NDN-RTC app design and protocol specification (DRAFT v0.2)

P. Gusev

September 14, 2013

Abstract

Contents

1 Overview

1.1 Participants and institutions

- Alex Horn
- Jeff Burke
- Jeff Thompson
- Peter Gusev
- Lixia Zhang
- Qiuhan Ding

1.2 Motivation

The video conferencing tool is one of those apps which is needed first of all for internal use and experimenting with real-time communication over NDN. As we are interested in increasing NDN popularity among non-research peers as well, we are trying to provide easy-to-setup NDN apps by employing web browser APIs. NDN-RTC is a JavaScript application which utilizes the WebRTC engine for media encoding/decoding and C++ NDN library for the transport layer.

1.3 Description

The main goal of NDN-RTC app is to provide real-time audiovisual communication between users and enable them to organize multiparty conferences. App should have simple intuitive user interface and should be easy-to-setup. The main functions provided by NDN-RTC app are represented by the Use-Case diagram on Figure 1. Main use cases of the app are the follows:

- **Discovery**
  - users should be able to discover other users of NDN-RTC app in order to start audio/video/text sessions with them;
  - users should be able to discover current ongoing multiparty conferencies.
- **Join conference**
  - users should be able to start new peer-to-peer or multiparty conference (audio-, video- or text-based);
  - users should be able to join existing conferences if they are allowed.

- **Leave conference**
  - users should be able to stop the conference they have initiated previously;
  - users should be able to leave the conference they have joined previously.

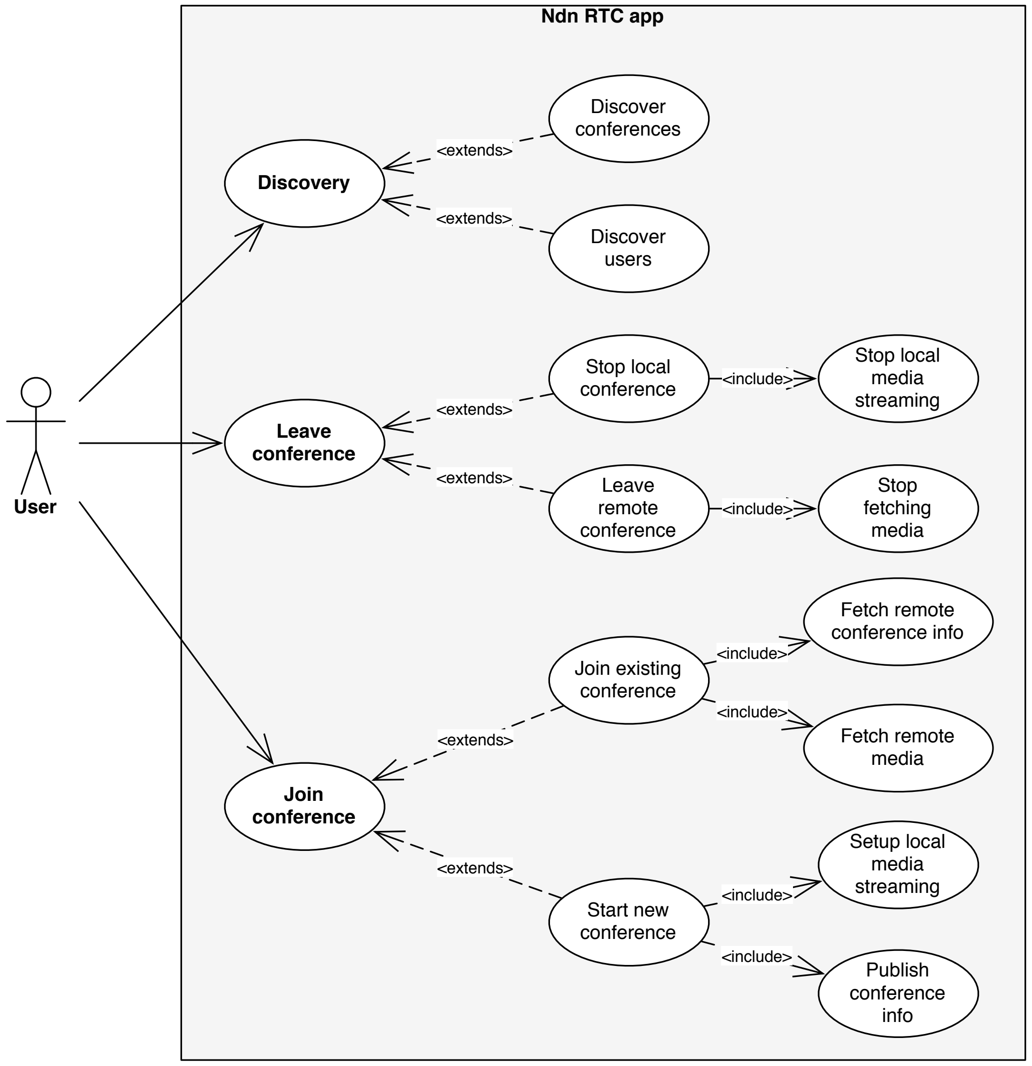

Figure 1: NDN-RTC Use-Cases

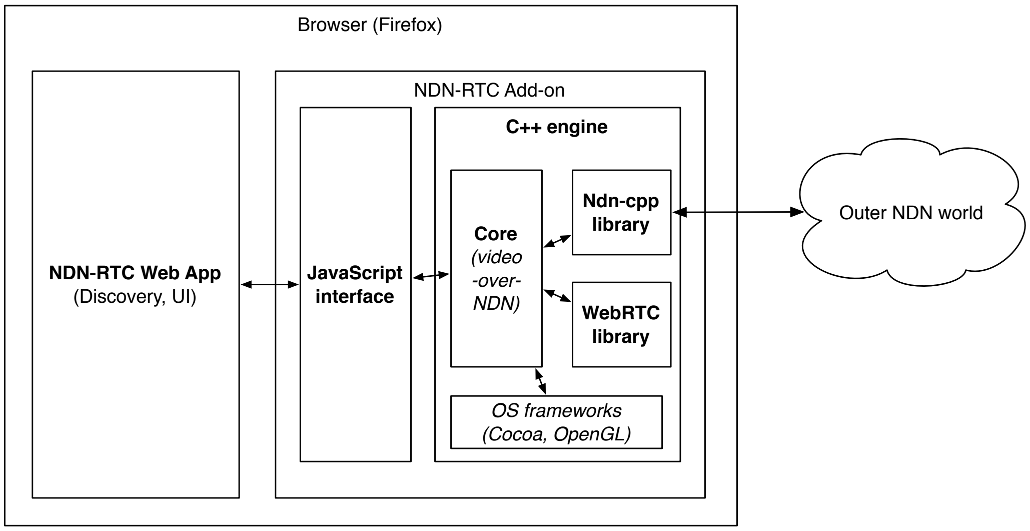

Figure 2: NDN-RTC architecture

In order to satisfy easy-to-setup requirement, NDN-RTC was designed as a combination of Javascript and C++ modules which together operate inside Firefox browser. C++ module is responsible for real-time media communication and implemented as an add-on, while Javascript app provides all discovery functionality and UI (see Figure 2).

Two modules of NDN-RTC are:

1. **NDN-RTC Add-on**
   Add-on consists of several submodules, namely:
   - **C++ engine**
     - *Core* - implements NDN-RTC protocol for enabling real-time media communication over NDN networks.
     - *NDN-Cpp library* - provides interfaces for NDN-specific network interaction.
     - *WebRTC library* - provides interfaces for media encoding/decoding, echo-cancellation, audio/video synchronisation, etc.
   - **Javascript interace** - provides Javascript API for web applications for accessing C++ engine functionality.
2. **NDN-RTC Web App**
   Provides general conference discovery between peers and UI.

One should note that NDN-RTC Add-on is required to be installed in a browser for proper functioning of NDN-RTC Web App. In other words, without the add-on NDN-RTC Web App will not be able to provide RTC-over-NDN functions for its users, whereas add-on is solely independent module and the API it provides can be employed in third-party web apps.

2 Protocol specification

2.1 NDN Namespaces

Naming is an important aspect of any NDN application. There are three different namespaces used in NDN-RTC app:

- **user namespace** - used for accessing individual user's media data and peer-to-peer session initiation (see Figure 3);
- **session namespace** - used for accessing conferences' info (both peer-to-peer and many-to-many) and encrypted ChronoChat room;
- **discovery namespace** - used for discovery mechanisms (see Section 2.2 for details).

Each user of NDN-RTC app is uniquely identified by a prefix URI which contains username (top level on the Figure 3). A user (further called a *producer*) publishes her media streams (means audio or video streams) under *streams* namespace as presented in Figure 3. Each stream is encoded with its own symmetric key (see Section 2.4 for details). Producer can publish one particular stream in several copies with different encoding parameters (for instance, to allow consumers fetch media stream wich is the most appropriate for current network conditions). The information about streams' encoding parameters can be accessed by issuing interest in producer's *info* namespace (***/<user prefix>/streams/video0/vp8-640/info***) which will be satisfied by content object presented in Table 1.

Any user who wants to fetch producer's media stream (further called a *consumer*) needs to express interest with URI corresponding to a certain media stream (see Section 2.3 for more details on RTC).

All type of communications inside NDN-RTC (whether it is peer-to-peer or many-to-many conference) are uniquely identified by *sessions*. Each session has an initiator and participants. Session should have at least two

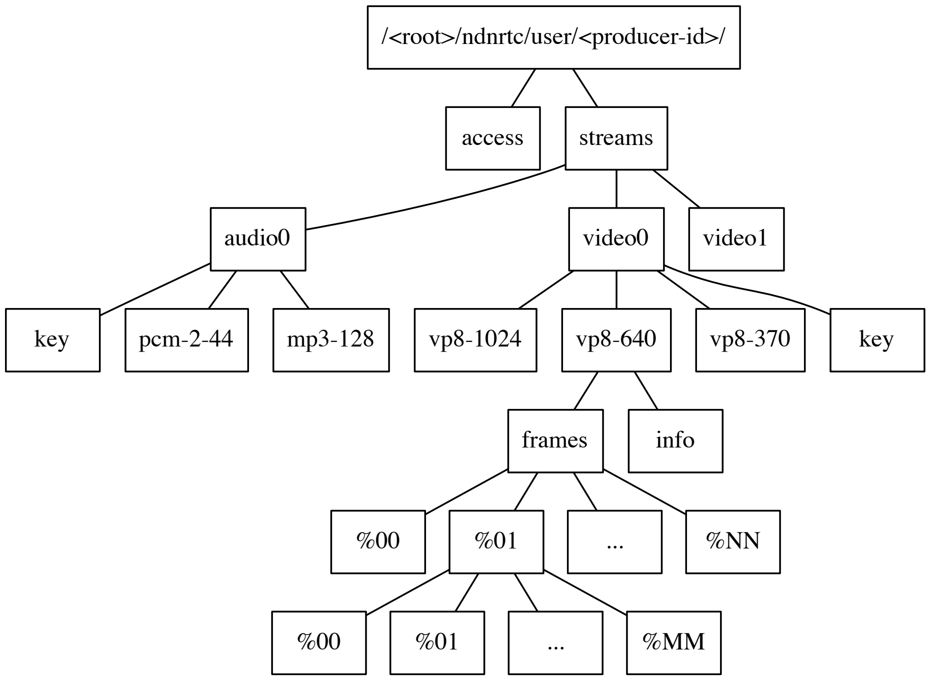

Figure 3: NDN-RTC user namespace

participants to be initiated. Each session has a description (see Figure 5), which contains such information as participant list, encryption key, conference name, description, etc. NDN-RTC sessions are placed under the session namespace:

/<root>/ndnrtc/session/<session-id>

This prefix is used as a ChronoChat URI. However, there are several modifications to the traditional ChronoChat protocol [**?**]. First, in addition to the existing ChronoChat namespace, a new *info* node is added. By issuing interests in this namespace (***/<root>/ndnrtc/session/<session-id>/info***) users can access conference announcement information. NDN-RTC adopts Audio Conference Tool approach [**?**] for secured conference announcements (see Section 2.4 for details).

Moreover, several additional control message types were added to the existing ChronoChat messages. These messages are used for propagating service information among participants like added/removed media streams, changing encryption keys, etc. (see Table 2).

Table 1: Data object example for stream's "info" namespace

/<user prefix>/streams/video0/vp-640/info
StartTime: 1379894400
Type: "video"
Codec: "vp8"
Framerate: 30
Bitrate: 4000
Producer's Signature

Table 2: NDN-RTC conference service messages

Type	Parameters	Description
StreamChanged	• media stream prefix • type of change: *added, removed*	Message is sent anytime user adds or removes a stream from an ongoing conference.
StreamKeyChange	• media stream prefix • frame number • encryption key	Message is sent when producer changes encryption key for a media stream. The new key should be used after the specified frame number.
ConferenceInfoUpdate		Message is sent when conference initiator changes any conference description data. Other participants can fetch new version of conference data after receiving this message.

There is a third namespace used in NDN-RTC which allows users to discover current conferences and each other called discovery namespace. Next section 2.2 explains it in detail.

2.2 Discovery

In order to start a chat session (either video, audio or textual) between two peers, at least one peer need to know name prefix of the other one. Likewise, users need a mechanism for conferences discovery. NDN-RTC app provides these two types of discoveries:

- **User discovery** - allows to see currently "active" or "visible" users of NDN-RTC.
- **Conference discovery** - allows to see current ongoing conferences.

Either type of discovery is essentially implemented in the same way using existing ChronoChat protocol with a restriction to allowed message types. Once a user started NDN-RTC app and would like to be visible for other users, she joins (sends a *Join* message) to a global service chat room with the prefix: ***/ndn/broadcast/ndnrtc/roster***. It's a special chat room which does not allow any type of data messages except *Join*, *Leave* and *Heartbeat*. In all other respects, it is treated as usual ChronoChat room with syncing, recovery, etc. and each user who joins this room (which basically in this context means "become visible to other users") is synced with chat room data automatically by the means of ChronoChat protocol. By providing name prefixes in *Join* messages (either for a user or a conference, see Figure 4) users can easily fetch these prefixes and initiate new or join existing sessions. Users who'd like to stay visible should send *Heartbeat* messages periodically. The same ideas of joining system roster chat room applies for conferences as well.

2.3 Real-time communication

1. Upon successful discovery of a video producer URI, the consumer issues an interest in the index namespace and gets data about the media stream's parameters ($FrameRate$ and codecs).

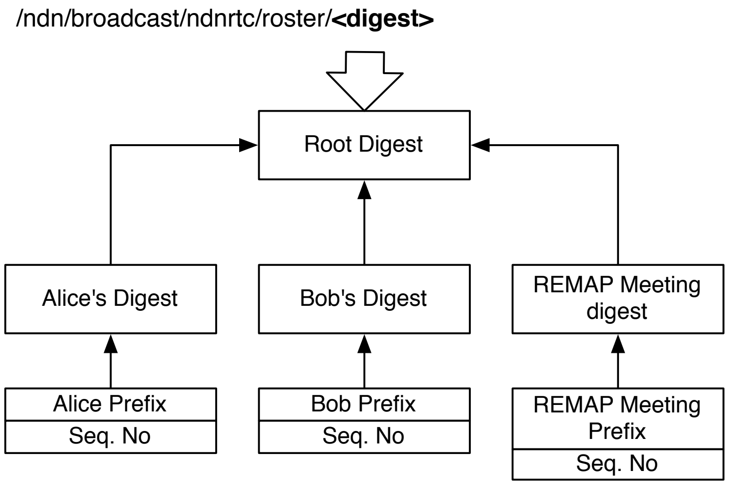

Figure 4: NDN-RTC discovery ChronoChat digest tree

2. The consumer issues $Interest_0$ with the *RigthMostChild=**true*** selector in the namespace for the frames like this: */root/mediadata* and waits until the first segment is received.

3. The consumer switches to **"Chase Mode"**

Chase Mode

1. THe consumer extracts the frame number - FN from the obtained *DataObject* and pipelines interests at a rate of $2 * FrameRate$ (twice the producer data rate) with *RightMostChild=**true*** in the namespace for segments like this: */root/mediadata/FN/0.* i.e. the pipeline contains interests $Interest_{FN}$, $Interest_{FN+1}$, $Interest_{FN+2}$,...

2. The consumer watches two parameters: *DeliveryRate* of frames, $RoundTripTime$ for interests:

   - If *DeliveryRate* is the same as $FrameRate$ and $RoundTripTime$ is not growing steadily, the consumer switches to **Fetch Mode** (see below)
   - If either *DeliveryRate* and/or $FrameRate$ are growing, the consumer chooses a lower quality (by modifying the prefix) and re-enables **Chase Mode**

Fetch Mode

1. The consumer extracts the latest frame number - LFN and sets up a *Frame interest* pipeline at a frequency of $2 * FrameRate$:

   for a 24 fps video:

   - **t = 0 sec**
     Interest for */root/video/LFN/0*, timeout = 1.5 sec
   - **t = 1/48s**
     Interest for */root/video/LFN+1/0*, timeout = 1.5 sec
   - **t = 2/48s**
     Interest for */root/video/LFN+2/0*, timeout = 1.5 sec
   - ...

   The number of segments per frame ($NumSegs$) is indicated using the *FinalBlockID* field of each segment's DataObject. The consumer sets up a *Segments interest* pipeline for each frame like this:

   - Interest for */root/video/LFN/0*
   - Interest for */root/video/LFN/1*
   - Interest for */root/video/LFN/2*
   - ...
   - Interest for */root/video/LFN/$SegNum$*

2. The consumer periodically issues a **"probe"** interest with the last known frame number and *RightMostChild=**true*** in order to detect a lag from the producer. If a considerable lag was detected, the consumer chooses stream parameters for a lower rate and switches to the **Chase Mode**

3. The consumer watches the value of $DeliveryRate$ and $RoundTripTime$ for the interests. If either of these values starts to grow, the consumer should choose stream parameters for a lower rate and switch to the **Chase Mode** in order to minimize the lag from the producer.

2.4 Security concerns

In order to provide secure communication between users, two main security problems should be solved:

1. Data encryption and verification
2. User authenticity

Encryption and verification. As NDN-RTC app requires communications over untrusted network, transmitted data should be encrypted. Moreover, as there is a requirement of real-time communication, encryption should be efficient enough in order to minimize the delay. In this case, choosing fast symmetric encryption over a slower asymmetric encryption is more preferable. Once user started to publish media stream (either audio or video), a new encryption key is created and the data is published encrypted by this key.

Data verification is achieved by leveraging NDN embedded mechanisms for per-packet signing. Therefore, consumer can verify received packet by checking it's signature.

Authenticity. NDN-RTC app should be perceived in a "producer-consumer" approach, i.e. there is a producer of media data somewhere on the network, and there is a consumer who'd like to get access to the producer's data. In this case, a producer should be the only authority who is in charge of giving access to his/her data.

As producer's data is encrypted using symmetric key, producer can control access to his data by sending this key only to trusted users. Any consumer who wants to get access for media stream from a producer needs to retrieve an encryption key first by issuing interest in the *key* namespace of a media stream by adding his name and a timestamp:

/<producer prefix>/streams/video0/key/<caller>/<timestamp>

Adding a current timestamp prevents network from responding with previously cached data. Encryption key is returned to the consumer in a data object encrypted by the consumer's public key.

A producer can change encryption keys for media streams anytime and notify other conference participants using *StreamKeyChange* service chat message (see Table 2). Per-stream encryption keys and ability to change them on-the-go makes feasible complex conference scenaria in NDN-RTC app: for instance, if a user participates in several conferences simultaneously, she can provide only audio stream in one conference, audio+video in another and screen sharing video in third conference. With per-stream keys other participants can not fetch media streams wich are not announced for the current conference.

Session announcement. Secured session announcement approach was borrowed from Audio Conference Tool (see [**?**] for details). Upon creation of new session, initiator generates a pair of keys K_e and K_d and symmetric session key K_s. Asymmetric keys are used for secure publishing of session's information, while symmetric key is used by all the participants for chat data encryption. A sample structure of session announcement packet is presented on Figure 5.

We assume, that public keys of all the participants accessible through the network and can be easily verified. Announcement data contains hashes of each participants' public key so they can quickly check, whether they are among invitees or not. If the user is among conference participants, she can retrieve encryption key K_d and decrypt conference information as well as session key K_s. Using session key K_s, user can join conference straight away (by sending encrypted *Join* message to the chat).

2.5 Session initiation

2.5.1 Peer-to-peer

Initial interaction of two users is depicted by Figure 6.

The user who would like to make a call issues an interest in an *access* namespace of a callee's prefix by adding his name and current timestamp:

/<root>/ndnrtc/user/<callee>/access/<caller>/<timestamp>

Timestamp is needed in order to prevent network from answering with previously cached data. If calle answers the call, she performs three consecutive steps:

- initiates a new session by generating session id and announcement data and publishes them in session's *info* namespace;

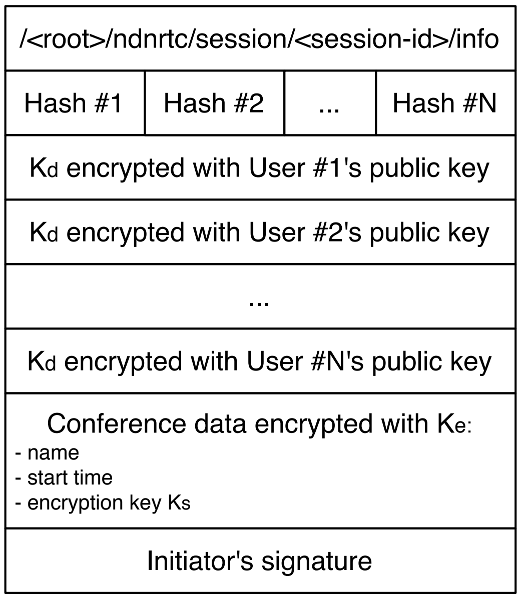
/<root>/ndnrtc/session/<session-id>/info
Hash #1
Hash #2
...
Hash #N
Kd encrypted with User #1's public key
Kd encrypted with User #2's public key
...
Kd encrypted with User #N's public key
Conference data encrypted with Ke:
- name
- start time
- encryption key Ks
Initiator's signature

Figure 5: Session announcement data

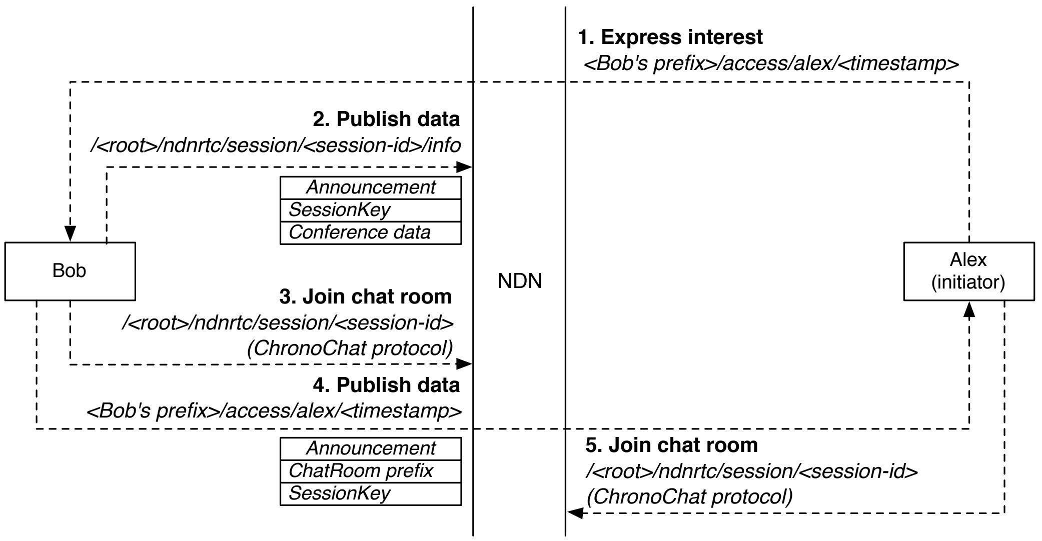

Figure 6: NDN-RTC peer-to-peer session initiation

- creates new chat room with prefix ***/<root>/ndnrtc/sessions/<session-id>*** and joins it (using ChronoChat protocol);

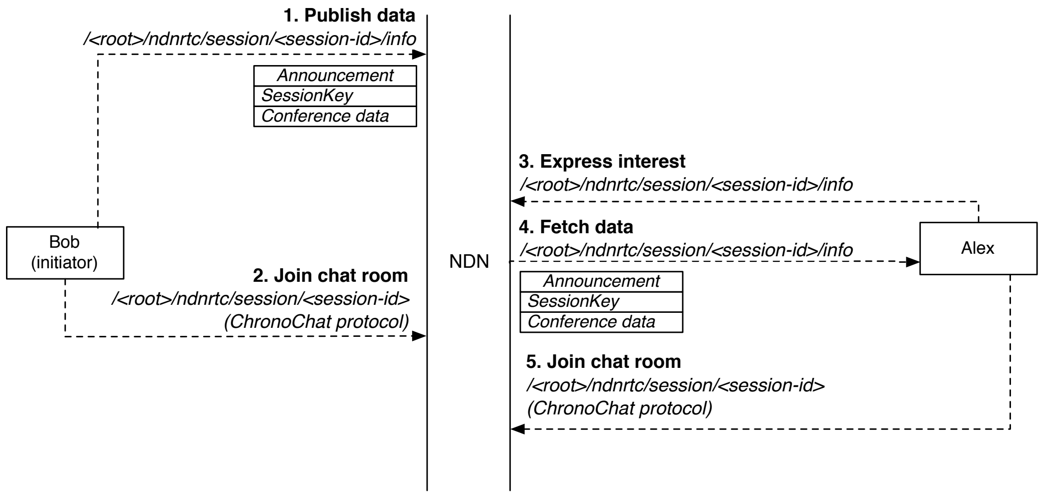

Figure 7: NDN-RTC many-to-many session initiation

- sends session announcement with chatroom prefix back to caller encrypted by the caller's public key.

Upon receiving a session announcement, the caller joins the chatroom and starts communication session by exchanging service or text messages.

2.5.2 Many-to-many

In many-to-many scenario, unlike in peer-to-peer mode, session initiator does not interact with any of the participants directly and creates new session by publishing announcement data object in session's *info* namespace. It is up to participants how they can discover the chatroom prefix (for instance, initiator may publish it in the discovery namespace or send it via e-mail). By obtaining chatroom prefix, participants are able to fetch session announcement data, extract encryption session key and join a conference. Many-to-many scenario is presented on Figure 7

3 App design

3.1 Overview

TBD

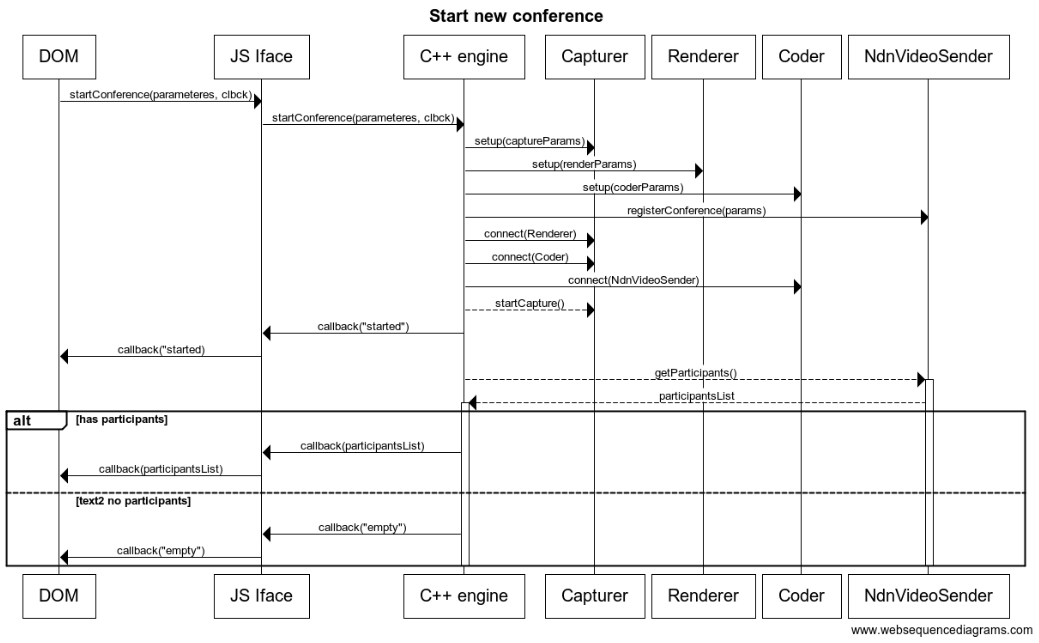

Figure 8: Sequence diagram for starting a conference

3.2 C++ Firefox add-on

TBD This sections describes the internal architectural approach for the C++ part of the add-on.

One can start learning how the add-on works by looking at the sequence diagrams of the main use-cases: Starting a conference (Figure 8), Joining an existing conference (Figure 9) and Leaving a conference (Figure 10).

3.3 Javascript Web application

3.3.1 Overview

TBD

3.3.2 UI design

TBD

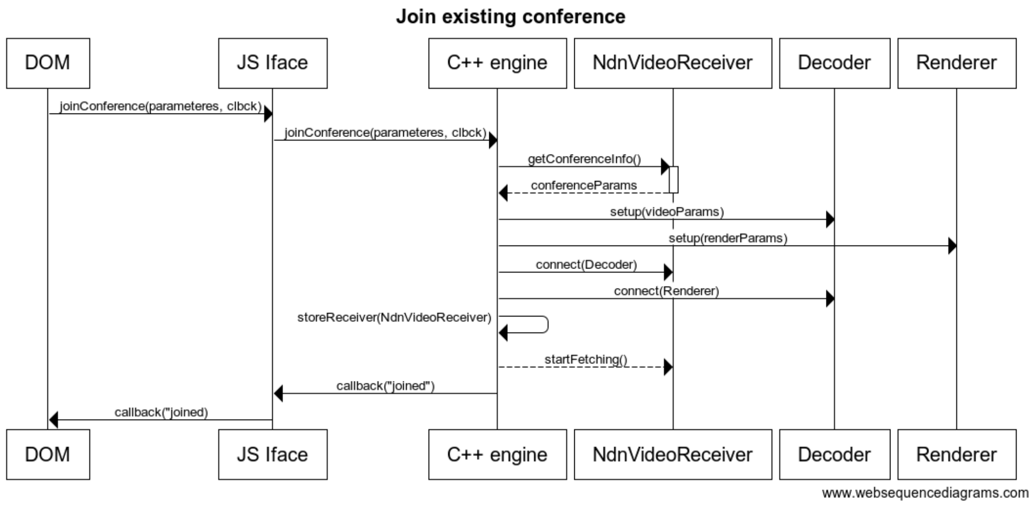

Figure 9: Sequence diagram for joining an existing conference

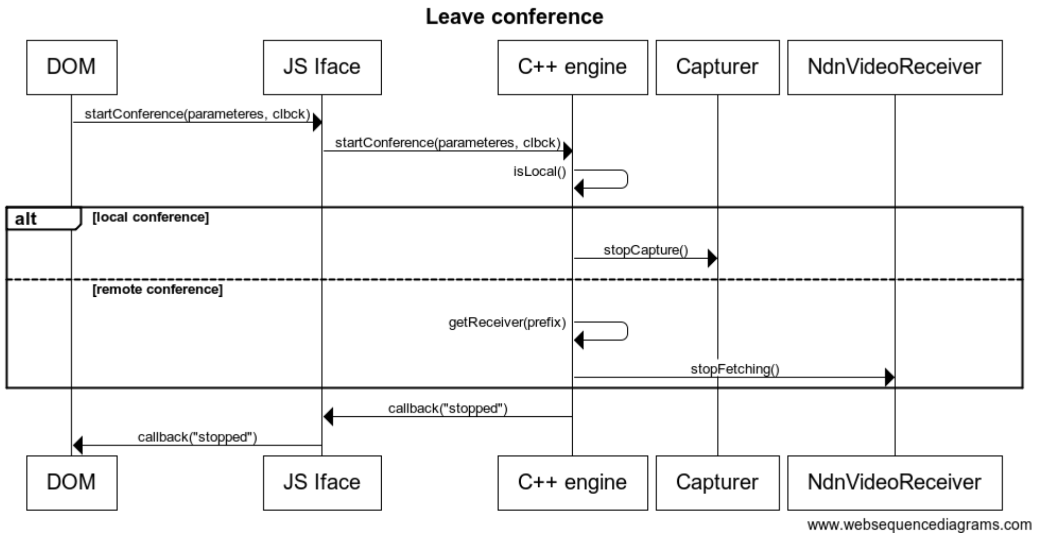

Figure 10: Sequence diagram for leaving a conference

4 Next steps

- Provide user authentication in video conferences
- Implement secure media transfer
- Scalable video encoding

5 References

[?], [?]