Hochschule für Angewandte Wissenschaften Hamburg
Hamburg University of Applied Sciences

Ausarbeitung Projekt 1 - WiSe 2010/11

Alexander Knauf

Problem Analysis and Concept for Extending the *MP2PSIP* RELOAD Stack

Fakultät Technik und Informatik
Department Informatik

Faculty of Engineering and Computer Science
Department of Computer Science

Contents

1 Introduction

SIP session management is generally built upon a dedicated server infrastructure of SIP proxies and registrars that are used to locate and interconnected the end user devices. The system performance of the latter and the ubiquitous availability of broadband Internet on flat rates enabled P2P solutions for several centrally managed services, such as voice and video conferencing. This development was recognized by the standardization organization IETF by chartering the P2PSIP working group to develop protocols that manage SIP session establishment in a decentralized P2P fashion. To elaborate and adopt a baseline protocol for P2P storage and messaging service, the working group is developing the RELOAD base [1] protocol.

Today, a very limited set of implementations of the RELOAD base protocol are available and are often out of date or incomplete. An advanced implementation of RELOAD is available from the *MP2PSIP* Project. The stack is written in the C# programming language and was developed for a feasibility study by a large German Internet provider. It is designed to demonstrate how the RELOAD protocol can be used at ISPs to deploy a decentralized SIP session management infrastructure utilized by mobile devices also running RELOAD as client. The stack implementation covers most of the RELOAD concepts, but has several protocol incompatibilities compared to the standard.

In this document, the current status, architecture and programming techniques of the *MP2PSIP* RELOAD stack will be analyzed and a concept to enhance the stack functionalities is presented. The analysis will show how the RELOAD base specification is realized with the 3rd generation programming language C#, using several benefits of the large .Net framework. Furthermore, this document will reveal some erroneously implemented specifications and proposes several refinements within an enhanced concept. The improved MP2PSIP RELOAD stack should serve as base for the approach of a distributed conferencing (DisCo) scheme with SIP [2] that provides ad-hoc multiparty conferences managed by its participating peers.

The remainder of this document is structured as follows. Section 2 provides a detailed analysis of the MP2PSIP project, while section 2.2 gives an overview of the entire stack implementation. Section 2.3 shows details of the implementation and section 2.4 summarizes the stack properties including a list of further missing features. In section 3, a concept to enhance the MP2PSIP project is presented. This document concludes and gives a look on future work in section 4.

2 Analysis of MP2PSIP Stack concerning an Implementation of DisCo

2.1 Objective of the Analysis

An alternative approach for centrally manged multimedia conferences is presented by the protocol scheme for Distributed Conferencing (DisCo) [2]. It distributes the traditional central manager of a multiparty conversation on several independent endpoints that use a common SIP URI as conference identifier. The DisCo approach is built upon a P2PSIP overlay for REsource Location And Discovery (RELOAD) [1] used to register the common conference ID. An advanced implementation of the RELOAD base specification is the *MP2PSIP* RELOAD Project. It was designed as a demonstrator by a large German ISP [1] to evaluate the capacities of the RELOAD base protocol. The objective of this work is to determine whether the MP2PSIP project could serve as the RELOAD base implementation to implement the DisCo approach and, if necessary, to present concepts to extend its design. Furthermore, the analysis will figure out whether MP2PSIP RELOAD is compliant to protocol standards for guaranteeing interoperability to other RELOAD implementations.

2.2 The MP2PSIP Project

Project Overview

The MP2PSIP project is composed of several modules that are build upon the RELOAD protocol core implementation. Figure 1 shows the main components of the MP2PSIP project and their dependencies or interactions between them. The *RELOAD Class* module is the core component of the project. It implements the RELOAD base [1] protocol and the specification of the SIP Usage [3] which serves also as test application for the MP2PSIP stack. *RELOAD class* is used by the *RELOAD MDI* and *RELOAD Service* components. The RELOAD MDI is a graphical user interface created with the Windows.Forms API [4]. It provides mechanisms to crate a new RELOAD peers or clients by just clicking a button. It further allows to execute several RELOAD operations, e.g., *join*, *leave* or *store* and *fetch* operations. The *RELOAD Service* component can be used as daemon to execute a single RELOAD peer persistently on a Windows OS. It does not have user interface and is designed to run as a bootstrap node for a RELOAD overlay instance.

The *Chord Monitor* tool is an ASP.NET web project and is used to visualize a running RELOAD overlay. It renders all RELOAD peers and clients on top of a Google map [5] and displays each

[1] For reasons of confidentiality, the industrial partner will not be named publicly.

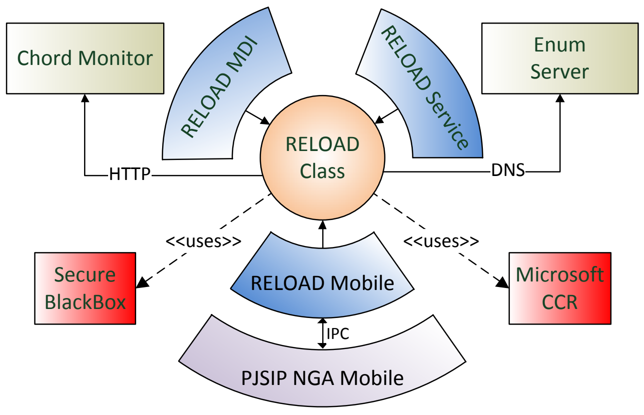


Figure 1: Project Overview

predecessor/successor relations and each established transport connection. The resulting graph arranges all participants of a RELOAD overlay according to the their overlay addresses in Chord [6] overlay typical ring topology. The monitoring tool obtains its rendering information from JSON objects sent from the RELOAD class component at every time a node state changes or a RELOAD message has been sent.

The RELOAD Mobile component is the mobile variant of the RELOAD core classes. It has several modifications and preprocessor directives in the source code that enables the compilation of the RELOAD stack on Windows Mobile 6.X devices. The mobile RELOAD variant is used by the *PJSIP NGA* mobile application. It is a full VoIP application written in C using the open source PJSIP stack [7] providing a SIP signaling and media library. In contrast to the RELOAD base protocol that foresees a username/password authentication does the mobile application use the International Mobile Subscriber Identity (IMSI) to uniquely identify a mobile RELOAD participant. IMSIs are used in GSM and UMTS networks as unique identifier for a participant in a mobile network. Mobile clients utilize the telephone number to establish SIP sessions. Because there exists no protocol standard to use mobile telephone numbers in a RELOAD overlay, the MP2PSIP uses an ENUM server that maps a telephone number to a SIP URI that can be registered in the overlay by the SIP Usage [3].

Stack Architecture

The RELOAD specification [1] defines a layered architecture that divides the network stack into several independent components. The MP2PSIP stack implementation follows this proposition,

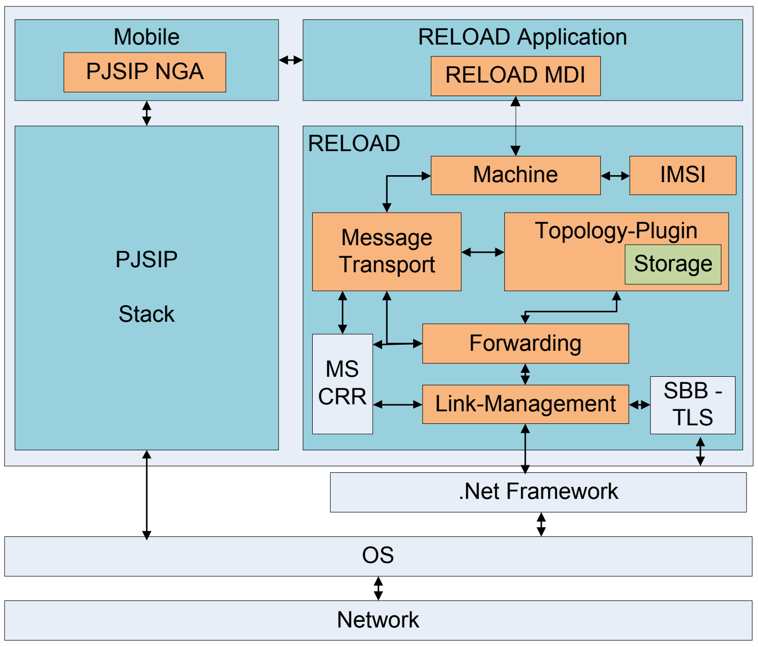


Figure 2: Architecture of the MP2PSIP RELOAD Stack

but skips the recommendation to separate the storage and Usage modules from the rest of the protocol stack.

An overview of the stack architecture is given in Figure 2. The implementation is divided into the mobile application and the desktop variant. Both are using the RELOAD stack including the *SIP Usage*. The mobile *SIP* client and RELOAD MDI applications utilize the RELOAD stack as abstract storage and messaging service. They communicate with the lower layers through the *Machine* component - a background worker that handles user requests asynchronously. The RELOAD stack itself is composed of a *message transport*, *topology-plugin*, *storage*, *forwarding* and *link-management* components as described in the RELOAD specification [1]. An additional IMSI module is utilized by the SIP Usage and resolves a mobile subscriber identity into SIP URIs. The stack uses two external components. The Microsoft Robotics *Concurrency and Coordination Runtime* (MS CCR) [8] library for asynchronous communication and the *SecureBlack Box* [9] library to enable a secure communication via TLS.

The MP2PSIP RELOAD project is running on top of the .Net framework, while the mobile SIP application is natively written in C. Both programs are simply communicating via the Windows registry for inter process communication. The mobile RELOAD variant listens via a change event handler whether the mobile SIP client is sending commands using the registry. If it is a fetch, the RELOAD stack tries to resolve the SIP URI engaging the RELOAD lookup abilities to retrieve the IP address and port of the called party. The contact information is then stored as another Windows registry entry, that will be used by the SIP application to create a SIP and media session. The RELOAD MDI user interface is communicating directly to the RELOAD stack by reference to the *Machine* background worker. User commands are queued and will be processed periodically by the RELOAD stack.

The next lower layer provides the *message transport* and the *topology* components. The message transport is responsible for creating message bodies and is invoked by the *Machine* component. The *topology-plugin* provides a generic interface for the key-based routing layer. The lowest layer of the RELOAD stack provides the *forwarding & link-management* components. The former implements the packet forwarding service, while the latter maintains the transport connections for the routing table. The RELOAD base protocol is designed to be independent from a concrete overlay algorithm. The enrichment and maintenance of the routing table is controlled by the upper topology-plugin that is aware of used overlay algorithm. The stack design allows a seamless exchangeability of the topology-plugin to switch to another routing algorithm.

Traffic between the overlay peers is transported by the link management module. It frames the overlay messages within a secure transport protocol. The actual stack implementation can run over TLS [10] as required by the RELOAD base specification [1], but can also use TCP for testing and evaluation scenarios. For TLS transport, the RELOAD stack uses the *SBB* [9] library to generate and maintain TLS connections.

The message transport, forwarding and link management components are communicating via the Microsoft *CCR* [8] that deals with asynchronous operations that are characteristic in a transactional protocol as RELOAD. Using CRR has the advantage that the forwarding and link components can be kept loosely coupled, and the stack implementation need not care about concurrency constraints.

A more detailed description about the implementation and analysis of the MP2PSIP stack follows in the next section 2.3.

2.3 Analysis of the RELOAD stack Implementation

Initializing the RELOAD Stack

The MP2PSIP RELOAD stack can be initialized by creating a new instance of the RELOAD.MACHINE class as shown in listing 1. The minimal configuration to run the stack is a delegate that has a RELOADGLOBALS.TRACEFLAGS enum and string as parameter and has no return values. The delegate must be set after stack instantiation by setting its *ReloadConfig.Logger* property (see line 8 in 1). The delegate implements a trace logger using the TRACEFLAGS to categorize the logging output, e.g., TRACEFLAGS.T_ERROR to indicate an internal error or exception. A minimally initialized RELOAD stack will use the default configuration that instantiates a RELOAD *client* application that tries to contact the default enrollment server and bootstrap peer. A detailed stack configuration can be done through the RELOADCONFIG member variable of the MACHINE class or the RELOADGLOBALS class that contains static variables.

```
using Lib.RELOAD;
namespace ReloadTest {
  class Program {
    static void Main(string[] args) {
      Machine machine = new Machine();
      // Set Logger delegate
      machine.ReloadConfig.Logger = new ReloadConfig.LogHandler(Logger);
    }
    void Logger(ReloadGlobals.TRACEFLAGS traces, string s) {
      /* Implementation of Logger */
    }
}
```

Listing 1: RELOAD machine init

While the former configuration focuses on user dependent properties (e.g., IMSI, SIP URI, *isClient*, *isPeer*) does the Globals class adjust the stack properties, e.g., retransmission timer or enrollment server address. Those values are partially settings for a specific RELOAD overlay instance and will be adjusted automatically through an overlay configuration document that is retrieved at enrollment.

Message Transport

The message transport component is composed of two main modules – the RELOAD.MESSAGES file and the RELOAD.TRANSPORT class. The Reload.Messages file is a container for multiple C# classes and structs that represent the messages definitions of the RELOAD base [1] protocol and the SIP Usage [3] specification. Multiple public classes and structs within within a single file is in contrast to the C# coding style [11], but an option to group related classes by their remit. A RELOAD message is represented by RELOADMESSAGE class and is a composition of the *ForwardingHeader* struct, a *RELOAD_MessageBody* class and the *SecurityBlock* struct. It is implemented as abstract class that contains a single instance member variable representing the message code and the two virtual member methods *Dump()* and *FromReader()*. It is a generalization for several classes implementing a RELOAD_MessageBody. The class hierarchy is shown in figure 3.

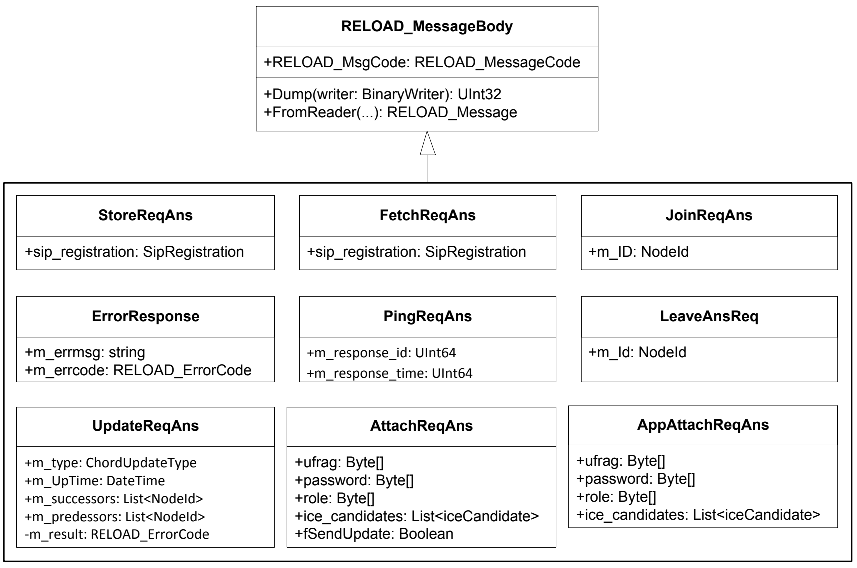


Figure 3: Class hierarchy of RELOAD message body implementations

Implementations of *Dump()* realize the serialization of the message content in network byte order and returns the length of the serialized message in bytes. The *FromReader* implementation is the corresponding deserialization functionality that returns a new instance of a concrete reload message body.

By comparing the message body implementations with the RELOAD base specification it can be recognized that the attributes of StoreReqAns, the FetchReqAns and PingReqAns differ from the proposed standard in the RELOAD base specification [1]. While the ping request definition is just following a previous draft version, the store and fetch request only contain a SipRegistration member field. The base specification [1] defines a more generic payload in for request and corresponding answer messages. As the result, the current implementation does not implement the RELOAD meta-model and is not capable of creating storing and fetching requests for other Usages than the SIP registration Usage.

The Message.Transport class implements the message transaction model for several protocol procedures, e.g., flows for joining the overlay or the RELOAD *App/Attach* procedures. An interesting programming technique facilitates the interconnection of the message transport with forwarding and link-management components. The Microsoft *Concurrency and Coordination Runtime* (CCR) library [8] provides the handling of the RELOAD messaging protocol. RELOAD is an asynchronous protocol in a transactional model in which each request get acknowledged with a corresponding response message. If there are several outgoing requests waiting for answer that may arrive in another order it can be difficult to handle those concurrencies. The RELOAD stack leaves this issue to the CCR library. A DISPATCHER class manages OS threads and balances the dequeue of classes implementing an ITASK interface that are queued within DISPATCHERQUEUE instance. Tasks can be enqueued using an ARBITER class as shown in listing 2.

```
 1 reloadSendMsg = create_store_req(new Destination(res_id), data);
 2 reloadDialog = new ReloadDialog(m_ReloadConfig, m_flm, node);
 3 /* m_DispatcherQueue initialized includes Dispatcher */
 4 Arbiter.Activate(m_DispatcherQueue,
 5   new IterativeTask<ReloadMessage, ReloadMessageFilter, int>(
 6     reloadSendMsg,
 7     new ReloadMessageFilter(reloadSendMsg.TransactionID),
 8     RetransmissionTime,
 9     reloadDialog.Execute));
10 ...
11 yield return Arbiter.Receive(false, reloadDialog.Done, done => { });
```

Listing 2: Store request using CCR

The displayed code snippet shows the *Store()* method of the message transport. The first two lines just show the initialization of the reload message body (the store request) and a RELOAD dialog object that allows the stack to follow the message transaction. The *Arbiter.Activate()* method in line 4, queues a new instance of an ITERATIVETASK (lines 5 to 9) that obtains the RELOAD message and a message filter as arguments for the *reloadDialog.Execute()* method. An iterative task can be initiated with multiple generic parameters followed by a delegate to a method that can be invoked by the dispatcher - here *reloadDialog.Execute()*. The *Execute()* method invokes any implementation of the forward and link-management component (e.g., TCP or TLS module) to encapsulate the RELOAD message in a transport protocol.

After arbiter activation, the dispatcher will execute the queued iterative task asynchronous to the invoking thread. Hence, the store method thread shown in listing 2 is not blocked between the enqueue of the iterative task and its execution by the dispatcher. If the store request was send and the corresponding answer is received, the ARBITER.RECEIVE() will be activated through the *reloadDialog.done()* method. The third argument of the *Arbiter.Receive()* display in line 11, could be a delegate or the definition of an anonymous method that is executed after receive. In this example, it is implemented as an anonymous method that has no further instructions. Using this programming technique, the implementor need not care about concurrent invocations of the lower send methods and an asynchronous message passing can be implemented just as a call of a static method.

Topology and Storage

The topology-plugin is a concept of the RELOAD base specification that decouples the messaging and storage service from the specific overlay algorithm. The MP2PSIP project follows the recommendation to maintains routing information in a separate module. The realization is implemented within RELOAD.TOPOLOGY.ROUTINGTABLE class that is a nested class of RELOAD.TOPOLOGY class. It contains several member methods that provide the key-based routing information, e.g., *SetFingerTable()* or *GetSuccessor()*, but also provides methods that actively manage connections engaging the *forwarding & link-management module*, e.g., to populate its routing table if it detects a change in the routing topology. Routing table entries are implemented as another nested class of RoutingTable. This RTABLEENTRY class contains four member variables:

icecandidates: This field is a list of ICE candidates that can be used to connect to the remote party. In contrast to other DHTs that store contact address to an endpoint (IP:Port) does RELOAD use the ICE protocol [12] for connection establishment. ICE *" Provide a means for two peers to determine the set of transport addresses that can be used for communication."* ([12] p.100).

dt: This field is a DateTime object to save the time of first attach to the remote host.

nodestate: This enum provides the information of remote host if it is *unknown*, *attaching*, *attached* or if the local host received update messages from the remote endpoint.

pinging: The last member is a boolean flag that indicates whether the remote host is still pinging – the keep-alive procedure of RELOAD.

The architecture of the MP2PSIP RELOAD stack combines the topology-plugin with the storage into a single component as indicated in figure 2. Data storage is performed by the topology component by providing each a member method for *Store()* and *Fetch()*. They are implemented as Setter and Getter for the Topology member variable called *m_StoredValue* that is a dictionary of SipRegistration instances with strings as their keys. Hence, the MP2PSIP RELOAD stack is only designed to store and fetch values for the SIP registration Usage. A more generic storage implementation that can handle various types of data including their meta-information is needed.

Forward and Link-management

The lowest layer seen from the RELOAD architecture are the forwarding and link-management components. Forwarding is implemented in the RELOAD.FORWARDINGLAYER class. It provides only the member method *ProcessMsg()* returning a boolean that indicates if true, that a received message must be forwarded or, if false, that a received message must be processed at this peer.

The link-management is more complex. Actually, the MP2PSIP project implements two link-management variants, the RELOAD.OVERLAYLINK.SIMPLEFLM that works with TCP and the RELOAD.OVERLAYLINK.RELOADFLM that runs on TLS using the Secure BlackBox [9]. Both implement the RELOAD.IFORWARDLINKMANAGEMENT interface that is displayed in listing 3.

```
public interface IForwardLinkManagement {
  bool Init();
  void Send(Node NextHopNode, ReloadMessage reloadMessage);
  void ShutDown();
  event ReloadFLMEvent ReloadFLMEventHandler;
  bool NextHopInConnectionTable(NodeId dest_node_id);
  List<ReloadConnectionTableInfoElement> ConnectionTable { get; }
}
```

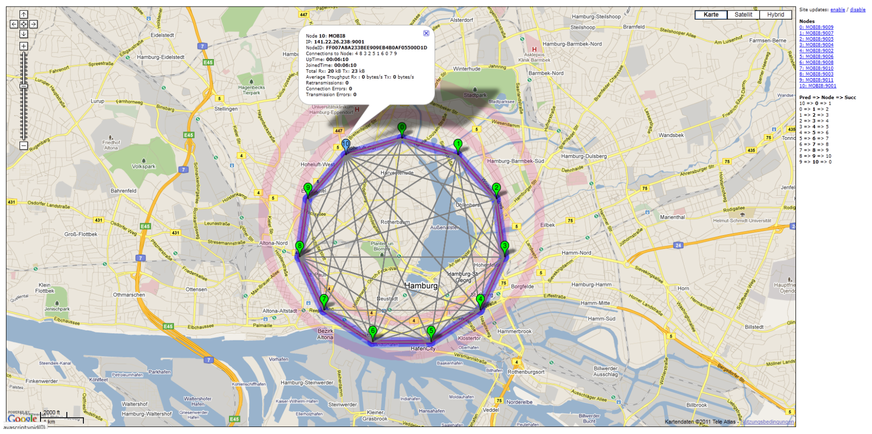
Listing 3: Link-management interface

Apart from the obligatory *Init()* and *ShutDown()* functionalities, a forward and link manager needs to implement a method for *Send()*, a RELOADFLMEVENTHANDLER to announce incoming message to the upper layers and the CONNECTIONTABLE property. The *NextHopInConnectionTable()* method seems to be dead source since no other class has any reference to it. Nevertheless, this interface is highly generic to the upper layers and its implementations seem to have no obvious bugs. Hence, this analysis will not go deeper into the details of this component.

2.4 Summary

The MP2PSIP RELOAD project provides several utilities for demonstration purposes like the *Chord Monitor* view (see figure 4) or the RELOAD MDI overlay management console. The RELOAD stack itself implements most of the base protocol specifications [1] to demonstrate a SIP registration scenario in RELOAD. The possibility to deploy the RELOAD stack onto a mobile device with a proper SIP client application is a useful feature. Using the Microsft CCR library to provide the messaging service is an elegant way to handle asynchronous send a receive processes.

Apart from the missing features described in the stack analysis 2.3 the following RELOAD properties were out out scope in MP2PSIP project.



```
Node 10: MOBI8
IP: 141.22.26.238:9001
NodeID: FF007A8A233BEE909EB4B0AF05500D1D
Connections to Node: 4 8 3 2 5 1 6 0 7 9
UpTime: 00:06:10
JoinedTime: 00:06:10
Total Rx: 20 kB Tx: 23 kB
Averiage Troughput Rx : 0 bytes/s Tx: 0 bytes/s
Retransmissions: 0
Connection Errors: 0
Transmission Errors: 0
```

```
Site updates: enable / disable

Nodes
0: MOBI8:9009
1: MOBI8:9007
2: MOBI8:9005
3: MOBI8:9004
4: MOBI8:9002
5: MOBI8:9006
6: MOBI8:9008
7: MOBI8:9010
8: MOBI8:9003
9: MOBI8:9011
10: MOBI8:9001

Pred => Node => Succ
10 => 0 => 1
0 => 1 => 2
1 => 2 => 3
2 => 3 => 4
3 => 4 => 5
4 => 5 => 6
5 => 6 => 7
6 => 7 => 8
7 => 8 => 9
8 => 9 => 10
9 => 10 => 0
```

Figure 4: Chord monitoring tool

ICE: Although the MP2PSIP RELOAD stack provides several components ready to run Interactive Connectivity Establishment (ICE) [12] for NAT traversal an concrete implementation of ICE is not present. The reason for this is the enormous complexity of the ICE protocol. A full ICE implementation would have exceeded the time and scope of the MP2PSIP project.

Certificate & TURN Usages: The base specification [1] defines the two mandatory Usages for Certification storage and TURN Usage. The certificate Usage stores a user's certificate in the overlay thus to avoid the need to send the certificate at each message. The TURN Usage specifies that a RELOAD peer should advertise itself in the overlay if its prepared to work as a TURN Server [13]. Because the MP2PSIP RELOAD stack is limited to store and fetch SIP Registrations, it is consequentially not able to support the Certificate and TURN Usages.

Access Control Policies: The overlay locations where a RELOAD peer is allows to store data values is limited by a small set of *Access Control Policies*. Those policies are most often bound to a user certificate that permit the storage of data at the Resource Id that corresponds to the hash over a username or its node ID. The MP2PSIP RELOAD stack is a good-natured overlay and allows the storage at a peer if the request is addressed with a resource Id that is in its range.

ConfigUpdate, Stat and Find Messages: The MP2PSIP RELOAD stack does not implement the *ConfigUpdate*, *Stat* and *Find* messages. ConfigUpdate is used if a peer detects that another peer's configuration document is out of date. It then sends a configuration update to that peer. Because the MP2PSIP project is not designed as a productive system and overlay peers will never run with an out of data configuration the lack of ConfigUpdate is not critical. The Stat request is used to retrieve the meta informations (stored Kinds, lifetime, etc.) at a specific resource Id. Although the RELOAD specification obliges the Stat request message it is not a critical for demonstration proposes. That also applies on the lack of a Find request. It is used to discover the resource Ids for Kinds the requester is interested in.

Although the MP2PSIP project does not implement the entire RELOAD protocol it is a good candidate for further implementations. By extending the stack with with additional features it could be the base further implementation of other Usages like the DisCo and Share [2, 14]. In the following sections, this document will propose several concepts for extending and fixing the MP2PSIP project.

3 Design to Enhance the MP2PSIP RELOAD Stack

3.1 Overview

The design objective for enhancing the functionalities of the MP2PSIP RELOAD stack is to enable a seamless implementation of further Usage requirements. This can be achieved by setting an abstract Usage layer on top of the stack architecture and a re-implementation of the components that were exclusively designed for a SIP usage. An adapted message component must be able to create, store, and fetch requests that contain any kind of application data. A on top Usage interface must enable a seamless implementation of new Usages without the need to adapt the lower protocol logic. The storage of data values must be migrated into a separate module and has to be enabled for the storage of data from various applications. By looking onto future implementations of the Usages for Shared Resources [14] and Distributed Conferencing [2] that use a non-standard access control policy the enhanced MP2PSIP RELOAD stack should be enabled to load and enforce Usage-defined access control rules.

3.2 Message Component

A fundamental problem of the MP2PSIP RELOAD stack to enable the implementation of further RELOAD Usages is the realization of the message transport component. The classes that represent the store and fetch operation just allow the transport of SipRegistration instances. Hence, new request definitions must be enabled to transport data of further RELOAD Usages and should be conform to the RELOAD base protocol [1]. The re-implementation will entail a complete re-implementation of all components that are related the RELOAD message transportation and the *Machine* component.

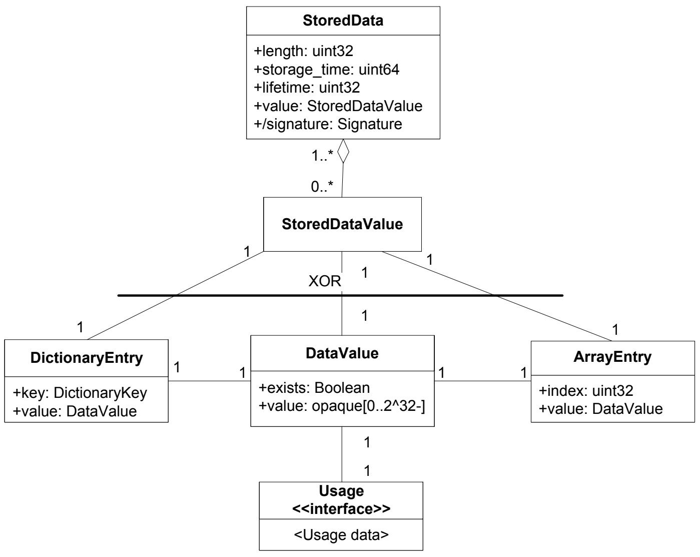


Figure 5: StoredData and StoredDataValue

StoredData structure

The new stack design needs to support the data structures that are defined in the RELOAD base specification [1]. In RELOAD an overlay resource is a container for several RELOAD Kinds stored under a common resource id. Each Kind belongs to one or more RELOAD Usages that provide an interface for the upper application using RELOAD. The key to realize this data hierarchy is the implementation of the STOREDDATA structure and its sub-structures as displayed in figure 5. It carries meta informations like storage time and lifetime of the stored value, but also contains the more concrete STOREDDATAVALUE structure. As proposed in the RELOAD specification, each stored data contains a signature element that enables a storing peer to validate the integrity and provenance of the data. To support all specified data models a stored data value can be either a dictionary entry, an array entry or just a data value object. Note, that the former two are also including a data value object but extend it with the either an index or a dictionary key value.

The StoredData struct will be the new base for message transport as well as for data storage. The data for each Usage will be encapsulated within the DATAVALUE. In RELOAD, the value should be encoded as an opaque string. The new stack design will keep this prerequisite while serializing a message body, but for an easier development, a DataValue object will carry an

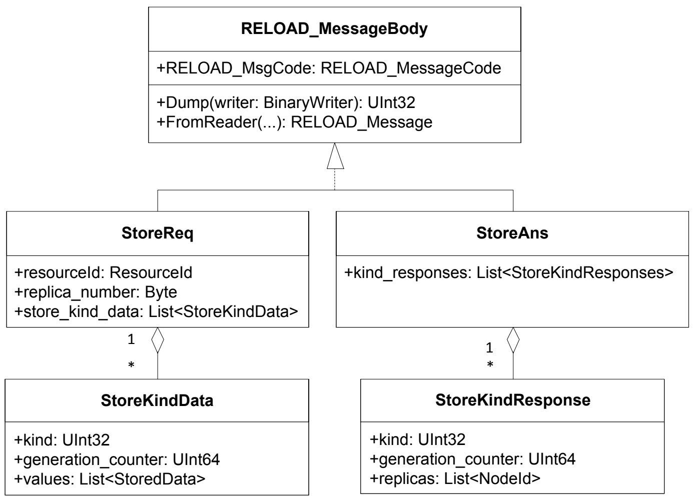

RELOAD_MessageBody
+RELOAD_MsgCode: RELOAD_MessageCode
+Dump(writer: BinaryWriter): UInt32
+FromReader(...): RELOAD_Message
StoreReq
+resourceId: ResourceId
+replica_number: Byte
+store_kind_data: List<StoreKindData>
StoreAns
+kind_responses: List<StoreKindResponses>
1
*
StoreKindData
+kind: UInt32
+generation_counter: UInt64
+values: List<StoredData>
1
*
StoreKindResponse
+kind: UInt32
+generation_counter: UInt64
+replicas: List<NodeId>

Figure 6: Store request / answer messages

instance of a class that implements a USAGE interface. More details about the realization of Usages are described in section 3.3.

New StoreReq/Ans

In the current design, the message definitions for the store request and the store answer are implemented as a single class carrying a SipRegistration instance. By looking into details, it can be seen that a store answer will be serialized as an empty message body. Just the RELOAD message code is set to a StoreAns message. The new message definition for storage foresees two classes instead of one in the current design as shown in figure 6. Both classes will further be realizing the abstract RELOAD_MessageBody class to be compatible to the remaining stack. The new StoreReq message will have the three attributes *Resource ID*, *Replica_number* and a list over STOREKINDDATA objects. The latter attribute is a meta structure used to store multiple STOREDDATA objects of a single RELOAD Kind. Hence, in the new design a single store request can contain multiple data values of several RELOAD Kinds.

The corresponding StoreAns message will be more than just an empty message body. It will contain a list of STOREKINDRESPONSE objects each acknowledging the corresponding StoreKindData of the initial store request. It further notifies the requester where the replicas of the data values have been stored in the overlay.

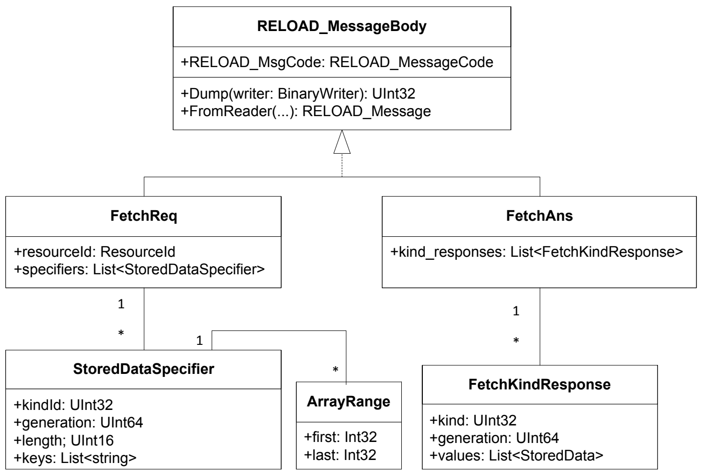


Figure 7: Fetch request and answer messages

New FetchReq/Ans

The design for the new FetchReq and corresponding answer message are also following the RELOAD base specifications as shown in figure 7. A FetchReq contains the resource ID been requested and a list of StoredDataSpecifiers. Dependent on the data model of the requested RELOAD Kind such data a specifier is a query for a list of specific dictionary keys (or all), one or more ranges of an array or just for the Kind ID if the data model is *single value*. The correspondingFetchAns message then contains a list of FetchKindResponse objects for each requested Kind. Each is carrying the result set in form of a list of stored data objects.

3.3 Generic Usage Interface

The objective while extending the RELOAD stack is to provide an abstract messaging and storage service that can be used by a variety of applications. To enable the extensibility of the MP2PSIP RELOAD stack for further Usage implementations, the new design adds a new Usage layer on top of the stack architecture. It mainly consists of two components, the Usage interface and the Usage manager as displayed in figure 8. The Usage interface specifies several operations that are required by the stack to handle a concrete Usage. For example, a Usage instance must know how to serialize and deserialize its Kind data or which application procedure must be performed after successful retrieval of the stored data value. In addition to the RELOAD specification, each instance of Usage must specify a UsageCodePoint value

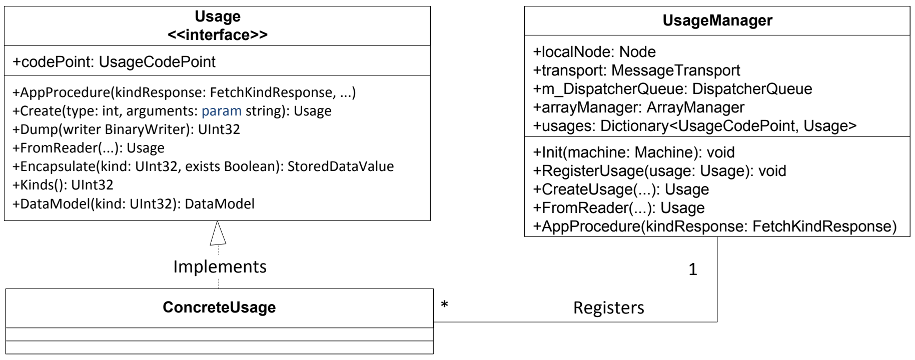

Usage
<<interface>>
+codePoint: UsageCodePoint
+AppProcedure(kindResponse: FetchKindResponse, ...)
+Create(type: int, arguments: param string): Usage
+Dump(writer BinaryWriter): UInt32
+FromReader(...): Usage
+Encapsulate(kind: UInt32, exists Boolean): StoredDataValue
+Kinds(): UInt32
+DataModel(kind: UInt32): DataModel
UsageManager
+localNode: Node
+transport: MessageTransport
+m_DispatcherQueue: DispatcherQueue
+arrayManager: ArrayManager
+usages: Dictionary<UsageCodePoint, Usage>
+Init(machine: Machine): void
+RegisterUsage(usage: Usage): void
+CreateUsage(...): Usage
+FromReader(...): Usage
+AppProcedure(kindResponse: FetchKindResponse)
Implements
ConcreteUsage
*
Registers
1

Figure 8: Usage manager- a factory pattern

that is unique within the stack implementation. It will be used by the USAGEMANAGER class to identify each Usage. It is needed because an identification by the Kind Id might not be unique at all times, since a RELOAD Kind can be reused by several Usages.

The Usage manager class is designed as a *factory pattern*. It is a producer and consumer, respectively, interpreter of instances implementing the Usage interface. Therefore, it provides the two methods *CreateUsage()* and *FromReader()*. While the former one is used to create new Usage instances from the upper application layer, does *FromReader()* select the correct *Usage.FromReader()* method on incoming RELOAD requests. The lower message transport component must not be aware of the concrete deserialization procedure for a incoming Usage data. Rather, it just delegates this problem to the Usage manger class. The concrete implementations of Usage are registered at instantiation of the RELOAD stack. Therefore, its *RegisterUsage()* method takes an instance of Usage as argument to obtain a prototype implementation. Those are stored in a dictionary in which the key is the Usage code point. The registration mechanism could also be used to re-register new Usages at runtime using dynamic class loading functionalities of the .Net framework.

3.4 Abstract Storage

The storage of data values must be handled transparent to the concrete RELOAD Kind that will be stored. As described in section 2.3 the current stack design is limited to the storage of SipRegistration objects and is implemented within the topology-plugin. The new design foresees a separate STORAGEMODULE that is a manager of multiple RESOURCE objects as shown in figure 9. The storage module provides a dictionary of Resource object using the resource id as dictionary keys. The storage module mainly provides the two base operations of

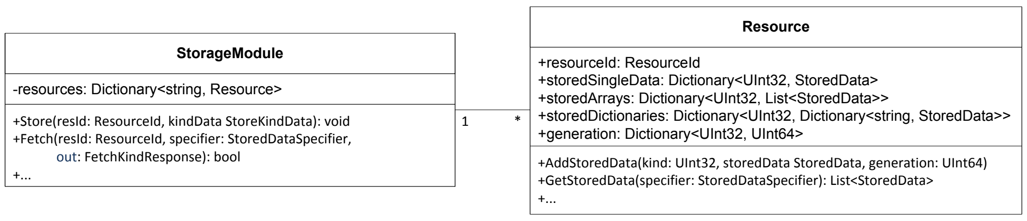

StorageModule
-resources: Dictionary<string, Resource>
+Store(resId: ResourceId, kindData StoreKindData): void
+Fetch(resId: ResourceId, specifier: StoredDataSpecifier, out: FetchKindResponse): bool
+...
1
*
Resource
+resourceId: ResourceId
+storedSingleData: Dictionary<UInt32, StoredData>
+storedArrays: Dictionary<UInt32, List<StoredData>>
+storedDictionaries: Dictionary<UInt32, Dictionary<string, StoredData>>
+generation: Dictionary<UInt32, UInt64>
+AddStoredData(kind: UInt32, storedData StoredData, generation: UInt64)
+GetStoredData(specifier: StoredDataSpecifier): List<StoredData>
+...

Figure 9: Storage module carrying Resource objects

of storing new resources by processing the incoming STOREKINDDATA objects and returning the stored data on receiving a fetch request. The RELOAD storage concept foresees the storage of multiple different Kinds at a single resource ID. This demand will be realized by definition of the Resource class that provides a separate dictionary for each RELOAD *data model*. The keys to these Resource objects values are UInt32 values representing the Kind ID. The dictionaries that are storing values of the array data model are themselves carrying a lists of STOREDDATA objects. For values that are stored as dictionary entries, the corresponding container is storing a separate dictionary for each RELOAD Kind. To maintain the generation counter for each stored RELOAD Kind each Resource object provides an additional dictionary storing the generation counter.

3.5 Access Control Policies

The RELOAD base protocol [1] defines a set of rules that control the permission to write at a specific overlay location called *Access control policies*. New Usages may define new access control policies for their Kind definitions. Hence, the design for the implementation of those policies should generic and expendable. Similar to the design for the Usages 8 will the proposed access control component use a kind of a *factory pattern* as shown in figure 10. An *AccessController* class maintains a set of classes implementing an *AccessControlPolicy* interface. By contrast to the Usage manager does the corresponding ACCESSCONTROLLER not create instances of ACCESSCONTROLPOLICY objects. Instead, it chooses which policy must be applied and engages the concrete policy object on an inbound store request. It will just return a boolean value indicating whether the requester is permitted to store at this peer and whether the carried data values need to be stored by this peer.

The logic of Access control policy instance can be implemented in two variants. First, as C# code or, second, as embedded ECMAScript. The former variant can be used to the four access policies standardized in the base draft [1]. The latter one, for additional policies defined by Usages. This approach is useful to implement the recommendation to distribute new access

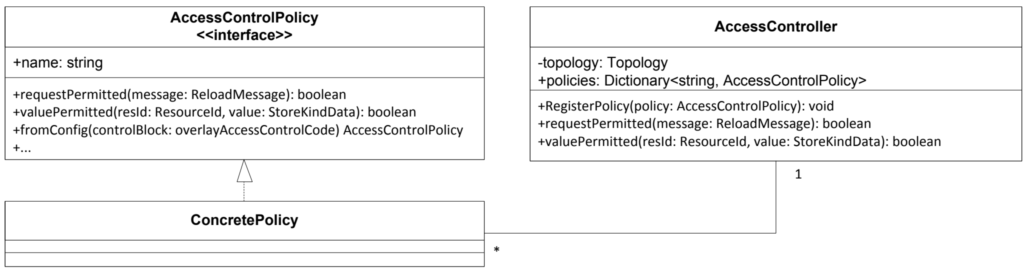

AccessControlPolicy
<<interface>>
+name: string
+requestPermitted(message: ReloadMessage): boolean
+valuePermitted(resId: ResourceId, value: StoreKindData): boolean
+fromConfig(controlBlock: overlayAccessControlCode) AccessControlPolicy
+...
AccessController
-topology: Topology
+policies: Dictionary<string, AccessControlPolicy>
+RegisterPolicy(policy: AccessControlPolicy): void
+requestPermitted(message: ReloadMessage): boolean
+valuePermitted(resId: ResourceId, value: StoreKindData): boolean
1
ConcretePolicy
*

Figure 10: Access Control Policies and Controller

control policies within the overlay configuration document [15]. While parsing the XML document the additional policies can be registered at the access controller. Several .Net wrapper exists to execute ECMAScript code within a C# program and could be used for this purpose.

4 Conclusion and Outlook

This document analyzed the implementation of a RELOAD stack that might be a candidate for further implementations of the distributed conferencing scheme DisCo [2]. DisCo proposes an alternative for central managed conferences where several participants are maintaining the signaling relations for a multiparty conference. The analyzed MP2PSIP project is designed as a demonstrator for a mobile VoIP application that uses the RELOAD specifications [1] for discovering the contact address of the remote endpoint. The MP2PSIP stack implements many aspects of the RELOAD protocol and provides a messaging and storage service. At second glance, however, the RELOAD stack implementation has several deficits. In its current state, it is not capable to serve any other applications than the SIP Usage [3]. This is caused by the message transport component whose stack operations for storing and fetching are implemented exclusively for a SIP registration usage. Additionally, the MP2PSIP project does not provide a mechanism to implement further Usages.

This document proposed a concept to enhance the functionalities of the MP2PSIP RELOAD stack. The extensions are designed to add an additional Usage layer on top of the stack architecture. A renewed message component will implement the message structures as defined in the RELOAD base draft and allows the transportation of various Usage specific data values. A new Usage interface and a Usage manager component designed as a factory pattern will allow a straightforward implementation of new Usages. A new storage module is a container for *StoredData* objects that carries any Usage specific data structures. Another RELOAD demand in scope of the MP2PSIP extension is the implementation of the access control polices. A new module will handle the provenance and integrity checks of incoming store requests.

The MP2PSIP RELOAD stack could be the base for further implementation of new Usages. The focus is to implement the Usages for Shared Resource (ShaRe) [14] and the DisCo Usage [2]. These Usages will be the base for a P2P conferencing application in a tightly coupled model. It will demonstrate that a P2P conference managed by several independent peers provides a better scalability and resilience than a central managed solution.

References

[1] 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.

[2] 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.

[3] 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.

[4] “Windows Forms.” http://msdn.microsoft.com/de-de/library/dd30h2yb.aspx, 2011.

[5] “Google Maps API.” http://code.google.com/intl/de-DE/apis/maps/, 2011.

[6] I. Stoica, R. Morris, D. Karger, M. F. Kaashoek, and H. Balakrishnan, “Chord: A scalable peer-to-peer lookup service for internet applications,” in *SIGCOMM ’01: Proceedings of the 2001 conference on Applications, technologies, architectures, and protocols for computer communications*, (New York, NY, USA), pp. 149–160, ACM Press, 2001.

[7] “PJSIP Stack.” http://www.pjsip.org/, 2011.

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

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

[10] T. Dierks and E. Rescorla, “The Transport Layer Security (TLS) Protocol Version 1.2,” RFC 5246, IETF, August 2008.

[11] “CSharp Coding Style Guide.” http://www.csharpfriends.com/articles/-getarticle.aspx?articleid=336#2, 2005.

[12] J. Rosenberg, “Interactive Connectivity Establishment (ICE): A Protocol for Network Address Translator (NAT) Traversal for Offer/Answer Protocols,” RFC 5245, IETF, April 2010.

[13] R. Mahy, P. Matthews, and J. Rosenberg, “Traversal Using Relays around NAT (TURN): Relay Extensions to Session Traversal Utilities for NAT (STUN),” RFC 5766, IETF, April 2010.

[14] 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.

[15] 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.

[16] 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.

List of Figures

Listings