P2P scenarios will provide a better scalability compared to centrally managed approaches.

References

[1] J. Rosenberg, “A Framework for Conferencing with the Session Initiation Protocol (SIP),” RFC 4353, IETF, February 2006.

[2] J. Rosenberg, H. Schulzrinne, G. Camarillo, A. Johnston, J. Peterson, R. Sparks, M. Handley, and E. Schooler, “SIP: Session Initiation Protocol,” RFC 3261, IETF, June 2002.

[3] A. Johnston and O. Levin, “Session Initiation Protocol (SIP) Call Control - Conferencing for User Agents,” RFC 4579, IETF, August 2006.

[4] A. Knauf, G. Hege, T. Schmidt, and M. Waehlisch, “A RELOAD Usage for Distributed Conference Control (DisCo),” Internet-Draft – work in progress 02, IETF, March 2011.

[5] C. Jennings, B. Lowekamp, E. Rescorla, S. Baset, and H. Schulzrinne, “REsource LOcation And Discovery (RELOAD) Base Protocol,” Internet-Draft – work in progress 13, IETF, March 2011.

[6] “Analysis and Enhancement Design of the MP2PSIP RELOAD Stack.” http://inet.cpt.haw-hamburg.de/members/alexander-knauf/knauf, 2011.

[7] C. Jennings, B. Lowekamp, E. Rescorla, S. Baset, and H. Schulzrinne, “A SIP Usage for RELOAD,” Internet-Draft – work in progress 05, IETF, July 2010.

[8] A. Knauf, G. Hege, T. Schmidt, and M. Waehlisch, “A Usage for Shared Resources in RELOAD (ShaRe),” Internet-Draft – work in progress 00, IETF, March 2011.

[9] “SecureBlackbox Suite.” http://eldos.com/sbb/, 2011.

[10] D. Cooper, S. Santesson, S. Farrell, S. Boeyen, R. Housley, and W. Polk, “Internet X.509 Public Key Infrastructure Certificate and Certificate Revocation List (CRL) Profile,” RFC 5280, IETF, May 2008.

[11] J. Maenpaa and G. Camarillo, “Service Discovery Usage for REsource LOcation And Discovery (RELOAD),” Internet-Draft – work in progress 02, IETF, January 2011.

[12] M. Petit-Huguenin, “Configuration of Access Control Policy in REsource LOcation And Discovery (RELOAD) Base Protocol,” Internet-Draft – work in progress 01, IETF, March 2011.

[13] “ECMAScript Documentation.” http://www.ecmascript.org/, 2011.

[14] “Jint - Javascript Interpreter for .NET.” http://jint.codeplex.com/, 2011.

[15] I. Stoica, R. Morris, D. Liben-Nowell, D. R. Karger, M. F. Kaashoek, F. Dabek, and H. Balakrishnan, “Chord: A Scalable Peer-to-Peer Lookup Protocol for Internet Applications,” *IEEE/ACM Trans. Netw.*, vol. 11, no. 1, pp. 17–32, 2003.

[16] “Microsoft Robotics - Concurrency and Coordination Runtime (CCR).” http://msdn.microsoft.com/en-us/library/bb905470.aspx, 2010.

[17] “ARM NEON Technology.” http://www.arm.com/products/processors/technologies/neon.php, 2011.

[18] “Mono Project homepage.” http://www.mono-project.com/Main_Page, 2011.

[19] “The PlanetLab homepage.” http://planet-lab.org/, 2010.

List of Figures

Listings