Hochschule für Angewandte Wissenschaften Hamburg
Hamburg University of Applied Sciences

Masterarbeit

Alexander Knauf

DisCo: A Protocol Scheme for Distributed Conference Control in P2PSIP based on RELOAD

Fakultät Technik und Informatik
Department Informatik

Faculty of Engineering and Computer Science
Department of Computer Science

Alexander Knauf

DisCo: A Protocol Scheme for Distributed Conference Control in P2PSIP based on RELOAD

Masterarbeit eingereicht im Rahmen der Masterprüfung
im Studiengang Informatik
am Department Informatik
der Fakultät Technik und Informatik
der Hochschule für Angewandte Wissenschaften Hamburg

Betreuender Prüfer : Prof. Dr. Thomas C. Schmidt
Zweitgutachter : Prof. Dr. Franz Korf

Abgegeben am 16. Januar 2012

Alexander Knauf

Thema der Ausarbeitung

DisCo: A Protocol Scheme for Distributed Conference Control in P2PSIP based on RELOAD

Stichworte

Peer-to-Peer, SIP, P2PSIP, RELOAD, Konferenzmanagement, Verteilte Systeme

Kurzzusammenfassung

Eine verbreitete Architektur für Multimedia Gruppenkonferenzen mit dem Session Initiation Protokoll ist das klassische Client/Server Modell, in dem ein zentraler und dedizierter Server die Signalisierung und Medienverteilung für die Teilnehmer übernimmt. Diese Arbeit stellt einen alternativen Ansatz für verteilt kontrollierte Konferenzen vor. Mehrere, von einander unabhängige, Endgeräte übernehmen gemeinsam und adaptiv zur Größe der Konferenz die Kontrolle, um eine bessere Lastverteilung zu erreichen und die Ausfallsicherheit zu erhöhen. Die Registrierung und der initiale Verbindungsaufbau zu einer verteilen Konferenz wird dabei über ein RELOAD P2PSIP Overlay vollzogen. Die Registrierungen sind mit relativen Positionsdaten der Konferenzmanager angereichert, um die Gesamttopologie der Konferenz zu optimieren und somit kürzere Latenzen und weniger Jitter zwischen den Teilnehmer zu erzielen.

Alexander Knauf

Title of the paper

DisCo: A Protocol Scheme for Distributed Conference Control in P2PSIP based on RELOAD

Keywords

Peer-to-Peer, SIP, P2PSIP, RELOAD, Tightly coupled conferences, Distributed Systems

Abstract

A widely deployed architecture for group conferences with multimedia transmissions is the traditional Client/Server model. A central and often dedicated Sever maintains signaling and media distribution among the conference parties. This work presents an alternative approach for distributed conference control. Several independent entities are controlling a single conference in a self-adaptive and cooperative fashion. This enables load distribution among the controlling entities and enhances the robustness by the lack of a single point of failure. The registration of the conference identifier and the connection establishment to a conference are achieved by a RELOAD P2PSIP overlay. The registrations are augmented with topological descriptors thus optimizing the entire conference topology with respect on minimizing delay and jitter.

Contents

1. Introduction

1.1. Motivation

Voice over IP refers to a technology to transmit voice over computer networks, which is nowadays highly relevant for human communication. Even though the Public Switched Telephone Networks (PSTN) are still a key component for regular telephone calls, Voice over IP (VoIP) solutions are gaining importance end-user communication. Nowadays, there exist a variety of different VoIP applications and protocols. On the one hand, Instant Messaging (IM) services that are developed and provided by a single company and are generally based on proprietary protocols. IM services are very popular since they have often an intuitive user interface, provide a present service announcing the currently available contacts and are in particular free of charge. On the other hand, exists VoIP services based on open and standardized protocols that are not particularly related to any specific company. They are characterized by a high interoperability among different service providers and are deployed in a variety of softphone applications or on special VoIP telephones. The advantages of these communication services are additional functionalities, e.g., third party call control (3pcc) or multiparty conferences and save expenses compared to the charges of PSTN telephony.

A widely deployed standard protocol for creating VoIP call is the Session Initiation Protocol (SIP) [1]. SIP is an application layer protocol used for signaling among the communication endpoints to create, modify and terminate calls. SIP was initially designed as a peer-to-peer protocol to create a session among end-user devices. However, the deployment model of SIP is provider centric and binds a SIP user agent to a host that maintains a dedicated infrastructure of SIP registrars and proxy servers. Registrars are needed to bind the SIP address of an user to its current IP address while proxies are used to located the remote endpoint. To serve multiparty conferences, a provider additionally maintains a dedicated conferencing server. It acts as central point of control for a conference and often provides module to mix and distribute the conference media among the parties. Since the maintenance of infrastructure produces ongoing costs, its use often requires customer charges. In particular, multiparty video conferences are nowadays only established as business application due to their costs. Further, the dependency on a dedicated infrastructure makes the service architecture vulnerable to system failures and threatens the continuous availability of multimedia services. Even though the service provider maintain redundant hardware to enhance the reliability, complete failures occur occasionally and interrupt service. Hence, the dependency on the service provider can be identified as a single point of failure.

To decouple the need of dedicated hardware for VoIP session establishment, several approaches where proposed [2, 3] for a distributed VoIP and IM architecture called Peer-to-Peer SIP (P2PSIP). The basic idea is to combine the high scalability of structured P2P overlay using Distributed Hash Tables (DHT) with IP telephony using SIP. The concept for decentralised

P2P signaling became such a relevance in the SIP community that the Internet Engineering Task Force (IETF) finally founded a new working group charted to develop a P2P protocol for the use of SIP. An emerging standard is the REsource LOcation And Discovery (RELOAD) [4] base protocol. RELOAD is an application layer protocol for P2P signaling providing mechanisms for data storage and lookup, as well as connection establishment among peers. In contrast to other P2P approaches, the RELOAD base protocol provides a security mechanism for central user authentication and data is sent over secure transport protocols. Furthermore, RELOAD explicitly limits the amount of data peers are allowed to store in the overlay to reduce the burden of becoming a peer in the overlay. Generally, overlay members have permissions to store at overlay locations that are related to their public key certificate. The privilege to store data at a specific overlay location, can not be delegated to further users. As a consequence, coordination and rendezvous processes among distributed peers are hardly to implement on RELOAD.

Using SIP session management capabilities and RELOAD as P2P signaling protocol provides a reliable and distributed base for a future VoIP applications. However, both protocols should be augmented with mechanisms for coordination, load balancing and state coherency to enable a distributed management for processes that are traditionally managed by a central component. In particular, multiparty conferencing could be realized in a decentralized fashion to reduce the need on a dedicated conferencing infrastructure.

1.2. Objectives

This work initially discusses the problem space for developing a protocol scheme for distributed conference control. The concepts for distributed conferencing and shared resource in RELOAD are designed as Internet drafts in the IETF and thus need to be compliant and consistent with the protocols it uses. Based on those discussions, requirements can be derived to develop a new protocol standard.

Afterwards, this work develops solutions to enable distributed conference control based on the RELOAD and SIP protocol. Therefore, it discusses the solution space to find out which approaches are best suited to the requirements established previously. This enables the definition of several protocol operations and data structures each for the RELOAD and SIP protocol. These are presented and explained in detail thus to show that chosen mechanism are compliant to the base protocols and to enable the reader to reimplement all protocol operations. Following, the RELOAD operations are implemented as C#/.NET project using a prototype RELOAD stack. Using this implementation, it is shown that the developed protocol mechanism optimize the conference topology and thus reduce the delay to joining a distributed conference.

All these steps contribute to development and deployment of a new protocol standard in the area of P2PSIP. The final task in this progress, is the implementation of all SIP routines which are partly finished within this work.

1.3. Contribution

This thesis presents a distributed and self-organizing protocol scheme to create session based multimedia conferences among peers. This principle for Distributed Conferencing (DisCo) refers to a multiparty conversation in a tightly coupled model that is identified by a unique URI and located at several independent entities. Multiple SIP [1] user agents uniformly control and manage the multiparty conversation. The mapping from a single identifier to several endpoints is achieved through a RELOAD [4] overlay. The DisCo principle requires the definition of two different *usages*. First a *usage* to enable cooperative write access to overlay resources. Second a *usage* for distributed conference control using SIP and RELOAD. The usage for cooperative write access due to the access model in RELOAD that permits overlay members an exclusive write access to few overlay locations related to a user public key certificate. The exclusively write permission can not be shared with further user by following the RELOAD base specification. The second usage for distributed conferencing is needed to map the Id/locator separation and to specify a scheme to manage distributed conferences.

The first *usage* called *Usage for Shared Resources in RELOAD* (ShaRe) [5] defines a generic mechanism that enables overlay users to share their exclusive write access to a specific overlay resource with further users. Write permission is controlled by an Access Control List (ACL) in which the legal owner of an overlay resource explicitly allows further users access. An ACL is maintained by the overlay participant that is responsible for storage of the resource to be shared.

The second *Usage* called *RELOAD Usage for Distributed Conference Control* (DisCo) [6] defines application and signaling procedures that enable distributed conferences based on RELOAD. This includes a SIP protocol scheme for distributed conference control, a new overlay resource and corresponding rules to handle it and a mechanism to provide a coherent conference state among the distributed controller. DisCo further provides a possibility for proximity-aware routing among the conference participants to reduce delay and jitter in the multimedia communications.

1.4. Organization

The remainder of this thesis is organized as follows. Section 2 provides an overview on basic SIP signaling procedures, followed by a description of standard architectures for session based

group conferencing. Further, it gives overview on the emergence of P2PSIP approaches towards the common P2P signaling standard named RELOAD. Section 3 discusses the generic and particular problem space for distributed conferencing with SIP and RELOAD and sets the requirements for the possible solutions. Section 4 addresses the issue of sharing resources in a RELOAD overlay. Therefore, it discusses two concurrent approaches, shows further deployment scenarios for shared resources and specifies a protocol scheme to achieve resource distribution. Section 5 presents the protocol scheme for distributed conference control based on RELOAD. After an overview on the DisCo concepts, it discusses the protocol design and specifies operation to register, join and leave a distributed conference. Further, section 5.4 discusses and explains the mechanisms for backward compatibility to plain RELOAD and SIP clients, followed by the SIP protocol scheme for conference maintenance. Section 6 introduces an event package for distributed conferences, used to synchronize the focus peers and to announce the conference state among the participants. Section 7 given an overview into the implementation of the DisCo protocol while section 8 presents several measurements and evaluations of the used RELOAD stack and DisCo implementations. This work concludes and gives a look on future work in section 9

2. Evolution of P2PSIP

2.1. Traditional SIP Signaling

The area of VoIP is closely related to one of the most successful protocol standards of the early 2000s – the Session Initial Protocol (SIP) [1]. SIP, from its primitives, is used to initiate sessions among applications that intent to communicate. The main contributions to session establishment of SIP are the following:

- Locate remote endpoint
- Contact the endpoint and determine its willingness to initiate a session
- Exchange meta information, e.g., media type and codes
- Modify established sessions
- Terminate existing sessions

SIP is an application layer protocol for signaling and is abstract from session it initiates. The message body of a SIP request generally contains session description information, e.g., the Session Description Protocol (SDP) [7] that provides a representation for media types and session parameters. SIP is a transactional protocol. Each request will be answered with zero or more *provisional* response messages and at least a *final* response. Thereby, each request message uses a *method* to indicate the desired transaction, while each response message contains a status code, indicating the willingness of the remote endpoint to answer the request. In SIP terminology, a *User Agent Client* (UAC) sends request messages to a *User Agent Server* (UAS) responding with answer messages. The user agents act in these logical roles for the duration of the transaction. If a UAC receives a request, it assumes the role of a UAS. A typical scenario for SIP is the call establishment via a SIP INVITE message as shown in the following section.

2.1.1. Call establishment

Figure 1 shows a typical scenario for SIP call establishment. The example shows two users, Alice and Bob, intending to create a VoIP call. The intermediate entities *atalanta.com* and *biloxi.com* are representing SIP *proxies* of the providers of each Alice and Bob. In this scenario it is assumed that Alice knows Bob's Address-of-Record (AoR) – Bob's SIP URI *sip:Bob@biloxi.com*.

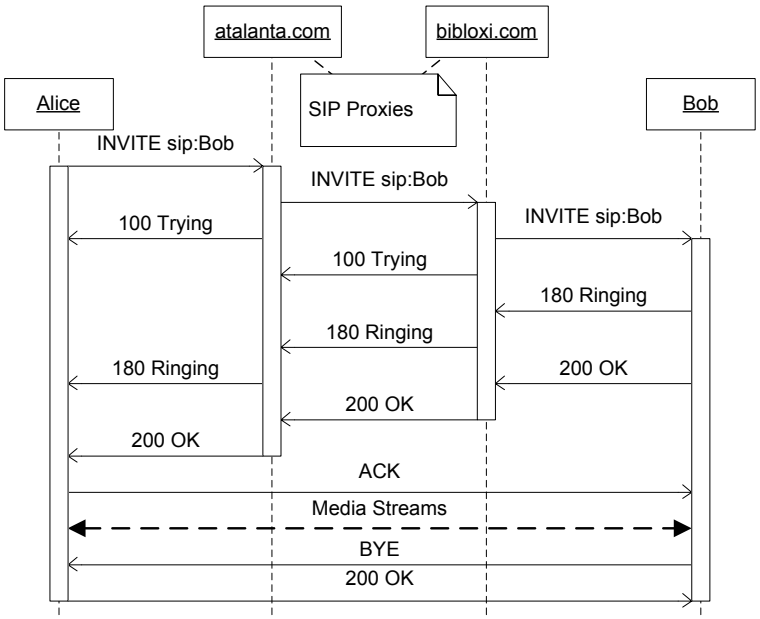


Figure 1: Call flow: Call establishment using SIP

```
1  INVITE sip:bob@bibloxi.com SIP/2.0
2  Call-ID: 0815@141.22.26.6
3  CSeq: 1 INVITE
4  From: "Alice" <sip:alice@atalanta.com>;tag=134652
5  To: "Bob" <sip:bob@biloxi.com>
6  Via: SIP/2.0/UDP 141.22.26.6:5060;branch=z9hG4bKf1
7  Max-Forwards: 70
8  Contact: <sip:alice@141.22.26.6>
9  Content-Length: 159
10 ...
```

Listing 1: SIP INVITE: Alice calls Bob

In order to call Bob, Alice sends a SIP INVITE request message to Bob 1. This initial INVITE contains several SIP header, e.g., *to* and *from* indicating who will be called and who is calling, a unique call identifier, a sequence number and a contact address locating Alice's UA. The session description in the SIP message body, will announce the preferred media types for the desired call. As Alice does not know a route to Bob, the INVITE request will firstly be routed to her *outbound proxy* atalanta.com. The outbound will resolve the host part of Bob's AoR (biloxi.com) by using DNS and forward the request to Bob's *inbound proxy*. Additionally, it will

record its contact into the SIP *Via* header. SIP uses via headers to be enabled to route a corresponding answer along the reserve route of the initial request. In the example, atalanta.com responds with a 100 *Tying* provisional response. It indicates to Alice that the INVITE request was received and that proxy will work on behalf of Alice to route the message to its destination. As the SIP INVITE request reaches the inbound proxy of Bob, the proxy queries its database if a user called Bob is actually registered and forwards the request. Once the INVITE reached the SIP user agent of Bob, it may notify Alice that it is trying to reach Bob (perhaps by a ring tone) by responding a 180 *Ringing* messages. This response will be routed back to Alice along the reverse path of the initial INVITE request. A user agent receiving a 180 *Ringing* will typically play any kind of ringback tone.

If Bob answers the call its SIP user agent will send a 200 *OK* response that will contain Bobs contact address and an agreement on media types for the desired session. Alice's UA confirms the final 200 response by sending an ACK message. This message is send directly to Bob since they had already exchanged their contact addresses and acknowledges Bob the retrieval of 200 *OK*. An ACK message in response to an 200 *OK* that was sent within an INVITE transaction, is considered its own transaction and thus does not required a further response. Finally, Bob and Alice can send the negotiated media and hence performing a VoIP call.

The SIP signaling session can be terminated by sending a BYE request routed directly to the remote UA. By comparing transaction ID the request, the receiver is aware of media session that has to be ended. To conclude the BYE transaction, Alice responds by an 200 *OK* and ends the call.

2.1.2. SIP Notification Mechanism

The Session Initiation Protocol [1] is a flexible protocol that can be extended with further functionalities. Nowadays, there exist about 170 standardization documents in the IETF [8] that are directly related to SIP. There even exists a standard that is a guideline [9] for authors intending to extend SIP. However, this section introduces an early extension to SIP whose mechanisms are used later on.

The SIP Specific Event Notification extension [10] provides a mechanism to subscribe to a resource of interest, e.g., a call state of a remote host, and to receive event notifications if the observed state changes. The state of an observed entity are represented by a so called *event packages* and are commonly defined as an XML document. Figure 2 shows a sample call flow for an event notification for a *registration* event package [11]. An application is interested to receive notifications if a SIP user agent registers its URI at the proxy/registrar server. The application might be service of the provider of that user to welcome users on registration.The application subscribes by sending a SIP SUBSCRIBE request message that contains a *Event* header field whose value contains the literals *reg*, e.g. *"Event: reg"*, which is name

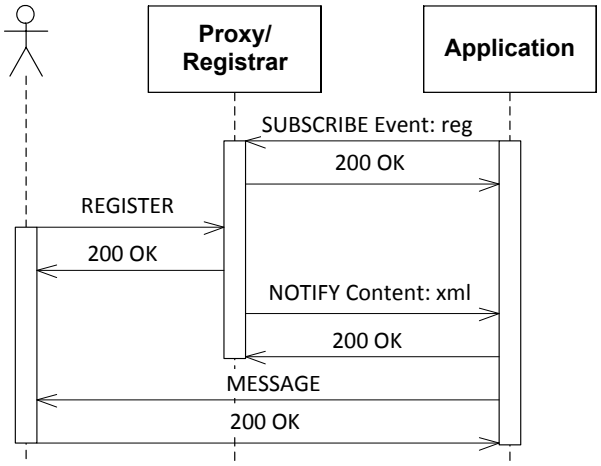


Figure 2: Call flow: Registration event notification

of the event package. The names of event packages are generally registered at the Internet Assigned Numbers Authority (IANA) [12] and hence well-known to applications supporting an event package. Since the registrar server supports the registration event package and the subscribing application is allowed to receive those events, it responds with a final 200 OK response message.

If a user stores its SIP record via an SIP REGISTER request, its registrar will send a SIP NOTIFY request to the subscribed application that contains the event notification represented by an XML document. The XML content than contains information about the registration, e.g., that the registration is active, the duration of registration and contact addresses of the registered user agent. Using the contacts, the service application can finally send any kind of welcome message to the registered user by sending a SIP MESSAGE request whose content is a plain text message.

2.1.3. Call Transfer

In several cases, a user may want another user to contact a third user to transfer a call. For instance, a user Alice is in a call with the user Bob and wants a user Carols to establish a call to Bob. This can be achieved using a SIP REFER request. The REFER method extension provides a *Refer-to* header field that contains an address which the recipient of the REFER should contact. A call transfer implicitly establishes a subscription for a *refer* event package that allows the *referrer* to follow the progress of the transfer. The call transfer from Alice to

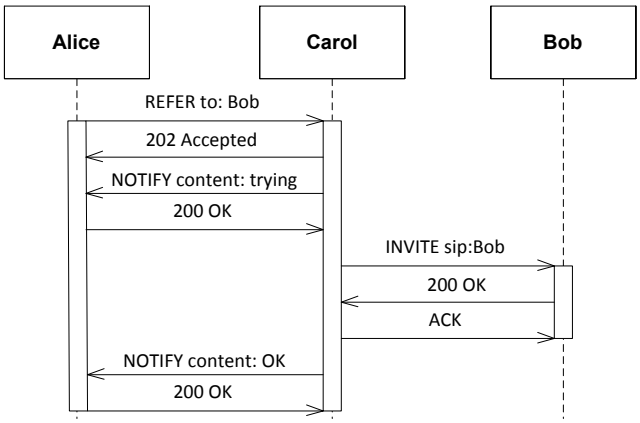


Figure 3: Call flow: Call transfer and refer notification

Bob is shown in figure 3. The REFER request by Alice is responded with a provisional 202 *Accepted* response to acknowledge the retrieval followed by the first event notification for the *refer* subscription. The body of the NOTIFY message contains a *message/sipfrag* [13] content. The *sipfrag* content is similar to the structure of SIP messages, but also allows well-formed subsets of SIP messages. For instance, the content of the first event notification is a *"SIP/2.0 100 Trying"* response-line. Since Carol accepted the reference, she initiates a call to Bob by performing the 3-way-handshake INVITE/OK/ACK. Carol finally notifies Alice about the success of the reference by sending a NOTIFY with the content *"SIP/2.0 200 OK"*.

2.2. Conferencing with SIP

2.2.1. Three-way Conference

A common scenario is a three-way conference in which a user agent maintains two dialogs to other user agents. In this scenario, a pair of users had already an established call and one of them wants to add another party. The user agent capable of media mixing in a 3-party call, re-INVITES its dialog partner to a new session. In this re-INVITE, the conference *focus* may negotiates new media parameter for the 3-way session and add an additional *isfocus* parameter to its *Contact* header, e.g., *"Contact: sip:alice@atlanta.com;isfocus"*. By receiving the INVITE with the *isfocus* parameter, the other party is aware that the user Alice will manage a multiparty conference. After the re-invitation, the conference initiator can invite the third party and will further mix and distribute the media of both endpoints.

In an alternative scenario, a three-way conference can be initiated by a third party joining an established SIP dialog. This can be achieved using a SIP *Join* header extension [14]. Prior

to INVITE request, the third party has to obtain the *call-id* and the values of the *From* and *To* header of the session it intends to join. These may be obtained by some non-SIP source, e.g., web-page, instant message, etc. The third party then sends a SIP INVITE containing the additional *Join* header to one of the parties. An example *Join* header is shown in listing 2.

```
1 Join: 815@atlanta.com
2         ;from-tag=1234
3         ;to-tag=4321
```

Listing 2: Example: SIP Join header to an already establish call

A three-way conference is applied for basic scenarios to establish an ad-hoc multiparty conversation among peers (Note that SIP proxies/registrar might be required at all). However, it represents a centralized topology that does not scale for larger conferences. Furthermore, as the user agent performing the central focus role disappear, the entire conference must be restarted. Hence, it represents a central point of failure.

2.2.2. Conferencing Frameworks

Advanced mechanisms for conferencing with SIP are defined within several frameworks [15, 16]. The Framework for Conferencing with SIP [15] specifies terminologies, architectures and protocol components needed for multiparty conferencing with SIP. It categorizes the following three conferencing models:

Loosely Coupled: In a loosely coupled model, the conference participants provide no signaling relations amongst each other. There is no central point of control and session parameter may be distributed via IP multicast. Loosely coupled models maybe desirable within the boundaries of an IP multicast domain, but are not deployable in a global scale since IP multicast is often filtered at the provider edges.

Fully Distributed: In a fully distributed model, each conference participant maintains a signaling relation to all further conference members. This model may suit small conferences in which the number of signaling messages is negligible. However, a coordination within larger conferences would produce a lot of signaling overhead and its merging would be complex.

Tightly Coupled: In a tightly coupled model, the conference is managed by a single point of control. It provides several conferencing functions, e.g., notification service, negotiates the media sessions among its clients and may perform media mixing functions as well. The tightly could model is widely deployed and extended by several functionalities. However, this model may not suit conferences in an ad-hoc or peer-to-peer scenario since a single peer might not be capable of performing all media functionalities.

A conferencing service as proposed in [16] is more than just SIP signaling. Central conferencing system generally provides five functionalities to participate, control and modify a conference as shown in figure 4 and described as follows:

Conference Object: A conference object is a logical representation of a specific multiparty conversation. It includes a so called *blueprint* that represents a configuration for a conference, e.g., supported media types, maximal number of participants per conference or the availability of floor control. A conference object further contains a conference state representation during an active conference.

Conference Control Server: A conference control server provides an interface to between the conferencing system and the clients. It is used by clients to create a conference by select and modify one of the *blueprints* provided by the conferencing system and to make a reservation for the desired multiparty conversation. Client and Server are communicating via a dedicated Conference Control Protocol (CCP). A conference object created by a user could be cloned or stored to reuse it for a further conference.

Floor Control Server: A floor control server enables authorized conference users (Floor Chairs) to manage access to conference resources, such as conference media, with the conference participants. Floor control is performed by a dedicated floor control protocol, e.g., the Binary Floor Control Protocol [17].

Foci: Foci are the logical conference entities that maintain the SIP signaling relations with the conference participants. Each *focus* maintains the membership operations (join, refer, leave) and the media negotiation for a single *conference object*.

Notification Service: The notification service uses the SIP specific event notification mechanism [10] to inform each subscribed participant about the current conference state. The conference state is generally represented by an XML *event package*, e.g., the Event Package for Conference State [18], which will be introduced in section 2.2.3.

The *Media Mixer* component is not explicitly part of the conferencing framework [16], but naturally part of the conferencing architecture. The media mixer is commonly located on the same device as the conferencing system or on a dedicated media server also known as Multipoint Control Unit (MCU). The media mixing could also be performed by the participants of a conference, but there are actually no standards or best practices along this topic. However, the media distribution is controlled by the *focus* to a conference object that has to ensure that each party is connected to the corresponding media.

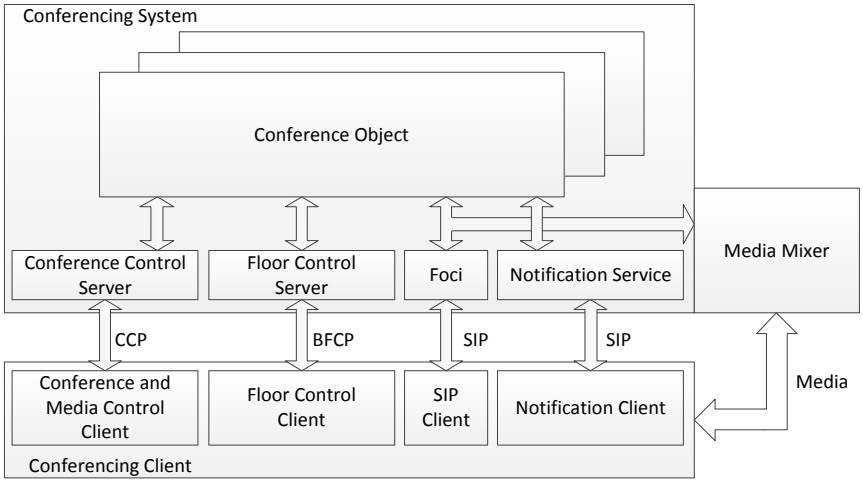


Figure 4: Overview: Centralized conferencing framework

2.2.3. Conference Event Package

In a SIP multiparty session, it is desirable that each conference member obtains a view on the entire state of the multiparty session. The conference status includes information about participants, existing media streams and if auxiliary services are available. A SIP solution to distribute the conference state among the members is a notification service that updates its subscribers about changes in the conference state.

This thesis provides a brief overview to the Event Package for Conference State [18] because its syntax and semantic will be partly reused in an event package for distributed conferences presented in section 6. The conference package enables users to obtain information about changes in state for a specific conference instance. It is build on the SIP specific event notifications [10] (or see section 2.1.2) and defines an XML scheme that represents the state of a actively running conference. Users interested in state updates send a SIP SUBSCRIBE request to the URI of the conference whose *Event* header field requests the *conference* event package, which is the registered package name at the IANA [12]. The subsequent NOTIFY message by the *focus* then contains an XML document in the message body representing the last recent state of conference. The Document Object Model (DOM) tree of the XML has a *conference-info* element as root that has six child elements each describing another aspect of the conference. The root element and its direct children will be described on a high level as following:

Conference-Info: This element is the root of the DOM tree and provides three essential information. First, which conferences state is represented by the XML document, second, an

indication if the XML provides the entire state or a delta to the previous state and, third, a version number to properly order the received notification.

Conference-Description: This child element is a container for several sub-elements that provide general meta-data to a conference. The meta-data comprises descriptions about the conference content (e.g., display-text or subject), URIs to auxiliary services or capacities of the conference (e.g., available media types or maximal user count). The information provided within a conference description is commonly set up before starting a conference by some kind of Conference Control Protocol as described earlier in section 2.2.2.

Host-Info: This child element of the root provides meta-data about the entity hosting the conference such as a display-text or an HTTP URI to an web page providing further information about the conference. The host-info element is also setup previously and the information provided will not change during a conference as long as its remains of the same conference host.

Conference-State: This element comprises three sub-element that enable the conference host to inform its subscribers about changes in the overall state. This includes a user count of all participants and if the conference is currently active or locked in terms of, if a participation request to the conference URI will succeed. In contrast to the previously described element, is conference-state element a more dynamical content that will change more frequently during a conference.

Users: This container element provides a dynamical count of *user* sub-elements, each representing a conference participant. The relationship from a user to its XML element is mapped through the user's SIP URI. Each *user* element provides an attribute that binds it a specific user participating the conference. It is further noticeable that this correlation also represents a user who has multiple devices and signaling relations joined in the conference. Therefore, each *user* element contains one or more *endpoint* element each representing a user's device. However, *user* elements further contain meta-data that describe a user's preferences (e.g., language) or roles in the conference in plain text to be read by humans.

Sidebars: The Sidebar-by-ref and Sidebar-by-value elements enable the XML document to map the relationship of the conference it describes to other conferences. A reference scenario for sidebars is, e.g., a user participating the conference who is simultaneously a *focus* for another conference. This scenario is called a *simplex cascaded conference*. However, as the mechanism for sidebars is not in scope of the later on presented event package for distributed conferences (see section 6).

Partial Notifications A technique that will be referenced within this thesis is the *partial notification* mechanism defined in the conference event package. Partial notifications are useful for XML elements whose children can frequently change during a active conference and to avoid sending the entire XML document at every state delta. These XML elements own two additional attributes. First, the *"state"* attribute and, second, an *element key* also designed as XML attribute. The former *state* attribute defines the three different indications *"full"*, *"partial"* and *deleted*. An element of state *full* indicates that the subsequent DOM tree will contain a complete representation of this aspect of the conference state. Accordingly, an element with state *partial* contains just a subset of elements that are representing the state delta to the previous conference state. An element of state *deleted* does not contain any children and indicates that the entity represented by this element does not exits anymore.

Element keys enable a unique identification of XML elements that have a common parent. This is essential for processing partial event notifications as the receiver is enabled to update only the state delta for the corresponding element in its local copy of the XML state document. For instance, a notification of the disappearance of a user Bob. Bobs state is represented by an XML element whose element key is an *entity* attribute containing Bobs SIP URI. A conference party Alice receives a SIP NOTIFY with the sample XML shown in listing 3. It contains the root *conference-info* element declaring its XML namespace in line 3 and indicates in line 4, which conference is represented by this document identified by the SIP URI *sip:conf-1234@example.com*. The *state* attribute indicates that the subsequent XML document describes a state delta. The only sub-elements needed to announce Bobs disappearance is a *users* element containing just a single *user* element. Alice is now informed that the user Bob is not longer participating the conference as indicated by the *entity* and *state* attributes shown in line 7.

```
1 <?xml version="1.0" encoding="UTF-8"?>
2   <conference-info
3    xmlns="urn:ietf:params:xml:ns:conference-info"
4    entity="sip:conf-1234@example.com"
5    state="partial" version="815">
6     <users>
7       <user entity="sip:bob@example.com" state="deleted"/>
8     </users>
9 </conference-info>
```

Listing 3: Conference-info example: Announcement of the disappearance of the user Bob

The SIP user agent of Alice may use this information to update the user interface to show the user Alice the new conference state.

2.3. Emergence of P2PSIP Approaches

2.3.1. Unstructured P2P Systems

A P2P system is by definition a *"self-organizing system of equal, autonomous entities (peers) which aims for shared usage of distributed resources in a networked environment avoiding central services"* [19]. This citation cut right the chase of matter for P2P networks and further describes a problem of P2P approaches – avoiding central systems.

Early developments of P2P systems, e.g., the SETI@home project [20] or original Napster Music Download service [21] were build around a central component. SETI@home used a central server to distribute recorded radio signals from space among multiple end-user devices to perform calculations. On finishing the calculation, the devices returned their results back to the SETI server. This architecture is in the true sense no peer-to-peer network, but it demonstrates the higher scalability of a distributed system compared with a single server system.

The initial music sharing approach of Napster, came even closer to the definition of a P2P network. Music files were located and shared among the peers within the Napster network. Napster just provided a central indexing server that maintained global knowledge about the location of each file. Peers searching for a piece of music, queries the index server to obtain the address of peers which provide that song. However, this approach violated the definition that a P2P system has to be self-organizing.

Another *pure* P2P System that avoids any central component was the Gnutella protocol [22] since version 0.4. The protocol enabled a fully distributed file-sharing service that used flooding algorithms to find the desired resources. Due to those flooding algorithms, the Gnutella protocol did not scale for very large P2P networks and produced false negative result while searching for desired resources.

2.3.2. Distributed Hash Tables

The key technology that enabled the success of P2P networks was the development of so called *structured P2P networks* that used distributed hash tables (DHT) as routing algorithm. DHT provide the following benefits:

Self-organizing: No central instance is needed for coordination

Scalable: Key-based routing (KBR) algorithms have often a logarithmically complexity

Load-balancing: Keys are distributed uniformly

Failure tolerant: Disappearance of nodes does not collapse the entire network

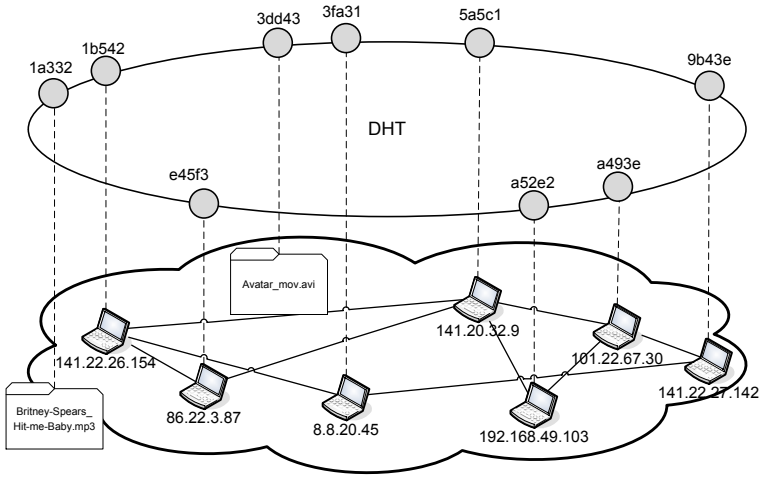


Figure 5: Key-based routing layer: Overlay network upon the IP network

DHTs provide a lookup service that is similar to hash tables that store *(key, value)* pairs. Each node and each data stored within a DHT is assigned a unique key that represents their address in an *overlay* network. The overlay addresses are commonly generated by using a consistent hash functions, e.g., the Secure Hash Algorithm (SHA1) [23]. For instance, a node joining an overlay generates its overlay address by hashing its current IP address and port, e.g., `hash('141.22.26.154:6084') = 0xFA25`– assuming a hash function that produces 16 bit long hashes. Addresses for data are likely generated by, e.g., calculating the hash over the filename. The hash function there ensures a uniform distribution of the overlay address along the entire address range. Hence, nodes and data values share a flat address space that abstracts contact addresses and data addresses. Figure 5 visualizes an overlay network whose KBR algorithm results in a circular address space. On the top, it shows an overlay network that has a 20 bits long address range from $0x00000$ to $0xFFFFF$ that is arranged on a circle. The bottom of the figure shows the connection graph among the devices joining the overlay and several files they provide. Each node in an overlay is thereby responsible to maintain a set data that for which the node is responsible. These responsibilities are generally bound to the overlay addresses, e.g., a node is responsible for data whose address are close to its own. Overlay data can be stored directly on its responsible node or as reference (contact address) to the node it holds. Analogously to data storage, does each node maintain a distributed routing table enabling to locate nodes and data in the overlay. Each node has a partial knowledge on the entire overlay topology. If it knows a direct route to the destination it contacts the remote node. Otherwise a node recursively forwards the messages to a node that is nearest to the hop destination it knowns, according to the used overlay algorithm.

Overlay routing and data storage is based on the distributed hash table algorithm of an overlay. There exist several popular DHTs like Chord, Pastry, CAN or Kademlia [24, 25, 26, 27] which have different features in terms of scalability or resilience against churn and network partitioning. However, this work will further reference on the Chord overlays since it is the default DHT algorithm used for the RELOAD base protocol [4] described in section 2.4.

2.3.3. Motivation for P2PSIP

In the years 2004 to 2005 several new approaches proposed a combination of SIP and structured P2P networks to establish a decentralized signaling topology called P2PSIP. The central SIP proxy[1] architecture is replaced by an structured P2P network [2]. This is mainly motivated by two reasons:

Configuration & Costs: The installation, configuration and maintenance of a dedicated SIP infrastructure is costly and can not be setup ad-hoc. In contrast, a self-organizing P2P does not require dedicated hardware or personnel for installation.

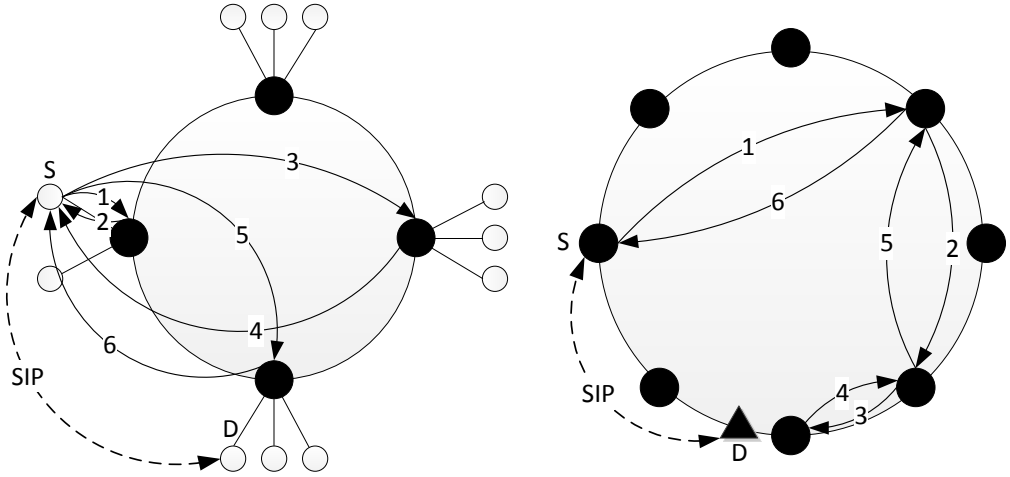
Resilience: SIP proxies are central points of failure while a P2P system can are robust against node failures.

Another disadvantage of the traditional SIP infrastructure identified by [3] is that all traffic, even internal or confidential, traverses through an external third party. In several companies, it is not desirable to route confidential VoIP calls or IM messages via an external server infrastructure, thus many companies have banned those services [3]. While large organizations might be able to maintain their own SIP infrastructure, it could be too costly for smaller companies to provide hardware and staff on their own. Users should be enabled to create VoIP and IM communication with minimal requirements on external hardware and setup times.

2.3.4. SIP over P2P

The first approaches [2, 3] for a combination of SIP and P2P overlays had a common base. They used standard SIP [1] messages which were partially augmented with SIP extension headers [28] as signaling protocol to maintain a structured P2P overlay. This SIP over P2P approach provides the advantage to reuse existing SIP stack implementations and to be compliant to ordinary SIP user agents. Additionally, SIP was already a widely deployed standard protocol that was recognized by firewalls or traffic shapers. New P2P protocols might be detected as unknown and hence filtered by those entities. For instance, each uses SIP REGISTER message to store SIP records in the P2P overlay.

[1] For convenience, we will further refer the proxy/register combination defined in [1] by using the term *proxy*



(a) Approach [2]: Hybrid overlay with iterative routing (b) Approach [3]: Flat overlay with recursive routing

Figure 6: Comparison: Hybrid–Iterate vs. Flat–Recursive routing

Both approaches [2, 3] used the DHT overlay just for resource location. Once the user agents received a contact address by the overlay, SIP session establishment is done directly between endpoints with standard SIP procedures. However, both early P2PSIP concepts prefer different routing strategies and overlay topologies as shown in figure 6.

Figure 6a shows a hybrid overlay approach that uses a iterative routing. Participants in such an overlay are split in in two roles – simple nodes and *super nodes*. Both types of nodes are based in the same P2PSIP implementation and are deployed on end-user devices. Each performs standard SIP registrations [1] and P2PSIP overlay registration simultaneously, while *super nodes* are additionally joined to a Chord DHT [24]. *Super nodes* are selected by their capacities, e.g., public IP address, large bandwidth or continues uptime. Simple peers try to register their SIP records by locating any *super node* and send them their registration message. The latter then use Chord routing procedures to store the record at a *super node* that is responsible for the data. The lookup procedure uses an iterative routing. As shown in 6a a node N send SIP INVITE or MESSAGE request addressed to the destination user agent D to any known *super node* S. The latter queries its local routing table and responds with a redirect message returning the nearest *super node* S' it knows to N. N repeats the procedure by sending the request to S' which may responds with another intermediate *super node* S''. Since D is registered on S'', the 5th request by N will return a contact address to D. Finally, N and D can establish a session using normal SIP signaling procedures.

In contrast, figure 6b shows the approach of Bryan *et al* [3, 29]. All nodes participating the P2P network are joined to a Chord DHT overlay despite of their capabilities. They likely try

to register through standard SIP and overlay mechanisms. To locate a remote user agent, requests are recursively forwarded to a node in the local routing whose overlay address is the closest to the destination address. The receiver of the request repeats this procedure using its own local routing table and forwards the message. The node that stored the SIP record of the destination node responds by sending the response back to last hop node. The message routes along the reverse path of the request until finally reaching the requesting node that is now enabled to initiate a SIP call to the desired party. A light modified variant of the this so called *recursive overlay response routing* (rorr) is the *recursive direct response routing* (rdrr) procedure. After an overlay request has reaches the node stored the SIP records it could send the response directly back to the requester if the original request contains a contact address.

All routing procedures presented above have their pros and cons. Iterative routing reduces the opportunity for Denial-of-Service (DoS) attacks compared to recursive routing mechanisms. There, an attacker just needs to repeat a lookup query to all nodes in its local routing table to initiate multiple lookup procedures routed over various different routes. Especially in P2P overlays that allow parallel lookups, e.g. Kademlia [27], a DoS attack will easily flood an overlay.

On the other hand, recursive routing algorithms provide an advantage if nodes are located behind a Network Address Translator (NAT). It can be assumed that nodes already joined in an overlay had traversed their NATs (e.g. by STUN [30] or TURN [31]) in order to establish transport connection to the nodes in their routing table. Hence, an overlay query profits from the fact that no additional NATs must be traversed while it is forwarded to its destination. If it a rdrr routed network, the final response might by require an additional traversal. In an iterative route only the initial query profits from an existing connection. All subsequent hops may require methods for reaching a node in private address range and thus have a worse response delay.

2.4. RELOAD – A P2PSIP Application Layer Protocol

2.4.1. A Common Solution

The appearance of several competing P2PSIP approaches [2, 3, 32, 33] and their increasing relevance, resulted in a new P2PSIP Working Group (WG) at the IETF chartered in March 2007 to develop a P2P protocol for the use of SIP. All drafts proposed at the P2PSIP WG had their pros and cons. The approach of using SIP as messaging protocol for a P2P network facilitates the backward compatibility to plain SIP client. A counter-argument is that SIP messages are heavyweight for P2P overlay signaling. Another discussed aspect is the routing mechanism of an P2P overlay network. It is preferable to have a hybrid-overlay or aflat and recursive routing? However, all approaches had a common objective – a new P2PSIP signaling protocol. In October 2008 Bruce Lowekamp, who was one of the authors *dSIP* P2PSIP approach initially presented at the SIPPING WG [34], finally presented the first draft for REsource LOcation And

Discovery (RELOAD) [35] that represents a consistent merge among all proposed drafts for P2PSIP and is today in 2012 almost a Request For Comments (RFC).

2.4.2. Overview on RELOAD

The REsource LOcation And Discovery (RELOAD) [4] is a P2P signaling protocol to discover and locate overlay resources. Currently, the protocol is defined as an Internet draft and still needs some work before is can be published as RFC. However, RELOAD will become a highly relevant signaling protocol. Various Internet drafts in the Real-time Application Area (RAI) are already build on this base document. Figure 7 gives an overview on the entities, roles and several RELOAD overlay operations. RELOAD provides abstract messaging and storage service that is self-organizing and maintained by multiple independent peers forming an overlay network. In contrast to the previous P2PSIP approaches, e.g., dSIP [34] by D. Bryan *et al.* or the Peer-to-Peer Protocol [32] by S. Baset *et al.* that used SIP messages for overlay communication, RELOAD defines a proper messaging model that abstracts from any specific application. This enables various types of applications to use RELOAD as signaling protocol and abstracts from a pure SIP [1] usage.

RELOAD contributes to the existence of several different approaches for P2P routing algorithms by providing a generic routing interface. This enables implementations of RELOAD that use different structured or unstructured P2P algorithms that adopt to the conditions of the application scenario. For instance, in an environment in which peers are frequently joining and leaving the overlay, it can be favorable to use Kademlia [27] as key-based routing layer to prevent partitioning of the overlay topology.

A fundamental problem in P2P systems is the lack of any authentication mechanisms. In contrast to other P2P overlays [24, 25, 26, 27], RELOAD uses a security model for user au-

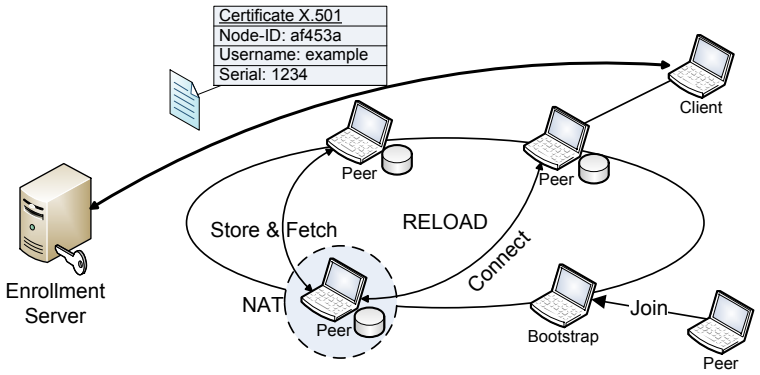


Figure 7: Overview: Roles and services provided by the RELOAD P2P protocol

thentication based on a Central Authority (CA) to initiate a trust delegation mechanism among the peers. Peers establish transport connections to further unknown, even suspect peers. Using a CA issuing public key certificates to overlay participants enables, authentication of peers joining the overlay and a pairwise authentication among peers. It further enables establishment of secure transport connections to prevent unauthorized readings of the overlay messages.

RELOAD is a P2P protocol performed by end-user devices, defines mechanisms for NAT traversal, since many, if not the majority of end-user devices are located in the private address range. NAT traversal is build on the Interactive Connectivity Establishment (ICE) [36] protocol to find candidates for "Traversal Using Relays around NATs" (TURN) [31].

RELOAD specifies two types of participants to meet distinct capabilities of different devices. It defines *peers* as overlay participants that are actively maintain routing and data storing. Peers are generally common desktop devices or even dedicated servers used for bootstrap procedures. So called *clients* can join the overlay network without the burden of data storage or routing. Weak devices like handhelds may enroll as clients instead of peers to save processing power, memory and, especially, saving battery power.

The following sections will explain the main concepts of RELOAD in more detail as they are fundamental for a understanding the approaches presented in this work.

2.4.3. Protocol Architecture

The RELOAD base [4] protocol defines its own architecture as a layered model similar to the TCP/IP Model [37] (aka. DoD Model) defining four abstraction layers for P2P communication. The proposed architecture describing the main components is shown in figure 8. The right column in the figure shows the proposed protocol architecture. The middle column classifies each component to a logical layer in the P2P communication model while the left column refers to Internet layer the components are situated. Starting top down, the following description will briefly introduce the architecture:

Application: This layer contains so called *Usages* that represent an interface to the applications that utilize the RELOAD protocol as abstract messaging and storage service. An application usage defines procedures of how the application will utilize RELOAD services and may define additional data structures (called *Kinds*) that will be stored in the overlay.

Transport: This layer contains three components for transportation of overlay messages. The *message transport* component provides a generic message routing service for the overlay. It is responsible for the end-to-end transactions of messages by monitoring all transaction and, if necessary, to retransmit messages. The *storage component* maintains data values that are retrieved and requested by the overlay network. On receiving a store

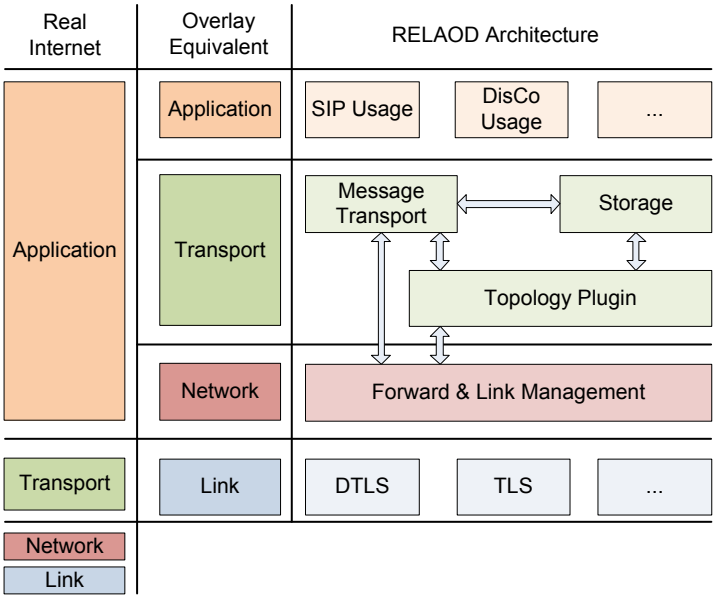


Figure 8: Architecture: RELOAD P2P layer model compared with DoD Internet model

request, the storage validates if the message originator is permitted at this peer by quiring each corresponding usage and the topology plugin. The latter represents is aware of the key-based routing layer of use. On behalf the topology-plugin, the stack implementation is aware how to route overlay messages and is aware of storage responsibilities of a peer. RELOAD specifies the Chord [24] algorithm as default key-based routing layer.

Network: This layer provides the packet forwarding by using the routing rules provided by topology-plugin. It established and maintains all physical transport connections to other peers. Furthermore, this layer implements congestion control to improve the reliability of the end-to-end transaction.

Link: This layer contains the underlying transport used for end-to-end communication. RELOAD defines the use of Transport Layer Security (TLS) and Datagram TLS (DTLS) as default transport protocols.

2.4.4. Usages

A significant features of RELOAD [4] is the possibility to serve as P2P data storage and connection service for a variety of different applications. This is achieved by allowing applications to define *usages* that specify how they use the underlying RELOAD protocol for their service

and specify data values to be stored in the overlay. For instance, the SIP usage [38] defines a data structure for SIP records and describes how the P2P messaging protocol is used to initiate SIP dialogs among endpoints. A usage represents some kind of guide for developers that want to implement an application on top of a RELOAD protocol stack.

2.4.5. Resources and Kinds

RELOAD [4] specifies a hierarchical address space for data stored within the overlay network. It separates overlay *Resources* as logical representation of an overlay location, from application specific *Kinds* providing their own ID. Each resource comprises a set of (*Kinds*) under a common *Resource Name* as identifier. The form of Resource names is specified by the usages and are generally plain text tokens. For instance, the resource name of a SIP record is the Address-of-Record of a overlay user, e.g., *sip:alice@dht.example.com*. The overlay addresses of Resources are created by using a consistent hash function. The overlay address of a resource is a so called *Resource-ID*. The default RELOAD configuration uses Chord as key-based routing thus resource-ids are the result of 128 bit SHA-1 [23] hash over the resource name.

The second depth in the storage hierarchy are *Kinds*. Kinds are generally simple data structures that are defined in C-like syntax. Each Kind within an overlay instance is identified with a unique integer value– the *Kind-ID*. These Kind identifiers can be registered at the IANA as a well-known kind or can be defined ad-hoc within the scope of an overlay instance. A sample Kind data structure is shown by listing 4.

```
1  enum { sip_registration_uri (1), sip_registration_route (2),
2  (255)} SipRegistrationType;
3   select (SipRegistration.type) {
4     case sip_registration_uri:
5       opaque              uri<0..2^16-1>;
6
7     case sip_registration_route:
8       opaque            contact_prefs<0..2^16-1>;
9       Destination       destination_list<0..2^16-1>;
10   } SipRegistrationData;
11   struct {
12      SipRegistrationType   type;
13      uint16                length;
14      SipRegistrationData   data;
15  } SipRegistration;
```

Listing 4: Sample: Definition of the SIP-REGISTRATION Kind [38]

The presented `SipRegistration` [38] structure is defined bottom-up starting at line 17. It It contains the three field: `type`, `length` and the nested struct `data` whose type is defined at line 11. The RELOAD messaging model defines several type primitives, e.g., `uint16` for a unsigned short or `opaque` representing a variable string of bytes. The `opaque <0..2^16-1>` syntax indicates the upper boundary of the data length in bytes and further implies an unsigned short prefix that indicates the length of the succeeding data value. RELOAD further defines complex types. For instance, the `Destination` type in line 10 represents a generic overlay location. `Select` condition are used to enable different variants of a single Kind data structure. The syntax in lines 1-2 shows the definition of the `SipRegistrationType` enum that separates two possible registration variants:

`sipregistration_uri:` In this case, the store data value to another SIP URI referring to the registered user. This registration type is used to delegate calls to further devices.

`sip_registration_route:` In this case, the registered user has registered from its SIP URI to a list of overlay destinations. Generally, those list contain a single destination containing the overlay address (node-id) of the registered user.

2.4.6. Messaging Model

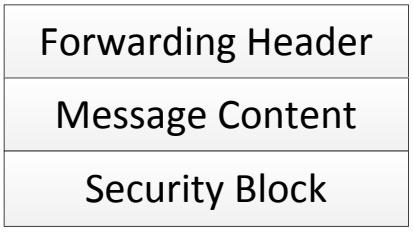


Figure 9: Structure: Composition of a RELOAD message

In contrast to the protocol origins [34, 32], RELOAD [4] provides a proper message protocol designed for P2P signaling. A RELOAD message is a composition of three message segments as shown by figure 9. The forwarding header[2] contains generic routing information that is processed by intermediate peers to forward a overlay message to its destination. The message content segment is used for end-to-end transactions among pairs of peers such as connection establishment and data storage and retrieval. The security block contains the public key certificates and corresponding signatures over the message content. It is used to enable the

[2] A more detailed description of the forwarding header is omitted in this work, since the mechanism for ShaRe and DisCo do not modify or require it.

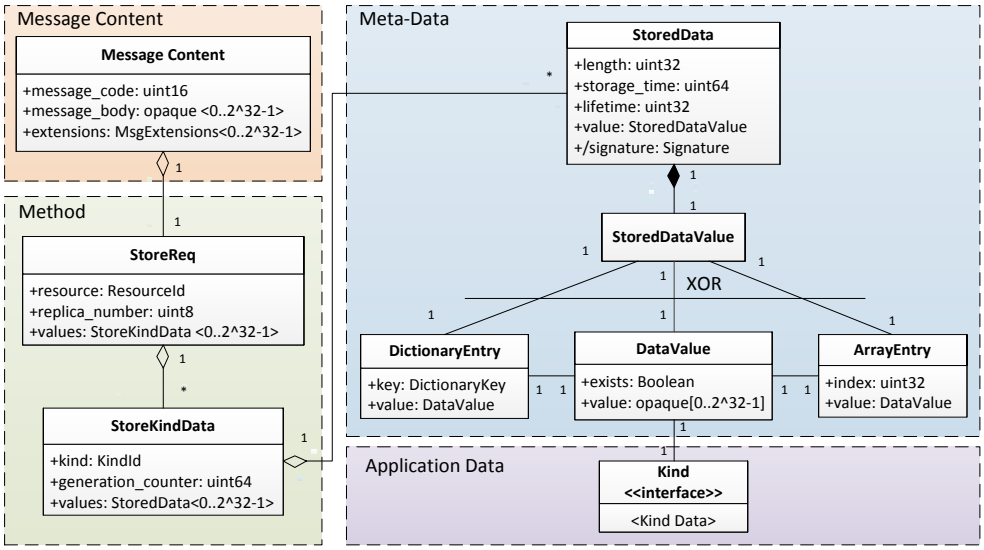


Figure 10: Message body hierarchy: Structure of a RELOAD store request

trust mechanisms in RELOAD to authenticate integrity and provenance of the originator of a message.

The messaging model of RELOAD is transactional. It acknowledges each request with a corresponding answer message or an error response if the transaction causes a failure. Exceptions are made for messages that were retrieved by peers that are not the desired destination. In those cases a message is silently dropped by the receiver to prevent malicious peers from scouting the network. However, the default routing strategy of RELOAD is the recursive overlay response routing (rorr) thus a response is routed along the reverse path of a corresponding request.

Message Content The message content structure is the container for all types of RELOAD messages. It is composed of a preceding `uint16` message code, message body and message extensions at the end of the body to support future protocol enhancements. It serves as header for the succeeding content by indicating message type and length of the rest PDU in an implicitly unsigned integer that precedes the each the message body and extension fields.

Method The content of the message body contains various request methods as described as follows:

Attach: Attach requests are used to establish transport connections to other peers. These are added into the *connection table* of a peer, but not into its routing table. This differentiation

is due to the different roles of overlay members. A RELOAD *client* just attaches its admitting peer to join the overlay. The admitting peer adds the client to its connection table and will forward messages to the client that are addressed to him. Overlay message that should actually be routed by the client according to the KBR, will be routed by the admitting peer.

Update: The Update request is used to update the routing table of a receiving peer. Updates are typically sent during the join procedure of a peer. For joining, a peer initially sends a series of Attach request to those nodes it had to contact with respect to the key-based routing layer. Subsequently, it sends Update requests to indicate that it will take over routing and storage responsibilities. Updates are responded with Update answers that can contain neighbour or routing table entries.

Join: A Join request is used to inform its receiver that the request originator has successfully joined the overlay and takes over the routing and storage responsibilities for a address range that had belonged to the receiver.

Leave: The contrary of join. Indicates that the message originator releases its overlay functions.

Fetch: A Fetch request is used to obtain a overlay resources.

Store: A Store request is used to put data values into the overlay network. Figure 10 shows a Store request that is separated into a `StoreReq` structure that contains multiple nested `StoreKindData`. The each nested `StoreKindData` structure contains the application data to be stored identified by its kind-id. It is thus possible to store several data value of different application within the same request.

Meta-Data The data storage and hence corresponding messages, provide meta-data for each stored value. A `StoredData` structure provides general meta values, e.g., length and lifetime but, however, also provides a signature to validate provenance and integrity for a piece of data. The data can be encapsulated in the three different data models, single value, array or dictionary as indicated by the *XOR* delimiter in figure 10. Accordingly, the structure for array values contains a index and a dictionary provides a dictionary key. The data is actually provided by the `DataValue` structure. A boolean `exists` flag indicates if true, that it contains application data. Otherwise it is an empty object overwriting an existing one. This is the default mechanism to delete values from the overlay. In this way, the meta data to a value remains and any peer receiving this empty object can distinguish that the data was deleted intentionally.

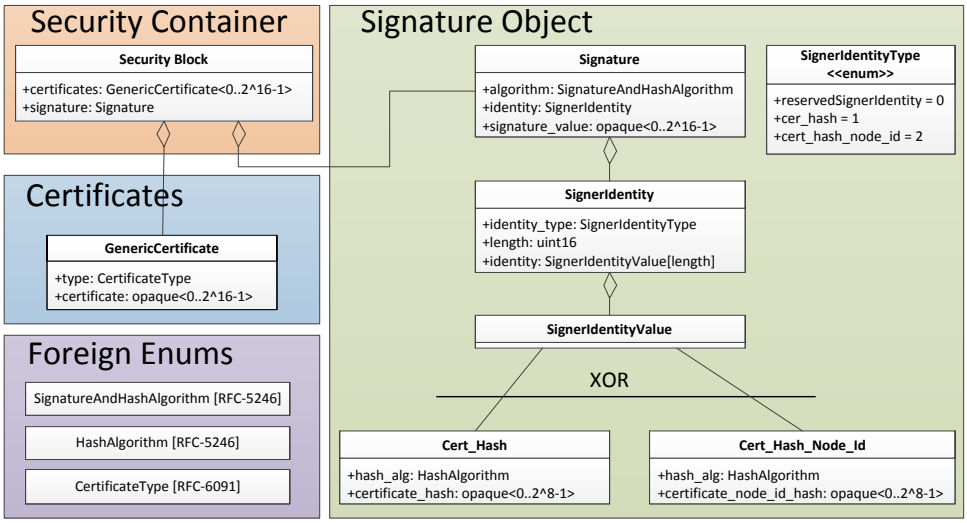


Figure 11: Structure: Security block including certificates and signature

Application Data The structure of the application data is defined by the usages to RELOAD. For instance, the struct of the `SipRegistration` shown in listing 4 would take place in the `opaque` value of the `DataValue`.

Security Block The security block of a RELOAD message comprises several data structure as shown in figure 11. The `Security Block` is a container for a list of certificates and a single `Signature` object. The certificates are represented in a generic certificate structure. Each indicates the type, whose default is X.509 [39] and the certificate as DER [40] encoded string. The certificate type enum is a foreign definition of the Transport Layer Security (TLS) Authentication standard [41].

The signature object is used to sign an entire RELOAD message and further, to sign the message content in store requests and fetch answer messages as shown in the `StoredData` structure of the *Meta-Data* section in figure 10. The signature has three purposes, it indicates the algorithm used to sign the message, refers to the certificate and signer of the data and contains the signature value itself. The indication of used signature algorithm is provided within the `SignatureAndHashAlgorithm` enum. The definition of this enum is made in Transport Layer Security standard [42] and thus not part of the RELOAD specification. However, each RELOAD implementation must at least be able to sign data using RSASSA-PKCS1-v1_5 signature algorithm [43] and hashed with SHA256.

The signer of a message and stored data object is identified by the `certificate_hash` or `certificate_node_id_hash` field that contains the hash value of the certificate that

was used to sign the data. Since each RELOAD message contains all certificates needed to verify the message or data, the receiver calculates the hash over each certificate to identify the certificate which was used to sign data. In this way, a receiver can verify all data structures contained within a request, even if the request was not originated by creator of the data. For instance, a store request sent to replicate data values contains the certificate of the replicator and the certificate of the peer that originated the data values. In this case, the `SignatureIdentity` of the security block in store request contains the hash value of certificate of replicator. The stored data object in the replica however, contains the hash over certificate of the originator of the data value. Thus a receiver can validate each signature by using the corresponding certificate.

The input values for the signature are each different of signing the entire message or a stored data value:

Message: The signature input for a message are the overlay-id, message transaction-id, entire message content and the signature identity in a continues string representation.

Stored Data: The signature input for a store data are the resource-id, kind-id, storage time and the store data value and signer identity as continues opaque string.

2.4.7. Enrollment & Security Model

P2P networks do generally not foresee any kind of authentication or security mechanisms. On the contrary, the popularity of P2P file sharing applications like eMule [44] or KaZaA [45] was driven by the apparently anonymity of users. In addition, such networks are known to distribute maleware. The RELOAD base protocol [4] is designed as a *serious* P2P network whose primary usage is for telecommunication. Hence, a requirement for a P2PSIP protocol is an unambiguous assignment of individual entities to establish transmission dialogs in a deterministic way. If a user Alice wishes to call a user Bob, the underlying protocol must ensure that only Bob retrieves the call.

The security model in RELOAD is based on X.509 [39] public key certificates that are issued by a central authority (CA). Overlay participant are authenticated by an *enrollment server* through their credentials ,e.g., username and password. This binds each overlay instance to an organization providing a minimal of dedicated hardware. This stands in contrast to the global scope P2P file-sharing networks. A sample enrollment and authorization procedure is shown in call flow 12.

The enrollment is initiated by an HTTPS Get request addressed a configuration server. It is generally addressed by an URL that is the concatenation of the overlay provider appended with the path */.well-known/p2psip-enroll*, e.g., *(*https://example.org/.well-known/p2psip-enroll). The *.well-known* segment is part of the *well known URI* framework for Internet services defined

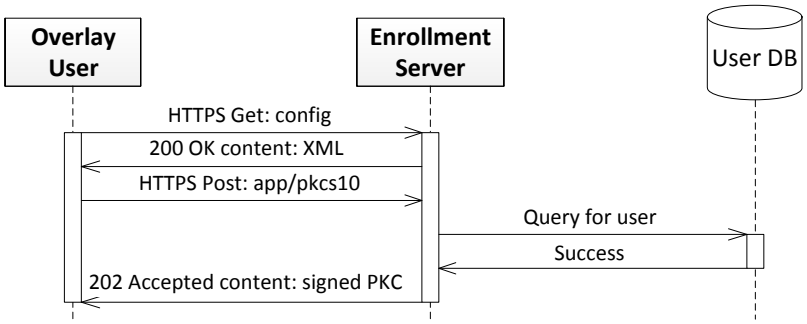


Figure 12: Call flow: RELOAD enrollment procedure

in [46]. The domain of the provider is commonly the domain name of the overlay instance and thus the host part of the Address-of-Records of users. The response by the enrollment server contains an overlay configuration document in XML format. It provides for example a list of bootstrap peers, the address of the enrollment server or a list of required Kinds an implementation must support to join this overlay. Further, the configuration document contains the root public certificate. The root certificate is used enrolling users generate a certificate signing requests to the enrollment server. The request is sent within an HTTPS Post containing the credentials of the user. If the user is recognized, the enrollment server signs the user certificate and issues a node-id within a certificate.

The user certificate allows the overlay participant to authenticate against other peers in the overlay and allows him to storage values at specific overlay addresses as described in the following section 2.4.8.

2.4.8. Access Control

The RELOAD protocol [4] specifies a set of rules that control storage of overlay resources. These so called *Access Control Policies* (ACP) are based on the public key certificate of a user and limit the locations where users are allowed to store data. What ACP for a particular Kind data is used, is defined by the usages. The assignment which ACP applies for what Kind is specified in the XML configuration document of an overlay. Each peer with storage responsibilities in the overlay must enforce the access control policies to ensure the correct storage of data. Access control policies are generally bound to the public key certificates to prove the identity of the originator of the data and to validate whether this particular user is allowed to write at resource-id. ACPs just apply for initial store request by the originator of a data value. If a data will be replicated, the peer receiving the replica must just validate if the replicating peer is a possible source for a replicating store request, concerning the key-based routing layer.

The RELOAD base protocol specifies four default policies and allows the specification of further policies by usages. Two of the default access control policies are the `USER-MATCH` and `USER-NODE-MATCH`. In the `USER-MATCH` ACP, a data value must be stored by the peer responsible for data storage at requested resource-id, if the security block carries a user certificate whose username value hashes to the resource-id. The `USER-NODE-MATCH` is similar, but additionally requires that the key of to a data value stored in a dictionary data model is equal to the node-id of the message originator. The two remaining ACPs have similar rules that reduce the storage permission of a user to resource-ids that are related the username or node-id of a peer. As a result, the overlay locations a user is allowed to write data are quite limited since users are issued few public key certificates by the overlay.

3. Challenges of Distributed Conferencing

3.1. Design Challenges

The protocol scheme specified in this work, distributes the control on a single conference to several independent endpoints. A subset of the entire conference participants manage signaling and media relations among the remaining session members. This distinguishes the DisCo principle from the traditional conferencing frameworks and opens a new problem space for a distributed conference control. This includes general coordination problems, compliance to the used protocol standards and the capabilities of the endpoints that perform the DisCo scheme.

3.1.1. Conference Transparency

The DisCo principle is splitting the identifier of a multiparty session which is in SIP the conference URI to several locators– the conference controller. This raises the question of what view will the conference participants have on the distributed conference. Plain SIP user agent clients cannot handle a SIP URI with several locators without the assistance of a third party (e.g., a *redirect server* or forking proxy [1]) that would violate the transparency while joining a conference. In a redirect sever scenario for distributed conferencing, the UAC would try to invite the conference URI and receive a 300 *Multiple Locations* response containing several contact addresses to the desired destination. Then, the UAC iteratively try to invite each of those addresses.

This is not what DisCo intends. In a distributed conference, a user shall call a conference URI and receive a single destination that will be its conference focus. The participation to a conference should be transparent each to the user and its SIP application. This is actually not possible in SIP and represents the first problem, which must be solved.

3.1.2. Coherency of State in a Distributed Conference

In distributed systems, decisions are often based on incomplete information of the entire state. For instance, in distributed hash tables routing decisions are based on routing tables that contain $log(N)$ of the joined peers, where N is the quantity of all peers in the overlay network. This is intended to reduce the load of routing and storages responsibilities for the peers. In other distributed systems it is desired that all endpoints have the same view on the entire state of the system. This is performed often by replicating the state among all endpoints. The replication is thereby transparent to the consumers of the system that retrieve a single logical date object regardless of how many replicas exist.

Distributed conferences count the latter category of distributed systems. Each controller in a conference is aware of the participants it serves but has no state information of the other conference foci. This incomplete knowledge is insufficient from the view of a conference party. For instance, a minimal requirement would be the information about who is participating the conference or who left the conversation. Hence, a distributed conferencing model must provide mechanisms to synchronize the most recent conference state to all its parties. Since a DisCo is a fully distributed system of independent peers a synchronization mechanism must concern the possibility of race conditions that may produce a incoherent representation of the entire state. Hence, a mechanism for distributing a state representation must provide its participants a consistent view on the entire conference state.

3.1.3. Peer Failures

Distributed conferencing is preliminary designed as a protocol performed by peers. Accordingly, the protocol must be adapted to the typical properties of P2P systems. A core problem in P2P networks is the continues appearance and disappearance of the nodes joining the network. This behaviour called *churn* must be considered in a distributed conference architecture. The peers actively managing the multiparty session are not required to maintain the conference service as they intent to leave. Furthermore, must a DisCo protocol specify restructuring mechanisms if peers disappears from a conference unexpectedly. Such failover procedures enhance the reliability of a distributed conference adapting to the effects of churn.

3.1.4. Load balancing

The protocol design assumes that several members of a multiparty session allocate their device capabilities in favour of maintaining the conference. It should be considered, however, that the end-user devices are not overloaded while performing the conference functions. The load caused, e.g., by mixing mixing and distributing the media streams should be evenly distributed among the participating endpoints. An approach for distributed conferencing should define mechanism to estimate to load of its controlling entities and to balance it. Joining conference parties should select a less loaded conference manager or the latter should be enabled to forward incoming calls to further managers before overloading.

3.1.5. DisCo in P2PSIP

The DisCo protocol presented in this work is based on the emerging protocol stand RELOAD [4]. RELOAD is used to announce the conference identifier in a P2PSIP overlay to avoid the need for dedicated SIP register and proxy servers. All peers controlling the conference should

be registered on a single record to announce their function as focus to joining peers. However, the access model of RELOAD has two incapabilities to this conference registration scheme:

- Users have exclusive access permission to overlay resources that cannot be shared
- Users can only access overlay resources that correspond to their public key certificate

The first limitation hampers the registration of several focus peers in a single record. This a problem to realize a distributed conference control in particular, and further, for any rendezvous procedures that need a common resource for coordination in RELOAD.

The second limitation represents a problem for the creation of individual conference identifier. It is be desirable that a multiparty session is identified by a descriptive Address-of-Record. A discussion about cats and dogs should indicate the topic within its conference URI, e.g., *alice-discussion-on-pets@example.org*. By applying the access control policies of RELOAD, it is impossible for a user to store any kind of Resource under another name.

3.1.6. Backward Compatibility

A general issue in network communication and specially in P2P networks is the heterogeneity of applications that might implement different extensions to a standard. In a RELOAD overlay, all parties are implementing at least the RELOAD base specification [4] to provide and maintain the overlay messaging and storage service. Apart from this, overlay clients and peers may implement different RELOAD extensions (e.g. the Direct Response and Relay Routing Extension [47]) or application usages and Kind data structures. The RELOAD protocol challenges the heterogeneity of implementation by the overlay configuration document that specifies the mandatory extensions and the required Kinds application must support. However, the required Kinds do not prohibit implementations to store and fetch data values of non-required Kinds. If a storing peer does not support a Kind of inbound store request, it just returns an error response indicating its incompatibility to the specific Kind.

Another challenge are plain SIP user agents not implementing the RELOAD base protocol. Those implementations should not be limited to join a distributed conference since it uses standard SIP signaling for conference maintenance. The DisCo protocol scheme specifies mechanisms that enable DisCo-unaware applications to join a distributed conference.

3.2. Organization of Focus Peers

3.2.1. Communication Delay

A general challenge in P2P systems are worse latencies compared with a similar centralized system. Especially in telecommunication are the end-to-end delay and jitter a critical issue for the quality of service. Typical delay tolerances recommended by the International Telecommunication Union (ITU) [48] for voice over IP are shown in table 1. It shows the one-way transmission delay to the transmission rating factor (E-Model) [49]. The E-Model is based on a large number of subjective tests to rate the quality of speech in telecommunications. The table shows that the user satisfaction is high as long the one-way delay is lower than ~250ms. By increasing latency, the satisfaction becomes worse. Actually, the ITU recommends to keep the total delay blow 400ms as users notice a delay while speaking.

VoIP Delay/ms	Transmission Rating Factor/R	Quality Category	User satisfaction
0 - ~200	100-90	Best	Users very satisfied
~200 - ~250	90-80	High	Users satisfied
~250 - ~400	80-70	Medium	Some users dissatisfied
~400 - ~550	70-80	Low	Many user dissatisfied
< ~550	60-50	Poor	Nearly all user dissatisfied

Table 1: Recommendation: Mouth to ear delay in telecommunication

A crucial factor for end-to-end delay is the delay for transportation of the data packets through the Internet. In a centralized conferencing scenario the data packages generally flow from the source to the media server and finally to the destination. In a distributed conference, the data packets may traverse several intermediate peers. A distributed conferencing architecture should define mechanisms to minimize the latencies due to IP routing.

3.2.2. Media Capacities

Another factor that should be taken in account while selecting an adequate entry point to a distributed conference are the media capacities of those devices. Each communication endpoint provides a set media types (voice, video) and corresponding codecs (e.g., speex[50], h.264[51]) that it offers the remote endpoint for establishing the media session. The latter responds with an intersection of media codecs he and the offerer are able to de/encode. This kind of offer/answer model [52] based on the Session Description Protocol [7] tries to negotiate common media parameters even though if the result returns the least best intersection.

In a distributed conferencing model both the participants and manager are ordinary home or mobile devices that might not have the same variety on media codecs as provided by a

dedicated server. However, as each device might have different capabilities, a joining user agent could select its entry point to the conference by reference to the media codecs the focus peers provide. Hence, conference parties should be aware of the existing media streams to optimize their media qualities.

3.2.3. Focus behind NATs

A large number of end-user devices are located behind a Network Address Translation (NAT). A focus peer could also be located behind a NAT thus limiting its reachability for users joining the conference. A solution for distributed conference should take NATs in account for the selection of an adequate focus peer.

3.3. Requirements on Distributed Conferencing

Based on the problem space defined in the previous sections, the following requirements for distributed conference control are outlined:

Transparent: The compatibility to established SIP implementations must be provided. The distribution of the conference identifier should be transparent to the clients.

Coherent: The conference state must be coherent on the controlling entities, as well as on the joined parties. A mechanism must be defined to keep the entire conference stat in sync.

Robustness: The conference service must be maintained even if controlling entities leave the multiparty session. This demands mechanisms to compensate the lost of focus peers.

Balanced: The entities providing the conference service should remain in an adequate workload. Mechanism must be defined to detect assisting peers and to transfer calls to them.

Controllable Sharing: The mechanism to share an overlay resource with further peers must be controllable by the resource owner. Any mechanism enabling a shared write access should ensure that only authorized users are permitted to write the shared resource.

Revocable: The shared write permission on an overlay resource must be revocable. This includes a mechanism that is able to revoke access for a single user, group of user or anyone.

Compliant: The mechanism to share resources should be compliant to the RELOAD base protocol. This compatibility should consider the access control concepts, messaging protocol and storage design (e.g., storage of small amount of data).

Responsive: The shared resource should be responsive as long as at least a single user is actively managing it. The lifetime of a data or duration of the shared access should not expire as its creator leaves the group.

Proximity-awareness: The entities intending to participate a distributed conference should aware of their proximity. A mechanism must be define to announce a topological descriptor thus further peers are enabled to make joining decisions on it.

Media-awareness: The entities in a distributed conference should be aware of all provided media type and codecs. These should be used to optimize the conference topology along the media capacities.

NAT-awareness: The entities in a distributed conference should consider the surrounding NAT. Mechanism must be defined for cases in which it is desirable to join a focus behind NAT. Furthermore, conference joining should consider the cases in which it is desirable to invite a farther focus if the optimal focus is behind NAT.

Ad-hoc: Any mechanism allowing variable resource names should enable users to create a new resource name ad-hoc.

Deterministic: Any variable selected resource name should always resolve to its originator. This is need to prevent hijacking of resource names of other users, e.g., preventing a user Chuck to register a variable SIP record *alice@dht.example.com*.

Restrictive/Configurable: Any mechanism for variable resource names should enable the provider of a RELOAD overlay instance to control the amount and form of variable names its clients are permitted to use. This prevents uncontrolled resource name allocations by overlay parties.

4. Shared Resources in RELOAD

4.1. Introduction

A design concept of RELOAD [4] is to limit the amount of data values that peers in a P2PSIP overlay need to maintain. The protocol achieves this constraint by access control policies. These are generally bound to a users public key certificate and permits him to store value at overlay locations that are related to a users Address-of-Record (AoR) or his overlay node-id. This write permission for a user is exclusive and not shareable with other users. However, a distributed conference is a cooperative service managed by several independent entities that need a common overlay resource to announce the conference.

This work presents an approach to share overlay resources with further users and to permit the registration of variable AoRs. The approach is designed as *Usage* for Shared Resources [5] in RELOAD. It enables rendezvous processes, where a single identifier is linked to multiple, dynamic instances of a distributed cooperative service. The approach for shared resources is a generic mechanism, applicable for distributed conferences in particular and for any future RELOAD usage with similar requirements.

The elaborated solution in this section addresses the problem statements for resource sharing in a RELOAD overlay. Therefore, it discusses two alternatives to map distributed write access in P2PSIP with respect to be compliant to the RELOAD protocol. It is also issues revocation of write permissions and determines if a distributed conference persists even if its initiator leaves it. Since distributed write access on a single resource can cause race conditions, a mechanism will be defined to coordinate concurrent write attempts on a shared resource. Further, the following sections show that the approach for shared resources remains controllable and configurable for an overlay provider to reduce the additional load caused by allowing variable resource names.

4.2. Design Pattern for Shared Resources

4.2.1. Self-signed Certificate Chain

Shared resources in RELOAD can be realized by distributing user certificates that are derived from the certificate used to store the registration of a distributed conference. Figure 13 shows the usages of self-signed certificates that authorize conference participants write access to the conference registration.

The creator of a conference intends to register a conference identifier that deviates from its own AoR. To retrieve permission, the creator sends a certificate signing request to the Enrollment

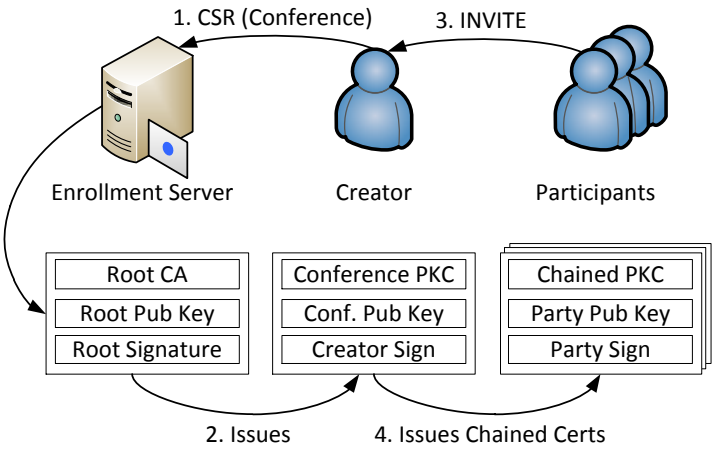


Figure 13: Chained Certificates: Shared write access by self-signed PKCs

server to obtain a new certificate. This conference certificate is used to store the registration for the distributed conference. Afterwards, users can resolve the conference ID using RELOADs lookup mechanism and SIP invite the conference creator. According to a predefined policy, the creator issues self-signed certificates that are chained to the conference certificate to the members of the multiparty session. These are used to authenticate against the overlay peer peer maintaining the registration as permitted user to store an additional mapping to the DisCo-Registration.

A prerequisite to achieve this authentication model, is an additional access control policy to the RELOAD base protocol. The policy must permit write access for users that can present a public key certificate that is derived from the certificate used to sign the registration. In this way, we achieve the two objectives for shared resources in RELOAD. Once, registration of a variable conference identifier and, twice, a shared write access on the overlay resource.

4.2.2. Access Control List

Shared resources in RELOAD can alternatively realized by so called *access control lists* (ACL). Access control lists are stored along with the overlay resources designated for a shared usage. An ACL contains the overlay usernames of the peers that are permitted write access to shared resource. This scenario assumes that creator of the shared resource owns a public key certificate that permits him store data at the overlay location. As shown in figure 14, Alice adds Bob into the access control list to permit him write access to shared resource. The particular overlay data structure that will be shared is referred by its kind-ID. She signs the ACL entry

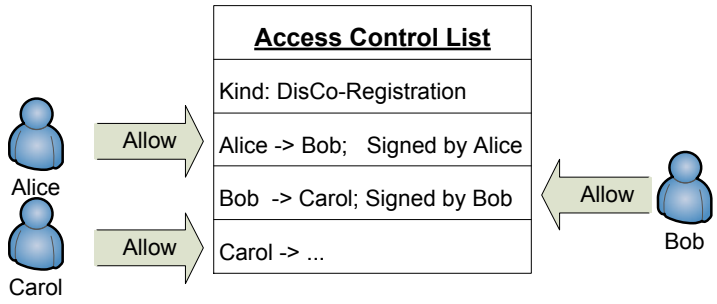


Figure 14: Access control list: Shared write access via list of permitted users

for Bob with her own private key to prevent it from unauthorized modifications. Bob further delegates write permission to Carol by adding a new ACL item to the list.

The access model requires the definition of a new access control policy in RELOAD. The policy must permit write access if a requesting user is registered in the corresponding ACL to the shared resource. Furthermore, the peer maintaining the resource could comprehend the entire trust delegation chain to validate if the creator of the shared resource was authorized to store the resource.

This approach would suite the DisCo requirement to be a shared resource, but offers no solution to register a variable conference identifier.

4.2.3. Resource Name Pattern

Variable conference identifier can be realized by an approach called resource name pattern. The approach makes use of the overlay configuration document to announce regular expressions that specify the form of valid resource names for each RELOAD Kind. In this way, a peer receiving a store request could validate of the chosen resource name for the retrieved data matches to the regular expression defined for this Kind.

This approach needs to prerequisites to enable this pattern matching. First, the plain resource name must be sent in within the store request, since the default RELOAD meta-data omit the transmission of the resource name. Second, a new access control policy that permits the storage of data if the resource name matches the regular expression defined in configuration document.

4.2.4. Comparison and Selection of an adequate Approach

The certificate chain approach covers the objectives of sharing a resource with further user and enables a variable resource naming for the conference identifier. Furthermore, certificate chaining is a common mechanism to delegate trust among third parries. However, two arguments are against the approach for self-signed conference certificates.

Conference certificate: The RELOAD base protocol defines that a certificates can only be issued while enrollment of a peer. Thus, a certificate signing request by a peer already joined to the overlay will be rejected. Without the additional conference certificate, the creator is unable to store a registration at the overlay resource-id that correlates to the conference identifier. The only remaining alternative would be to generate the chained certificates for the conference participants based on the user certificate of the creator. In this case, the requirement of a variable conference ID could be achieved through the approach resource name pattern.

Revocation: The requirements for distributed conferencing specified that any mechanism to share a resource must provide a further mechanism to revoke the shared write access. Using chained certificate, this requirement is hardly to implement. An approach could be to limit the expiration time for a chained certificate to a short value. Through the expiration of the chained certificate and without reissuing a new once, a conference creator can revoke write permissions to specific users. A problem with this approach raises up, if the conference creator leaves the multiparty session and thus will not reissue write permissions. This would violate the requirement of a responsive conference. A distributed conference must continue as long a peers are willing to manage it. An alternative approach to short expiration times is announcement of a revocation list. A revocation list could be publicly stored in the P2PSIP overlay indicating bad peers that are no longer permitted to access the conference resource. However, if the conference creator leave the session or even the RELOAD overlay, no peer would be permitted to further maintain the revocation list.

The alternative access control list a mechanism is build a common practice to control write permission of various operating system. ACLs have in advantage that revocation of write permissions are achieved by deleting the corresponding ACL item in the list. In this way, an authorized user could revoke single items or entire branches of the trust delegation tree. Further, the approach meets the requirements of keeping the conference service responsible in the case that its creator disappears. The requirement to provide a variable conference identifier could be achieved in combination this the approach for resource name pattern.

However, even with access control lists it is hardly to revoke write access for certain users if the conference creator left the overlay. However, the remaining branches of the trust delega-

tion tree are still controlled by authorized peers. This enable a continues maintenance of the conference service without the presence of its creator.

In summary, the approach for chained certificates would suite the requirements of distributed conferencing if an enrollment server would issue conference certificates. Revocation could then be handled by an advanced mechanism using revocation lists. On the other hand, access control list provide a more flexible manner to coordinate and control shared access to an overlay resource. The advantage comes from the generic mechanism, to share any Kind of overlay resource and that authorized peers can maintain the shared resource independently. As result, access control lists are the used to define a RELOAD usage for shared resources.

4.3. Scenarios for Co-Managed Overlay Resources

4.3.1. Third-Party Registration

The Session Initiation Protocol [1] allows to add new bindings between an Address-of-Record and further contact addresses. This enables a third party to register its contact to an existing SIP record and to act on behalf of that user agent if desired. A typical scenario is described by the following example. As shown in figure 15, a secretary accepts calls on behalf of her boss as he is currently not available. The company provides an own SIP server infrastructure and uses IP telephones at all desks. The SIP phone of the boss of the company registers at the server by sending a SIP REGISTER whose *To* and *From* header contain the URI of the boss. The *Contact* header contains the IP address of the VoIP phone. This is a standard procedure to register a SIP address.

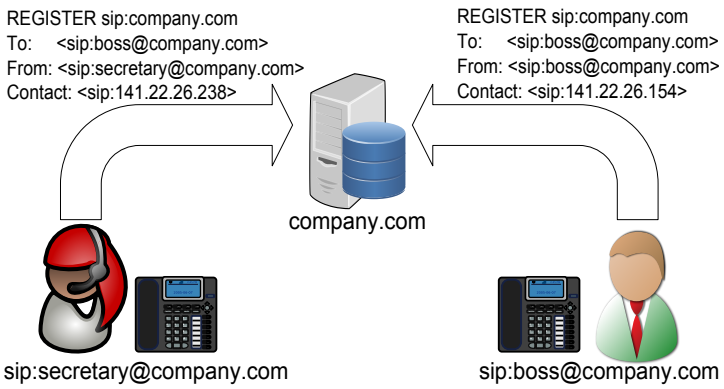


Figure 15: Shared resource scenario: SIP third-party registration

The internal policy in this company allows the secretary to answer calls on behalf of her boss. Hence, her SIP phone is allowed to register as third party by setting the *To* header to the URI

of her boss and the *From* header to her own URI (*sip:secretary@company.com*). Both can now configure whether incoming call will be forwarded to the secretary, her boss or both (forking).

This simple scenario does not work if the company provides a RELOAD overlay for VoIP. In RELOAD, the owner of the public key certificate has an exclusive write access to the corresponding resource. Simply passing the certificate to the secretary does not work either as the secretary (hopefully) not knowns the corresponding private key to sign the message. Concerning the SIP usage for RELOAD [38] the boss could configure a forwarding thus redirecting all calls to his secretary. The problem is he is doing it without the permission of his secretary (she might be gone for a copy job) and that the VoIP phone of the boss is not able to receive calls anymore. Shared overlay resources in RELOAD could map the third-party registration behavior of SIP.

Initially, the boss would register its SIP URI to the node-id of its VoIP/RELOAD phone and afterwards register the username of his secretary in a corresponding access control list. Then, she can additionally register her node-id to the registration of her boss as she is available to answer calls.

4.3.2. Message Board

A native mechanism for group communication are text-based message boards or forums. A group of registered or anonymous users maintain a digital message board to exchange opinions or report bugs and archive them for further discussions. The RELOAD base protocol [4] could be used to create P2P message boards whose content is stored among overlay peers as shown in figure 16. The hierarchical structure of a message board could by mapped on an overlay by storing each thread as a separate resource. The main thread could refer to the subthreads to enable a message board navigation. Each thread could be stored under the topic-text in an URI scheme, e.g., *msg:$user.$topic@$overlay/$sub-thread/$subsub-thread* where `$user` identifies the initiator of the meassage board. The hash over the URI then specifies the resource-id and thus the peer that will store the posts of a thread.

The default access control policies in RELOAD would not allow any kind of distributed message boards. Users would need a separate public key certificate for each thread and sub-thread they want to create following the presented URI scheme. A mechanism for shared resources as presented by this work would allow a distributed write access and variable resource names following the URI scheme.

The URI scheme could be set as resource name pattern for a *message board Kind* in the configuration document. The initiator of a message board would be permitted to store thread in the overlay and, additionally, to store access control lists onto each thread location. The latter than contain the username of the overlay peers that are permitted to add an modify the message board.

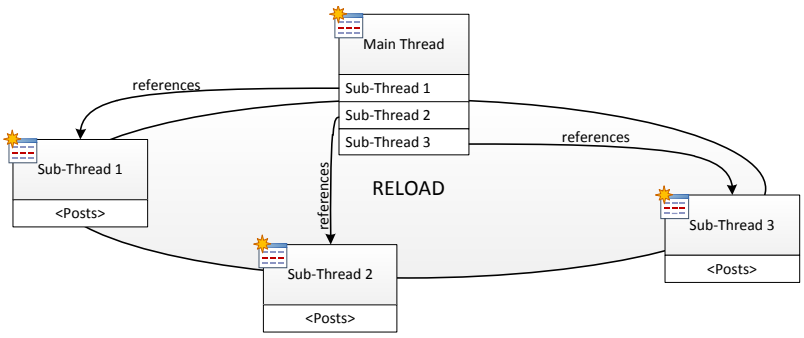


Figure 16: Shared resource scenario: Distributed message board on RELOAD

4.4. Management of Concurrent Write Attempts

The access control list and the resource that is shared by the ACL can be written by several independent peers. This implies possibility that two or more peer can concurrently store or modify those resources. Hence, concurrent store request on a resource demand a coordination procedure to avoid race conditions on the stored data. The coordination procedure defined here, resolves race condition by assigning each authorized peer a separate part of the overlay resource. As described in section 2, a resource comprises several RELOAD Kinds and each Kind can be arranged in the three different data models:

- Single Value
- Dictionary
- Array

A single value cannot be separated into independent pieces and is this by definition not allowed to be used as a shared resource.

If the shared resources uses the dictionary data model, the Kind data will be separated by the dictionary key. Per definition, a peer has exclusive write permission to the dictionary value, whose key is equal to the node-id of the accessing peer. This prevents a dictionary value from concurrent write attempts and keeps the data coherent.

If the data model of the shared resource is an array, the array index is used to separate the array item from concurrent writing. Per definition, array indexes must be generated as described in the following algorithm:

1. Obtain the Node-ID of the certificate that will be used to sign the stored data.
2. Take the least significant 24 bits of that Node-ID

3. Concatenate an 8 bit long short individual index value to those 24 bit of the Node-ID

The resulting 32 bits long integer need to be used as the index for storing an array entry in a shared resource. The 8 bit individual index can be incremented individually for further array entries and allows for 256 distinct entries per Peer. The mechanism to create an array index is related to the pseudo-random algorithm to generate a Synchronization Source (SSRC) identifier of the transport protocol for real-time applications (RTP) [53] for calculating a collision probability. It generates an array index that most probably not clash with another index of the same array. The probability that two peers produce the same array index can be approximated by the with formula [53],

$$P(C) = 1 - (\frac{-N^2}{2^{L+1}}) \tag{1}$$

with L is length of the identifier and N the number of peer. Assuming 500 peers adding values to an array, the probability P is $7,4 * 10^{-3}$ that two peers generate a equal array index.

4.5. Access Control List in RELOAD

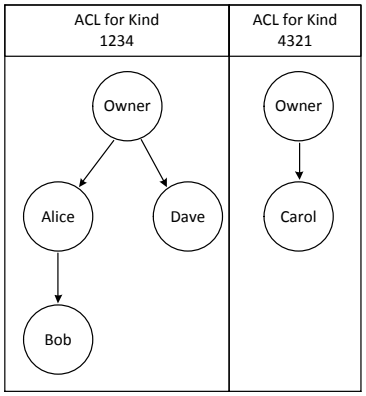
4.5.1. The ACL Kind

The concept of Shared Resources in RELOAD [5] adopts a new RELOAD *Kind* representing Access Control Lists (ACL). An ACL is a self-managed and shared resource that consists of an array of `AccessControlListItem` structures as shown in listing 5. Each entry delegates write access for a specific Kind data to a single RELOAD user. An ACL enables the RELOAD user who is authorized to write a specific Resource-ID to delegate his exclusive write access to a specific Kind to further users of a RELOAD instance. Each Access Control List data structure therefore carries the information about who obtains write access in the `to_user` field in line 5, the Kind-ID of the Resource to be shared in the `kind` in line 7, and whether delegation includes write access to the ACL itself in the boolean `allow_delegation` flag in line 9. The latter condition grants the right to delegate write access further for the Authorized Peer. Access Control Lists are stored at the same overlay location (resource-id) as the Shared Resource and use the RELOAD array data model. They are initially created by the Resource Owner.

An access control list is also representing a shared resource that needs to be fulfill the requirements for shared resources presented in section 4.4. Hence, the array indexes are formed according to the isolated data storage algorithm, its uses the USER-CHAIN-ACL access control policy and is compliant to the third requirement by containing the `ResourceNameExtension` structure as initial field.

Access Control List				
#index	to_user	kind	ad	signed by
0x123abc001	Owner	1234	true	Owner
0x123abc002	Alice	1234	true	Owner
0x123abc003	Dave	1234	false	Owner
0x123abc004	Owner	4321	true	Owner
0x123abc005	Carol	4321	false	Owner
...	...	...	...	...
0x456def001	Bob	1234	false	Alice
...	..	...	...	...

(a) Array representation



(b) Tree representation

Figure 17: Example: Access control list array including entries for two different Kind-IDs

```
1  struct {
2    /* Contains the resouce name in plain-text */
3    ResourceNameExtension res_name_ext;
4    /* The delegator */
5    opaque to_user<0..2^16-1>;
6    /* The kind-id of the shared kind */
7    KindId kind;
8    /* If true, allowed to store further ACL items in the list*/
9    Boolean allow_delegation;
10 } AccessControlListItem;
```

Listing 5: Kind structure: A single access control list item

Table 17a shows an example of an access control list while figure 17b represents according trust delegation tree. The `res_name_ext` field is omitted to simplify illustration. The array entry at index `0x123abc001` displays the initial creation of an ACL for a Shared Resource of Kind-ID 1234 at the same Resource-ID. It represents the root item of the trust delegation tree for this shared RELOAD Kind. The root entry MUST contain the username of the Resource owner in the `to_user` field and can only be written by the owner of the public key certificate associated with this resource-id. The `allow_delegation` (ad) flag for a root ACL item is set to 1 by default. The array index is generated by using the mechanism for isolating stored data as described in Section 3.1. Hence, the most significant 24 bits of the array index (`0x123abc`) are the least significant 24 bits of the node-id of the resource owner.

The array item at index `0x123abc002` represents the first trust delegation to an authorized peer that is thus permitted to write to the Shared Resource of Kind-ID 1234. Additionally, the

authorized peer Alice is also granted (limited) write access to the ACL as indicated by the `allow_delegation` flag (ad) set to 1. This configuration authorizes Alice to store further trust delegations to the Shared Resource, i.e., add items to the ACL. On the contrary, the trust delegation to user Dave in index `0x123abc003` not allowed Dave to append the ACL (`ad = 0`). The array index `0x456def001` illustrates trust delegation for Kind-ID 1234, in which the authorized peer Bob is not allowed to grant access to further peers. Each authorized peer signs its ACL items with its own private key, which makes the item ownership transparent.

To manage Shared Resource access of multiple Kinds at a single location, the Resource Owner can create new ACL entries that refer to another Kind-ID as shown in array entry index `0x123abc004`. Overwriting existing items in an access control list that reference a different kind-id revokes all trust delegations in the corresponding subtree. Authorized peers are only enabled to overwrite existing ACL item they own. The resource owner is allowed to overwrite any existing ACL item, but should be aware of its consequence that he might revoke the write access for still authorized peers. Furthermore, the ACL represented by the arrays store in RELOAD need to be loop free. Self-contained circular trust delegation from A to B and B to A are syntactically possible, even though not very meaningful.

4.6. Variable Resource Names

4.6.1. Resource Names Pattern

The mechanism to enable a variable naming scheme for overlay resources is based on regular expressions. Regular expression are used to define a certain naming pattern on which the variable resource names must match. The pattern are globally defined for each RELOAD instance within the XML configuration document and thus maintained by the overlay provider. Pattern for variable resource names are arbitrary URIs and must contain the `@` sign that separates the user-info from the host identity accordingly to the generic URI syntax [54] and uses the following variables:

$USER: This variable contains the legal username of the owner of the X.509 certificate [39] who initially created the shared resource. The username can be taken from the `SubjectAltName` in the X.509 certificate using the `uniformResourceIdentifier` type (c.f. [39] section 4.2.1.6.).

$DOMAIN: This variable contains the domain name of the overlay instance and thus specifies the host identifier of the variable URI.

Several sample pattern for a variable resource could be formed like shown in listing 6. The variable sub-string in the resource name can be very restrictive like shown in lines 1 and 7 or allow users to specify arbitrary string concatenations as shown in lines 3 and 5.

```
1 $USER-[0-9]-loc@DOMAIN        e.g., alice-3-loc@example.org
2
3 \d{1,4}\.$USER@DOMAIN         e.g., afk.bob@example.org
4
5 $USER-confercne-.*@$DOMAIN    e.g., carol-conference-ondogs@dht.de
6
7 $USER@$USER\.@DOMAIN          e.g., dave@dave.domain.org
```

Listing 6: Sample Pattern: Regular expressions to define resource naming pattern

The overlay providers are free to define any combination of the $USER/$DOMAIN and variable sub-strings. However, should a naming pattern consider an overlay wide uniqueness of resource names. For instance, a user with a overlay username *bill.gates@dht.ms.com* could get hijacked by a user within Address-of-Record *gates@dht.ms.com* using the second pattern in listing 6. A simple method to avoid name hijacking lies on the overlay operator side by defining reserved substrings used to delimit the pattern variables from user defined parts of a resource name. For instance the string *-conference-* as shown in line 5 in listing 6 could be reserved to for constructing conference URIs and shall not be includes in usernames of overlay parties. This naming scheme enable operators to control the number or form of allowed resources names for their overlay users.

4.6.2. Resource Name Extension

Access control in the RELOAD base specification [4] uses the *Signature* object presented in section 2.4.6 to validate if a user has write access on an overlay resource. The signature referrers to the public key certificate that authorizes its owner to store values at specific resource-ids. Access control policies like the USER-MATCH and USER-NODE-MATCH allow the storage if the hash over username in certificate matches the requested resource-id.

The meta-data in the RELOAD messaging protocol do not provide any default mechanism to transfer the name of a resource in plain text. It is either not transmitted within the certificate nor within the meta-structures of the store request. Without having the resource name in clear text, a receiver of request ain't able to validate if the chosen resource name matches to the corresponding regular expression of the naming pattern (see previous section 4.6.1).

To resolve this limitation, the ShaRe specification [5] defines the optional resource name extension struct shown in listing 7 that carries the resource name. The initial `type` field in the structure indicate what in which format the succeeding resource name is present. Currently, the only defined type is *pattern* and indicates that the following data structure contains an opaque $< 0..2^{16} - 1 >$ field containing the Resource Name of the Kind being stored. The type "pattern" further indicates that the resource name matches to one of the variable resource name pattern

defined for this Kind in the configuration document explained following. The unsigned short `length` value contains the length of the remaining data structure. It is only used to allow for further extensions to this data structure. Hence, the `ResourceNameType` enum and the `ResourceNameExtension` structure can be extended by further Usages to define other naming schemes, e.g., a type *free* indicating no restriction on the naming scheme.

Any application that defines a RELOAD Kind that intends to use variable resource names must use the resource name extension as initial field. Hence, when a peer is receiving a request transporting data, it takes the resource name out of the extension field and validates if its format is compliant to the corresponding resource name type.

```
1  enum { pattern  (1),
2  (255)} ResourceNameType;
3
4  struct {
5    ResourceNameType type;
6    uint16           length;
7    select(type) {
8      case pattern:
9        opaque resource_name<0..2^16-1>;
10
11       /* Types can be extended */
12   }
13 } ResourceNameExtension
```

Listing 7: Kind Extension: Extension containing the resource name in plain text

XML Resource Name Extension A strength of the RELOAD base specification [4] is the configuration of peers and clients towards the overlay properties before joining the RELOAD instance. This configuration XML includes a *kind-block* element specifying and configuring a set of RELOAD Kinds that must be supported by peers joining the P2PSIP overlay. The usage for shared resources [5] uses this mechanism for the indication if certain Kinds are enabled for variable resource naming and, if true, deploying the corresponding regular expressions of the allowed naming pattern. Therefore, ShaRe defines an extension to the *kind-block* XML element as shown in 8. The XML extension represented in the Relax NG notation as the document schema of the RELOAD configuration XML.

The extension defines its own XML namespace as shown in line 3. The Uniform Resource Name (URN) is a sub-namespace of RELOADs *config-base* namespace. Each *kind-block* element is a aggregation of several *kind-parameter* as indicated by the notation `=&` in line 7. Hence, line 7 extends a `variable-resource-name` element to the possible kind parameters. The extension element as a boolean `enable` attribute and can contain a variable

number of pattern elements. The enable flag indicates RELOAD implementations that for the Kind described within this *kind-block*:

- There exist resource name pattern it must support.
- Kind data structures sent within store requests or fetch answers contain a preceding ResouceNameExtension sturct as initial field as defined in listing 7.

As a result, depending on the configuration for a specific Kind in the XML document, implementation have to parse incoming request in different manners. This protocol scheme is useful to avoid sending the resource name extension field, if a Kind does not support variable names.

```
1  <!-- VARIABLE RESOURCE URN SUB-NAMESPACE -->
2
3  namespace share = "urn:ietf:params:xml:ns:p2p:config-base:share"
4
5  <!-- VARIABLE RESOURCE NAMES ELEMENT -->
6
7  kind-parameter &= element share:variable-resource-names {
8
9      attribute enable { xsd:boolean }
10
11     <!-- PATTERN ELEMENT -->
12
13     element pattern { xsd:string }*
14 }?
```

Listing 8: XML Extension: Variable resource name extension to the configuration document

A pattern element shown in line 13 must be present if the "enabled" attribute of its parent element is set to true. Each element defines a pattern for constructing extended resource names for a single Kind. It is of type xsd:string of the W3C recommendation for data types [55] and interpreted as a regular expression. In this regular expression, $USER and $DOMAIN are used as variables for the corresponding parts of the string in the certificate username field (with $USER preceding and $DOMAIN succeeding the @). Both variables must be present in any given pattern definition. If no pattern is defined for a Kind or the "enabled" attribute is false, allowable resource names are restricted to the username of the signer for shared resource.

A sample of the XML extension for variable resource names in shown in listing 9. The required-kind element is the container for each kind-block element carrying a single kind element describing the Kind that must be supported by implementations. As in the sample, all RELOAD implementation must support the *DisCo-Registration* that uses the USER-CHAIN-ACL access control policy. The max-count and max-size sub-elements in line 6

and 7 limit the amount of DisCo-Registration a single peer must store. Lines 8 to 12 define that variable resource names are allowed and the corresponding pattern. To authenticate the provenance of the `kind` element, the succeeding `kind-signature` element contains a signature over the Kind element. It is calculated using the private key file that is associated to the root public key certificate of the overlay provider. Since the root certificate is available for each overlay party, they are able to verify the signature.

```
1  <required-kinds>
2    <kind-block>
3      <kind name="DISCO-REGISTRATION">
4        <data-model>DICTIONARY</data-model>
5        <access-control>USER-CHAIN-ACL</access-control>
6        <max-count>2</max-count>
7        <max-size>100</max-size>
8        <share:variable-resource-names enable="true">
9          <pattern>
10           $USER-conf-[0-9]@$DOMAIN
11         </pattern>
12       </share:variable-resource-names>
13     </kind>
14     <kind-signature>
15         VGhpcyBpcyBub3QgcmlnaHQhCg==
16     </kind-signature>
17   </kind-block>
18 </required-kinds>
```

Listing 9: XML Example: Variable resource name extension for a DisCo-Registration Kind

4.7. Protocol Operations

4.7.1. Granting Write Access

Sharing write access with other RELOAD users to a specific Kind at a resource-id can solely be issued by the owner of the corresponding public key certificate called the *resource owner*. A resource owner can share RELOAD Kinds by using the procedure shown in figure 18. The resource name extension field is omitted in the call flow for a simplified visualization.

- The resource owner (RO) may stores initially the resource to be shared. The subsequent answer message just acknowledges the storage of the Kind with id 1234.
- The resource owner stores a root item of an access control list at the resource-id of the shared resource. The root item contains the resource name extension field, the

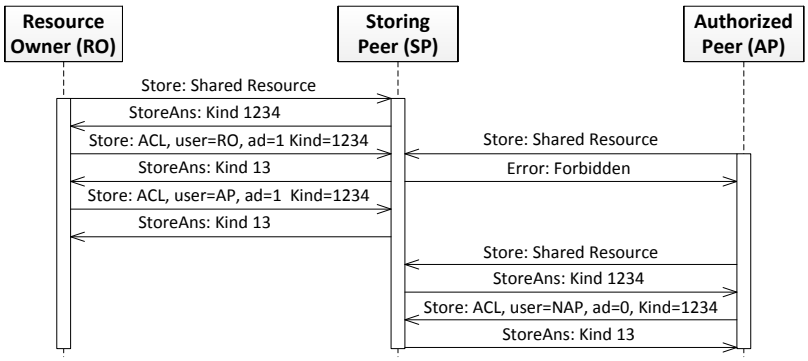


Figure 18: Call flow: Resource owner sharing a resource

username of the resource owner and kind-id (e.g., 1234) of the shared resource. The allow_delegation (ad) flag is set to 1 as default for a root ACL item. The array index of the root item is formed as described in the algorithm for isolated data stored shown in section 4.4. The store answer message by the storing peer again just acknowledges the reception of the ACL Kind, whose id is 13 in this example.

- The next ACL item for this kind-id stored by the resource owner will delegate write access to authorized peers (AP). This ACL item contains the same resource name extension field, the username of the authorized peer and the kind-Id of the shared resource. Optionally, the resource owner sets the "ad" to 1 (the default equals 0) to enable the authorized peer to further delegate write access. Each succeeding ACL item created by the resource owner can be stored in the numerical order of the array index starting with the index of the root item incremented by one.

Figure 18 further shows an attempt to access the shared resource before AP is listed in the access control list. Hence, the storing peer will not allowed the request and reject in with an *Error Forbidden* response.

The authorized peer can now write the shared resource and as is received delegation allowance ("ad"=1) it can extend the access to an existing Shared Resource as follows.

- The authorized peer can store additional ACL items at the resource-id of the shared resource. These ACL items contain the resource name extension field, the username of the newly authorized peer (NAP), and the kind-id of the Shared Resource. The "ad" flag is set to 0 thus not allowing NAP to further delegate write access. The array index MUST be generated as described in section 4.4. Each succeeding ACL item can be stored in the numerical order of the array index.

The protocol operations shown in figure 18 are shown as a sequence of independent requests. However, the RELOAD protocol enables to compile several operations within a single request as long they use the same method. For instance could all store request by the resource owner be achieved within a single store request.

A store request by an authorized peer that attempts to overwrite any ACL item signed by another peer is unauthorized and causes an Error_Forbidden response from the ptoring peer. Such access conflicts could be caused by an array index collision. However, the probability of a collision of two or more identical array indices will be negligibly low using the mechanism for isolating stored data (see section 4.4).

4.7.2. Revoking Write Access

The RELOAD base specification [4] does not foresee any mechanism to explicitly remove data form the overlay. Instead, the data intended for removal gets overwritten by an empty stored data object. This has in advantage that the "removed" data value still provides meta-data, e.g., storage time and the signature object of the removing peer.

Hence, write permissions are revoked by storing a *empty* value at the corresponding item of the access control list. Revoking a permission automatically invalidates all delegations performed by that user including all subsequent delegations. This allows to invalidate entire subtrees of the delegations tree with only a single operation. Overwriting the root item with a non-existent value of an access control list invalidates the entire delegations tree.

An existing ACL item MUST only be overwritten by the user who initially stored the corresponding entry, or by the resource owner that is allowed to overwrite all ACL items for revoking write access.

4.7.3. Validation of an Access Control List

Access control lists are used to transparently validate authorization of peers for writing a data value at a shared resource. Thereby it is assumed that the validating peer is in possession of the complete and most recent ACL for a specific resource/kind pair. The corresponding procedure consists of recursively traversing the trust delegation tree and proceeds as shown in the pseudo code example in listing 10:

1. Obtain the username of the certificate used for signing the data stored at the Shared Resource.

2. Validate that an item of the corresponding ACL (i.e., for this Resource/Kind pair) contains a `to_user` field whose value equals the username obtained in step 1. If the Shared Resource under examination is an Access Control List Kind, further validate if the "ad" flag is set to 1.
3. Select the username of the certificate that was used to sign the ACL item obtained in step 2.
4. Validate that an item of the corresponding ACL contains a `to_user` field whose value equals the username obtained in step 3. Additionally validate that the "ad" flag is set to 1.
5. Repeat steps 3 and 4 until the `to_user` value is equal to the username of the signer of the previously selected ACL item. This final ACL item is expected to be the root item of this ACL which SHALL be further validated by verifying that the root item was signed by the owner of the ACL Resource.

```
1  function validate_acl(acl, certs ,signer_id): bool
2    // acl:        The entire access control list
3    // certs:      All certificates to validate request
4    // signer_id:  Id of the siger of the shared resource
5    var signer_cert, last_name;
6    get_pkc(in certs, in signer_id, out signer_cert);
7    var curr_name = signer_cert.username;          // Step 1
8    REPEAT:
9      var in_list = false;
10     var depth = 0;
11     FOR each item in acl:
12       IF item.to_user == curr_name THEN:         // Step 2,4
13         IF acl.kind_id == ACCESS_CONTROL_LIST AND item.ad != 1 THEN:
14           Break;
15         IF depth > 0 AND item.ad != 1 THEN:
16           Break;
17         get_pkc(in certs,in item.signature.id, signer_cert);
18         last_name = curr_name;
19         curr_name = signer_cert.username;        //Step 3
20         in_list = true; depth = depth +1;
21         Continue;
22   UNTIL (last_name == curr_name OR !in_list)     //Step 5
23   return in_list;
```

Listing 10: Pseudo code: Algorithm for validating an access control list

The trust delegation chain is valid if and only if all verification steps succeed. In this case, the creator of the data value identified by the `signer_id` in 10 of the shared resource is an authorized peer. The ACL validation can be omitted if the signer of the shared resource is the resource owner. In this case, the validation is achieved via the public key certificate of the resource owner.

Depending on the role of a peer validating the access control list, it has a different knowledge about the ACL itself and obligation on the validation procedure. In the following, this thesis figures out the differences between peers storing the shared resource and peers requesting for the shared resource.

Storing Peers Storing peers, i.e., peers at which Shared Resource and ACL are physically stored, are responsible for controlling storage attempts to a shared resource and its corresponding access control list. To assert the USER-CHAIN-ACL access policy presented in the following section 4.7.4, a storing peer must perform the access validation procedure described in listing 10 on any incoming store request using the most recent access control list for every Kind that uses the USER-CHAIN-ACL policy. It has further to ensure that only the resource owner stores new ACL root items for shared resources or overwrites an existing ACL item.

Accessing Peers Accessing peers, i.e.,peers that fetch a shared resource, might validate for its own security that the originator of a shared resource was authorized to store data at a certain resource-id by processing the corresponding ACL. To enable an accessing peer to perform the access validation procedure described in pseudo code 10, it first needs to obtain the most recent access control list in the following way.

1. Send a RELOAD *Stat* request to the resource-id of the shared resource to obtain all array indexes of stored ACL Kinds. The corresponding *StatAns* message contains all meta-data, e.g., storage time, dictionary keys and array indexes of all stored data objects stored at the resource-id.
2. Fetch all indexes of existing ACL items at this resource-id by using the array ranges retrieved in the *Stat* answer.

Peers can cache previously fetched access control lists up to the maximum lifetime of an individual item. Since stored values could have been modified or invalidated prior to their expiration, an accessing peer better refreshes its knowledge using *Stat* requests to check for updates prior to using the data cache.

4.7.4. USER-CHAIN-ACL Access Control Policy

The RELOAD base protocol [4] defines a set of default access control policies that regulate the write permissions for peers in the overlay. Even though the protocol *anticipates that only a small number of generic access control policies are required* (c.f. [4] section 6.3.), it allow the definition of further policies if required.

The usage for Shared Resources [5] presented in this thesis specifies an additional access control policy to the RELOAD protocol. The USER-CHAIN-ACL policy allows authorized peers to write a shared resource, even though they do not own the corresponding certificate. Additionally, the USER-CHAIN-ACL allows the storage of Kinds with a variable resource name that are following one of the specified naming pattern. Hence, on an inbound store request on a Kind that uses the USER-CHAIN-ACL access policy, the following rules apply:

- A given data value must be written by the storing peer if either one of default USER-MATCH (c.f. [4] section 6.3.1.) or USER-NODE-MATCH (c.f. [4] section 6.3.2.) access policies applies. The latter policy must be used if the data model of the shared resource is a *dictionary* as otherwise the USER-MATCH would mask it.
- If neither of those policies applies, the storing peer must validate if the resource name of the kind data is conform the variable naming pattern. Hence, the kind data must be written if the certificate of the signer contains a username that matches to one of the variable resource name pattern (c.f. section 4.6.2) specified in the configuration document and, additionally, the hashed resource name matches the resource-id. The resource name of the Kind to be stored must be taken from the in this case mandatory resource name extension field in the corresponding Kind data structure.
- If even the latter conditions does not apply, the originator of the store request might be an authorized peer that does not poss a public key certificate allowing him write access. Then the storing peer must store the kind data if the ACL validation procedure described in listing 10 has been successfully applied.

All definition of access control lists, variable resource names and their corresponding XML extensions are just specified to enable the access control policy described above. Shared resources in RELOAD were initially intended to serve the usage for distributed conferencing as base for a secure mechanism to map a group access in RELOAD. However, seem the ShaRe Internet draft to be a reference for future applications using a shared write access. For instance will the third party registration scenario presented in 4.3.1 soon be available as new Internet draft within the standardization progress in the P2PSIP working group in the IETF.

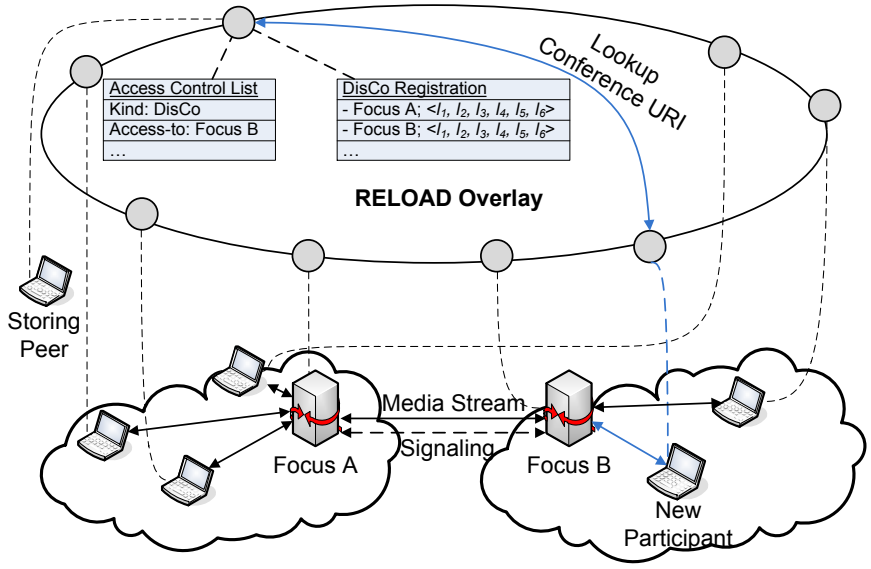


Figure 19: Reference scenario: Focus peer A and B jointly managing a DisCo

5. Distributed Conference Control based on RELOAD

5.1. Overview

Distributed conference control (DisCo) [6] defines a protocol scheme for managed P2P conferences with SIP [1] using the discovery and locations service of the RELOAD base protocol [4]. A *DisCo* refers to a multiuser conference over IP in which the controlling entity called focus is located at many independent endpoints. The distribution of the point of control onto multiple *focus peers* is held transparent to users and to its VoIP application and provides backward compatibility to regular SIP applications. Here, the conference is identified by a single Uniform Resource Identifier (URI). In DisCo, its locator is separated onto several focus peers. A reference scenario of a distributed conference is shown in figure 19. The mapping of the conference URI to one or more focus peers is stored in a RELOAD data structure for distributed conferencing denoted as DisCo Registration. The DisCo Registration and a corresponding access control list is maintained by a *storing peer* which is part of the RELOAD overlay, but not necessarily member of the conference.

Each DisCo Registration stores the overlay address (node-id) of a focus peer and a topological descriptor called *coordinate* value that announces a relative position in the network. It is used to optimize the interconnection graph among the conference participant and its responsible focus peers. A RELOAD peer that intents to participate in a multiparty conversation

uses RELOADs lookup functionality to resolve the conference URI. It retrieves a dictionary that contains all actively managing focus peers and all registered but not actively managing focus peers. The joining peer then may chooses the closest focus peer to join the conference referencing to its own *coordinate* value. In this instance, it reduces network latencies to its controller thus minimizing the delay and jitter for the subsequent media streams. Following, the joining peer establishes a transport connection using RELOAD functionalities, while the instantiated transport socket is passed to the SIP stack. Using this transport, the latter creates a SIP session to participate the distributed conference. The focus peer announces the arrival of a new participant and negates the media parameters to finally connect the party with the conference media.

Participants of a conference are obliged to put their device capabilities at the disposal of the conference. This enhances the scalability of the service while a growing multiparty session. Following a conference policy, several participants obtains the permission by already established focus peers to store data at the overlay location of the DisCo Registration thus to add their own mappings (*AoR -> node-id*) into the shared DisCo resource. This is achieved by adding the overlay usernames of those parties into the access control list. As shown in figure 19, the focus peer B has obtained permission by A to register as a focus.

As a result, participants in a DisCo are arranged in a two-layered hierarchy in which several focus peers maintain a subset of the entire conference members. The controlling peers distribute the conference media among their clients and distribute the upstream media of their clients to the other conference controller. They further maintain an additional signaling relation for synchronizing the entire state of the conference.

5.1.1. Scope of DisCo

Distributed conferences are designed to enable ad-hoc multiuser conferences in P2P scenarios. The motivation for avoiding a centrally managing entity might be reasoned by different aspects. First of all, a P2P conferencing solution benefits from the characteristics of P2P networks:

- Self-organization
- Fault-tolerant
- Scalability

The users of a P2P conference can ad-hoc establish new multimedia sessions without booking and configuring a dedicated conferencing service. A coordinated P2P session can adapt to the size of the conference and compensate node failures. The DisCo principle can be compared with further P2P Instant Messengers like Skype [56]. Skype is build on a P2P network that is

mainly self-organized by the peer joining it. Further, Skype provides a free voice conferencing service and point-to-point video chat. However, its video conferencing service claims for user charges and demands a potent devices to distributed the conference media for maximal 10 participants[3]. The DisCo protocol in contrast, is an open standard and can be implemented by everyone. An open source implementation could enable a conferencing application that scales to a large number of participants without assistance of a central server. The lack of dedicated sever enables the establishment of a video conferencing service without user charges.

Another reason for avoiding a third-party service provider are privacy policies within companies. A company might not want to perform confidential conferences about over third-party supplier. While larger companies are able to host their own dedicated conferencing infrastructure, it might be a financial problem for smaller companies to host hardware and personnel. In DisCo each focus peer managing the conference is also a participant of the group session. This takes in advantage that a distributed conference is self-managed and does not require a third-party supplier. The RELOAD overlay can be instantiated within the company network to avoid to the registration of the conference URI in a public RELOAD overlay. Thus, a confidential meeting can be achieved without the need of third-party supplier while reducing the costs to provide the service to a minimal amount.

5.1.2. Concurrent Work in the IETF

An alternative Internet draft on distributed conferencing (DCON) [58] submitted to the IETF proposes signaling protocol to interconnect multiple centralized conference servers. In this concept, SIP user agents request a dedicated XCON server [16] to join a certain conference URI. If conference is not hosted by the Server, it delegates the request to a DCON application that is arranged within a DCON not further specified overlay. The overlay is joined by further XCON instances, each hosting independent conferencing domains. Using the overlay, a DCON module locates the server managing the desired conference and performs a participation request behalf of the SIP user agent. If accepted, the conference media is sent from the host of the remote conference, via the XCON server hosting the user agent, and finally, to the requesting user agent.

The conferencing approach for DisCo and DCON are designed for different deployment scenarios. DCON assumes an infrastructure of several dedicated XCON servers that provide their clients access to all conferences hosted by these server. DisCo assumes a group of independent peers, creating a self-managed conference service.

[3] Skype recommends video conferences with up to 5 parties [57]

5.2. Protocol Design

5.2.1. Architecture

To design an architecture for distributed conference control, first, the roles and relations of the participating entities must be identified:

Focus Peer: A focus peer is a participant and a manager of a distributed conference. It must support the functionalities to resolve and register a distributed conference ID in a P2PSIP overlay. Further, it must be enabled to accept conference calls and provide mechanisms to announce changes of the conference state. A focus peer must be aware of the distribution of the conference to enable synchronization mechanism with the other controller.

DisCo-aware Participant: A DisCo-aware participant is a peer that joined a distributed conference by resolving the DisCo Registration in a P2PSIP overlay. It provides the same functionalities as a focus peer, but is not actively involved in the conference maintenance.

RELOAD-aware Participant: A RELOAD-aware participant is a peer that joined the conference through a P2PSIP overlay, but is not aware of the its distribution. The conference distribution must be transparent to such a user to be compliant to its RELOAD and SIP implementations.

Plain SIP User Agent: A plain SIP UA joined the conference through standard SIP signaling. It is either not aware of the RELOAD protocol nor the DisCo protocol schemes. The conference distribution must be transparent to such a user to be compliant to its SIP implementations.

Storing Peer: The storing peer is the RELOAD peer responsible for the address range of the hashed conference identifier and not necessarily participating in the multiparty session. It must be aware of the RELOAD aspects of DisCo to maintain the DisCo Registration and its corresponding access control list.

The resulting architecture by identifying the entities for distributed conference control is shown in figure 20. The figure omits the RELOAD-ware participant to simply the illustration. The conference state is a logical representation of the roles an relations among the distributed conference. Each focus peer maintains a coherent copy of the conference state. Its synchronization is achieved by a SIP notification [10] mechanism described in more detail in section 6. The synchronization via SIP event has in advantage to reuse existing SIP implementations. The latter must be extended by a state agent for distributed conferencing to performing the synchronization procedures. The same state agent is used to announce the conference state to the DisCo-aware participants. Hence, by implementing a single module we obtain two functionalities needed for DisCo sessions.

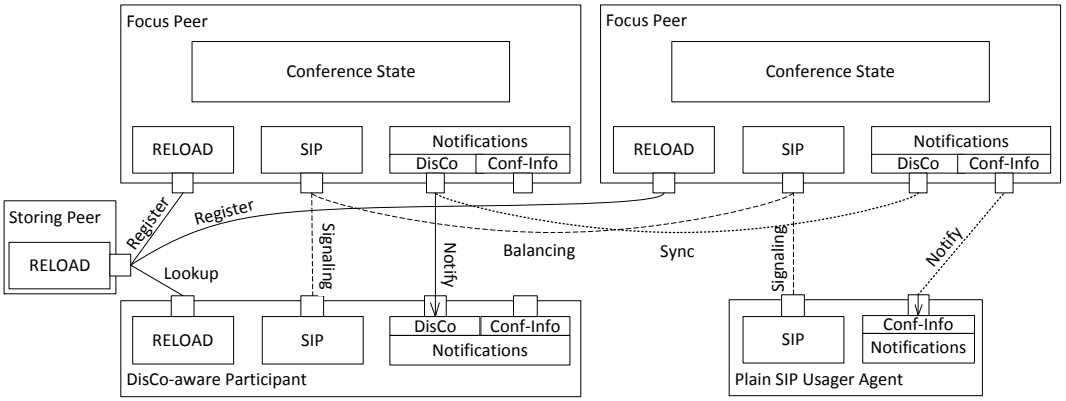


Figure 20: Architecture: Roles and interactions within a distributed conference

Since each conference focus provides a copy of the conference state, they can use it to provide load balancing and failover mechanisms. A focus peers reaching its threshold of serving new clients can determine a less loaded focus from the state representation.

Focus peers further maintain a single SIP signaling relation to each directly connected participant. The SIP dialogs are used to negotiate the media parameters to the subsequent media session. This is plain SIP behavior and hence compliant to all entities in a DisCo.

DisCo-aware participants retrieve the overlay addresses of the focus peers through the RELOAD lookup service and initiate transport connections via the RELOAD messaging service. DisCo participants are subscribing for DisCo event notifications to obtains a view on entire conference state. The provided module for the *conference-info* event package [15] is only used if it assumes the role of a focus peer to the conference.

Plain SIP user agents can join a distributed conference via plain SIP signaling mechanisms. However, the conference must either be registered at a dedicated SIP proxy, or the IP address of one of the focus peers must be passed by an external mechanism. This due to the lack of *SIP-to-RELOAD* gateway peers in P2PSIP . Gateways peer are mentioned within the P2PSIP concepts [59] draft, but no more detailed is available yet. SIP user agents subscribe for the conference-info event package [15] to obtain the global knowledge of the conference. However, as the event package for conference state was not intended to map a distributed conferencing scenario. Hence, the view on the conference state is limited as further described in section 6.

5.2.2. The DisCo Registration

A data structure used to register a distributed conference must map the separation of conference ID to several locators. This requirement implies a data model that can carry multiple values that are encapsulated within an outer generic structure. The RELOAD protocol [4] provides the three data models `single value`, `array` and `dictionary` that can be used to encapsulate data. Accordingly, a DisCo Registration could utilize either an array or dictionary as data model.

The array data model has several open issues with respect on carrying a DisCo Registration.

- How are the array indexes formed?
- How can the registration be prevented from race conditions while concurrent writing?
- How can a joining participant retrieve all array entries at once?

The last issues is justified by the RELOAD messaging protocol. For value stored in the array data model there exists no *wildcard* fetch mechanism to retrieve all array entries at once. Instead, a RELOAD peer can query for one or more array ranges. An array fetch on the entire address range from $index = 0$ to $index = 2^{32} - 1$ would be replied by a response containing the requested array entries, plus ~4 billion empty array entries.

The dictionary data model fits for all three open issues of the array. The dictionary could be set to the node-id of registered focus peer. This is an unambiguous assignment of the dictionary value and separates each mapping from concurrent writing. Furthermore, this approach follows the SIP Registration usage [38] for RELOAD and thus compliant to RELOAD protocol design.

Furthermore, a DisCo Registration could reused an existing RELOAD Kind to map the conference ID. For instance, the SIP Registration Kind is also arranged as dictionary and is enable to map a SIP URI to several locators. However, none of the previously defined Kinds provides the possibility to register a peers node-id and to announce its relative position in the network. As a result, we need to define a new RELOAD Kind data structure that fits the requirements for distributed conferencing.

This work proposes a lightweight DisCo Registration structure that enables an unambiguous mapping for each registered focus peer. The DisCo Registration and its surrounding data model are shown in listing 22. Line 1 shows the RELOAD definition of dictionary key type. The maximal size of the key is large enough to carry the node-id (128 bits) of a focus peer. The inner `DataValue` struct is a further meta-data of the overlay data model. The boolean `exists` indicates if the carried `value` contains data. Line 13 to 17 show the definition of the DisCo Registration. It contains the resource name extension field presented in listing 7, the node-id of the conference creator and an opaque coordinate value. The latter is designed as opaque string value to carry any type of topological descriptors.

```
1  typedef opaque DictionaryKey<0..2^16-1>;
2  /* The distionary data model */
3  struct {
4    DictionaryKey     key;
5    DataValue         value;
6  } DictionaryEntry;
7  /* The data value structure*/
8  struct {
9    Boolean           exists;
10   DisCoRegistration value<0..2^32-1>;
11 } DataValue;
12 /* The DisCo Registration structure*/
13 struct {
14   ResourceNameExtension res_name_ext;
15   opaque coordinate<0..2^16-1>;
16   NodeId node_id;
17 } DisCoRegistration;
```

Listing 11: DisCo-Registration: Data structure to register a distributed conference

The data structure fulfills all previously established requirements for a DisCo Registration. The `res_name_ext` field allows the registration of a variable conference ID by following the definitions of ShaRe presented in sections 4. Using a dictionary data model, all focus peers can register their node-id within the same data structure to map the ID/Locator split of DisCo. By defining a generic field to carry topological descriptor, the focus peer can announce their relative coordinates with respect to optimize the conference topology.

5.2.3. Routing to a Focus

If the conference URI is resolved to several focus peers by the RELOAD lookup functionalities, a joining peer must connect to any of the available peers. Here, we identify three mechanisms to initiate a SIP session to a focus:

Iterative: In an iterative mechanism, the joining peer choses one the focus peers retrieved by the DisCo Registration and sends an AppAttach request to initiate a SIP dialog. This approach is favorable with respect to the minimal signaling overhead if the request is accepted by the focus. Otherwise, the joining peer must repeat this procedure after a timeout, causing unnecessary delay.

Source-routed in RELOAD: In a source-routed approach, a joining peer could use the RELOAD destination list feature to build a source-route along the order of the preferences of the joining peer. A source-routed AppAttach request would traverse the focus

peers until one of them accepts the request. However, the destination list in RELOAD is not intended to be interrupted by intermediate peers. The default overlay routing specifies that a requests containing more than one destination in the list, must be forwarded without processing the contained data [4]. Hence, a source-routed AppAttach request will always have the last focus in the list as its destination thus overlay source-routing is no option.

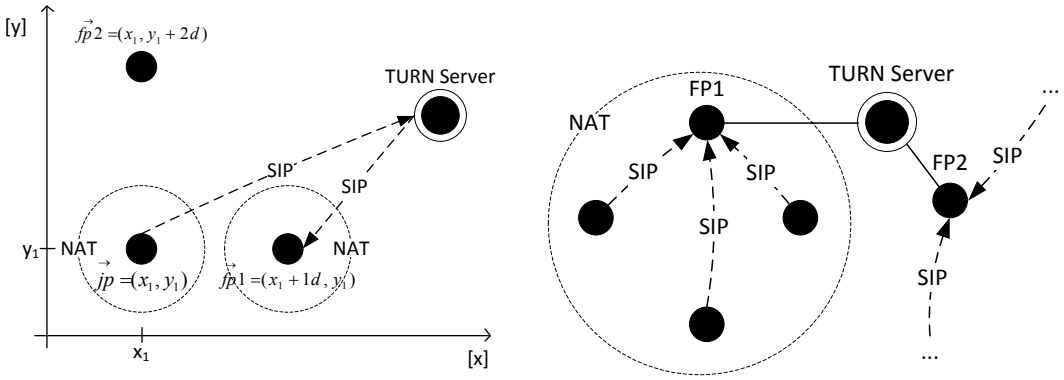
Source-routed in SIP: In a SIP source-routed approach, a joining peer would sent AppAttach requests to all focus peers simultaneously. Then, the peer could extract the IP addresses of the focus peers responded the AppAttach requests. The obtained addresses could be used to create a SIP source-route that traverses all focus peers. The latter, aware of this source-routing procedure could intercept the SIP request and accept the conference call. This approach is the worst with respect to signaling overlay, but prevents the joining peer from retransmissions.

As a result of this discussion, the iterative focus selection is the most favorable. It produces the least overhead and follows the default AppAttach procedures of RELOAD. A focus peer not accepting the call could alternatively use the DisCo call delegation mechanism to transfer the caller to anther focus peer. The SIP source-routing approach produces a factor N overhead to initiate a conference call, with N is the number of available focus peers. The approach is not scaling well for very large multiuser conferences in which almost any peer is subscribed as potential focus.

5.2.4. Adding Focus Peers

An advantage of distributed conference control compared with centralized conferencing refers to the distribution of the load that occurs during operation of a multiparty session. All participants in a distributed conference can potentially become a focus peer for their conference and thus obtain the responsibility for signaling and media mixing for a subset of the conference parties. Participants with adequeate capacities (CPU, Memory, Network) of the conference should become a focus peer according to their capacities. For instance, a mobile device connected over UMTS would not be a suitable candidate to register as focus peer. However, a more sophisticated mechanism to estimate potential, workload and bandwidth of focus peers is left to future work.

Another challenging issue are potent desktop device which is located behind a NAT. Figure 21 shows two different scenarios that illustrate under which circumstances a focus behind a NAT is suitable or not. The figure in 21a shows a case where a joining peer jp that has a relative network position $jp = (x_1, y_1)$. jJp could connect to the one of the focus peers $jp1$ and $jp2$, with relative coordinates $jp1 = (x_1 + 1d, y_1)$ and $jp2 = (x_1, y_2 + 2d)$ with d is the distance between jp and $fp1$. Jp chooses $fp1$ as its focus peer based on its proximity. Unfortunately,



(a) Bad case: Coordinates circumvented via NAT traversal (b) Good case: Several parties behind the same NAT

Figure 21: Comparison: Focus peers behind NATs

are both peers located behind two different NATs which both will detect during the RELOAD *AppAttach* procedure and subsequent ICE checks [36]. Additionally, the behavior of the NAT is very restrictive and forces the peers to perform the TURN protocol [31] for further connectivity. Both peers must establish a transport connections to a TURN server that will relay the data packets among the endpoints. As a result, media streams will traverse over an circuitous route and increase the delay and jitter. The second focus peer $fp2$ would have been the better choice because he is not behind a NAT.

As a consequence, it could defined that it is not desirable to register as focus peer, if the application resides in a NAT. However, not all symmetric NATs require relaying through a TURN server. If the peer applications determine their public IP in front of the NAT, e.g., using STUN [30], the peers are enabled to setup direct links among each other.

A reasonable circumstance for focus peers behind NAT is shown in figure 21b. Supposing a large company network behind a restrictive NAT performing a distributed conference. Several parties of the conference reside in the same company network and would select accordingly to proximity a focus peer in their domain. Even though the that focus $fp1$ relays through an external TURN server to the remaining conference, the established interconnection graph is optimized in terms of delay, jitter and efforts for NAT traversal.

As a result, a conference party should to register as potential focus peer even though it resides in the private address space. The joining peers should determines if a chosen focus resides in its own NAT (c.f. figure 21b) or it must relay the streams via an external TURN server (c.f. figure 21a). These information can be obtained by the ICE Checks [36] performed while RELOAD AppAttach procedures.

5.2.5. Proximity-awareness

An issue in distributed conference control is the proximity-awareness of the participants. Each RELOAD peer within the context of a distributed conference should be aware of its relative position in the network topology. These *topological descriptors* are used to optimize the connection graph among the DisCo parties in respect of the underlying network. The objective is to reduce the issues of delay and jitter and to provide an adequate quality of the multimedia service.

One class of proximity approaches [60] is built on landmarks. To determine the relative network position *p* of a node, the round-trip times (RTT) are measured against a fixed set of well known landmarks $l_0, l_1, .., l_n$. These measurement results will be ordered according to the landmark index with the result of a *landmark vector* $< l_1, l_2, .., l_3 >$.

The obtained relative position can then be stored in the overlay. Assuming the relative position *p* for node-ID id_n, then $p' = hash(id_n)$ is an overlay identifier, for position *p* of node n. Node *n* will then be mapped with p' into the overlay region, stored on the node that is responsible for this address range. The nodes n_1 and n_2 are close to each other if the difference of $|p_1 - p_2|$ is low, with p_1 and p_2 were retrieved by a lookup operation on $p_1{}'$ and $p_2{}'$.

The DisCo protocol scheme contributes to landmark approaches by defining a `landmarks` extension to the overlay configuration document presented in listing 12.

```
1  <!-- LANDMARKS URN SUB-NAMESPACE -->
2
3  namespace disco = "urn:ietf:params:xml:ns:p2p:config-base:disco"
4
5  <!-- LANDMARKS ELEMENT -->
6
7  parameter &= element disco:landmarks {
8    attribute version { xsd:int }
9
10   <!-- LANDMARK-HOST ELEMENT -->
11
12     element landmark-host {
13       attribute address { xsd:string },
14       attribute port { xsd:int }
15     }*
16  }?
```

Listing 12: XML extension: List of landmarks to determine relative position in the network

The `landmarks` element in line 7 is a container for `landmark-hosts` elements. The `version` attribute is incremented if the landmarks host contained within corresponding child

elements changed. The child element contains an address and port field to which the peers can perform their RTT measurements.

This enables DisCo-aware peers to determine their topological description using a landmark approach. The resulting landmark vector can be announces within the DisCo Registration to enable joining peers a proximity-aware conference joining.

As specified in the previous section, the DisCo participants iteratively try to establish a SIP dialog to their chosen focus. If the latter is is fully booked, it will try to further delegate the joining party to another focus. The delegation is thereby performed with respect to the coordinates of the joining peer. Therefore, the SIP INVITE request can contain the coordinate as an URI-parameter called 'coord' in the contact-header in base64 encoded form [63]. An example contact URI shown below.

```
1 "sip:alice@example.com;coord=PEknbSBhIHRvcG9sb2dpY2FsIGRlc2NyaXB0b3I+".
```

Listing 13: Coord-parameter: Encoding of the coordiate value as base64

The called focus uses the 'coord'-parameter to determines the next available focus closest to the calling peer using the received descriptor and the coordinates of the other focus peers.

5.3. Protocol Operations

5.3.1. Conference Creation

The creation of a distributed conference requires several individual steps that have to be done by the conference creator. The following procedures assume that the creator posses the public key certificate permitting him to write data at the desired resource-id.

Registering a DisCo After the user has chosen a conference URI that complies to the allowed namespace and determined its coordinates, the DisCo application can begin the registration procedure in the RELOAD overlay shown in the sequence diagram in figure 22. The registration procedure is structured as described follows:

Stat Request: Prior to registration, the conference creator checks whether at the desired resource-id still contains a DisCo registration and corresponding access control list from a previous conference by sending a *StatReq* of RELOAD message (cf. [4] section 6.4.3.1.). The request will be routed through the overlay to the resource-id which is the result of hashing the desired conference URI. The storing peer responds with a *StatAns* message (cf. [4] section 6.4.3.2.) containing the meta-data for all Kinds it is currently holding. It is important to note that the storing peer will just return the meta-data for the

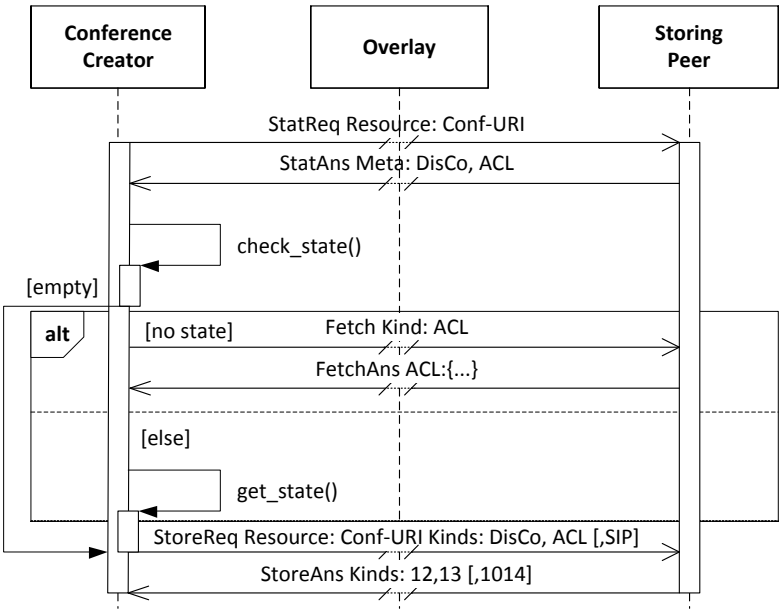


Figure 22: Message flow: Registration of a distributed conference

requested resource-id and not for all stored data objects it holds. Hence, if the answer contains meta-information from a previous conference, the creator has to fetch all DisCo and ShaRe Kinds. If *StatAns* contains no result, the requested resource-id is empty and the conference creator can proceed with a *StoreReq* (cf. [4] section 6.4.1.1.). Otherwise, it must check the state of previous registration.

State Check: In general, a DisCo application should be aware of previous conference registration by caching the last recognized DisCo and ACL data objects. If the registration state retrieved by the *StatAns* equals the cached state, the registration can proceed with the storage of of the DisCo and ACL Kinds. However, could both Kinds been changed during the disappearance as the conference will persists as long as peers are managing session and could have been added new focus peers. In such a case, the creator must send a *FetchReq* message explicitly requesting for all access control list items. By receiving all ACL items from the subsequent *FetchAns* message, the creator can reconstruct the existing trust delegation tree whose root item should have been signed by itself.

Store Request: Finally, the registration completes by sending a *StoreReq* containing the DisCo-Registration Kind shown in listing 11, the ACL Kind and, if possible, a SIP registration Kind enabled to use variable resource names. The access control list Kind contains at least the root item initiating or refreshing the ACL trust delegation tree. If the

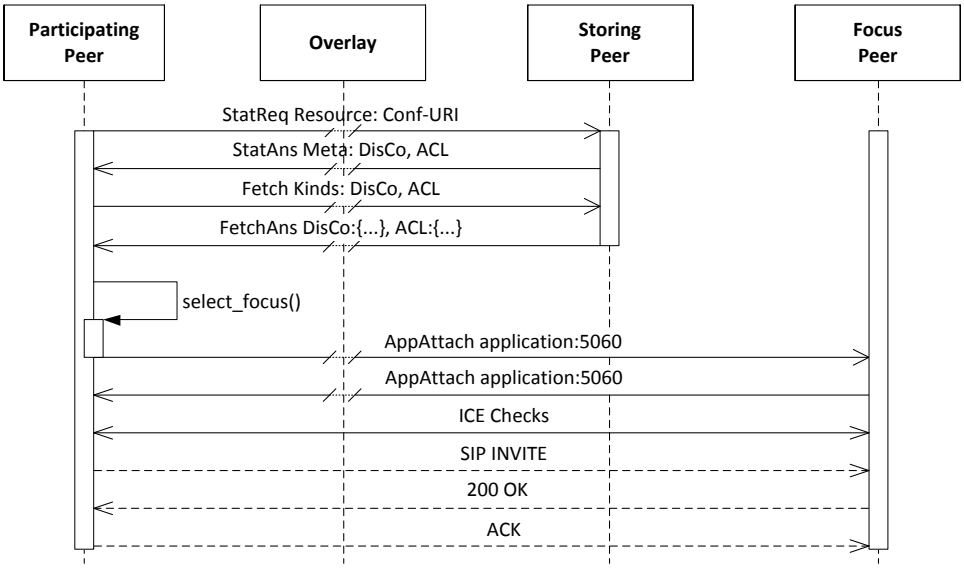


Figure 23: Message flow: Selecting focus and establishing a transport

creator is already aware of specific users who will participate the conference in focus role, it can add further ACL items to the initial store request. The optional SIP registration (see listing 4) should be of type `sip_registration_route` and register a destination list containing at least the node-id of the conference creator. In analogy to the DisCo Kind, the creator is free to store an additional access control list allowing the future focus peers to register their node-ids in the SIP-Registration Kind. This serves as backup service enabling DisCo-unaware RELOAD peers to participate the distributed conference.

The lifetime of a distributed conference is not limited by the participation time of its creator. As long as the root item of an access control list to a DisCo-Registration is not expired (cf. `StoredData` [4] section 6.), authorized peers allowed to further maintain DisCo-Registrations at the storing peer and even store new focus registrations.

Once the conference creator has registered its node-id for a DisCo, it should be ready to receive incoming participation request over RELOAD messaging service and via SIP signaling. The creator is then responsible to negotiate media parameters and to connect all parties to the conference media.

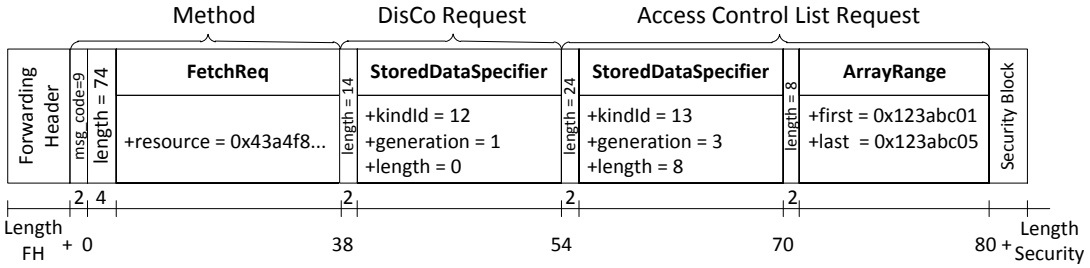


Figure 24: Message Structure: A FetchReq message represented along the byte stream

5.3.2. Joining the Conference

The joining procedure to a registered DisCo is separated into several steps and is performed using at least three application layer protocols RELOAD [4], SIP [1] and a transport protocol for the conference media, e.g., Real-Time Transport protocol (RTP) [53]. This section will describe the former two protocol procedures in detail shown in figure 23.

Assuming a user obtained the conference URI by some external mechanism, a joining peer sends a RELOAD *StatReq* message to the resource-id related to the conference URI. The message will be routed along the overlay to the peer storing the DisCo Registration and corresponding access control list. The additional *StatReq* is necessary to obtain the array indexes of the access control list items, since RELOAD provides no *wildcast* mechanism for arrays. Afterwards. the joining peer sends a *FetchReq* message (cf. [4] section 6.4.2.1.) for the DisCo and ACL Kinds to the resource-id of the conference URI. The *FetchReq* should include any specific dictionary keys, but explicitly fetches for ACL items. These are used by joining peer to comprehend the trust delegation tree of the shared conference resource and enable him to verify if the DisCo Registration is stored by authorized focus peers

Figure 24 shows a sample of a *FetchReq* as it would be represented after serialization. The forwarding header follows the message body starting with the 2 byte message code and a 4 byte long length field containing the size of the rest of the message body. The message code is encoded as `uint8` in network byte order and identifies the method of this message as fetch request message. It contains the 32 bytes long resource-id of the desired data objects and one or more `StoredDataSpecifier` structures each of with a maximal payload of $2^{16} - 1$ bytes. The size of each specifier is indicated in a preceding length field of 2 bytes. The sample message contains a specifier requesting for the DisCo kind-id (=12) whose generation counter is 1. The generation counter gets incremented each time a store request updates the stored data. The rest of this specifier is the length value whose size is 0. This indicates the receiving stack a wildcard fetch on a dictionary, since the kind-id of the DisCo registration must be known and matches *dictionary* as corresponding data model.

The next specifier again precedes with a length field followed by the queried kind-id requesting the access control list Kind (=13). In contrast to DisCo specifier, contains ACL query a payload of 8 bytes length represented by the succeeding `ArrayRange` object. It requests for the array range from $0x123abc01$ which might be the ACL root item of the trust delegation tree, to array index $0x123abc05$. The array range already implies that there exists at least four possible focus peers to join the conference. Four, because the conference creator must not be necessarily be registered as focus but needs to initiate the access control list since it is the resource owner.

Note that the additional length field within the data specifier object are not designed to be redundant with each preceding length specification. Instead, it can be used by further extensions of new data models to indicate their length.

The response on the fetch request is the according *FetchAns* message carrying all DisCo-Registrations and the entire access control list.

Application Attach The final operation perform by the RELOAD messaging service is the so called *AppAttach* procedure. The application attach is the default mechanism in RELOAD to establish a direct transport connection among two endpoints. The *AppAttach* overlay message therefore carries the ICE [36] credentials *ufrag* and `password`, an *application* field containing the port number of the desired transport (5060 = SIP port) and a couple of ICE candidates through which the connection will traverse. The `ufrag` and `password` contain a random at least 4 byte username and 22 bytes long password used to identify each endpoint when creating a direct connection among them. Each ICE candidate belongs to a host that will be used to establish the transport connection, e.g., TURN or STUN servers used for NAT traversal. In the simplest case, an ICE candidate is the local host if the peer has a public IP address.

The focus peer responds the request with a another *AppAttach* message, containing its own ICE parameter and candidates. After exchanging the ICE parameters, both peesr try to establish the connection as indicated by the *ICE Checks* message flow in figure 23. However, a detailed description of the ICE procedures [36] will be out of scope of this document. The result of ICE is a direct transport connection over which the participating peer can send a SIP INVITE request to the focus peer to finally join the conference.

5.3.3. Leaving a Conference

Participants leave a distributed conference sending a SIP BYE request to its connected focus peer. The latter acknowledges by responding a 200 OK response and stops sending the conference media to leaving peer. This is the default SIP behavior as described [1] and thus transparent to exiting SIP implementations.

A focus peer maintains signaling and media session for its connected parties. Thus, as a focus leaves several users would loose their connection to the conference. Te ensure a continues service, the protocol scheme for distributed conferencing provides a procedure for leave management described in section 5.5. After the leave management is completed, a focus peer sends a SIP BYE request to each of its participants and to its own focus.

A focus peer must additionally remove its DisCo and SIP registrations. This is achieved by sending a RELOAD *StoreReq* containing for each Kind an empty data value that overwrites its records. This remove procedure is called storage of "nonexistent" values in RELOAD [4].

Any access control list items remain existent up to their individual lifetime. This ensures that participants and storing peers (in case of a topology switch) can still validate (c.f. listing 10) the entire trust delegation tree represented by an ACL.

5.4. DisCo-Unaware Participants

5.4.1. RELOAD-aware Applications

Peers joined to an RELOAD overlay obtained the overlay configuration document that defines the Kinds a peer must support to participate the overlay. A minimal overlay configuration will require at least the support of the SIP registration Kind. However, overlay peers are permitted to store Kinds that are not listed in the required Kinds. A minimal configuration opens up the possibility that peers are aware of the SIP Kind, but not for a DisCo Kind thus DisCo-unaware RELOAD peers.

As mentioned in section 5.3.1, the conference creating peer addresses this problem by storing an additional SIP-Registration Kind and deposits a corresponding access control list at the same resource-id as the DisCo-Registration. Hence, the minimal requirements for DisCo-unaware peers is the implementation of the SIP usage [38] and the protocol scheme for shared resources presented in section 4. In this way, a peer or client resolves the conference URI by the SIP usage and validates the retrieved SIP Registration along the access control list.

The subsequent connection establishment to the focus then follows the protocol scheme defined in the SIP usage as shown in figure 25. The *FetchAns* message retrieved from the overlay contains a dictionary with the two SIP-Registrations of the type `sip_registration_route` (c.f., listing 4) for the focus peers A and B and the corresponding ACL. The default SIP usage behavior is to take all destination lists and sent an *AppAttach* request to each of listed node-ids within the lists. In the example, both focus A and B just stored a single node-id within the destination list. Accordingly, the joining peer sends the *AppAttach* to the node-id of focus A. The capacities to serve another user agent by focus A are exhausted and it will silently drop the received message. In this way, it emulates devices

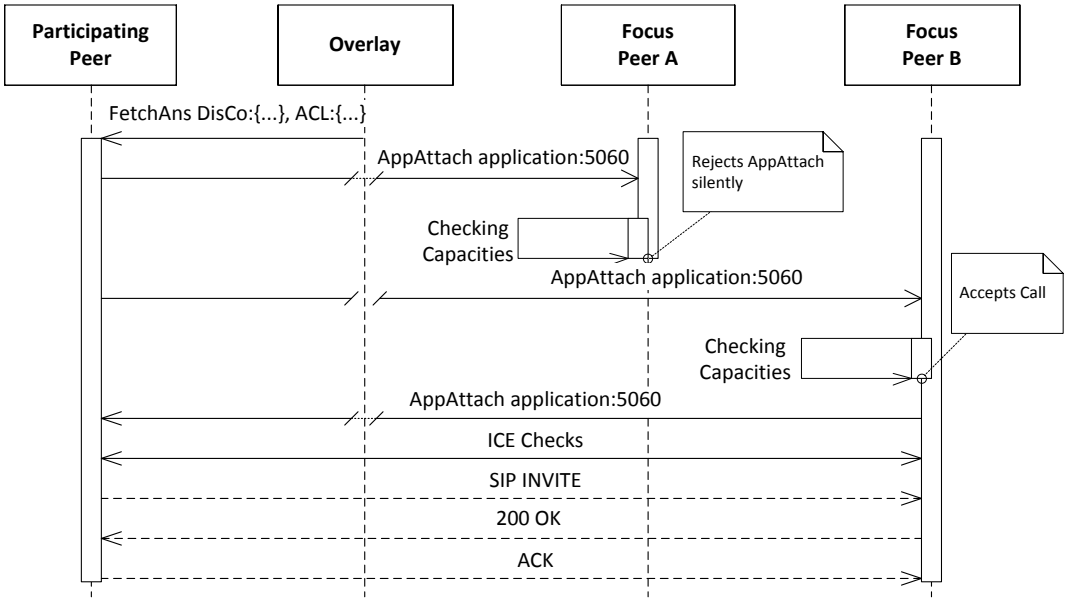


Figure 25: Message Flow: Joining a conference by the SIP-Usage

that is registered by switched off. The next *AppAttach* is sent to focus B. It determines its capacities and accepts the application request by responding with the corresponding answer message. The rest of the procedure for connection establishment is similar to the that of the DisCo usage. This protocol scheme is transparent to the requesting RELOAD application and thus backwards compliant to implementation just supporting the SIP and ShaRe usages.

5.4.2. Plain SIP User Agents

The major issue for a plain SIP implementation is the lookup for the conference URI to any focus peer serving the distributed conference. In general, the conference URI to a DisCo is registered only within a RELOAD overlay and disable plain SIP user agents to resolve the URI. In the following, we will discuss the alternatives to enable the participation for plain SIP user agents.

A straightforward solution is to pass the IP address of a focus peer by any external mechanism, e.g., Instant Messenger, E-Mail or a simple PSTN telephone call. The user then simply calls the conference URI by setting the domain to obtained IP address. The SIP application will generate a SIP INVITE request as shown in the following listing 14.

```
1  INVITE sip:disco@141.22.26.154 SIP/2.0
2  Via: SIP/2.0/UDP 141.22.26.238;branch=z9hG4bK2293940223
3  To: <sip:disco@141.22.26.154>
4  From: <sip:alice@atlanta.com>;tag=4321
5  Call-ID: 815@atlanta.com
6  CSeq: 1234 INVITE
7  Max-Forwards: 70
8  Contact: sip:alice@141.22.26.238
9  Content-Length: 123
10 ...
```

Listing 14: SIP INVITE: Legacy SIP user agent calling a DisCo

This solution does not require any SIP proxies, P2PSIP overlays and works on SIP signaling only. As focus peer are expecting incoming SIP calls anyway it identifies the INVITE message as request to join the distributed conference by the request-line and *To* header and can connects caller to the conference media.

However, determining the IP address of a device, sending it through any out of band mechanism to the caller and finally constructing the conference URI, is a complicated procedure with respect to usability. Hence, this could be an alternative for experts aware of SIP and IP addresses.

Another alternative approach to resolve a DisCo URI uses SIP proxies or redirect servers that act as gateways to a RELOAD overlay. The approach was initially mentioned in the Concepts and Terminology for Peer to Peer SIP draft [59] but not described it in more detail. Conventional SIP user agents communicate to the P2PSIP overlay through adapting peers which interact in between of the overlay and plain SIP. Those adapters accept SIP requests and resolve the next hop by using the RELOAD lookup functionalities. If the address is resolved to a destination, the adapter peer may proxies or redirects the caller to the destination.

By using gateway peers, a SIP INVITE request to join a distributed conference would be routed to a gateway peer that might be provided by the operator of the RELOAD instance. Then, the gateway translates the SIP request to a RELOAD *FetchReq* message requesting for conference URI as DisCo Kind, or alternatively as SIP Kind. The challenge is to select an adequate focus peers as entry point for the caller. The latter and the gateway peer will have no preferences, e.g. by relative network position to a certain focus.

A gateway acting as proxy or redirect could simply fork the SIP request to all focus peers by sending each an *AppAttach*. This would be equivalent to a default connection establishment as proposed by the SIP usage in which a peer sends an *AppAttach* to all received destination peers. The proxy could use the first established TCP connection to either proxy the initial SIP request or to redirect the caller to the remote endpoint of the TCP connection. All other

successfully establish transports to the remaining focus peers can be canceled. Alternatively, could a forking proxy use all established TCP connection to fork the initial SIP INVITE of the caller to all focus peers.

Even though forking SIP requests is a applied practice, could it cause unwanted domino effects in a distributed conference. Focus peers in a DisCo try to delegate calls to further conference controller as they are running out of capacities (c.f. section 5.5). A forked SIP request reaching several focuses that are overloaded produces multiple call delegations probably reaching a single and less loaded controller. Hence, a fork on SIP would produce avoidable call delegations.

The forking of a SIP request on the RELOAD layer followed by proxy or redirect operation could be a straightforward solution to enabled RELOAD-unaware SIP user agent to join a DisCo. The additional operations by adapter peers are fully transparent to the joining user and its SIP application. However, there are currently no detailed specifications or implementation for SIP-to-RELOAD adapter peers available. Hence, the usage of gateways peers is an area of research for future work.

5.5. Conference Management

5.5.1. Conference Access

The RELOAD base protocol [4] specifies several access control policies controlling write permissions onto the overlay network. Vice versa, are there no restrictions or policies regulating the read permissions on overlay resources. Hence, all legal peers and clients of an overlay can fetch any DisCo or SIP registrations and try to participate a distributed conference. As a consequence, it is desirable for distributed conference to specify a policy that defining who is allowed to participate and, as appropriate, to define conference credentials.

In a light-wight access model, the validation whether the originator of an incoming SIP call to a distributed conference is allowed to join, can be clarified verbally among the active parties. In a more advanced model, the participants of a conference are known previously to creation and listed within a conference object. The conference object can be represented by an XML document like the XML event package for distributed conferences presented in the later section 6. Thus, the DisCo applications of conference managers are aware of all permitted users and can automatically grant access to the multiparty session.

A third access model uses SIP authentication procedures to join a conference. The conference creator defines a shared secret that it passes to all parties of a conference. The users configure their DisCo application to use the shared secret to authenticate against the focus peers as shown in call flow 26. The participating peer resolved the conference URI and established a

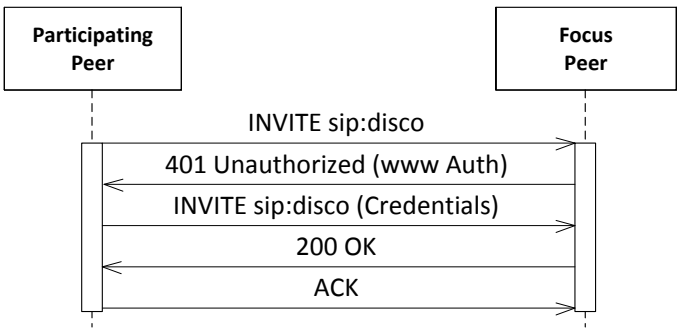


Figure 26: SIP Authenticate: Participant authenticating against focus peer

direct transport via RELOAD and sends a SIP INVITE to the chosen focus peer. The latter is aware of the conference policy and responds with a 401 Unauthorized message indicating the authentication scheme. The participating peer then re-originates a new INVITE request containing the conference credentials, e.g., a shared password to authenticate itself. Finally, both complete the transaction by sending 200 OK, respectively, the final ACK from the participating peer.

5.5.2. Call delegation

A main advantage of DisCo conferencing is the possibility to distribute the load to maintain the service onto several independent entities. In a worst case scenario, it is possible that many joining peers select the always the same focus as their conference entry point and thus exhaust its capacities to serve clients. To avoid overloading of focus peers, distributed conference control enables load-balancing by a mechanism for call delegation. Incoming participation requests are transferred to another established focus peer or conference participants that are registered as potential focus peers in the overlay. Call delegations use SIP REFER requests [64] that are achieved transparently to the transferred party. The transparency due to additional session information that is passed by the referring user agent and a source route set by the user agent accepting the REFER request. The sequence diagram shown in figure 27 shown a sample of a call delegation.

The referred peer RP had chosen the referring focus RF as its entry point to the distributed conference from a previous RELOAD lookup for the conference URI. On receiving the SIP INVITE, RF checks its own capacities to serve another conferencing client. Because RF reach its threshold, it temporarily accepts the call sending a 200 OK but sets the media *on hold*. This is achieved within the SDP answer setting the `a=` attribute to `sendonly` as shown in listing 15 line 8.

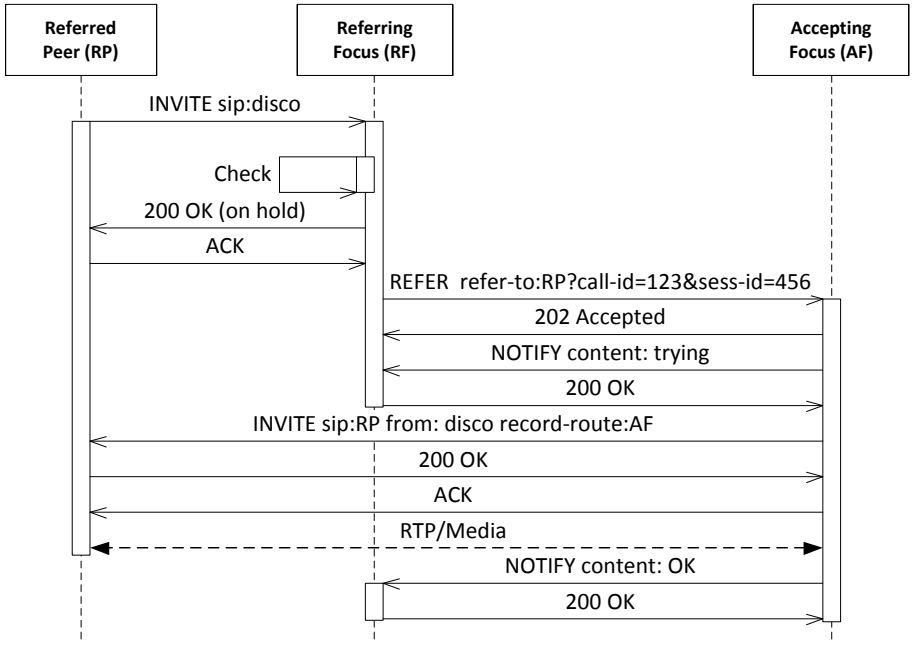


Figure 27: Call delegation: Transfer of a party due to load-balancing

Hence, after the ACK message of RP, no media session will be established. RF then delegates the call by sending a SIP REFER to the accepting focus AF. The URI carried within *refer-to* header field, has two additional parameter:

```
1 v=0
2 o=disco 456 456 IN IP4 141.22.26.27
3 s=
4 c=IN IP4 141.22.26.27/127
5 t=0 0
6 m=audio 49172 RTP/AVP 97
7 a=rtpmap:97 iLBC/8000
8 a=sendonly
```

Listing 15: 200 OK: SDP answer setting media on hold

call-id: This URI parameter contains the *call-id* the initial SIP dialog among RP and RF. It uniquely identifies the session and will be used by AF to the subsequent SIP re-INVITE.

sess-id: This URI parameter contains session identifier of the initial offer/answer. The identifier is the same value as the second o= (origin) parameter as shown in listing 15 in line

2. Note, that the origin field normally contains a timestamp of the Network Time Protocol (NTP) but is set to `456` to simplify the figure.

AF acknowledges the REFER request by responding a 202 Accepted followed by a SIP NOTIFY message indicating that the reference is pending. The accepting focus then re-INVITEs RP to the distributed conference, setting the call-id of the INVITE and the sess-id of the contain SDP offer to the values retrieved by the previous REFER request. Additionally, AF add an *record-route* header field to the re-INVITE that refers to it own SIP URI or contact address. A record route is generally added by *stateful* SIP proxies to remain in the path of further SIP requests within a dialog of user agent client. In case of a focus, the additional *record-route* header grants that all requests sent to the conference URI will be routed through the focus peer. A user agent client sending a SIP request must initially determine a so called *route set*. A route set is a collection of SIP URI though which a request must traverse. A *record-route* header forces a SIP user agent to add the URI contained in the header to the route set ordered along the path to destination. Hence, a user agent will route a request firstly to the URI or contact address of the focus peer in anticipation that the latter will route the message to the conference URI. Instead, if a focus receives a request addressed to the conference URI it will intercept it and respond it on behalf of the conference.

This represents a separation of the conference identifier and the locator of a conference in SIP. For the participating SIP user agent does the signaling relation appear as it is communicating to a single entity. But actually, after RF has transparently referred RP, RP is in a signaling relation with AF. The intention behind this ID/Locator separation is to remain compliant to the standards and hence to existing SIP implementations while performing a distributed conference. Just the SIP behavior of the focus peers must be adjusted to maintain a DisCo in SIP. Since further SIP requests enables him to maintain signaling with its participants in the role of focus.

After the 3-way handshake INVITE, OK, ACK the user agent RP and AF have negotiated their preferred media types and codecs and establish the media session. Finally, notifies AF the referring focus about the successfully call delegation.

5.5.3. Leave Management

A special issue in distributed conference control is the leave management of focus peers that are actively maintaining signaling and media connections to several conference parties. Depended on whether a focus peer is retired expected or unexpected, the DisCo specifies a proactive and a reactive leave management.

Proactive Leave Management A focus peer that leaves the conference is responsible to rearrange all its connected participants to the remaining conference controller. Therefore, a focus uses the global knowledge provided by the XML event package for distributed conference presented in section 6. It represents the entire state of a multiparty session and announces the local states of all other focus peers. This including their remaining capacities to serve further clients and their relative coordinates in the underlying network. Based on this global knowledge, a focus peer delegates all participants to the remaining focus peers as described in the following strategies:

Filling Vacancies: The first strategy is the delegation of participants to those focus peers that are actively managing the conference. Thereby, the delegating peer should consider the vacancies of those focus peers and their relative distance to the parties been delegated. By calculating the order of closest focuses for each participant and a subsequent distribution onto the further focus peers achieves the best delegation result with respect to management overhead and conference topology shifts.

Activating Focuses: If all remaining vacancies are exhausted by delegating participants, the leaving focus delegates the its client to the participants that are registered are registered as focus. It should redistribute the last remaining participants on as few potential focus peers as possible to avoid enlarging the diameter of conference topology. It thereby concerns the relative distance between the potential focuses and the participants. Furthermore it takes in account not to overload a focus by delegation to many parties as the delegating focus has no information about the capacities of those peers yet. However, as all peers in a conference should have more or less an equivalent computing power and hence the leaving peer will have to delegate very few parties.

As all or at least the most of the parties are rearranged to the other conference controller, the latter can leave the conference as described in section 5.3.3.

Reactive Leave Management If a focus peer leaves unexpected, e.g., by network or system failures the conference parties rejoin the conference. The following two strategies can be simultaneously performed by the lost participants and the remaining conference controller.

Rejoin: If a participant detects the loss of its responsible focus (e.g. missing signaling and/or media packets) it should repeat the joining procedure as described in section 5.3.2. This ensures at least an early rejoin to the distributed conference.

Re-invite: Focus peers detecting the disappearance of a focus will detect try to re-invite lost parties. The re-invites should not be performed concurrent with the other focuses since the INVITE requests by focus peers appear as a single entity for SIP user agent. Hence, should be re-invites performed according to proximity concerning the relative coordinates

of an user agent. If no proximity information for a peer is available, concurrent focus peers follow a tie-breaker heuristic to resolve the concurrency. The focus whose node-id is the numerically smallest should re-invite the lost parties to the maximum of vacancies it has. Thereafter, the peer with the second smallest node-id fills re-invites to peer until its maximal capacities, and so forth. The node-id of remote focus peers can be taken from the XML event package for distributed conferences.

5.6. Media Management

5.6.1. Model for Media Distribution

This section describes a basic scheme for media negotiation and distribution, which is done in an ad-hoc fashion. Each focus peer forwards all media streams it receives from the conference to all connected peers it is responsible for and similarly all streams from its peers to its responsible focus. This results in the media stream naturally following the SIP signaling paths.

When a new peer has been attached to a focus, new media streams may be available to the focus, which need to be forwarded to the conference. To accomplish this, the new media streams need to be signaled to the other participants. This is usually done by sending a SIP re-INVITE that modifies the media sessions by adding the new streams to the SDP. This renegotiation can be costly since it needs to be propagated through the whole conference. Also, distributing all media streams separately to all participants can be quite bandwidth intensive. Both problems can partially be mitigated by focus peers performing mixing of media streams, thus trading bandwidth and signaling overheads for computational load on focus peers.

5.6.2. Offer/Answer Model

A peer joining a conference negotiates media types and media parameters with its designated focus using the standard SDP offer/ answer protocol [52]. The focus should offer all existing media streams that it receives from the conference.

A new participant does not necessarily know about all media streams present in a conference beforehand, and thus some of the media streams might not be included in the initial SDP offer sent by the joining peer. An SDP answer sent by the corresponding focus can not contain any media streams not matching an offer (cf. RFC-3264 [52] Section 6). A joining peer which is aware of the distributed nature of the conference, should not send an SDP offer in the initial INVITE message, but instead send an empty INVITE to which the focus replies with an OK, containing the offer. This prevents the need for a second offer/answer-dialog to modify the

session. But for compatibility the normal behavior with the INVITE containing the complete offer must be supported.

For new media streams, (i.e., sent by the new participant) the focus should only offer media types and codecs which are already used in the conference and which will probably be accepted when forwarded to neighboring peers, unless the focus is prepared to do transcoding or mixing of the received streams.

A focus has two options when distributing media streams from a new participant to the conference. The focus can either mix the new streams into his own, thus averting the need to modify the already established media sessions with neighboring peers or in case the focus is not willing or able to do mixing of the media streams, he should modify the media sessions with all attached peers by sending a re-INVITE, adding the new media streams coming from the newly joined participant to the SDP. This should subsequently be done by all other focus peers upon receiving the new stream, resulting in the stream being distributed across the conference.

6. Management of a Coherent Conference State

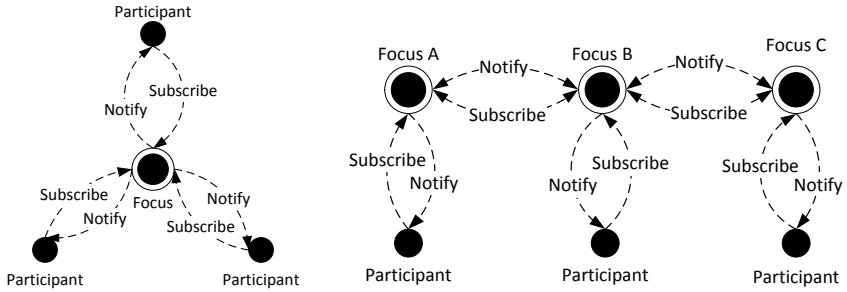
6.1. Introduction

The constitution of all distributed knowledge about signaling and media relations among the conference participants and focus peers defines the global state of a distributed conference. It is composed of the union of every local state at the focus peers. To enable focus peers to provide conference control operations that modify or require the global state of a conference, RELOAD usage for distributed conference control defines a mechanism for inter-focus state synchronization. Global knowledge is essential to comprehend the entire state of a conference in order to adapt on changes. For instance, the disappearance of a focus peer requires reconstruction procedures, which are only possible if all peers have an even knowledge base.

Even though the conference state could be announced via broadcast or multicast among the peers, the change event distribution is based on mutual subscriptions for SIP events. The reason for SIP events is the restricted scope of broadcast and multicast domains. A distributed conference can be performed on a global scale thus demanding a synchronization mechanisms not limited by provider boarders.

The Event Package for Distributed Conferences and allows to preserve a coherent knowledge of the global conference state. This XML based event package named *distributed-conference* must be supported by each DisCo-aware application to enable the state synchronization within a distributed conference. Participants of a distributed conference should subscribe for DisCo-event package to display the conference state to the user. To provide backward compatibility to conference members that do not support the distributed-conference event package, a translation to the Event Package for Conference State [18] called *conference-info* is provided.

In a previous work [65, 66] we proposed a focus synchronization by using event package for conference state that was augmented by a multi-focus extension. Further evaluations and feedbacks pointed out that an extended *conference-info* XML document does not meet the requirement for distributed conferencing. In particular, it does not fit race conditions while concurrent and maybe opposed change events. The coherency of the distributed conference state demands a versioning scheme is able to provide the partial ordering of concurrent change events. Furthermore, the DOM tree of conference-info package is not proposed to for concurrent write change events that could produce race conditions. This motivated the specification of a new event package adapted to the requirements of distributed conferencing.



(a) Central Notifier (b) Several Notifiers in DisCo

Figure 28: Distribution Models: Advertisement of change event in SIP

6.2. Distribution of Change Events

SIP event notifications [10] are generally built on a publish/subscribe model. A central service notifies a group of subscribed user agents about changes on the object of interest. In DisCo, the object of interest is the conference state which is partly distributed onto several independent entities each of them assuming the role of a notifier for change events. Hence, focus peers a that are actively managing a distributed conference are subscribers and notifiers for DisCo events. Figure 28 compares conventional event notification with the state synchronization in a DisCo. In traditional SIP conferencing 28a, the central focus has a complete view on the participating users as it solely maintains the signaling relations. The participants subscribe the focus for an event package, e.g., the conference-info package to obtain a similar view as the focus.

In a distributed conference 28b, the participants likewise subscribe to their focus peer to obtain notifications about changes in state. As shown in the figure 28b the focus peers A, B and C maintain each a subscription to their neighboring focus. Focus A had subscribed B and focus B subscribed focus A and C. If the state at focus A has changed, e.g., because it accepted a new participant, it sends B an SIP NOTIFY containing the state delta represented by the XML document. B will update its local XML document and will propagate the change event to focus C and to its subscribers. In this way, change events are flooded via the intermediate focus peers to the entire conference. This event propagation is loop free by using a version vector that maps the partial ordering of each change event. If a subscriber receives a change events several times, it can identify duplicates by the version vectors of the events.

The inter-focus communication paths are generally following the evolution of conference construction. A detailed example for such a scenario is shown in the call flow in figure 29. The participant B selected focus A as its entry point to conference after it fetched the DisCo-Registration from the RELOAD overlay. After the SIP 3-way handshake and media estab-

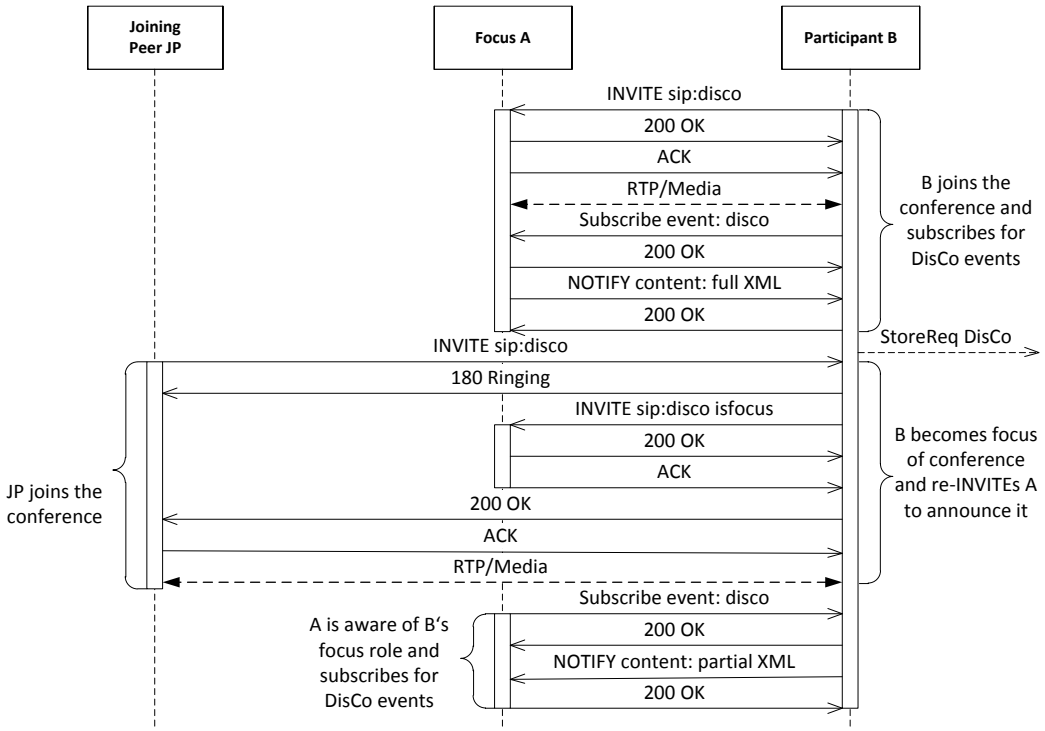


Figure 29: Mutual subscriptions: Party becoming focus and synchronizing state

lishment, B sends a SIP SUBSCRIBE request to A to obtain notifications for the DisCo event package indicated in the *Event* header field. After accepting, the focus A sends the first notifications that contains an XML document representing the entire state of the multiparty session. Subsequently registers B its RELOAD node-id in the overlay as indicated by the figure sending a *StoreReq*. The registration as potential focus peer may had based on the received conference state, for instance it recognized that focus A is reaching its threshold to serve additional clients.

Following, a joining peer JP selected B as its focus to participate in the conference by sending an INVITE request. B responds with a provisional 180 Ringing to acknowledge the retrieval of the call but will not answer the call yet. Before replying, B re-INVITES focus A to possibly renegotiate the media parameter and to publish its switch to focus role by adding the `isfocus` parameter to its contact address. Thereafter the new established focus peer B answers the INVITE by JP and connects him to the conference media.

Since focus A is aware of Bs change of role, it autonomously subscribes for DisCo events at B. The latter responds by sending its first SIP notification whose body contains the state delta from the last receive conference state to the last recent state including the appearance of JP.

By performing this subscription procedure, the synchronization paths naturally follow the process of evolution of a conference. In a worse case scenario, the resulting focus topology forms a chain of conference controller. However, the DisCo event package provides the global knowledge of all signaling and media relations among the peers in a multiparty session. This knowledge can be used to optimize the conference topology to a more sophisticated interconnection graph (e.g., tree or hypercube topology). Such optimization strategies are out of scope of this document and left to future work.

6.3. Event Package for Conference State

6.3.1. Overview

The *distributed-conference* event package is designed to convey information about roles and relations of the conference participants. Conference controllers obtain the global state of the conference and use this information for load balancing or conference restructuring mechanisms in case of a focus failure. Figure 30 gives a general overview of the DOM tree while appendix A contains the entire XML schema definition.

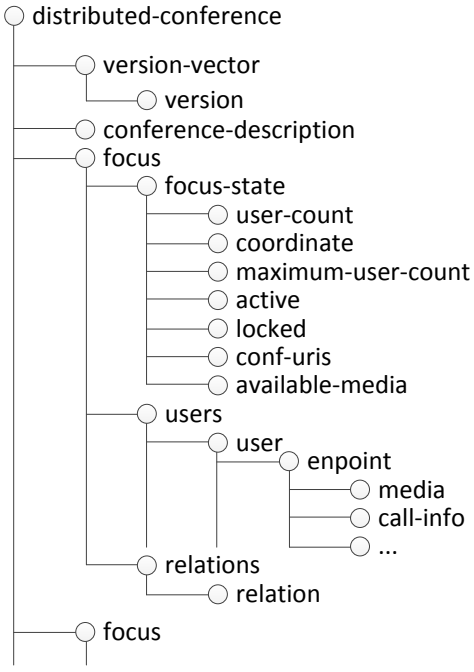


Figure 30: Overview: Event package for distributed conferences

The document structure is designed to allow concurrent change events at several focus peers. To prevent race conditions, each focus peer has exclusive writing permission to the *focus* sub element that describes itself. It is achieved by a unique mapping from a focus peer to its XML element using the *element keys* mechanisms for partial notification defined in the event package for conference state [18] and described in section 2.2.3. A focus peer is only allowed to update or change that <focus> sub element, whose *entity* element key contains its RELOAD username. This restriction also applies to the child elements of the *version-vector* element. Each *version* number belongs to a specific focus peer maintaining the version number.

The local state of a focus peer is described within a *focus* element. It provides general information about a focus peer and its signaling and media relations to participants and focus peers. The conference participants are aggregated within *users* elements, respectively, *user* sub elements.

General information about the conference as a whole, is provided within a *conference-description* element. In contrast to the *focus* and *version-vector* elements, conference description is not meant for concurrent updating. Instead, it provides static conference descriptions that rarely change during a multiparty session.

More detailed descriptions about the event package and its elements are provided in the following sections.

6.4. Description of XML Elements

6.4.1. <distributed-conference>

The <distributed-conference> element is the root of a distributed conference XML document. It uses the following attributes:

entity: This attribute contains the conference URI that identifies the distributed conference. A SIP SUBSCRIBE request addressed to this URI initiates an subscribe/notify relation between participants and their related focus peer.

state: This attribute indicates whether the content of a distributed conference document is a *full*, *partial* or *deleted* information. It is in accordance with the conference-info event package [18] to enable the partial notification mechanism.

The <distributed-conference> child elements are <vector-version>, <conference-description> and the <focus> elements. An event notification of state *full* includes all these elements. An event notification of state *partial* contains at least <version-vector> and its sub elements.

6.4.2. <version-vector>/<version>

Background Version vectors are based on vector clocks [67] and are used to follow the changes of data in concurrent distributed system. It enables the receiver to determine an update precedes another, succeeds it or if a local copy and the received are concurrent. Hence, systems receiving data that uses vectors for version control can track the causality of retrieved data. Therefore, each endpoint in the distributed system exclusively maintains its own version number in the vector that is incremented by one if the local state changes. As a result, version vector V has n elements with n is the number of entities in the system. An element i in a vector accordingly represents the updates generate at a System S_i. On propagating the update to the other systems, the entire version vector its attached to enabled the receivers to validate the causality. An example [68] is shown as follows:

$$System\ 1 \quad V_1 = \{2,0,3\} \tag{2}$$
$$System\ 2 \quad V_1 = \{2,1,3\} \tag{3}$$
$$System\ 3 \quad V_1 = \{2,1,5\} \tag{4}$$

A vector V_x is greater a vector V_y if at least one element i is greater and all remaining elements of V_x are at least equal to V_y. Thus in the example applies $V_1 < V_2 < V_3$ identifying system 1 as out of date concerning the local states of systems 2 and 3. Assuming that the vector of system 2 had been $V_1 = \{2,2,3\}$, the vectors V_2 and V_3 were concurrent and demand a resolution.

Version-vector Element The event package for distributed conferences uses the <version-vector> and its <version> sub elements to enable a scheme for coherent version control. Each <version> element contains a unsigned integer that describes the state of a specific focus peer and contains the following attributes:

entity: This attribute contains the username of the focus peer whose local version number is described by this element.

node-id: This attribute contains the node-id of the focus peer.

Whenever the local status of a focus peer changes (e.g., a new participant arrived) the version number of the corresponding <version> element is incremented by one. Each change in the local state also triggers a new event notification containing the entire <version-vector> and the changes contained in a <focus> element.

The recipient of a change event needs to update it local XML document. If a received <version> number is higher that the local, it updates the <version> element and its associated <focus> element with the retrieved elements. All other elements remain constant.

If the length or any version number of the retrieved <version-vector> is different from the local copy it indicates a incoherent knowledge about the entire conference state. If the retrieved <version-vector> contains any unknown focus peers and any local version numbers for the known focus peers is lower, the receiver requests a *full* XML notification.

If any local <version> number is retarded more than one, the receiver requests a *full* event notification from the sender. The full state notification updates all <focus> elements whose corresponding <version> element is out of date.

The version vector is further designed to merge concurrent vectors. Each vector number also contains the URI of the focus peer. Thus the receiver of a concurrent vector can comprehend which elements of its local state are greater or lower than of the received element.

6.4.3. <conference-description>

The <conference-description> element provides general information and links to auxiliary services for the conference. The following sub elements provide human-readable text descriptions about the conference:

<display-text>: Contains a short textual description about the conference.

<subject>: Contains the subject of the conference.

<free-text>: Contains a longer textual description about the conference.

keywords: Contains a list of keywords that match the conference topic. The XML schema definition and semantic is imported from the *conference-info* event package [18].

The <service-uris> sub element enables focus peers to advertise auxiliary services for the conference. The XML schema definition and semantic is imported from the *conference-info* event package [18].

The <conference-description> element uses the *state* element key to enable the partial notification mechanism.

6.4.4. <focus>

Each <focus> element describes a focus peer actively controlling the conference. It provides general information about a focus peer (e.g., display-text, languages, etc.), contains conference specific information about the state of a focus peer (user-count, available media types, etc.) and announces signaling and media information about the maintained participants. Additionally, it describes signaling or media relations to further focus peers.

The <focus> element uses the following attributes:

entity: This attribute contains the username of the RELOAD peer acting as focus peer. It uniquely identifies the focus peer that is allowed to update or change all sub elements of <focus>. All other focus peers are not allowed to update or change sub elements of this <focus> element. A SUBSCRIBE request addressed to the username initiates a conference state synchronization with the focus peer.

node-id: This attributed contains the node-id of the peer acting as conference focus.

state: This attribute indicates whether the content of the <focus> element is a *full*, *partial* or *deleted* information. A *partial* notification contains at maximum a single <focus> element.

Following, a detailed description of the sub elements of the <focus> element.

<focus-state> The <focus-state> element aggregates a set of conference specific information about the RELOAD user acting as focus peer. It uses the *state* attribute to enable the partial notification mechanism and has the following sub-elements:

<user-count>: This element contains the number of participants that are connected to the conference via this focus peer at a certain moment.

<coordinate>: This element contains the coordinate value of the focus peer which is stored in the overlay.

<maximum-user-count>: This number indicates a threshold of participants a focus peer is able to serve. This value might change during a conference, depending on the focus peers current load.

<conf-uris>: This element can contain other conference URIs in order to access the conference via different signaling means. The XML schema definition and semantic is imported from RFC4575 [18].

<available-media>: This element is imports the <conference-media- type> type XML scheme definitions from RFC4575 [18]. It allows a focus peer to list its available media streams.

<active>: This boolean element indicates whether a focus peer is currently active managing the conference. If this value is false, the focus peers is registered and ready to accept incoming participation requests.

<locked>: In contrast to <active>, this element indicates that a focus peer is not willing to accept further participation or call delegation request.

<users>/<user> The <users>, respectively, each <user> element describes a single participant that is maintained by the focus peer described by the parent <focus> element. The <users> element XML schema definition and its semantic is imported from the *conference-info* event package [18]. The <users> element is briefly described in section 2.2.3, however, a detailed description of this elements is out of scope in this document.

<relations>/<relation> The <relations> element serves as container for <relation> elements, each describing a specific connection to another focus peer. It enable a conference controller to trace back the entire interconnection graph among the focus peers. The parent element <relations> uses the *state* attribute to enable the partial notification mechanism and the following attributes are defined:

entity: This attribute contains the user name of the remote focus.

node-id: This attribute contains the node-id of the remote focus peer.

The content of each <relation> is a comma separated string that describes the tuple <CONNECTION-TYPE:IDENTIFIER>. The CONNECTION- TYPE is a string token describing the type of connection (e.g. media, signaling, etc.) and the IDENTIFIER contains a variable connection identifier. It is a generic method to announce any kind of connection to a remote focus. This specification defines following tuples:

<media:label>: This tuple identifies a single media stream. The *label* variable contains the SDP *label* attribute [18] that was used within the offer/answer process while joining the conference.

<sync:call-id>: This tuple indicates a specific subscription for the event package for distributed conferences. The 'call-id' variable contains the call-id of the SIP subscription dialog. Using this relation, a focus peer can track back the mutual subscriptions for DisCo event among all conference controller.

6.5. Translation to Conference-Info Event Package

The DisCo event package described in the previous sections, imports several XML element definitions of the Event Package for Conference State [18] due to two reasons. First, the semantic of these elements are matching the demands to describe the global state of a distributed conference and, second, it facilitates a backwards-translation to the *conference-info* event package. This enables a backward compatibility to DisCo-unaware SIP implementations. Each focus peer can provide a separate notification service granting to conventional SIP user agents a view on the conference state.

The following sections describe the translation to the Event Package for Conference State by defining translation rules for the root element and its direct sub-elements. For a better understanding, the following sections use a notation ci.<ELEMENT> to refer to a sub-element of the *conference-info* element, and disco.<ELEMENT> to refer to an element of the distributed-conference event package.

6.5.1. ci.<conference-info>

The root element of Event Package for Conference State uses the attributes *entity*, *state* and *version* and is the counterpart of the <distributed-conference> root element in the DisCo Event Package. The former two attributes *entity* and *state* are equal in both root elements and can be seamlessly translated.

The *version* attribute is incremented by one at any time a change event is detected locally or by an event notification of a remote focus.

6.5.2. ci.<conference-description>

The <conference-description> element exists in both event packages, conference-info and distributed-conference. Thus, the following elements are seamlessly translatable: <keywords>, <display-text>, <subject>, <free-text> and <service-uris>.

The sub elements <conf-uris>, <maximum-user-count> and <available- media> in conference-info have there counterparts below the /focus/focus-state element of the distributed-conference event package. Each describes a local state of a focus peer in the conference. Hence, the intersection of every disco.<conf-uris>, disco.<available-media> and the sum over each disco.<maximum-user- count> element of each disco.<focus> element in distributed- conference, specifies the content of the corresponding conference- info elements.

6.5.3. ci.<host-info>

The ci.<host-info> element contains information about the entity hosting the conference. For participants in a distributed conference, the hosting entity is their focus peer. Thus, the ci.<host-info> element contains information about the focus peer.

6.5.4. ci.<conference-state>

The ci.<conference-state> element allows subscribers obtain information about overall state of a conference. Its sub elements ci.<user-count>, ci.<active> and ci.<locked> are reused as sub elements of
focus
focus-state to describe the local state of a focus peer in a distributed conference. The translation rules from the distributed-conference to the conference-info event package are the following:

<user-count>: The sum over each value of the disco.<user-count> element defines the corresponding ci.<user-count>.

<active>: The boolean ci.<active> element is the logical concatenation over all disco.<active> elements by an OR-operation.

<locked>: The boolean ci.<locked> element is the logical concatenation over all disco.<locked> elements by an AND-operator.

6.5.5. ci.<users>/ci.<user>

The distributed-conference event package imports the definitions of the ci.<users> and ci.<user> elements under a parent disco.<focus> element for each focus peer in a conference. Thus, the aggregation over all disco.<users> elements specifies the content of the corresponding ci.<users> element.

6.5.6. ci.<sidebars-by-ref>/ci.<sidebars-by-value>

The *conference-info* event package provides a representation for users in a conference that are simultaneously joined within another *sidebar* conference. The DisCo event package does not provide a sidebar representation thus for the ci.<sidebars-by-ref> and ci.<sidebars-by-value> apply the rules as specified in RFC4375 [18].

7. Implementation

7.1. Libraries to Implement DisCo

The protocol scheme for distributed conferencing defines behaviors and data structures for the usage on the SIP [1] and RELOAD base protocol [4]. Since the Session Initiation Protocol is an established and widely deployed signaling protocol, several commercial and open source implementations exist in almost every programming language. For example, the implementation of an early DisCo SIP scheme called *SDCON* [69] was implemented on the protocol-level SIP Stack *JainSIP* [70] for Java.

In contrast to SIP, the RELOAD base protocol is still an Internet draft and has had have several changes in the past. This might be a reason why there were very few implementations of RELOAD and most of them were not published. The only partial implementation of RELOAD was published by Amit Ranpise [71] and just covered the data storage module and a RELOAD message parser written in C++. Hence, the circumstances to implement and prove the DisCo concepts were not preexisting. Finally, we obtained an advanced RELOAD implementation by a cooperation with *TLabs/T-Systems* of the Deutsche Telekom AG.

7.1.1. MP2PSIP Project aka RELOAD.NET

The used RELOAD stack implementation is the *MP2PSIP* project. It was initially developed as a confidential project within the Deutsche Telekom AG, Laboratories to evaluate the usage of a P2PSIP network that is supporting VoIP infrastructure. The project is scheduled to be published soon as open source stack named *RELOAD.NET*. The protocol stack is an .NET/C# implementation that includes the modules as shown in figure 31.

RELOAD Class: The RELOAD Class subproject comprises the core implementation of the RELOAD base protocol. This includes a usage layer, the message transport, storage, the topology plugin implementing Chord [72] key-based routing and the forward and link-management controlling the underlying transport layer. The core of the stack is not a full RELOAD implementation. It omits, e.g., the implementation of Interactive Connectivity Establishment (ICE) [36] used for NAT traversal, due to ICE itself is a new protocol standard and thus is not available in any programming language. In previous works [73, 74], the MP2PSIP project (aka RELOAD.NET) had been extended with several functionalities to be suitable for a implementation of DisCo and Share usages. It includes an abstract storage and usage layer to enabling the implementation of further application on top of RELOAD.NET as well as optimizations in the key-based routing layer.

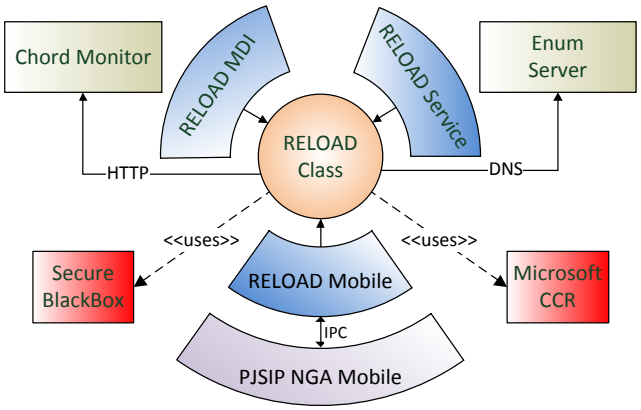


Figure 31: RELOAD.NET: Component Overview

RELOAD MDI: The RELOAD MDI is a graphical user interface for emulating and debugging the core implementation. It enables to start several RELOAD peers/clients on a single device and to achieve simple store and fetch operations.

RELOAD Service: The RELOAD Service is a subproject to deploy the core implementation as Windows Service. It is designed to run, e.g., as bootstrap peer.

RELOAD Mobile: The mobile module of the RELOAD.NET stack contains the RELOAD Mobile subproject and the PJSIP NGA softphone. The former is a variant of the core implementations using the same source code but adapted to the .NET *compact* framework. The adaption to Windows Mobile 6.X devices is mostly handled by pre-compiler directives, e.g., an adaption to the different environment variables of Windows and Windows Mobile. The PJSIP NGA is a SIP client based on the PJSIP SIP stack [75] written in C++. It provides regular VoIP using SIP for signaling, as well as the registration and lookup functionalities of RELOAD.

Further related to the RELOAD.NET protocol stack is a monitoring tool *Chord Monitor* and enum server. The enum server is a ordinary *bind9* daemon that translates fully qualified phone numbers to SIP identifier. It is used to enable a user to call his buddies via SIP/RELOAD by typing their mobile phone number. The Chord Monitor is an ASP.NET and JavaScript project based on the Google Maps [76] API. It renders all RELOAD peers and clients on top of a Google map [5] and displays each 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 overlay typical ring topology.

The core classes use to two external libraries to provide the RELOAD service. The *Eldos Secure BlackBox* (SBB) [77] and the Microsoft Concurrency and Coordination Runtime (CCR) [78]. SBB is a commercial security framework for .NET/C#. The stack uses its capabilities for TLS [42] secure transport connections and the handling of X.509 certificates [79]. The MS CCR is a library from out the Microsoft Robotics Studio. It facilitates the handling of concurrent procedures by providing a dispatcher class managing multiple method delegates passed to the dispatcher via queues.

However, the RELOAD.NET stack is still work in progress any several features are not stable yet. The implementation and the further evaluation of the DisCo protocol scheme were hampered by bug-fixing the RELOAD stack. Nevertheless, it is still one of the most advanced RELOAD base implementations and may by improved by publishing it as open source software.

The RELOAD.NET stack can be initialized by creating a new instance of the `Reload.Machine` class as shown in listing 16. 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 16). 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.

```
1  using TSystems.RELOAD;
2  namespace ReloadTest {
3    class Program {
4      static void Main(string[] args) {
5        Machine machine = new Machine();
6        // Set Logger delegate
7        machine.ReloadConfig.Logger = new ReloadConfig.LogHandler(Logger);
8      }
9      void Logger(ReloadGlobals.TRACEFLAGS traces, string s) {
10       /* Implementation of Logger */
11     }
12 }
```

Listing 16: Initialization: The RELOAD machine class

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.

The initial MP2PSIP project was not designed to serve any application built on top of it via an adequate interface. In an older version, the stack had only communicated via the Windows registry to the PJSIP NGA mobile VoIP application. To enable a communication from the RELOAD stack to the upper application three method *delegates* were added to the `Machine` class as shown in listing 17.

```
1 /* Returns the dialog object containing the transaction parameter */
2 public delegate bool DStoreCompleted(ReloadDialog dialog);
3 /* Returns the list of Usage objects, representing application data */
4 public delegate bool DFetchCompleted(List<Usage> usageResult );
5 /* Returns an ICE candidate containing an IP address and port */
6 public delegate void DAppAttachCompleted(IceCandidate ice);
```

Listing 17: Delegates: Enabling application to receive events

The `DStoreCompleted` delegate returns on a store answer message a `ReloadDialog` object. It contains in example, the source and destination addresses (node-ids), the transaction-id and a status indicating the success of the store request. A list of `Usage` instances is returned to the application on a successful fetch request. A Usage is an abstract interface whose implementations provide several methods to handle the received data. For example, each usage implementation does de-/serialization of the RELOAD Kind data structures and implements the procedures to obtain the destination address requested by the application. The address of the destination has finally returned within an `ICECandidate` object returned by the `DAppAttachCompleted` delegate. The ICE candidate including an `IPAddress` object is used by the application to create the application session. This is a design decision to facilitate the subsequent signaling procedures and not following the RELOAD base protocol [4] specifies. Following RELOAD, the `DAppAttachCompleted` delegate should return an initialized socket object used by the SIP stack to create a INVITE request. However, it is difficult to pass an C# socket to the used SIP stack natively written in C. The reasons for a native C SIP stack are explained in the following section.

7.1.2. PJSIP Stack/ Sipek Wrapper

The second protocol used by the DisCo is the Session Initiation Protocol [1]. Hence, to implement a prototype for distributed conference control, it demands a SIP stack ideally written in C# for better compatibility to the RELOAD.NET stack. However, are most of the available SIP stack for .NET under a commercial license. An alternative to a native C# implementation is a C#

wrapper of an open source project written in C/C++. A wrapper recommended wrapper facade is the *pjsipDll/Sipek* project under GPL2 license and published as *Google Code*. The pjsipDll is a plain C wrapper interface to the high level API of the PJSIP stack [75]. The PJSIP SIP stack is an advanced SIP implementation including libraries for media negotiation and transport via SDP/ RTP [7, 53], utilities for NAT traversal and implements several SIP standards[4]. The Sipek project is build on the pjsipDll wrapper and provides an API for call establishment and messaging.

The Sipek API provides the `CCallManager` class as main interface to the underlying *pjsipDll* wrapper and respectively, the native PJSIP stack. Listing 18 shows a minimal initialization of the stack.

```
1  void InitAndCall(string uri, IPhoneConfig config) {
2    /* Create call manager through a singlton */
3    var manager = CCallManager.Instance;
4    /* Add method delegate to receive events */
5    manager.CallStateRefresh +=
6              new DCallStateRefresh(CallManager_CallStateRefresh);
7    manager.IncomingCallNotification +=
8              new DIncomingCallNotification(IncomingCallNotification);
9    /* Init registrar and add a method delegate */
10   pjsipRegistrar.Instance.AccountStateChanged +=
11             new DAccountStateChanged(Instance_AccountStateChanged);
12   /* Configure manager, registrar and proxy */
13   manager.Config = config;
14   pjsipRegistrar.Instance.Config = config;
15   pjsipStackProxy.Instance.Config = config;
16   /* Initialize */
17   manager.Initialize(pjsipStackProxy.Instance);
18   /* Call */
19   manager.CreateSmartOutboundCall(uri, 0);
20 }
```

Listing 18: Sipek: Initiation call manager

The reason for two independent project libraries is caused by different behaviors of the .NET runtime and Mono runtime environment. In particular uses the Mono variant other binaries as the .NET version. The underlying TLS stack of the RELOAD Class sub-project is a dedicated Mono build. The pjsipDll whose sources are native C are specially compiled for Linux OS based systems to obtain dynamic *.so* library instead of an *.dll* binary. The remaining sub-projects are described as follows:

[4] See complete implementation list on the pjsip homepage [80]

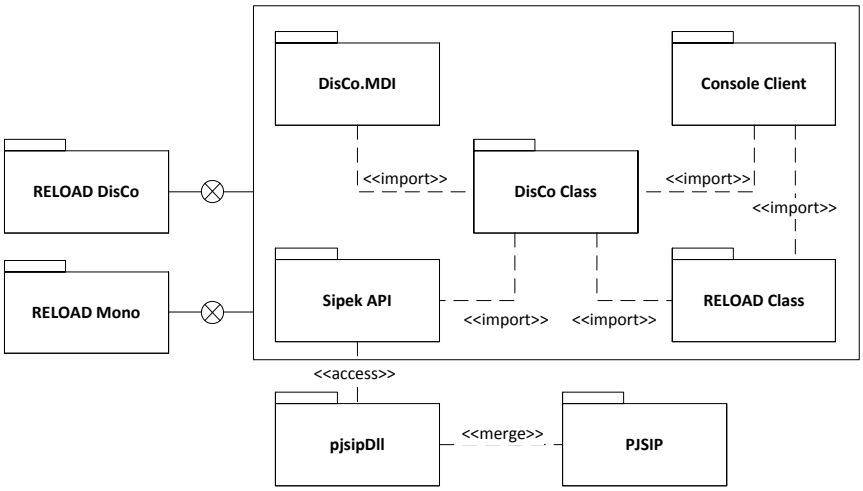


Figure 32: Packet Diagram: Project overview

DisCo MDI: The DisCo MDI project provides graphical user interface for creating SIP calls and conferences using RELOAD base protocol and plain SIP. It imports the DisCo Class project to perform SIP calls and RELOAD storage and lookup functions.

Console Client: The Console Client component is a terminal application designed for testing and automated execution. It can be configured to display a user an console interface to perform RELOAD operations (join, store, fetch, etc.). Alternatively, it can be configured to accepts console parameters to perform a predefined behavior. E.g., it can start as bootstrap node or initial creator of a distributed conference. The console client imports the DisCo class project to be enabled to run the DisCo and Share usages and additionally imports the RELOAD Class directly. This additional import enables the console client to directly invoke particular functions for the RELOAD Class project for testing and evaluation.

DisCo Class: The DisCo Class project is class library and implements the DisCo and ShaRe usages. It further provides an additional abstraction layer to the applications importing this class library. E.g., the storage of a DisCo-Registration [6] is implemented as a single invocation of a method that performs the steps of the registration procedure as described in section 5.3.1. It imports the Sipek API to perform call establishment and the Reload Class library for P2PSIP operations.

Sipek API: The Sipek class library provides an interface to the callback functions of the pjsipDll. It is added to the RELOAD DisCo/Mono project library to directly edit its source to enable the implementation of the SIP protocol scheme for distributed conferences.

RELOAD Class: This sub-project is the core RELOAD implementation of the RELOAD.NET project as described in section 7.1. The core library was continuously improved and extended during implementation of the DisCo protocol.

The external components of the project are the pjsipDll and the PJSIP stack. They are complied once for each operating system and the created dynamic libraries are accessed by the Sipek API.

7.1.3. XML Schema Converter

The RELOAD overlay configuration document is specified within a complex XML schema in Relax NG notation. An efficient method to obtain a class representation in source code is the usage of *XML to source* tools. The MS Visual Studio environment provides an *xsd.exe* tool to convert XML document to XML schema, and to convert an XML schema to C# or Visual Basic representation. However, is the *xsd.exe* tool not able to parse and process the Relax NG schema notation. This limitation was solved by the usage another XML tool called *Trang* [81]. Trang is an open source tool for XML schema conversation.

To obtain a C# representation for XML schema defined the ShaRe and DisCo [5, 6] and RELOAD base [4], all three Relax NG schema were combined in one *.rng file and converted via Trang to an XML schema. The latter was then used to generate a C# source code representation. For a further usage in the application, an retrieved XML document can be parsed to a C# class using the .NET `XMLSerializer` class. A sample of its usage is shown in listing 19. The `overlayelement` is the class representation of the RELOAD configuration document generated by `xsd.ex` and is passed to the serialized to serialize the obtain XML to an object.

```
1 var serializer = new XmlSerializer(typeof(overlayelement));
2 var document = (overlayelement)serializer.Deserialize(tr_xml);
```

Listing 19: XML Serializer: Obtaining a class representation of the configuration document

7.1.4. DisCo Class Design

The class library for distributed conference control combines the two class libraries Sipek and RELOAD Class to an interface for the usage by applications. Therefore it implements two connectors to each of those libraries and an API to the application. The component diagram shown in figure 33 illustrates the internal structure of the DisCo Class project. It includes the

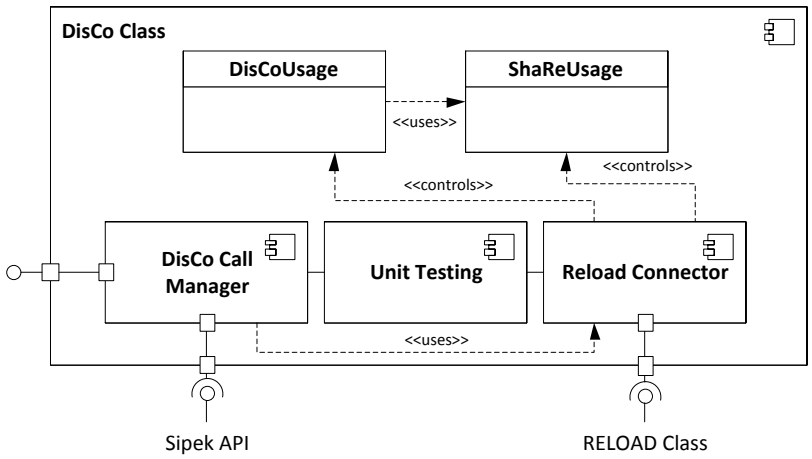


Figure 33: Component Diagram: Internal structure of the DisCo Class project

implementation of the RELOAD DisCo and ShaRe usages, a `DisCoCallManager` component and the connector interface to the RELOAD Class project. Additionally, it provides a unit testing module to prove the correct implementation of several methods.

The `DisCoUsage` and `ShaReUsage` classes are implementing the RELOAD protocol procedures to register and resolve a conference identifier for a DisCo. Both classes implement an abstract *Usage* interface thus to be compliant to the RELOAD core implementation. That latter provides a `UsageManager` class that is build as a factory pattern. The `Reload Connector` component registers both usage classes at the manager class of the RELOAD core and further listens on the method delegates shown in listing 17 to obtain RELOAD events.

The `DisCo Call Manager` component is instantiates the Sipek SIP stack and augments it with the RELOAD/DisCo protocol. It maintains a reference to the Reload connector component to register conference IDs in the RELOAD overlay that are obtained from the user application, e.g., the DisCo MDI graphical user interface.

7.2. Implementation of Usages

7.2.1. Shared Resources

The RELOAD usage for Shared Resources [5] as described in section 4 consists mainly of three aspects listed below:

- An access control list (ACL) data structure each representing a trust delegation from one overlay user to anther user
- An algorithm to create an array index that separates each individual ACL registration from concurrent writing
- An access control policy regulating the store attempt on a shared resource

ACL Data Structure The first item is implemented by the two C# structs `ACLItem` and `ResNameExtension`. C# provides the possibility to add contractors, methods to structs. Both structs use this feature and implement a constructor to directly set all field on instantiation as shown in listing 20. This allows a more comfortable and save instantiation of the structure if all values for an ACL are known. Otherwise a developer would have to fist declare the structure and subsequently assign each value and may forgets to set a data field.

```
1  public struct ACLItem {
2      public ResNameExtension resNameExt;
3      public string toUser;
4      public UInt32 kindId;
5      public Boolean ad;
6      /* Struct contructor for the ACL Item */
7      public ACLItem(ResNameExtension resNameExt,
8                     string toUser, UInt32 kindId, bool ad) {
9        this.resNameExt = resNameExt;
10       this.toUser = toUser;
11       this.kindId = kindId;
12       this.ad = ad;
13     }
14 }
```

Listing 20: ACLItem: C# struct represention

Forming the Array Index The `ACLItem` struct is maintained within a `ShareUsage` implementing the `Usage` interface. The latter is one of the improvements of the RELOAD.NET stack done in a previous work [74]. One of the methods a `Usage` realization has to implement is the *Encapsulate()* method. The function of this method is to wrap a Usage implementation into an abstract `StoreDataValue` object thus the underlying RELOAD core implementation must be aware of the concrete Usage object it will send or receive. Further, a stored data value is carrying the meta data specified for the data model of an RELOAD Kind. In the case of the ACL item which is an array, the `StoredDataValue` object has to be initiated with array index and the mandatory *exists* flag. Hence, the *Encapsulate()* method has to implement the algorithm for creating the individual array index. The realization is shown in listing 21.

```
1  public StoredDataValue Encapsulate(Boolean exists) {
2        /* Obtain the peer node-id from the UsageManager */
3        var localId = myManager.m_ReloadConfig.LocalNodeID;
4        byte[] bIndex = new byte[4];
5        /* Obtain the most significant 24 bit of my node-id */
6        Array.Copy(localId.Data, bIndex, 3);
7        /* Add the indivial index */
8        UInt32 index = myManager.MyArrayManager.CurrentIndex(
9            new ResourceId(item.resNameExt.rname), this.item.KindId);
10       bIndex[3] = (byte)index;
11       /* Convert byte array to UInt32 valie */
12       index = BitConverter.ToUInt32(bIndex, 0);
13       /* Create StoredDataValue object an return it to stack */
14       return new StoredDataValue(index, this, exists);
15 }
```

Listing 21: ACL Index: ShaReUsage object creating array index

The *Encapsulate* method obtains the information by its invoker, if this ACL item contains data (extist = True) or if this ACL item will overwrite an existing ACL will null (exist = False). The latter case is the default RELOAD operation to remove values from the overlay. To calculate the array index, the method obtains the node-id of the peer from its `UsageManager` object as shown in line 3. Via the array copy operations, the method obtains the most significant 24 bits of the node-id. The bits will delimiter the array index from array items stored by further peers accessing the shared resource as all perform the same operation. Following, the ShaRe usage implementation request its manager for current array index. This operation must be performed by the usage manager since it is the only component that maintains a global view on all usages. Therefore, the manager needs the resource-id and kind-id to increment the corresponding counter. Afterwards, the index be concatenated to the previous 24 bits and re-converted to an UInt32 as shown in lines 10 to 12. Finally, the index, exist flag and the usage object are passed to the constructor of the `StoredDataValue` and returned to the RELOAD stack.

Request Authorization A further mechanism defined in the usage for shared resources is the authentication of the originator of an inbound store request or fetch answer. The validation rules defined in RELOAD access control policies need to be performed by the peer responsible for storing the data and it is recommended that fetching peers validate the provenance of the retrieved data. Acces control policies are realized in the by core RELOAD.NET project via an `AccessController` class. The design and implementation of this component was part of the improvements developed during a previous work [74]. On instantiation of the controller class, it just provides the functionality to prove the integrity and prove-

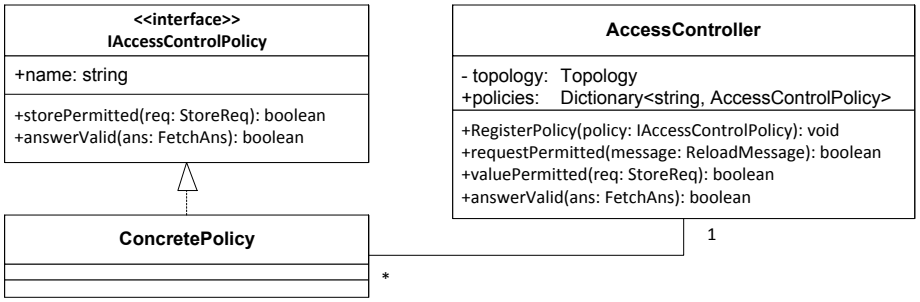


Figure 34: Access Control: Factory pattern for adding new policies

nance of the sender of the request, but not functions to validate the access control policies for the Kind data that request contains. The latter functionalities are obtained by implementations of an `IAccessControlPolicy` interface whose realizations are registered at the `AccessController` class. This factory like pattern is shown in figure 34. A realization of the `IAccessControlPolicy` must have a public property returning the name of the concrete policy, a *storePermitted()* and a *answerValid()* method. The name of an access policy is the string token defined in the corresponding Internet draft or RFC to a RELOAD usage. In case of the ShaRe usage, the corresponding string token is `"USER-CHAIN-ACL"`. This enables a unique mapping for the access controller class. The latter is aware of mapping kind-id to the name of an access control policy () from the RELOAD configuration document. Hence, if it receives an inbound request, it obtains the kind-id of the contained data, maps it to the name of the access control policy and invokes the corresponding `IAccessControlPolicy` instance to validate the retrieved Kind data.

The reason for a separate method for due to different procedures to obtain all data to validate a data value. In case of the ShaRe usage, a storing peer maintains the entire access control list locally. A peer requesting for data, has to fetch all ACL items from the storing peer to perform the ACL validation procedure. The latter is implemented as follows 22:

```
1 pivate Boolean ACLvalidation(List<GenericCertificate> certs,
2     List<StoredData> aclList, StoredData sharedRes) {
3   Boolean inList = false;
4   Boolean userEqToUser = false;
5   signerId = sharedRes.signature.Identity;
6   X509Certificate2 signerCert = CertTools.GetPKC(certs, signerId);
7   string username = CertTools.GetRFC822Name(signerCert);
8   string toUser = "";
9   while (!userEqToUser && !inList) {
```

```
10      var aclDepth = 0;
11      foreach (StoredData sd in aclList) {
12        var share = (ShareUsage)sd.Value.GetUsage;
13        var item = share.Kind;
14        if (item.toUser == username) {
15          if (sharedRes.kindId == ACLItem.KindId && !item.ad) {
16            inList = false; break;
17          }
18          if (aclDepth > 0 && !item.ad) {
19            inList = false; break;
20          }
21          signerCert = CertTools.GetPKC(certs, sd.Signature.Identity);
22          toUser = username;
23          username = CertTools.getRFC822Name(signerCert);
24          if(username == toUser)
25            userEqToUser = true;
26          inList = true;
27          aclDepth++;
28        }
29        else
30          inList = false;
31      }
32    }
33    return userEqToUser;
34  }
```

Listing 22: ACL validation: Authentication of the originator the shared resource

The method obtains a list of all user certificates needed to validate a received data value, the entire access control list related to the shared resource and the shared resource to be validated. Initially, the method invokes the `CertTools.GetPKC()` method to obtain the certificate that was used to sign the shared resource and translate it into an `X509Certificate2` object (line 7). The latter is representation for public key certificates in .NET. The username of a RELOAD peer is encoded within the this certificate and returned by another utility method `CertTools.GetRFC822Name()`. The following loops repeats the ACL *walk* until the `username` in the certificate that signed the last inspected ACL item is equal to the `toUser` field of the current ACL item. If the shared resource under test is an ACL item, the condition in line 15 validates whether the signer of the preceding ACL items was allowed to delegate the write access to the access control list. For all succeeding validations, the last condition must be true to validate the trust delegation chain up the root ACL item. Then, the `toUser` field becomes the username and username becomes the username of the signer of the ACL item. If both are equal, we reached the condition to end the while loop and thus can confirm, that the user originator of the shared resource was permitted store.

7.2.2. Usage for DisCo

A core feature of the protocol scheme for distributed conferencing is the registration the conference URI that is mapped to several focus peers in the overlay. The registration is realized in the `DisCoRegistration` class shown in listing 23. It contains the `ResNameExtension` class used to validate the pattern for variable resource names, an abstract definition of a coordinate value and the node-id of the registered focus peer.

```
1  pivate Boolean ACLvalidation(List<GenericCertificate> certs,
2  public class DisCoRegistration {
3    /* Meber fields */
4    public ResNameExtension resNameExt;
5    public ICoordinate coordinate;
6    public NodeId nodeId;
7    /* Constructor */
8    public DisCoRegistration(ResNameExtension resName,
9        Coordinate coord, NodeId nodeId) {
10     this.resNameExt = resName;
11     this.coordinate = coord;
12     this.nodeId = nodeId;
13   }
14 }
```

Listing 23: DisCo-Registration: Authentication of the originator the shared resource

The resource name extension is the class defined by the ShaRe usage described in the previous section. The `ICoordinate` interface requires from its realizations of two methods *toOpaque()*, *GetClosestNode()* and a *topoAlgorithm* class property containing a string token identifying the topology algorithm. The *toOpaque()* method shall return a string that can be parsed by another coordinate class that uses the same algorithm for generating a topological descriptor. For instance, an algorithm named *landmarking* could return a comma separated string representing the measured round-trip times to a set of predefined landmark hosts. A well-defined and standardized scheme to identify and serialize topological descriptions is it not defined yet and left to future work. The *GetClosest()* method takes a list of DisCo-Registration as arguments and returns the focus peers whose coordinates match next best. Listing 24 shows the implementation of calculation of the closest focus peer in an n-dimensional Cartesian space.

```
1  public NodeId GetClosest (List<DisCoRegistration> coords) {
2    NodeId potFocus = null;
3    int closest = 0;
4    foreach(DisCoRegistration reg in coords) {
5      var coord = reg.Coordinate;
```

```
6       int dist = 0;
7       for(int i = 0; i < coord.Vector.Length; i++) {
8         dist += (int)Math.Pow((vector[i] - coord.Vector[i]), 2);
9       }
10      if(potFocus == null){
11        potFocus = reg.NodeId;
12        closest = dist;
13      }
14      if(dist <= closest) {
15        potFocus = reg.NodeId;
16        closest = dist;
17      }
18    }
19    return potFocus;
20  }
```

Listing 24: GetClosest(): Implementation of the landmark algorithm

DisCo Usage Class The `DisCoUsage` class is a realization of the `Usage` interface which enables its registration at the usage manager factory of the core RELOAD.NET stack. On registration at the manager, the DisCo class begins to determine the relative position of the peer by pinging the landmarks hosts retrieved from the configuration document. The ping operation is handled asynchronously by using the Microsoft Concurrency and Coordination Runtime (CCR) [78]. The usage of the MS CCR should be shown in the following example in 25.

The method header of *Determine()* returns `IEmumerator<ITask>` object and takes `ReloadConfig` instance as parameter. The return value makes this method an *Iterator-Block* used in C# to support the *foreach* loop. In contrast to non-iterators, iterators are finalized by a `yield` keyword. For instance in line 3, the yield break operation ends this iterator if the DisCo usage already had determined its coordinates. A core feature of the CCR is the generic `Port<T>` class shown in line 5. It can be seen as a queue to input arguments into a callback method. If an operations is finished the port is used to post the completion to any interested receivers and returns the generic object back to the receivers.

Line 6 obtains the landmark hosts used to determine the coordinates vector in the n-dimensional Cartesian space. The loop shown in lines 8 to 12 starts the asynchronous by using the CRR static `Arbiter` class. It enqueues n iterative tasks object to the dispatcher queue handled by the CCR scheduler. The generic parameter of the iterative task are the arguments of the method delegate whose implementation shown in the next listing 26. Passed to *PingAsync()* is the i-th landmark host, the result string array and the host position in the landmark vector. This is necessary, because all interactive task will be executed concurrent

and might finish in another order as the order of their invocation. The `yield return` statement in line 18 waits for the completion of all started iterative tasks. Event though the yield return blocks this task, it does not block the entire system. Note that on execution, this task is managed by the CCR scheduler and a blocking task will just free resources for other tasks. However, if all ping tasks are finished, the *lamba* expression within the *Arbiter.Receive()* forms the coordinates vector and creates a new `Coordinate` instance.

```
1  public IEnumerator<ITask> Determine(ReloadConfig config) {
2    if(localCoords != null)
3      yield break;
4    try {
5      finished = new Port<string[]>();
6      var hosts = config.Document.Overlay.configuration.landmarks. ↩
           landmarkhost;
7      dim = hosts.Length;
8      var coords = new string[dim];
9      for(int i = 0; i < hosts.Length; i++) {
10       Arbiter.Activate(config.DispatcherQueue,
11         new IterativeTask<landmarkhost, int>(
12         hosts[i], coords ,i, PingAsync));
13     }
14   }
15   catch (Exception ex) {
16     config.Logger(ReloadGlobals.TRACEFLAGS.T_ERROR, "Determine: "+ex. ↩
           Message);
17   }
18   yield return Arbiter.Receive(false, finished, coords => {
19     var res = new StringBuilder();
20     foreach(string rtt in coords)
21       res.Append(rtt+",");
22     res.Remove(res.Length-1, 1);
23     localCoords = new Coordinate(res.ToString());
24     if(PositioningCompleted != null)
25       PositioningCompleted(localCoords);
26   });
27 }
```

Listing 25: Async Ping: Using CCR to multitask requests (1)

The next listing 26 shows the asynchronous ping operation. Likely to the *Determine()* method, the *PingAsync* is an iterative task that is handled by CCR scheduler. It function is to ping the landmark host obtained by the method argument and to put the returned RTT into the `coords` result array. The member variable `count` is incremented each time the ping operation has a result. As the counter is equal to the dimension of the coordinates vector (line 16), it uses

the `Port<T>` *finished.Post()* to announce the completion of the coordinates vector. This post is the trigger for the previously shown determine task to continue and to create a coordinate object from the retrieved RTTs.

```
1  private IEnumerator<ITask> PingAsync(landmarkhost host,
2     string[] coords, int order) {
3   var ping = new Ping();
4   try {
5     var reply = ping.Send(host.address, 1000);
6     if(reply.Status == IPStatus.Success) {
7       coords[order] = reply.RoundtripTime.ToString();
8     }
9     else {
10      /* Failure operation */
11    }
12    count++;
13  } catch(Exception ex) {
14    /* Error operation */
15  }
16  if (coords.Length == count) {
17    finished.Post(coords);
18  }
19  yield break;
20 }
```

Listing 26: Async Ping: Using CCR to multitask requests (2)

7.3. Demo Application

Finally, the demo application for distributed conferencing is presented in figure 35. The demo application was initially presented at the *36th IEEE Local Computer Networks* [82] conference and during the *Hamburg Lange Nacht des Wissens 2012*. It is a prototype implementation that uses the RELOAD.NET stack for registration of SIP URI and distributed conference URIs and the VoIP functionalities of the Sipek API. On startup, it joins the *t-reload.realmv6.org* RELOAD overlay, whose enrollment server and bootstrap node are hosted at the HAW-Hamburg. In the demo scenario, two desktop devices are acting as focus peers maintaining a single conference session. The conference participants are Windows Mobile 6.X executing the mobile variant of the RELOAD.NET stack. The lookup of the conference URI returned for each mobile device another focus peer through which they joining the distributed multiparty.

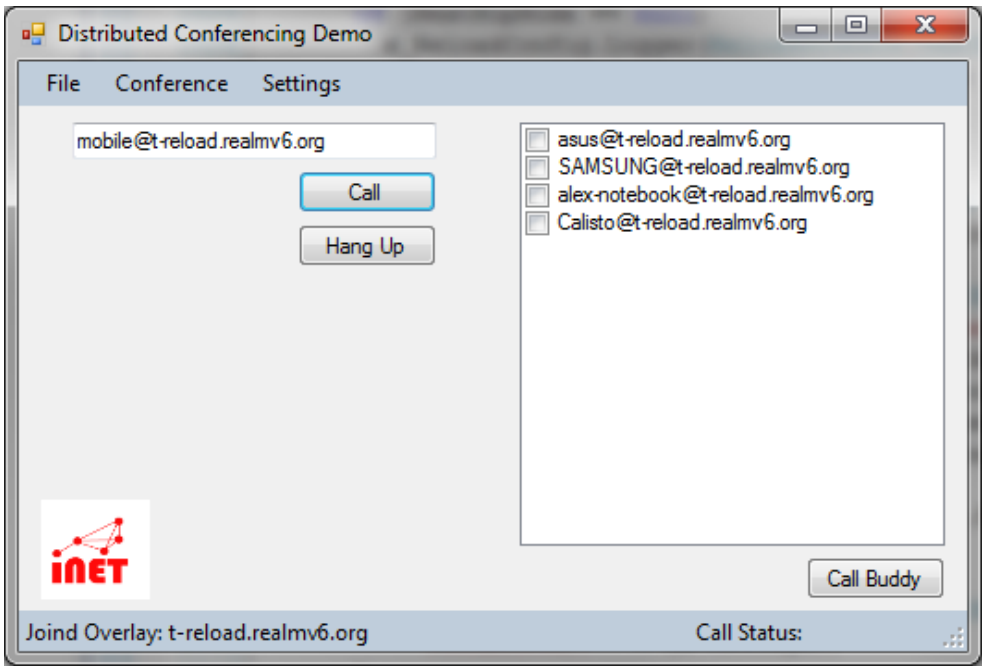


Figure 35: Demo Application: Distributed conferencing softphone prototype

8. Measurements & Evaluations

8.1. Objectives of Measurements

The evaluations in this work, should show the global scaling behavior of the RELOAD.NET stack and the implemented usages for distributed conference and shared resources. In order to achieve the most realistic results, measurements are performed in the *Planet-Lab* [83] testbed. Planet-lab is global compound of *approx.* 530 sites, actually providing 1022 nodes. By using this testbed it is possible to create realistic deployment scenarios of a global RELOAD overlay and to determine the advantages of the proximity-awareness of the participants in a distributed conference. The variables measured in these scenarios are:

- Delay to join a RELOAD overlay, without delay to configure and enroll a peer
- Delay to store a DisCo-Registration and corresponding access control list
- Hop-count of intermediate peer to store a registration
- Delay to retrieve a DisCo-Registration and ACL
- Hop-count for retrieving a DisCo-Registration/ACL
- Delay to initiate a connection to a selected focus peer (AppAttach)
- Hop-count for the upper operation
- Delay to establish a transport connection to join a distributed conference

The measurements show the different behaviors of the tested RELOAD.NET stack and DisCo protocol. The distinct scenarios German-wide, European-wide, North American-wide and finally on a global scale. Additionally, the test scenarios raveled the robustness of the RELOAD.NET implementation to sustain a P2PSIP overlay.

8.2. Mono Port

The RELOAD.NET project and thus the usages for distributed conferencing and shared resources are implemented in C#/.NET. The .NET runtime environment is designed to be executed on Windows operating systems and intends an abstraction of the used programming language. This is a particular problem for a deployment on the Planet-Lab testbed, since the virtual machines that are created for a Planet-Lab user, are Linux based Fedora [84] distributions.

The only solution is a porting of the RELOAD.NET stack to a C#/Mono project. Mono [85] is an open source runtime environment for .NET based applications. The target is to provide an abstract runtime that executes .NET programs transparently on any device. However, the Mono project is not covering the entire .NET framework and thus .NET features are not available on Mono. This has had consequences on the porting of the RELOAD.NET stack on Mono.

The first statement is that the version of the Mono runtime needs to be at least 2.X. Lower versions have a different implementation of thread pools which are massively used by the used concurrency library (CCR) [78]. However, this version of the Mono project is not available in the Fedora repositories. Hence, the source code has to be compiled on Planet-Lab node to the resulting binaries needed to be distributed among the PL nodes used for the measurements.

The second statement is that the supplied TLS library uses native Win32 binaries to create secure transport sessions. These are not provided by the Mono runtime thus treated a deployment on Linux. The vendor of the Secure Blackbox library offers a pure Mono distribution of its TLS framework for a high fee. However, the vendor has kindly passed time-limited access to the library on request for a scientific evaluation.

The third statement is that the Mono runtime does not have a similar behavior on each Linux distribution. For instance, for sending an ICMP request a user needs to have root privileges on Fedora which is not required if Mono runs on a Ubuntu distribution. Furthermore, many small differences in the implementation of .NET and Mono caused a lot of bug-fixing to port the RELOAD.NET stack finally on Mono and Linux OS.

8.3. Measurement Setup

8.3.1. Measurement Architecture

The entire measurement is a setup of the following three components as shown in figure 36:

PL-Manager: The PL-manager is a self-developed Python script to manage a distributed measurement setup in the Planet-Lab (PL) testbed. It establishes SSH connections to each PL given by a configuration file and invokes further Python scripts on the remote hosts. The PL-manager initiates the entire RELOAD overlay iteratively. It starts the remote script and waits until the created RELOAD peers are instantiated and joined to the overlay before it continues with the next site. If all overlay peers are joined, the PL-manager invokes another remote python script sending a further command to the peers. Analogously to the overlay creation, all commands are sent iteratively along the joining order. If the last procedure has finished, the manager ends all overlay peers and repeats the entire procedure until a predefined count.

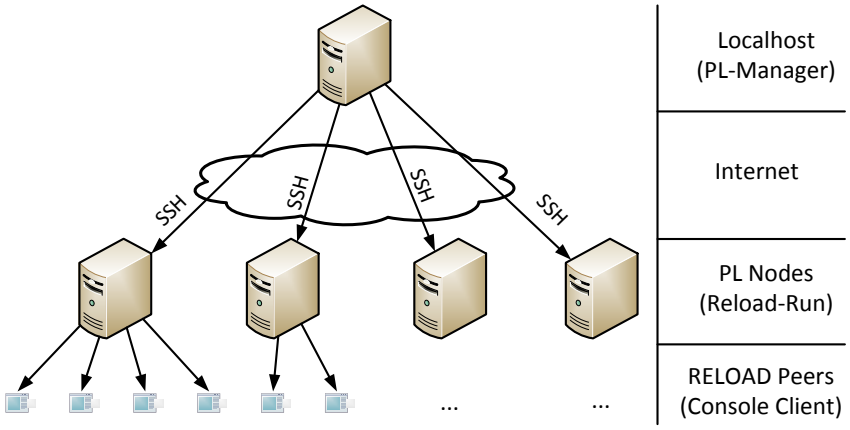


Figure 36: Measurement Architecture: Three-layers to instantiate a RELOAD overlay

Reload-Run: The Python RELOAD-run is deployed on the Planet-Lab to instantiate a variable number of RELOAD peers. By invocation of the PL-manager, the script starts each peer as subprocess and communicates to the PL-manager via its own `stdout` to acknowledge the successful startup of the peers. If an invoked peer is unable to join the overlay, the script terminates the process and continues with the next peer. Afterwards, RELOAD-run waits for further commands by listening to a stream socket. If it receives a command called *operation*, it uses the `stdin` to communicate with the invoked subprocesses to perform a RELOAD method (Fetch, Store). After finishing achieved operations, it waits for an *end* command to shut down all peers.

Console Client: The lowest layer in the measurement architecture, is the console application of the RELOAD.NET stack. It takes several console arguments that specifies it behavior on startup and on receiving the `operation` command. All applications are setup to log only the measure variables. However, the peers are configured to send a minimal overlay telemetries to the Web-monitoring tool, to enable the detection of peer failures.

8.3.2. Measurement Configuration

Each measurement scenario uses the same configuration parameters. The target of the configuration is the emulation of a distributed conference with 5 participants on each Planet-Lab. The peers were only deployed on Planet-Lab nodes that exhibit an appropriate load to minimize the effect of local system disturbances. The nodes are instantiating the peers according to the following setup:

CAIDA Monitor	Location
mnl-ph.ark.caida.org	
nrt-jp.ark.caida.org	Asia
she-cn.ark.caida.org	
dub-ie.ark.caida.org	
lej-de.ark.caida.org	Europe
her-gr.ark.caida.org	
pna-es.ark.caida.org	
sea-us.ark.caida.org	
mty-mx.ark.caida.org	
amw-us.ark.caida.org	North America
yto-ca.ark.caida.org	
wbu-us.ark.caida.org	
hlz-nz.ark.caida.org	Oceania
gig-br.ark.caida.org	South America
scl-cl.ark.caida.org	

Table 2: Landmark selection: Chosen from CAIDA measurement monitors

- The first peer assumes the role of the bootstrap node
- The second peer is the initial creator of a conference by registering a predefined conference URI
- The Peers 3 to 5 just join the conference
- The first peer started on each subsequent PL node, joins the conference and registers as additional conference focus
- Each subsequent peer started on a node just joins the conference

As a result, every 5th participant in a conference is a focus peer. The relation of one focus handling 4 participants is orientated at the Skype recommendation for video group conferencing [57]. In this configuration, the expected delay to join the conference is longer for the first peer stated on a node than for the node started subsequently. They should select the first peer as their focus due to their proximity-awareness. The topological descriptors are generated by estimating round-trip times against a predefined set of landmark host obtained while RELOAD enrollment. The quality of landmark approaches depend on an appropriate number of landmark nodes and their placement. For the particular measurement configuration, 15 landmarks are chosen from the set of CAIDA [86] monitor as shown on table 2.

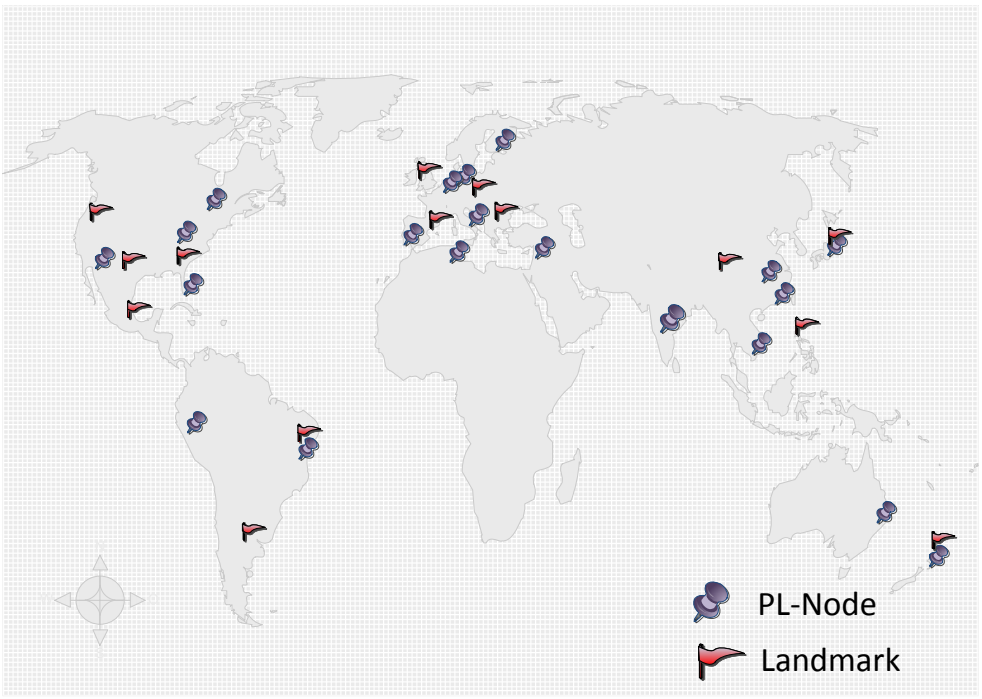


Figure 37: Peer Deployment: The distribution of the PL-nodes in the global scope

The CAIDA nodes are highly reasonable as they are not located behind NATs or firewalls and they are uniformly distributed around the world. Hence, the positive effect of selecting the focus by proximity should increase with the global distribution of the measurement scenario.

8.4. Measurements

8.4.1. Measurement Scopes

To evaluate the performance of the RELOAD.NET stack and the proximity-awareness benefits of distributed conference control, measurements are made in several scopes.

Germany: Representing a nationwide distribution

Europe: Representing a continentalwide distribution

North America: Representing a further continental distribution to be compared with Europe

Global: Representing the maximal possible distribution

The distribution of the Planet-Lab nodes in the global scope is shown in figure 37. The 20 nodes are mainly distributed among Europe and North America. This due for two reasons. At first, these regions are the most frequent users of the Internet [87, 88]. Secondly, the availability of adequate PL-nodes is rare on the remaining continents. For instance, only one PL-node in Africa (Tunisia) fits in the requirements (free CPU and Memory) for a RELOAD measurement. However, there is at least one PL-node located at each of the five main continents.

Figure 37 also shows the positions of the CAIDA [86] monitors that are used to generate the relative coordinates of the peers. They are even distributed among the global scope as the PL-nodes.

8.4.2. Joining a RELOAD Overlay

The first measurement represents the average delay to join a RELOAD overlay. The joining delay is a concatenation of the delays for the following procedure:

- Attaching the bootstrap peer including onwards forwarding to the admitting peer
- Attaching to a set of RELOAD peers to enrich a peers routing table
- Joining the admitting peer to participate the overlay finally

A previous measurement presented in [74] included the delay to contact the enrollment server in the joining time. For this measurement, the enrollment delay would mask the P2P delay behavior. In particular, it would interfere the compatibility of the European with the North American measurement since the enrollment server is static deployed on a server in Germany.

In the measurements scopes of Germany, Europe and the global distribution, the bootstrap node is deployed on the server at the HAW-Hamburg in Germany. For the measurements in North America, the bootstrap is deployed on a Planet-Lab node at the University of Washington, USA. Hence, the measurement results for North American and Europe remain comparable.

The measurement results are shown in figure 38. The graphs represent the Cumulative Distribution Frequency (CDF) in the ordinate and the average joining delay in the abscissa. The CDF indicates the relative frequency of peers that successfully joined the overlay after a specific time interval. The figure shows the different delays to join the overlay with reference to their distribution described as following:

Germany: 90% of the peers are successfully joined to the overlay of approx. $4s$. This represents the best result among the measured scenarios. The least joining delay is generated by the initially invoked Planet-Lab node hosting the bootstrap node with a delay of approx. $1,1s$.

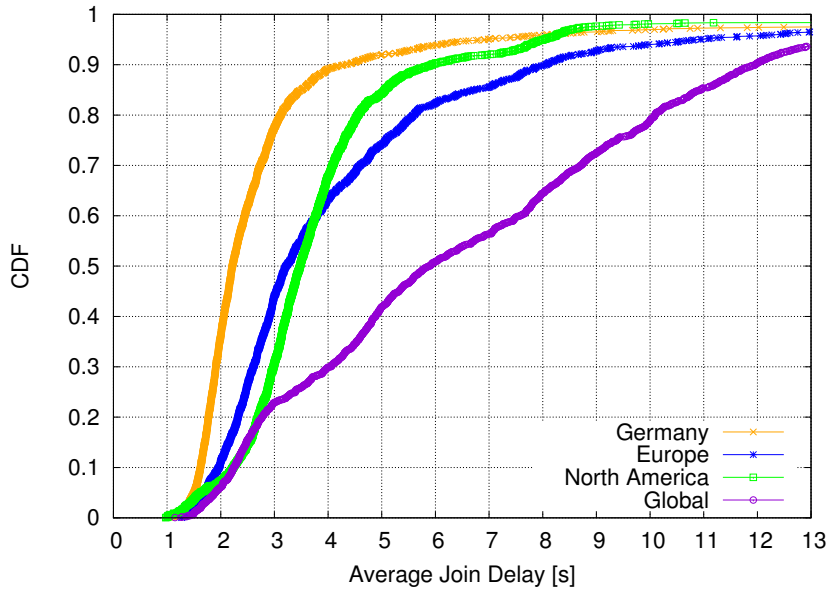


Figure 38: Joining: Average delay to join a RELOAD overlay in several scopes

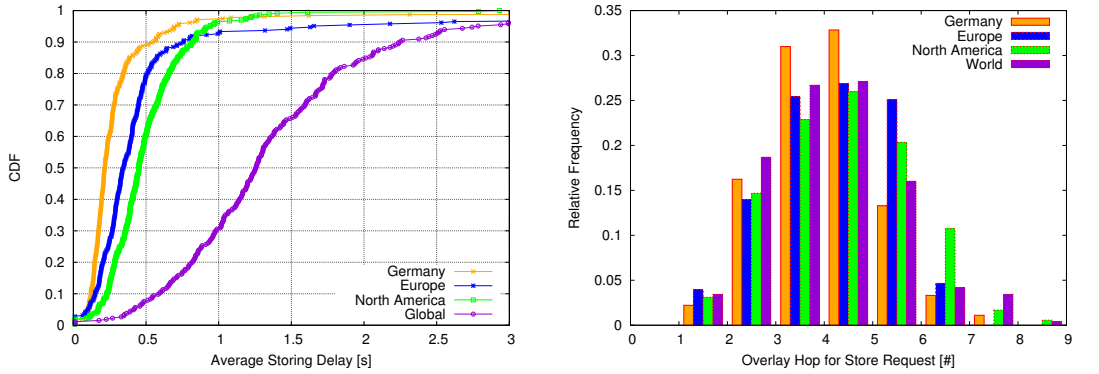
Europe: 90% of the peers in a European scope are joined to the overlay after approx. $8s$. In this scenario, delay to join worsens at ~60% overlay size.

North America: 90% of the peers are joined to the overlay within approx. $6s$. The frequency in this scenario has a smooth behavior.

Global: 90% of the overlay peers in global scenario joined the overlay under $12s$. The graph tend to a linearly behavior in comparison to the remaining measurement results.

The measurements for Germany, Europe and North America show an adequate scale behavior. The long tails of their CDF due to peers that needed a second or third try to connect to their admitting peer (AP). If the TLS connection establishment to the AP is not successful on the first try, a joining peer takes a timeout of 10 seconds and retries. Another source of trouble are intermediate peers that route an Attach response message back to a joining peer. The implementation of the return routing procedure seems to ignore responses if they have to be process concurrently.

The delay in the global scenario is also influenced by retransmissions. However, some locations have very long joining times without failures. For instance, the Taiwanese node `planetlab2.iis.sinica.edu.tw` needed in average approx. $11s$ to join the overlay, whereby the measured values have no outliers. This suggests an adequate quality of connection to the remaining overlay that is influenced by large network delays.



(a) Store Delay: CDF to store a DisCo-Registration

(b) Store Hops: Average overlay hops for a store request

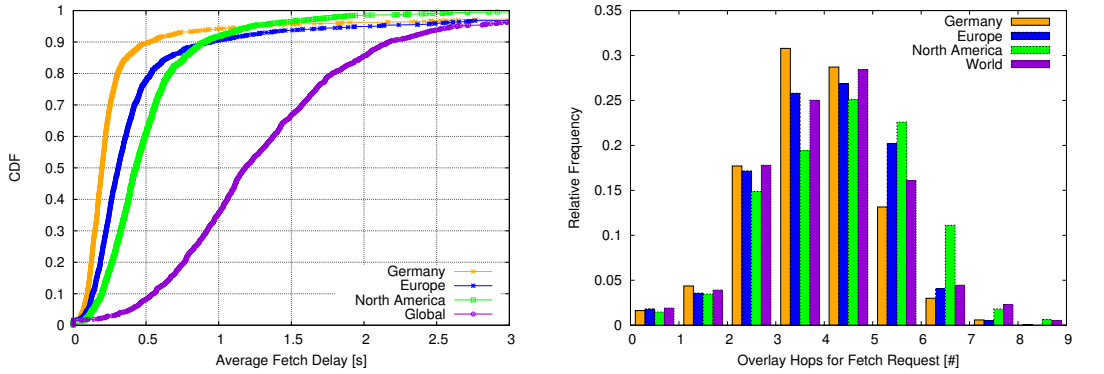
Figure 39: Store Request: Registration of a distributed conference

8.4.3. Storing a DisCo-Registration

The figures in 39 show the overlay behavior to store a DisCo-Registration. Figure 39a shows the CDF in reference to the delay to register a distributed conference, while figure 39b relative frequency of the corresponding overlay hops shows. The delay represents the round-trip time of a Store request and its subsequent Store answer message. The hop count is the number of hops that are traversed by the store answer. Since RELOAD implements recursive return routing, the hop count for a request and answer message is equal.

The average delay to store a data value in Germany, Europe and North America shows an adequate scale behavior. In Germany, a store takes less than $0,5s$ causing 90% of cases. In the continental measurements, a store request is finished after approx. $0,75s$ causing 90% of cases. The longest storing delays are achieved by the measurement in global scope. The hops count in all measurements is mostly in the same range. Causing 75% of cases, the hops count is in between 3 to 5 hops.

In comparison to the joining delay, store requests in Europe have mostly less delay than in North America. This indicates that several peers in Europe were deployed on PL-nodes that had initially problems with connecting to the overlay. However, if these peers have established all their necessary connections, succeeding requests have to traverse shorter physical distances than in the North American scenario. Nevertheless, the store request in Europe has had more erroneous tries causing the longer tail in the CDF compared to the American scenario.



(a) Fetch Delay: CDF to Fetch a DisCo-Registration (b) Fetch Hops: Average overlay hops for a Fetch request

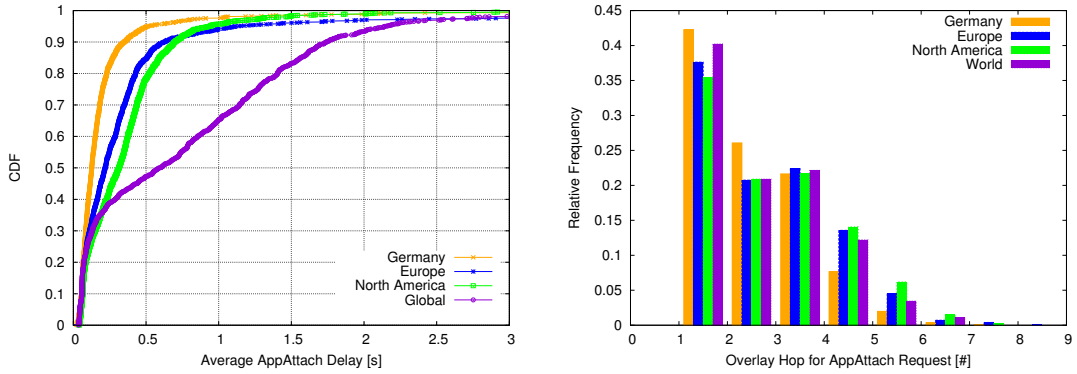
Figure 40: Fetch Request: Retrieval of the DisCo-Registration

8.4.4. Fetching a DisCo-Registration

The measured metrics to obtain a DisCo-Registration and access control list are shown in figure 40. Analogously to measurements of the store request, sub-figure 40a the delay for a fetch request and figure 40b the corresponding overlay hops. The presented behavior in both measurements is similar to those of the store request. A Fetch request takes in average less than approx. $0,5s$ in the German scenario and less than $0,8s$ in the continental scopes. The average Fetch delay in the global scenario is with a probability of 90% finished after a duration of approx. $2,25s$. The overlay hops counts to retrieve the data approx. even to those of the store request. The curves for measurements results of the Fetch request are smoother compared to those of the store request. This dues to the number of samples taken during measurement. A store request is performed by every 5th peer, while a Fetch is performed by all peers with except of the conference creator and the bootstrap node.

8.4.5. AppAttach to Focus Peer

The final RELOAD operation performed by the peer to participate a distributed conference is the AppAttach request. After a peer had chosen its focus peer according to its relative network position, he sends an AppAttach request. The request will be routed through the overlay, even if the destination node is located at the same device. This dues to the RELOAD protocol that does not provide additional proximity information. The measurement results are shown in figure 41.



(a) AppAttach Delay: Delay to initiate a transport to focus (b) AppAttach Hops: Avg. hops for an AppAttach request

Figure 41: Fetch Request: Retrieval of the DisCo-Registration

The average delay in sub-figure 41a shows, that AppAttach requests to focus peers are completed after approx. $0,35$ causing 90% of cases in the German scenario. In the continental measurements, an AppAttach takes about $0,65s$, while it takes about $1,75s$ in the global scenario. This represents an average reduction of approx. 23% for the AppAttach compared with the preceding Fetch request delay. This correlates with the average hop count to perform an AppAttach request as shown in figure 41b. About 35% to 40% of the requests needed one single hop to reach their destination.

The reason for this reduction is a relative small size of the overlay. Each peer maintains a connection table containing all peers they ever had learned, e.g., during routing messages for other peers. Further, each peer maintains a routing table which is a subset of the connection table. The difference between them is the routing decision. A peer uses its routing table exclusively to route messages originated by himself or to route messages for others. The connection table is only used to send a message, if the destination node-id is equal to an entry in the connection table. The previous stored and fetched requests are always routed to a resource-id thus a peer needs to use its routing table to obtain the next hop to destination. In case of an AppAttach request, the destination is a node-id. The node-id could probably have a match in the connection table, hence the AppAttach request is send directly to the designating focus peer. The connection table is periodically cleaned from entries that are not demanded for the routing table. However, this maintenance routine was set to repeat every half hour and thus to long to take effect during a measurement iteration.

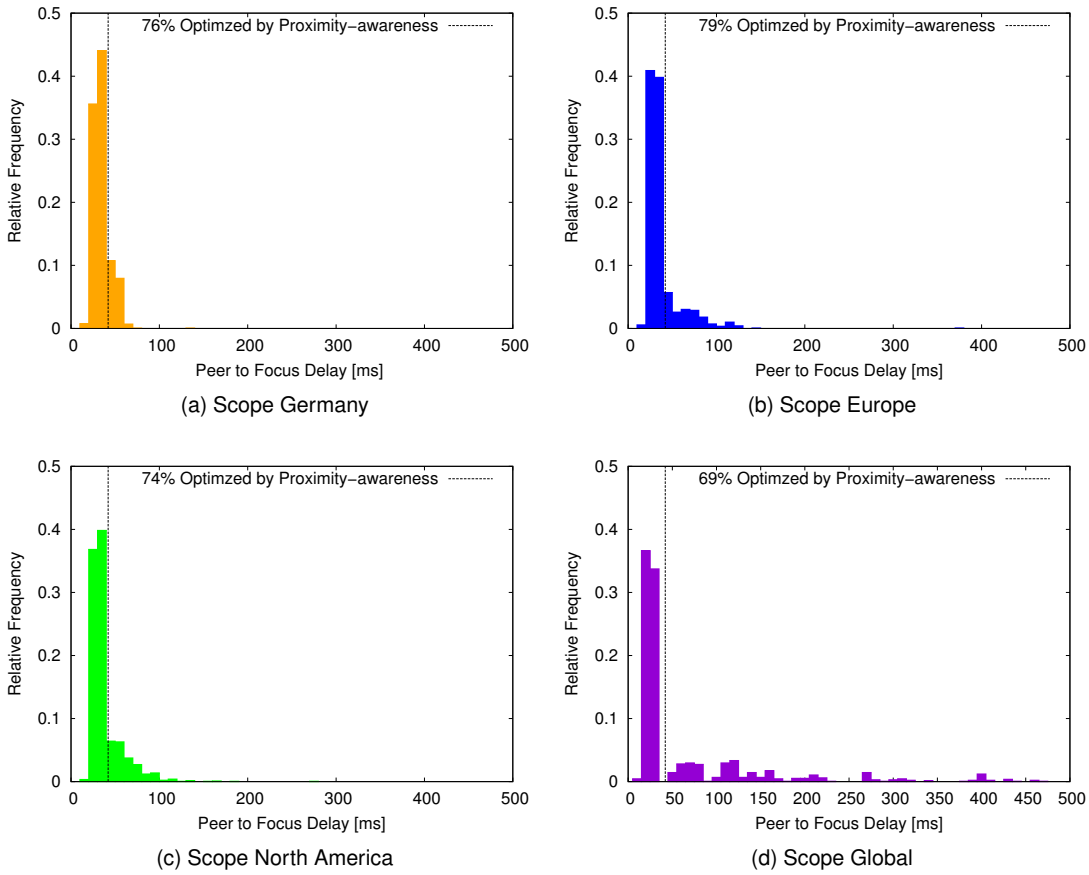


Figure 42: Peer to Focus Delay: Comparison of Scenarios

8.4.6. Connection Establishment to Focus Peer

The AppAttach procedure returns the IP address of the selected focus peer and the SIP port (5060) on which he will accept incoming calls. For the measurements the final call to the focus is emulated by sending a ICMP echo request plus a random value. The random value is based on previous measurement result performed in [65] that determined an average of approx. $30ms$ to establish a SIP session. The SIP establishment approximation is not ideal, but is sufficient to demonstrate the effect of proximity-awareness of the peers.

The measurement results for several distribution scenarios are shown in the figures in 42. Each figure shows the relative frequency to the delay in milliseconds to establish session to the chosen focus peer. The bulk of delays is in an interval of $\{x \in \mathbb{R} \mid 25ms \leq x \leq 35ms\}$ that represents the delay to join a focus hosted on the same PI-nodes as the joining peer. The high

probability of selecting the locally hosted focus is based on the proximity-information of the DisCo-Registration. Since focus and joining peer should generate even landmark coordinates, peers that are invoked after the first will always choose the first peer as their focus. The remaining connection delays are the delay times of each peer initially invoked on a PI-node. Those had to join the conference via a remote host. The amount of peers that selected their local focus are indicated in the figure. For instance in figure 42a, 76% of the peers had chosen the focus located at the same device. The amount of continental measurements even have results with 79% in Europe and 74% in North America. Contrary to expectations, only 69% of the peers in the global measurement choose their local focus. The smaller amount due to peer failures. The PL-nodes were under varying load during the measurements and caused occasionally peer failures. If a failure happened to a peer that should have resisted as focus, the subsequently started peer had to choose a remote focus. This caused additional and reasons the results.

8.5. Evaluation

The measurement results show the scaling behavior of the RELOAD.NET stack and the benefits of joining a distributed conference based on proximity-awareness. By increasing the distribution of the overlay peers from a national to a continental scope, the delays to join, store and fetch are almost doubling. This statement is supported by the uniformity of the overlay hops in the different scenarios. While the hops count is uniform, the delays increases and thus an effect on the higher network latencies. This trend is even clearer in the global scenario. There, the latencies are increased by an approximated factor of three compared to the continental measurement result. Peers located in the far east (e.g., Taiwan) or far west (e.g., Ecuador) in relation to the bootstrap node have an average joining delay of approx. $11s$ without connection errors. However, several PL-nodes have an uneven workload causing occasional peer failures that enlarge the joining delay due to overlay topology switches.

An unexpected effect is detected for the AppAttach procedure. Due to a too long set connection maintenance, entries in the connection table of a peer were used to directly AppAttach to the focus peer. A future measurement with a more adequate configuration will show that the overlay hop count to a focus peer will be more similar to those of the store and fetch request.

As previously excepted is the peer-to-focus delay in most cases around approx. $30ms$. This corresponds to the average delay to establish a SIP session on a local host. The conference participants choose their focus according to the coordinate vector. If no erroneous coordinate vector are determined, a participant will always choose the focus peer that is located at the same host. For the entire conference, 69% in the global scope up to 79% of the peers in the European scope selected the focus next to them. If the amount is adjusted to the 20% of nodes that must choose a remote focus peer it can be seen that 89% to 99% of the peers

choose their ideal focus peer. Hence, most peers had a delay below the ITU recommendation for *Voice-to-Ear* delay previously presented in table 1 in section 3.

The measurement of the global scenario is strongly influenced by variating workloads of the PL-nodes, causing node failures of the overlay. In a more stable setup, the results for a global scope should be similar to those of the nation and continental.

9. Conclusion and Outlook

This work presented an approach for distributed conference control based on the RELOAD protocol. Distributed conferencing (DisCo) is designed as a P2P signaling protocol enabling multiuser conferences on a large scale. To obtain a base to develop the DisCo scheme this work initially discussed the generic problem space of distributed conference to identify issues to be solved by the protocol. The identified challenges concerned issues in the RELOAD base protocol and the Session Initiation Protocol.

To enabled the registration of a conference identifier that maps a single URI to several independent entities a new RELOAD usage for shared resources was developed. The usage adds two functionalities to the RELOAD base protocol. At first, it provides a mechanism to share certain resources with further users. Secondly, it defines a scheme that enables variable resource names that orientate on the default access control policies of RELOAD. The protocol schemes to register, participate and cooperatively maintain a DisCo are discussed and defined in the approach for distributed conference control. The DisCo scheme issues proximity-awareness, load balancing and failover procedures for ad-hoc P2P conferences. Proximity-awareness is based on topological descriptors that are announced within a DisCo-Registration data structure. They are used to optimize the conference topology with respect on delay and jitter which are critical issues in multimedia communications. Load balancing procedures provide enable a uniform workload on the peers managing the conference. Since each individual entity in a P2P network is not obliged to continue a conferencing service, the network must compensate the disappearance of peers. The final challenges issued by the DisCo protocol is synchronization of the entire conference state. This is achieved through an XML event package for distributed conferencing, conveying the roles and relations among the participants.

The protocol scheme for distributed conference control was implemented in a C#/.NET project to demonstrate DisCo principle and to evaluate the protocol. The implementation based on the RELOAD.NET stack which is one of the first advanced RELOAD implementation in the P2PSIP community. The DisCo demonstrator applications based the Sipek API. It is a .NET wrapper for several high level functions of the PJSIP stack, a very advanced SIP stack written in C.

The implementations were used to perform measurements to evaluate the scaling behavior of the RELOAD and DisCo protocol. The measurements were performed on the Planet-Lab testbed. Planet-lab is global compound of approx. 530 sites actually providing 1022 nodes. Using this testbed, it is possible to create realistic deployment scenarios of a global RELOAD overlay and to determine the advantages of the proximity-awareness of the participants in a distributed conference. Measurements were setup to represent a scenario of a large conference with 100 participants in which every 5th node assumed the role of a focus peer. The results showed an adequate scaling behavior with respect to latency and overlay hop count. By the proximity-awareness of the overlay peers, the delay to finally establish a transport connection

to a chosen focus was optimized causing 69% of cases in a global scope and up to 78% of cases in European scope. In several cases, the delay to create a connection was reduced by a factor of ten. This shows the potential of proximity-awareness in distributed conference control.

For future work, the implementation of the RELOAD.NET transport layer should be enhanced the robustness of the stack against connections errors. The measurements revealed a weakness in the implementation of the TLS transport module. Furthermore, the implementation of the remaining DisCo functionalities is of particular interest. By implementation of the XML event package, further measurements can demonstrate the potential of load balancing and failover procedures. To achieve this, it must be considered if the used SIP API should be extended to wrap further functionalities from the PJSIP stack to .NET. Or, if there exist any other open source implementation for .NET that can easier been extended with DisCo SIP procedures.

Finally, the concepts for shared resource and distributed conference control are works in progress in the IETF. These documents will be continuously maintained to be ready for an adoption by the P2PSIP working group.

A. Appendix

```
ï»¿<?xml version="1.0" encoding="UTF-8"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema"
           xmlns:ci="urn:ietf:params:xml:ns:conference-info"
     xmlns="urn:ietf:params:xml:ns:distributed-conference"
     targetNamespace="urn:ietf:params:xml:ns:distributed-conference"
     elementFormDefault="qualified"
     attributeFormDefault="unqualified">
  <!--
       This imports the definitions in conference-info
   -->
  <xs:import namespace="urn:ietf:params:xml:ns:conference-info"
        schemaLocation="http://www.iana.org/assignments/xml-registry/ ↩
            schema/conference-info.xsd"/>
  <xs:import namespace="http://www.w3.org/XML/1998/namespace"
        schemaLocation="http://www.w3.org/2001/03/xml.xsd"/>
  <!--
      A DISTRIBUTED CONFERENCE ELEMENT
   -->
  <xs:element name="distributed-conference"
              type="distributed-conference-type"/>
  <!--
       DISTRIBUTED CONFERENCE TYPE
   -->
  <xs:complexType name="distributed-conference-type">
    <xs:sequence>
      <xs:element name="version-vector"
                  type="version-vector-type" minOccurs="1"/>
      <xs:element name="conference-description"
                  type="conference-description-type"
                  minOccurs="0" maxOccurs="1"/>
      <xs:element name="focus"
      type="focus-type"
      minOccurs="0"
      maxOccurs="unbounded"/>
      <xs:any namespace="##other" processContents="lax"/>
    </xs:sequence>
    <xs:attribute name="state" type="ci:state-type"/>
    <xs:attribute name="entity" type="xs:anyURI"/>
    <xs:anyAttribute namespace="##other" processContents="lax"/>
  </xs:complexType>
  <!--
```

```
41        VERSION VECTOR TYPE
42     -->
43    <xs:complexType name="version-vector-type">
44      <xs:sequence>
45        <xs:element name="version"
46                    type="version-type"
47                    minOccurs="1"
48                    maxOccurs="unbounded"/>
49        <xs:any namespace="##other" processContents="lax"/>
50      </xs:sequence>
51      <xs:anyAttribute namespace="##other" processContents="lax"/>
52    </xs:complexType>
53    <!--
54        CONFERENCE DESCRIPTION TYPE
55     -->
56    <xs:complexType name="conference-description-type">
57      <xs:sequence>
58        <xs:element name="display-text"
59                    type="xs:string" minOccurs="0"/>
60        <xs:element name="subject" type="xs:string" minOccurs="0"/>
61        <xs:element name="free" type="xs:string" minOccurs="0"/>
62        <xs:element name="keywords"
63                    type="ci:keywords-type" minOccurs="0"/>
64        <xs:element name="service-uris"
65                    type="ci:uris-type" minOccurs="0"/>
66        <xs:any namespace="##other" processContents="lax"/>
67      </xs:sequence>
68      <xs:attribute name="state" type="ci:state-type"/>
69      <xs:anyAttribute namespace="##other" processContents="lax"/>
70    </xs:complexType>
71    <!--
72        FOCUS TYPE
73     -->
74    <xs:complexType name="focus-type">
75      <xs:sequence>
76        <xs:element name="display-text"
77                    type="xs:string" minOccurs="0"/>
78        <xs:element name="associated-aors"
79                    type="ci:uris-type" minOccurs="0"/>
80        <xs:element name="roles"
81                    type="ci:user-roles-type" minOccurs="0"/>
82        <xs:element name="languages"
83                    type="ci:user-languages-type" minOccurs="0"/>
84        <xs:element name="focus-state"
```

```
85                  type="focus-state-type" minOccurs="0"/>
86        <xs:element name="users"
87                  type="ci:users-type" minOccurs="0"/>
88        <xs:element name="relations"
89                  type="relations-type" minOccurs="0"/>
90        <xs:any namespace="#other" processContents="lax"/>
91      </xs:sequence>
92      <xs:attribute name="entity" type="xs:anyURI"/>
93      <xs:attribute name="node-id" type="xs:string"/>
94      <xs:attribute name="state" type="ci:state-type"/>
95      <xs:anyAttribute namespace="##other" processContents="lax"/>
96    </xs:complexType>
97    <!--
98         VERSION TYPE
99     -->
100   <xs:complexType name="version-type">
101     <xs:simpleContent>
102       <xs:extension base="xs:unsignedInt">
103         <xs:attribute name="entity" type="xs:anyURI"/>
104         <xs:attribute name="node-id" type="xs:string"/>
105         <xs:anyAttribute namespace="##other" processContents="lax"/>
106       </xs:extension>
107     </xs:simpleContent>
108   </xs:complexType>
109   <!--
110        FOCUS STATE TYPE
111    -->
112   <xs:complexType name="focus-state-type">
113     <xs:sequence>
114       <xs:element name="user-count"
115                   type="xs:unsignedInt" minOccurs="0"/>
116       <xs:element name="coordinate"
117                   type="xs:string" minOccurs="0"/>
118       <xs:element name="maximal-user-count"
119                   type="xs:unsignedInt" minOccurs="0"/>
120       <xs:element name="conf-uris"
121                   type="ci:uris-type" minOccurs="0"/>
122       <xs:element name="available-media"
123                   type="ci:conference-media-type" minOccurs="0"/>
124       <xs:element name="active" type="xs:boolean" minOccurs="0"/>
125       <xs:element name="locked" type="xs:boolean" minOccurs="0"/>
126       <xs:any namespace="##other" processContents="lax"/>
127     </xs:sequence>
128     <xs:attribute name="state" type="ci:state-type"/>
```

```
129     <xs:anyAttribute namespace="##other" processContents="lax"/>
130   </xs:complexType>
131   <!--
132        RELATIONS TYPE
133    -->
134   <xs:complexType name="relations-type">
135     <xs:sequence>
136       <xs:element name="relation"
137                   type="relation-type"
138                   minOccurs="0" maxOccurs="unbounded"/>
139       <xs:any namespace="##other" processContents="lax"/>
140     </xs:sequence>
141     <xs:attribute name="state" type="ci:state-type"/>
142     <xs:anyAttribute namespace="##other" processContents="lax"/>
143   </xs:complexType>
144   <!--
145        RELATION TYPE
146    -->
147   <xs:complexType name="relation-type">
148     <xs:simpleContent>
149       <xs:extension base="xs:string">
150         <xs:attribute name="entity" type="xs:anyURI"/>
151         <xs:anyAttribute namespace="##other" processContents="lax"/>
152       </xs:extension>
153     </xs:simpleContent>
154   </xs:complexType>
155 </xs:schema>
```

References

[1] 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.

[2] K. Singh and H. Schulzrinne, "Peer-to-peer internet telephony using sip," in *Proc. of the int. workshop on Network and operating systems support for digital audio and video (NOSSDAV '05)*, (New York, NY, USA), pp. 63–68, ACM, 2005.

[3] D. A. Bryan, B. B. Lowekamp, and C. Jennings, "Sosimple: A serverless, standards-based, p2p sip communication system," in *Proc. of the 1st Int. Workshop on Advanced Architectures and Algorithms for Internet Delivery and Applications(AAA-IDEA '05)*, (Washington, DC, USA), pp. 42–49, IEEE Computer Society, 2005.

[4] C. Jennings, B. Lowekamp, E. Rescorla, S. Baset, and H. Schulzrinne, "REsource LOcation And Discovery (RELOAD) Base Protocol," Internet-Draft – work in progress 18, IETF, August 2011.

[5] A. Knauf, G. Hege, T. Schmidt, and M. Waehlisch, "A Usage for Shared Resources in RELOAD (ShaRe)," Internet-Draft – work in progress 02, IETF, October 2011.

[6] A. Knauf, G. Hege, T. Schmidt, and M. Waehlisch, "A RELOAD Usage for Distributed Conference Control (DisCo)," Internet-Draft – work in progress 03, IETF, July 2011.

[7] M. Handley, V. Jacobson, and C. Perkins, "SDP: Session Description Protocol," RFC 4566, IETF, July 2006.

[8] "SIP Standards." http://www.packetizer.com/ipmc/sip/standards.html, 2011.

[9] J. Rosenberg and H. Schulzrinne, "Guidelines for Authors of Extensions to the Session Initiation Protocol (SIP)," RFC 4485, IETF, May 2006.

[10] A. Roach, "Session Initiation Protocol (SIP)-Specific Event Notification," RFC 3265, IETF, June 2002.

[11] J. Rosenberg, "A Session Initiation Protocol (SIP) Event Package for Registrations," RFC 3680, IETF, March 2004.

[12] "The Internet Assigned Numbers Authority (IANA) homepage." http://www.iana.org, 2011.

[13] R. Sparks, "Internet Media Type message/sipfrag," RFC 3420, IETF, November 2002.

[14] R. Mahy and D. Petrie, "The Session Initiation Protocol (SIP) 'Join' Header," RFC 3911, IETF, October 2004.

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

[16] M. Barnes, C. Boulton, and O. Levin, "A Framework for Centralized Conferencing," RFC 5239, IETF, June 2008.

[17] G. Camarillo, J. Ott, and K. Drage, "The Binary Floor Control Protocol (BFCP)," RFC 4582, IETF, November 2006.

[18] J. Rosenberg, H. Schulzrinne, and O. Levin, "A Session Initiation Protocol (SIP) Event Package for Conference State," RFC 4575, IETF, August 2006.

[19] A. Oram, *Peer-to-Peer Harnessing the Power of Disruptive Technologies*. Sebastopol, CA, USA: O'Reilly Media, 2001.

[20] "The SETI@home homepage." http://www.setiathome.berkely.edu, 2011.

[21] "The Napster homepage." http://www.napster.com, 2011.

[22] "The Gnutella Protocol Development homepage." http://rfc-gnutella.sourceforge.net/developer/index.html, 2003.

[23] D. Eastlake and P. Jones, "US Secure Hash Algorithm 1 (SHA1)," RFC 3174, IETF, September 2001.

[24] 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.

[25] S. Ratnasamy, M. Handley, R. M. Karp, and S. Shenker, "Application-Level Multicast Using Content-Addressable Networks," in *Networked Group Communication, Third International COST264 Workshop, NGC 2001, London, UK, November 7-9, 2001, Proceedings* (J. Crowcroft and M. Hofmann, eds.), vol. 2233 of *LNCS*, (London, UK), pp. 14–29, Springer–Verlag, 2001.

[26] A. Rowstron and P. Druschel, "Pastry: Scalable, distributed object location and routing for large-scale peer-to-peer systems," in *IFIP/ACM International Conference on Distributed Systems Platforms (Middleware)*, vol. 2218 of *LNCS*, (Berlin Heidelberg), pp. 329–350, Springer–Verlag, Nov. 2001.

[27] P. Maymounkov and D. Mazières, "Kademlia: A peer-to-peer information system based on the xor metric," in *Proc. of the 1st Int. Workshop on Peer-to Peer Systems (IPTPS '02)*, (Cambridge, MA, USA), pp. 53–65, 2002.

[28] D. Bryan, "A P2P Approach to SIP Registration and Resource Location," Internet-Draft – work in progress 03, IETF, October 2006.

[29] D. A. Bryan, M. Zangrilli, and B. B. Lowekamp, "Challenges of DHT Design for a Public Communication System," Technical Report WM-CS-2006-03, College of William and Mary, June 2007.

[30] J. Rosenberg, R. Mahy, P. Matthews, and D. Wing, "Session Traversal Utilities for NAT (STUN)," RFC 5389, IETF, October 2008.

[31] 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.

[32] S. Baset, H. Schulzrinne, and M. Matuszewski, "Peer-to-Peer Protocol (P2PP)," Internet-Draft – work in progress 01, IETF, November 2007.

[33] M. Zangrilli and D. Bryan, "A Chord-based DHT for Resource Lookup in P2PSIP," Internet-Draft – work in progress 00, IETF, February 2007.

[34] D. Bryan, "dSIP: A P2P Approach to SIP Registration and Resource Location," Internet-Draft – work in progress 00, IETF, February 2007.

[35] C. Jennings, B. Lowekamp, E. Rescorla, S. Baset, and H. Schulzrinne, "REsource LOcation And Discovery (RELOAD) Base Protocol," Internet-Draft – work in progress 00, IETF, October 2008.

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

[37] R. Braden, "Requirements for Internet Hosts - Communication Layers," RFC 1122, IETF, October 1989.

[38] C. Jennings, B. Lowekamp, E. Rescorla, S. Baset, and H. Schulzrinne, "A SIP Usage for RELOAD," Internet-Draft – work in progress 06, IETF, July 2011.

[39] 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.

[40] ITU, "ASN.1 encoding rules: Specification of Basic Encoding Rules (BER), Canonical Encoding Rules (CER) and Distinguished Encoding Rules (DER)," recommendation, ITU, 2008.

[41] N. Mavrogiannopoulos and D. Gillmor, "Using OpenPGP Keys for Transport Layer Security (TLS) Authentication," RFC 6091, IETF, February 2011.

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

[43] J. Jonsson and B. Kaliski, “Public-Key Cryptography Standards (PKCS) #1: RSA Cryptography Specifications Version 2.1,” RFC 3447, IETF, February 2003.

[44] “The eMule Project homepage.” http://www.emule-project.net, 2011.

[45] “The KaZaA homepage.” http://www.kazaa.com, 2011.

[46] M. Nottingham and E. Hammer-Lahav, “Defining Well-Known Uniform Resource Identifiers (URIs),” RFC 5785, IETF, April 2010.

[47] X. Jiang, N. Zong, R. Even, and Y. Zhang, “An extension to RELOAD to support Direct Response and Relay Peer routing,” Internet-Draft – work in progress 05, IETF, March 2011.

[48] ITU, “Recommendation G.114 - One-way transmission time,” recommendation, ITU, 2003.

[49] ITU, “Recommendation G.107 - The E-model, a computational model for use in transmission planning,” recommendation, ITU, 2005.

[50] “The Speex projectpage.” http://www.speex.org, 2009.

[51] “Advanced Video Coding for Generic Audiovisual Services,” Tech. Rep. Recommendation H.264 & ISO/IEC 14496-10 AVC, v3, ITU-T, 2005.

[52] J. Rosenberg and H. Schulzrinne, “An Offer/Answer Model with Session Description Protocol (SDP),” RFC 3264, IETF, June 2002.

[53] H. Schulzrinne, S. Casner, R. Frederick, and V. Jacobson, “RTP: A Transport Protocol for Real-Time Applications,” RFC 3550, IETF, July 2003.

[54] T. Berners-Lee, R. T. Fielding, and L. Masinter, “Uniform Resource Identifier (URI): Generic Syntax,” RFC 3986, IETF, January 2005.

[55] “XML Schema Part 2: Datatypes Second Edition,” W3C Recommendation, World Wide Web Consortium, October 2004.

[56] “The Skype homepage.” http://www.skype.com, 2009.

[57] “Group video calling Product Datasheet.” http://download.skype.com/share/business/guides/gvc-product-datasheet.pdf, 2011.

[58] S. Romano, A. Amirante, T. Castaldi, L. Miniero, and A. Buono, “A Framework for Distributed Conferencing,” Internet-Draft – work in progress 09, IETF, June 2011.

[59] D. Bryan, P. Matthews, E. Shim, D. Willis, and S. Dawkins, “Concepts and Terminology for Peer to Peer SIP,” Internet-Draft – work in progress 03, IETF, October 2010.

[60] S. Ratnasamy, M. Handley, R. M. Karp, and S. Shenker, "Topologically-aware overlay construction and server selection," in *Proc. of 21st Annual Joint Conference of the IEEE Computer and Communications Societies (INFOCOM '02)*, (Washington, DC, USA), pp. 1190–1199, 2002.

[61] Z. Xu, C. Tang, and Z. Zhang, "Building topology-aware overlays using global soft-state," in *Proc. of the 23rd Int. Conf. on Distributed Computing Systems (ICDCS '03)*, (Washington, DC, USA), p. 500, IEEE Computer Society, 2003.

[62] A. Knauf, G. Hege, T. C. Schmidt, and M. Wählisch, "A Virtual and Distributed Control Layer with Proximity Awareness for Group Conferencing in P2PSIP," in *Proc. of IPTComm 2010*, Digital Library, (New York), pp. 122–133, ACM, August 2010.

[63] S. Josefsson, "The Base16, Base32, and Base64 Data Encodings," RFC 4648, IETF, October 2006.

[64] R. Sparks, "The Session Initiation Protocol (SIP) Refer Method," RFC 3515, IETF, April 2003.

[65] A. Knauf, "Scalable, Distributed Conference Control in Tightly Coupled SIP Scenarios," Bachelor Thesis, Hamburg University of Applied Science, September 2009.

[66] A. Knauf, G. Hege, T. Schmidt, and M. Waehlisch, "A RELOAD Usage for Distributed Conference Control (DisCo)," Internet-Draft – work in progress 01, IETF, December 2010.

[67] C. J. Fidge, "Timestamps in Message-Passing Systems that Preserve the Partial Ordering," in *11th Australian Computer Science Conference*, (University of Queensland, Australia), pp. 56–66, February 1988.

[68] D. Ratner, P. Reiher, and G. J. Popek, "Dynamic Version Vector Maintenance," Tech. Rep. CSD-970022, University of California, Los Angeles, October 1997.

[69] A. Knauf, T. C. Schmidt, and M. Wählisch, "Scalable Distributed Conference Control in Heterogeneous Peer-to-Peer Scenarios with SIP," in *Mobimedia '09: Proc. of the 5th International ICST Mobile Multimedia Communications Conference*, ACM Digital Library, (Brussels, Belgium), pp. 1–5, ICST, Sept. 2009.

[70] "The NIST JAIN-SIP homepage." http://jain-sip.dev.java.net/, 2009.

[71] Amit Ranpise, "RELOAD - implementation of Storage module and message encoder decoder." http://www.ietf.org/mail-archive/web/p2psip/current/msg05669.html, 2010.

[72] 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.

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

[74] “http://inet.cpt.haw-hamburg.de/teaching/ss-2011/master-projekt-i/alex-projekt2.pdf.”

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

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

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

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

[79] R. Housley, W. Polk, W. Ford, and D. Solo, “Internet X.509 Public Key Infrastructure Certificate and Certificate Revocation List (CRL) Profile,” RFC 3280, IETF, April 2002.

[80] “PJSIP homepage, feature list.” http://www.pjsip.org/sip_media_features.htm, 2010.

[81] “The Trang homepage.” http://www.thaiopensource.com/relaxng/trang.html, 2008.

[82] A. Knauf, G. Hege, T. C. Schmidt, M. Wählisch, L. Grimm, T. Kluge, and P. Pogrzeba, “Transparent Conferencing Without Central Control - Demonstration of the DisCo Approach in P2PSIP,” October 2011.

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

[84] “The Fedora homepage.” http://fedoraproject.org/, 2012.

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

[86] “The Cooperative Association for Internet Data Analysis homepage.” http://www.caida.org/home/, 2010.

[87] “The Statista homepage.” http://de.statista.com/statistik/daten/studie/39490/umfrage/anzahl-der-internetnutzer-weltweit-nach-regionen, 2012.

[88] “The Internet World Stats homepage.” http://www.internetworldstats.com/stats.htm, 2012.

List of Figures

Listings

Danksagung

An dieser Stelle möchte ich mich zunächst bei meiner zukünftigen Ehefrau bedanken, da sie mich wären der gesamten Arbeit immer moralisch getragen hat.

Dann gilt ein besonderer Dank an Prof. Dr. Thomas C. Schmidt, der meine Arbeiten stets gefördert hat und ohne den ich nicht so viel erreicht hätte.

Desweiteren, bedanke ich mich auch bei all den Mitarbeitern der Arbeitsgruppe INET, die mich bei meiner Arbeit immer inspiriert und unterstützt haben.

Versicherung über Selbstständigkeit

Hiermit versichere ich, dass ich die vorliegende Arbeit im Sinne der Prüfungsordnung nach §24(5) ohne fremde Hilfe selbstständig verfasst und nur die angegebenen Hilfsmittel benutzt habe.

Hamburg, 16. Januar 2012	
Ort, Datum	Unterschrift