Analysis of RTCWeb Data Channel Transport Options

Hasan Mahmood Aminul Islam

Helsinki December 10, 2012

UNIVERSITY OF HELSINKI
Department of Computer Science

HELSINGIN YLIOPISTO — HELSINGFORS UNIVERSITET — UNIVERSITY OF HELSINKI

Tiedekunta — Fakultet — Faculty Faculty of Science	Laitos — Institution — Department Department of Computer Science
Tekijä — Författare — Author Hasan Mahmood Aminul Islam	
Työn nimi — Arbetets titel — Title Analysis of RTCWeb Data Channel Transport Options	
Oppiaine — Läroämne — Subject Computer Science	

Työn laji — Arbetets art — Level	Aika — Datum — Month and year December 10, 2012	Sivumäärä — Sidoantal — Number of pages 63 pages + 6 appendices

Tiivistelmä — Referat — Abstract

The Web has introduced a new technology in a more distributed and collaborative form of communication, where the browser and the user replace the web server as the nexus of communications in a way that after the call establishment through web servers, the communication is performed directly between browsers as peer to peer fashion without intervention of the web servers. The goal of Real Time Collaboration on the World Wide Web (RTCWeb) project is to allow browsers to natively support voice, video, and gaming in interactive peer to peer communications and real time data collaboration.

Several transport protocols such as TCP, UDP, RTP, SRTP, SCTP, DCCP presently exist for communication of media and non-media data. However, a single protocol alone can not meet all the requirements of RTCWeb. Moreover, the deployment of a new transport protocol experiences problems traversing middle boxes such as Network Address Translation (NAT) box, firewall. Nevertheless, the current implementation for transportation of non-media in the very first versions of RTCWeb data does not include any congestion control on the end-points. With media (i.e., audio, video) the amount of traffic can be determined and limited by the codec and profile used during communication, whereas RTCWeb user could generate as much as non-media data to create congestion on the networks. Therefore, a suitable transport protocol stack is required that will provide congestion control, NAT traversal solution, and authentication, integrity, and privacy of user data. This master's thesis will give emphasis on the analysis of transport protocol stack for data channel in RTCWeb and selects Stream Control Transmission Protocol (SCTP), which is a reliable, message oriented general-purpose transport layer protocol, operating on top of both IPv4 and IPv6, providing congestion control similar to TCP and additionally, some new functionalities regarding security, multihoming, multistreaming, mobility, and partial reliability. However, due to the lack of universal availability of SCTP within the OS(s), it has been decided to use the SCTP userland implementation.

WebKit is an open source web browser engine for rendering web pages used by Safari, Dashboard, Mail, and many other OS X applications. In WebKit RTCWeb implementation using GStreamer multimedia framework, RTP/UDP is utilized for the communication of media data and UDP tunnelling for non-media data. Therefore, in order to allow a smooth integration of the implementation within WebKit, we have decided to implement GStreamer plugins using SCTP userland stack..

This thesis work also investigates the way Mozilla has integrated those protocols in the browser's network stack and how the Data Channel has been designed and implemented using SCTP userland stack.

ACM Computing Classification System (CCS): C.2.2 [Network Protocols]
C.2.4 [Distributed System]

Avainsanat — Nyckelord — Keywords
SCTP, Webkit, GStreamer, Data channel, NAT traversal, RTCWeb

Säilytyspaikka — Förvaringsställe — Where deposited

Muita tietoja — övriga uppgifter — Additional information

Acknowledgements

I would like show my gratitude to everyone who supported me throughout my thesis work. I would like to specially thank my supervisor Professor Sasu Tarkoma, University of Helsinki, for his splendid supervision, inspiration and valuable feedback throughout the work. I am also grateful to Toni Ruottu from University of Helsinki for his valuable feedback to my thesis work.

I would like to show special appreciation to my instructor Salvatore Loreto, PhD. from Nomadic Lab, Ericsson Research, Finland, for his constant supervision and valuable feedback during the thesis period. I would like to thank all my co-workers at Nomadic Lab, and specially Jouni Maenpaa for his constant support during the work.

I would also like to show special gratitude to Michael Tuexen from Muenster University of Applied Science, Randell Jesup from Mozilla for their valuable input and cooperation during my thesis work. I am also grateful to Tiina Niklandar and Pirjo Moen for their valuable advice during my Master degree program.

Finally, I thank my parents for their persistent affection and moral support throughout the period of my study. Further gratitude to my friends, family members, and everyone else who supported and inspired me during my whole life.

Helsinki, December 10, 2012

Hasan Mahmood Aminul Islam

Contents

3 GStreamer Plugin Creation Step

List of Figures

List of Tables

Abbreviations and Acronyms

DCCP	Datagram Congestion Control Protocol
DTLS	Datagram Transport Layer Security
ICE	Interactive Connectivity Establishment
IP	Internet Protocol
MTU	Maximum Transfer Unit
ROAP	RTCWeb Offer/Answer Protocol
RTCP	Real-time Control Protocol
RTP	Real-time Transport Protocol
SACK	Selective Acknowledgement
SCTP	Stream Control Transmission Protocol
SDP	Session Description Protocol
SIP	Session Initiation Protocol
SRTP	Secure Real-Time Transport Protocol
SSN	Stream Sequence Number
STUN	Session Traversal Utilities for NAT
TCP	Transmission Control Protocol
TSN	Transmission Sequence Number
TURN	Traversal Using Relays around NAT
UDP	User Datagram Protocol

1 Introduction

1.1 Motivation

The Web has introduced a new technology in Web Architecture with a vision to pose a more distributed and collaborative form of communication, where the browser and the user replace the web server as the nexus of communications. Internet Engineering Task Force (IETF) and World Wide Web Consortium (W3C) are working together on extending the web architecture. In particular the Real Time Collaboration on the World Wide Web (RTCWeb) project aims to allow browsers to natively support interactive peer to peer communications in the form of voice-video or gaming, and real time data collaboration. In such architectures, browsers become the source and sink of both media packets that flow directly from one browser to another while web server are only involved during the ”setup” phase of the communication. The servers are used for call establishment and transporting control information. Once the connection is established, the communication is performed directly between browsers without intervention of the web servers. RTCWeb is the initial attempt to introduce peer to peer communication in the Web Architecture. The server is still required as a rendezvous point for the browsers and to download the JavaScript that contains the application logic. Once the JavaScript is downloaded, "everything" happens in a peer to peer fashion among the browsers involved in the application. HyperText Markup Language HTML version 5 (HTML5) is transforming browsers to become a rich, integrated service environment that natively supports audio and video, persistent local storage, and offline web applications.

RTCWeb includes media data (e.g., audio, video) as well as non-media data (e.g., character screen position within an multiplayer HTML5 video game, text file, text chat). The communication path for media data is referred as media channel and for non-media data is referred as data channel. Data channel is designed to provide a generic transport service allowing web browsers to exchange generic data in a bidirectional peer to peer fashion that supports in-sequence, out-of-sequence, reliable and unreliable data transmission. The issue investigated in this work is how to handle non media data in the best suitable way in the context of RTCWeb. This issue is still under consideration among the researchers of RTCWeb.

Presently, there are several transport protocols suitable for media as well as data channel such as TCP [Pos97], UDP [Pos80], Real-time Transport Protocol (RTP) [SCFJ03], Secure Real Time Transport Protocol (SRTP) [BMN+04], Stream Control Transmission Protocol (SCTP) [SXM+07], and Datagram Congestion Control Protocol (DCCP) [KHF06]. However, still there is no such protocol which is best suitable for browser to browser data communication following all the requirements in the context of RTCWeb. While RTP over UDP is mostly used in real time communication, only UDP is not preferable for several constraints requested by data communication such as reliability, ordering, data integrity and especially lack of congestion control mechanism. TCP based protocol presents more security risk as it could be used by the web implementers to run attacks against Domain Name System (DNS), or other HTTP elements by opening many connections and leaving them dormant under heavy load [Bel10]. Therefore, a suitable transport protocol stack that provides reliability, ordering, congestion control mechanism, NAT traversal solution, and authentication, security and privacy, is highly appreciated for data channel.

1.2 Problem Definition

If RTCWeb media and data channels are used in parallel within RTCWeb connection, the data channels may create substantial negative impact on the media streams since a web user can generate as much data as the user decides to do. consequently, data channel may create congestion on the network. Therefore, a suitable congestion control mechanism is required for RTCWeb connection so that data channel could fairly compete with media streams. Congestion control mechanism could able to provide fairness to all traffic involved in the network. Congestion control mechanism for data channel is highly essential when there exist multiple data channels used in parallel with media channel in order to lessen queuing delay, and fairly compete with available bandwidth.

Therefore, congestion-control mechanisms in RTCWeb should consider the following issues during transmission [TLJ12b, AJ12]:

- There should be suitable congestion control mechanisms either individually or in conjunction with the media streams so that data transport cannot create congestion on media streams.
- Both reliable and unreliable, ordered and unordered data transmission should be supported.
- All media streams as well as data streams should be congestion controlled

and congestion control mechanisms should provide streams to act fairly with TCP. RTCWeb may include multiple data flows which would be individually congestion controlled.

- Congestion control algorithms should work for data channels even if there exist no media channels or those are inactive in one or both directions.
- Potential prioritization on each data stream with respect to other streams as well as media streams.

There are some challenges in browser to browser communication since no single transport protocol could cover all the use case requirements of RTCWeb. The deployment of a new transport protocol experiences problems traversing middle boxes such as Network Address Translation (NAT) box [SE01], firewall. Additionally, there is lack of universal identification system such as telephone numbers or email addresses in communication. Therefore, RTCWeb clients need to implement NAT traversal mechanism, since most of the RTCWeb clients will be web browsers residing behind a NAT and/or firewall. NAT traversal is a general term for technologies that set up and preserve Internet protocol connections passing through NAT boxes.

However, it is necessary to have a suitable stack of transport protocol that is able to provide the above congestion control requirements, NAT traversal solution, and authentication, integrity, and privacy of user data.

1.3 Research Goals and Contributions

At initial stage of this research, study on different kinds of transport layer protocols such as SCTP, DCCP and other DCCP variants, TCP, RTP, SRTP etc., was performed. Currently, there is a set of W3C JavaScript APIs available for browser to browser communication called PeerConnection. A PeerConnection allows two users to communicate directly through browsers. Communications are coordinated by signaling channel via the web server. Once the connection is established, the browsers use PeerConnection for exchanging media data as well as non-media data directly without the intervention of the server.

The current implementation for transportation of non-media data does not include any congestion control on the end-points. The goal of this work is to select a suitable stack of protocols that provides congestion control on each peer, NAT traversal solution, and authentication, integrity, and privacy of user data. This research selects SCTP for data transmission. Due to the lack of universal availability of SCTP within the OS(s), it has been decided to use the SCTP userland implementation. In order to provide security and confidentiality SCTP will be implemented on top of DTLS over UDP. The encapsulation over UDP facilitates the NAT traversal issue.

In this research, the WebKit implementation is investigated for browser to browser communications. WebKit is an open source web browser engine. Currently, it is used by Safari, Dashboard, Mail, and many other OS X applications. In WebKit implementation, PeerConnection uses RTP over UDP for the communication of real time media data and UDP tunneling for non-media data. The implementation uses GStreamer for real time communication. Therefore, in order to allow a smooth integration of the implementation within WebKit, GStreamer plugin has been implemented to be able to send and receive data using SCTP over UDP.

This thesis also investigates the way Mozilla has integrated those protocols in the browser's network stack, and how the data channel has been designed and implemented.

1.4 Thesis outline

The thesis is logically structured in a way to provide the reader with suitable background knowledge before diving deep into the details of this thesis. After introducing the work in Chapter 1, an overview of different transport protocols, which serve different types of congestion control mechanisms, will be presented in Chapter 2. Chapter 3 will discuss basic NAT traversal technologies that is a great concern while planning to choose a transport protocol or transport protocol stack for data channel in RTCWeb. Then, an overview of RTCWeb architecture is presented briefly in Chapter 4 and pointed out which part of this architecture is the main focus of this research. In Chapter 5, this thesis focus on the analysis of data channel protocol and how SCTP can be used for browser to browser communications resolving the problem of legacy NAT traversal. The design and implementation of GStreamer solution for data channel with result analysis will be described in Chapter 6. Finally, the discussion will be concluded with suggestion of possible future works in Chapter 7.

2 Congestion Control and Avoidance

Transmission Control Protocol (TCP) is the most popular transport layer protocol in Internet applications. The congestion control mechanisms provided by TCP work well specially in the wired networks but often deteriorates performance in wireless environment, where packets can be lost for various reasons other than congestion [BPSK97]. Moreover, TCP is too aggressive in terms of in-sequence and ordered delivery for multimedia applications that demands congestion control without ordered reliable delivery, since TCP follows strict ordering of user data. Therefore, currently most multimedia applications use UDP as transport layer protocol, but UDP is not preferable for several constraints necessary for data communication such as reliability, ordering, data integrity, and especially lack of congestion control mechanism. Some new transport protocols such as SCTP, DCCP have been standardized to satiate the requirements of multimedia applications. In this chapter, different types of congestion control mechanisms will be outlined for several familiar transport protocols.

2.1 Transmission Control Protocol (TCP)

TCP [Pos97] is a connection-oriented, reliable standard transport protocol. TCP is highly preferable to Internet applications such as world wide web, email, remote administration, and file transfer. TCP is also referred to as byte oriented transport protocol i.e., TCP segment is organized as a continuous sequence of bytes. TCP is able to recover data that is damaged, lost, duplicated, or delivered out of order by the Internet communication system. For this purpose TCP uses sequence numbers and acknowledgements from a receiver. The basic idea is that each octet is assigned a sequence number in a segment that is transmitted. TCP sender keeps a copy on a retransmission queue after sending it over the network and starts a timer. If the acknowledgement is received before the timer expires, the segment is dequeued. Otherwise, the segment is retransmitted.

TCP provides flow and congestion control mechanisms through the use of congestion window [Pos97]. TCP starts a retransmission timer when an outbound segment is passed down to IP. If there is no acknowledgement from the receiver for the data in a given segment before the timer expires, then the segment is retransmitted. TCP retransmissions occur on the network all the time. If the network is not under congestion, the sender increases the window size by a fixed number every round trip time. In response to congestion detection, the sender decreases the transmission rate by a multiplicative factor, for instance, the congestion window is decreased by half. This algorithm is known as Additive increase, multiplicative decrease (AIMD) of the sending rate. The acknowledgement (ACK) number from receiver determines a range of acceptable sequence numbers beyond the last segment successfully received.

TCP Reno is a variant of basic TCP congestion protocol [APB09]. It applies four congestion control mechanisms: slow-start, congestion avoidance, fast retransmit and fast recovery [Jac88, Jac90]. Slow-start and congestion avoidance algorithms are utilized to control the amount of outstanding data being pushed into the network. TCP sender uses congestion window (`cwnd`) to limit the amount of data in sender side to be injected into the network before receiving an acknowledgement (ACK). Flow control is achieved through receiver's advertised window (`rwnd`) on the amount of outstanding data. Another state variable, the slow start threshold (`ssthresh`), defines the margin to promote the sender switching from the slow-start to congestion avoidance algorithm.

At the beginning of transmission into a network with unknown conditions, TCP applies slow-start algorithm to determine the available capacity of the network instead of injecting large burst of data congesting the network. This algorithm is also used after loss recovery by the retransmission timer. A non standard, experimental TCP extension states that the initial value of `cwnd` can be defined as the following equation [AFP02]:

$$cwnd = min(4 \times SMSS, max(2 \times SMSS, 4380 bytes)) \tag{2.1}$$

where sender maximum segment size (SMSS) is the size of the largest segment that the sender can transmit. This value can be based on the maximum transmission unit of the network, largest segment the receiver is willing to accept or other factors. The size does not include TCP/IP headers and options [APB09]. The initial value of `ssthresh` is set arbitrarily high and reduced upon congestion detection. Algorithm 1 is used to determine whether slow-start or congestion avoidance algorithms are applied. Algorithm 1 shows that the adjustment of `cwnd` is performed on every incoming non-duplicate ACK. The slow start algorithm continues until the value of `cwnd` exceeds the value of `ssthresh` and `cwnd` is increased by SMSS.

Congestion avoidance algorithm allows the sender to increase the `cwnd` by SMSS per RTT. The value of `ssthresh` is adjusted while detecting segment loss using retransmission timer and the given segment has not yet been resent by way of the

Algorithm 1 TCP slow-start and congestion avoidance

```
upon arrival of a new ACK
if cwnd < ssthresh then
    apply slow-start algorithm
    cwnd +=SMSS
else
    if cwnd >= ssthresh then
        apply congestion avoidance
        cwnd+=SMSS*SMSS/cwnd
    else
        apply slow-start or congestion avoidance
    end if
end if
```

retransmission timer, the value is set by Equation 2.2.

$$ssthresh = max(FlightSize/2, 2 \times SMSS) \tag{2.2}$$

where, `FlightSize` is the amount of outstanding data in the network. In summary, Algorithm 1 depicts that TCP sender applies slow-start algorithm to increase the `cwnd` from 1 SMSS to the new value of `ssthresh` after the retransmission of dropped segments is completed. When the value of `cwnd` exceeds or touches the value of `ssthresh`, congestion avoidance algorithm takes place again.

Fast Retransmit/Fast Recovery

When TCP receiver detects arrival of an out of order segment, it immediately sends duplicate ACK to TCP sender that includes the expected sequence number. This ACK is a duplicate of an ACK which was sent previously. From the sender's point of view, a duplicate ACK can be caused by a lost segment or just a reordering of segments. When incoming data segments fill in all or part of a gap in the sequence space, TCP receiver immediately sends an ACK.

Fast Retransmit algorithm is used to detect and repair losses based on incoming duplicate ACKs. Fast Retransmit and Fast Recovery of TCP Reno is used together as shown in Algorithm 2. The arrival of 3 duplicate ACKs determine the lost of a segment and TCP starts retransmitting a missing segment without waiting for the retransmission timer. The `cwnd` is set to `ssthresh` plus 3*SMSS. Therefore, Fast Recovery takes place to promote the transmission of new data until a non-duplicate ACK arrives.

Algorithm 2 Fast Retransmit and Fast Recovery algorithm for TCP Reno

```
1)
if three duplicate ACKs are received then
    ssthresh = max (FlightSize / 2, 2 * SMSS)
    Retransmit the lost segment
    cwnd=ssthresh+ 3*SMSS
end if
2) Upon arrival of each additional duplicate ACK
cwnd += SMSS
3) Transmit a segment if the value of cwnd and the receiver's advertised window
allows.
4) When an ACK indicating new data arrives:
cwnd = ssthresh
```

Generally, TCP Reno is not able to recover multiple losses of packets in a single flight. In TCP Reno, Fast Recovery exits upon the reception of new ACK. TCP NewReno is the modification of the standard implementation of the Fast Retransmit and Fast Recovery algorithms [HFGN12]. The modification introduces partial acknowledgements and a new variable `recover`. Acknowledgement for a retransmitted packet will acknowledge some but not all of outstanding packets being transmitted. It is known as partial acknowledgement. The value of `recover` records the highest sequence number transmitted in step 1 in Algorithm 2. When a TCP sender receives three duplicate ACKs, the value of `ssthresh` is reduced to half of the current congestion window and the TCP sender enters fast retransmit mechanism to recover the lost segment. When an ACK arrives, TCP New Reno will determine whether it acknowledges all of the data up to and including `recover`. If it is not affirmative, then the packet acknowledged by partial acknowledgement is retransmitted. This process continues until an acknowledgement, denoting the highest sequence number already transmitted, arrives and thereafter TCP NewReno will leave the fast recovery setting the value of `cwnd` to `ssthresh`. Then congestion avoidance algorithm takes place.

Both TCP Reno and TCP NewReno are not so efficient while multiple losses are experienced in the network. When a packet is dropped in the network, the subsequent successful arrival of packets are not acknowledged until the expecting sequence number arrives. Moreover, a new Reno TCP sender has to wait an entire RTT to recover each lost packet when there are multiple loss of packets in a window. This phenomena leads to redundant retransmission since some of the packets between missing packets may be received successfully.

TCP Selective Acknowledgement (SACK) [MMFR96] is able to notify the sender implicitly about the missing sequence numbers as well as all segments that have reached successfully. Therefore, TCP sender need to retransmit only missing segments.

2.2 User Datagram Protocol (UDP)

User datagram protocol (UDP) [Pos80] is a transport protocol that allows one application program to transmit data to a second application program with a minimum of protocol mechanisms without prior connection setup. UDP is an unreliable transport protocol and does not guarantee in order delivery of user messages. Unlike TCP, UDP has no inherent order as all packets are independent of each other and hence, does not have any built-in congestion control and avoidance algorithms. Real time applications, video conferencing, voice over IP (VoIP), that do not demand high reliability and are tolerant to packet loss, prefer UDP for transmission of application data.

UDP is faster than TCP as it allows continuous streaming without any acknowledgements from receiver side whereas TCP is to adjust window size and round trip time (RTT) depending on the conditions of the network. UDP supports tunnelling for new transport layer technologies, such as SCTP [SXM$^+$07], DCCP [KHF06] in order to traverse middle boxes between end-points in the network.

2.3 Datagram Congestion Control Protocol (DCCP)

The Datagram Congestion Control Protocol [KHF06] is a UDP-like transport layer protocol that provides bidirectional, unicast connections of congestion controlled, unreliable datagrams. DCCP is suitable for applications, such as IP phones, video conferencing, video on demand (VoD), online games, etc., that require lower delay, and do not demand high reliability. TCP is not suitable for these applications so that it incurs long delays due to its conservative congestion control mechanism. It should be noted that DCCP provides reliable handshakes for connection setup and tear down.

However, one of the most attractive salient features of DCCP is that the congestion control mechanisms are modularly separated from its core, and each DCCP endpoint is free to choose different congestion control methods according to its preference. Each congestion control mechanism is denoted by a unique ID (CCID) : a number in between 0 and 255. CCIDs 2,3 and 4 are currently defined, CCIDs 0, 1 and 5-255 are reserved. The negotiation of a suitable congestion control mechanism or other feature negotiation between two DCCP end-points is reliable.

DCCP congestion control ID 2 [FK06] denotes TCP like congestion control [APB09] that includes the variant of selective acknowledgement (SACK) [MMFR96, BAW$^+$12]. CCID 2 is advantageous for those applications that are adaptive to abrupt changes in the congestion window and could take advantage of the available bandwidth in rapidly changing environment such as streaming media. Applications that prefer a large amount of bandwidth to transfer as much data as possible with a least possible short duration, should use CCID2.

CCID 3 [FKP06] is receiver based TCP friendly rate control mechanism (TFRC). It provides TCP-friendly sending rate optimizing the abrupt changes in sending rate of TCP or TCP-like congestion control [HFPW03]. This variant provides much lower fluctuation of throughput over time in contrast to TCP. CCID 3 is more preferable to CCID 2 in cases where applications need to minimize abrupt changes in sending rate in order to have smooth throughput, such as multimedia applications with small or moderate receiver buffering before playback. CCID 3 will not be an appropriate choice where applications suppose to change the sending rate by varying the packet size rather the packet sending rate.

CCID 2 and 3 are not dependent on the previous history to estimate the current allowed sending rate, but CCID 2 estimates the currently outstanding data over the network to determine its sending rate, while CCID 3 measures the length of recent loss intervals(definition is found in [HFPW03]).

CCID 4 [FK09]can be referred as a modified version of CCID 3. CCID 4 is more preferable to CCID 2 and 3 for applications that adjust the sending rate by varying the segment size instead of changing the sending rate in packets per second in response to congestion. This variant is defined as TFRC-SP (TCP friendly rate control for small packets). Both CCID 3 and 4 uses the TCP throughput equation [HFPW03] for their congestion control. CCID 3 measures the length of recent loss intervals, whereas CCID 4 additionally include nominal packet size of 1460 bytes, a round trip estimate in TCP throughput calculation.

DCCP does not provide any protection against attackers on a connection in progress. If any application desires security, it has to be dependent on other security mechanisms like IPSec [KA98], application level cryptography, etc. depending on the required security level.

2.4 Stream Control Transmission Protocol (SCTP)

Stream Control Transmission Protocol (SCTP) [SXM+07] is a reliable, message oriented, general-purpose transport layer protocol, operating on top of IPv4 and IPv6, providing a service similar to TCP with some new functionalities regarding security, multihoming, multistreaming, mobility, and partial reliability. SCTP supports multistreaming i.e., reliable in sequence delivery within each streams. User messages are partitioned into streams and are transmitted sequentially independent of other streams. The term ”stream” [SXM+07] is used in SCTP to refer to a sequence of user messages, in contrast to its usage in TCP, where it refers to a sequence of bytes, that are to be delivered to the upper layer protocol in-order with respect to other messages within the same stream. A lost of message in one of the streams in one SCTP association between two endpoints does not block delivery of messages in any of the other streams. Internally, each message from an SCTP user is assigned an Stream Sequence Number (SSN) to support in-sequence delivery within a given stream. SCTP provides fragmentation mechanism for user messages if it does not conform to the path Maximum Transmission Unit (MTU). SCTP assigns a Transmission Sequence Number (TSN) to each data chunk that is independent of any SSN. Each TSN is acknowledged by the receiving end, even if there exist gaps between the sequence number, in order to ascertain reliable transmission separate from sequenced delivery within stream.

Each SCTP packet consists of a common header and data chunks containing either user data or SCTP control information. SCTP packet may contain multiple data chunks according to MTU size. The common header has fixed length containing port numbers, a verification tag, and a checksum. SCTP uses verification tag to protect an association against blind attacks.

An SCTP association is initiated by a request from one SCTP endpoint using four-way handshake. Each SCTP endpoint negotiates several parameters such as verification tags, address information, number of streams, supported protocol extensions etc. The initialization process is based on cookie mechanism described by Karn and Simpson [KS99]. The cookie mechanism uses four way handshaking between two endpoints. This mechanism is utilized to provide protection against synchronization attacks. Synchronization attack is a type of denial of service (DoS) attack in which a sender transmits a volume of connections that cannot be completed. The initialization process starts by sending INIT chunk from SCTP sender and the SCTP receiver responds immediately with an INIT ACK chunk. Upon reception of INIT ACK, sender extracts the State cookie and sends it back using COOKIE-ECHO chunk. The SCTP receiver then replies with a COOKIE-ACK chunk. Upon the reception of COOKIE-ACK, the association between SCTP sender and receiver is established for the transmission of subsequent data.

SCTP does not support half-open state like TCP i.e., one SCTP endpoint may not continue transmitting data while the other end is shut down. The association on each endpoint will stop receiving data only when one endpoint executes shut down.

SCTP provides Multihoming support that allows multiple transport addresses for each SCTP endpoint, i.e., one or both endpoints in an association can be reached through more than one transport address. The list of addresses are negotiated during SCTP association. Basically, one address is considered as primary address and used for transmitting user messages. Other addresses are mainly utilized for retransmission to overcome the failures from an inactive destination address.

Congestion Control Mechanism

Each SCTP endpoint uses slow−start and congestion avoidance algorithm to control the amount of data being pushed into the network. Like TCP, an SCTP endpoint uses several control variables such as receiver advertised window size(`rwnd`), congestion control window (`cwnd`), slow-start threshold (`ssthresh`) and additionally partial_bytes_acked in order to facilitate `cwnd` adjustment during congestion avoidance.

The slow-start algorithm is used at the beginning of data transmission or after a message loss has been detected by the retransmission timer. The initial `cwnd` before transmission is set to min (4∗MTU, max (2∗MTU, 4380 bytes)) and no more than 1∗MTU after a retransmission timeout. The initial value of `ssthresh` may be arbitrarily large. The value of `cwnd` is increased by at most the lesser of the total size of the previously outstanding DATA chunk(s) acknowledged, and path MTU, only when `cwnd` is less than or equal to `ssthresh`, an received SACK advances the cumulative TSN Ack point, and the sender is not in Fast Recovery.

Congestion avoidance stage starts when `cwnd` is greater than `ssthresh` and `cwnd` is incremented by 1∗MTU per RTT. The procedure is as follows:

i. partial_bytes_acked is set to zero at initialization.

ii. If `cwnd` is greater than `ssthresh`, then partial_bytes_acked is increased by the total number of bytes of all new chunks acknowledged in received SACK including chunks acknowledged by TSN and by Gap Ack Blocks. Otherwise, `cwnd` is incremented by MTU and partial_bytes_acked is set to (partial_bytes_acked-cwnd). Each Gap Ack Block acknowledges a subsequence of TSNs received following a break in the sequence of received TSNs. Gap Ack Blocks in the SCTP SACK carry the same semantic meaning as the TCP SACK.

iii. When the receiver end acknowledged all of the transmitted data, partial_bytes _acked is reset to zero.

When SCTP sender detects congestion and determines a missing or lost packet from SACK, the control variables are set as follows:

$$ssthresh = max(cwnd/2, 4 \times MTU) \tag{2.3}$$

$$cwnd = ssthresh \tag{2.4}$$

$$partial\_bytes\_acked = 0$$

When retransmission timer expires on a particular address, SCTP endpoint is not allowed to send more than one SCTP packet until it receives acknowledgement for the successful delivery of the missing packet. The `cwnd` and `ssthresh` are set as follows:

$$ssthresh = max(cwnd/2, 4 \times MTU)$$

$$cwnd = 1 * MTU$$

Fast Retransmit and Fast Recovery

SCTP endpoint performs delayed acknowledgement mechanism. Delayed acknowledgement means the receiver doesn't immediately send acknowledgement for every single received data chunk. Whenever the sender receives SACK indicating missing TSN sequence numbers, it will wait for two further subsequent SACKs in order to ascertain those missing TSN(s).

When three consecutive SACKs indicating missing TSNs are received, SCTP sender will do the following:

i. Determine the missing chunk(s) for retransmission.

ii. If SCTP is not in fast recovery, then `ssthresh` and `cwnd` of the destination address(es) are adjusted by Equation 2.3 and 2.4.

iii. Determine how many of the earliest (i.e., lowest TSN) data chunks marked for retransmission that will fit into one SCTP packet allowed by path MTU, and retransmit them accordingly.

iv. Restart the retransmission timer only if the last SACK acknowledged the earliest outstanding TSN number, or the endpoint is retransmitting the first outstanding data chunk sent to that address.

v. Mark the data chunk(s) that are being fast transmitted.

vi. If the endpoint is not in fast recovery, enter fast recovery and the highest outstanding TSN is considered as Fast Recovery exit point. Fast recovery is completed upon the reception of an SACK that acknowledges all TSNs up to and including this exit point.

It is important to note that the number of outstanding TSN's is indirectly bounded by `cwnd` in SCTP. Therefore, the effect of Fast Recovery in TCP is realized automatically without any adjustment to the `cwnd`.

2.5 Comparison

Table 2.1 outlines the comparison of features among transport protocols described in this chapter. SCTP provides in sequence, ordered or unordered delivery, flow control, reliability and bidirectional data transfer like TCP. It also enhances a set of capabilities like partial reliability, security, multi-streaming and multi-homing. Some level of multihoming and mobility support can be attained through Multipath TCP [Bag11] that describes the extensions proposed for TCP. Multipath TCP enables two endpoints of a given TCP connection to use multiple paths to exchange data. DCCP provides a low expense, congestion control mechanisms for unreliable data transfer. Moreover, the congestion control methods are modularly separated from its core that supports each end-points to choose a different congestion control methods it prefers. UDP is not preferable due to lack of congestion control mechanisms. DCCP is more preferable to UDP for having congestion control functionalities for unreliable data flows.

Table 2.1: Feature comparison of Transport Protocols.

Features	**TCP**	**UDP**	**DCCP**	**SCTP**
Communication	Byte oriented	Minimal message oriented	Message oriented	Message oriented
Unordered data delivery	No	Yes	Yes	Yes
Connection-oriented	Yes	No	Yes	Yes
Reliability	Yes	No	Unreliable with minimal CC	Yes
Partial Reliability	No	No	No	Yes
NAT traversal	Yes	Yes	No	No
Protection against SYN attack	Sensitive to SYN attack	No	No	No SYN attack
Congestion Control mechanisms	AIMD, TCP Reno,NewReno, SACK	No	AIMD, TFRC, TFRC-SP	TCP SACK
Multistreaming	No	No	No	Yes
Multihoming	Some level of multihoming through multipath TCP	No	No	Yes

Moreover, conventional TCP has some limitations such as head of line blocking where sending independent messages over an order-preserving TCP connection causes delivery of messages sent later to be delayed within a receiver's transport layer buffers until an earlier lost message is retransmitted and arrives, vulnerable to SYN attack [Edd07]. SYN attack is a type of denial of service (DoS) attack in which a TCP sender transmits a volume of connections that cannot be completed. Whereas, SCTP avoids head of line blocking for its multistreaming feature, less conservative to ordering of real time data for its partial reliability feature, and provides protection against SYN attack with its built-in cookie mechanism. A good analysis on potential advantages/disadvantages of incorporating SCTP into existing TCP over Satellite network is found in [AAI02]. The experimental results of this article shows that the retransmission mechanism of SCTP experiences slightly better throughput than TCP in their simulation environment.

3 NAT Traversal

It is necessary to perform IP address translation when a network's internal addresses cannot be used outside the network. The reason behind the non-usability of Internal address is either they are invalid for use in outside network, or the security and privacy of internal addresses should be preserved from the outside networks. NAT traversal is a general term for technologies that establish and maintain Internet protocol connections passing through network address translation (NAT) gateways. Network address translation (NAT)[SE01] is a process of mapping IP addresses from one address space to another, in order to provide transparent communication through routing device. Real time communication such as multimedia communications experience significant problems for NAT traversal. The rudimentary source of this problem is that these applications such as Voice over IP, multimedia over IP (e.g. SIP, H.323), and on-line gaming, carry IP addresses in their payload while traversing NATs. NATs experience problem translating new transport layer technologies such as SCTP, DCCP. IP packets with unknown or too new transport protocol types, are dropped while traversing middle-boxes. This chapter will discuss basic NAT traversal technologies.

3.1 NAT

The simplest type of NAT that provides one-to-one translation of IP addresses, is known as basic NAT or one-to-one NAT. Many-to-one NAT is such NAT that allows many hosts residing in private network to share one public IP address. This type of NAT has to maintain a translation table so that incoming packets can be directed to originating host. Therefore, in order to avoid ambiguity, this kind of NAT needs to alter higher level information such as port numbers. This translation is known as Network address and port translation (NAPT) [SH99].

IP addresses can be divided into two types: Private IP addresses which are used within private IP network and Public IP addresses which are used to connect servers in the public Internet. Network address translation provides IP address mapping between internal/private network and external/public network. NAT also provides mapping several private addresses onto one unique global public address and transparent routing [SH99] between end-points. The term "Transparent routing" differs from the "routing" provided by traditional router device in a way that traditional router routes packets within single address space whereas NAT device facilitates forwarding between disparate address space.

NAT devices are of different types and different implementations use different types of NAT boxes [RWHM08]. Each outgoing session from an internal endpoint through NAT is assigned an external IP address and port number so that subsequent response packets can be forwarded to internal endpoint. The key variation comes from the criteria for reuse of a mapping for new sessions to external endpoints, after establishing a first mapping between an internal X:x address and port, and an external Y1:y1 address tuple. Let us assume that first session uses the mapping of the internal IP address and port X:x to X1′:x1′. The endpoint from X:x forwards to an external address Y2:y2 and X:x is mapped to X2′:x2′ on the NAT. It is complex for describing the NAT behaviour for various combinations of the relationship between X1′:x1′ and X2′:x2′, and the relationship between Y1:y1 and Y2:y2. NAT box with Endpoint-independent mapping behaviour reuses the port mapping for subsequent packets sent from the same internal IP address and port (X:x) to any external IP address and port and X1′:x1′ equals X2′:x2′ for all values of Y2:y2. NAT box with Address-Dependent mapping reuses the port mapping for subsequent packets sent from the same internal IP address and port (X:x) to the same external IP address, regardless of the external port. Here, X1′:x1′ equals X2′:x2′, if and only if Y2 equals Y1. NAT box with Address and Port-Dependent Mapping reuses the port mapping for subsequent packets sent from the same internal IP address and port (X:x) to the same external IP address and port while the mapping is still active. In this case, X1′:x1′ equals X2′:x2′ if and only if, Y2:y2 equals Y1:y1. Some NATs attempt to preserve the port number used internally when assigning a mapping to an external IP address and port, where x1=x1′, x2=x2′. This port assignment behaviour is referred as ”port preservation”.

Various techniques exist for NAT traversal. In the following subsection, several techniques for NAT traversal as preferred in RTCWeb will be discussed.

3.2 Session Traversal Utilities for NAT (STUN)

Session Traversal Utilities for NAT (STUN) [RMMW08] is a protocol that provides a tool for other protocols in dealing with NAT traversal. The end-points residing in private network can utilize this tool to discover the external IP addresses and port allocated by NAT that is corresponding to its private address.

STUN is referred as a client/server protocol. It permits two types of transactions.

One is request/response transaction, where a client sends a request to a server, and the server returns a response. In indication transaction, either client or server sends an indication. This transaction does not generate any response. Both transactions include a randomly chosen 96-bit ID. A binding method can be used in both transactions. In case of request/response transaction, a binding request, which may pass through one or more NATs, is sent from a STUN client to a STUN server. Consequently, the source transport address of the request received by the STUN server will be the external IP address and port as translated by NAT. This translated address is referred as reflexive transport address. The STUN server stores reflexive transport address into an XOR-MAPPED-ADDRESS attribute in the STUN binding response and sends the binding response back to the client. The response passes through the NAT that will modify the destination transport address in the IP header, but XOR-MAPPED-ADD attribute within the body of STUN response will remain unaffected. This is the way of finding reflexive transport address translated by outermost NAT with respect to STUN server.

STUN can be run on top of TCP or UDP. Running over UDP may cause STUN messages to be dropped by the network, since UDP is unreliable by nature. The client application itself needs to retransmit the request message in order to ensure reliability of STUN request/response transactions. STUN indications are not retransmitted and hence, not reliable. In some usages, STUN is multiplexed with other data over a TCP connection, it is necessary to implement STUN on top of any kind of framing protocol, specified by the usage or extension. This framing protocol helps the agent to extract STUN messages and application layer messages. The usage will specify how the client knows to apply framing protocol and what port to connect to.

3.3 Traversal Using Relays around NAT (TURN)

Traversal using Relays around NAT is a relay extensions to STUN. TURN was invented to support multimedia sessions using SIP signaling. TURN is a protocol that provides help for end-points behind NAT to request TURN server to act as a relay when a direct communication path using hole punching technique cannot be found. Hole punching technique [SKF08] is a technique that establishes connection between two hosts across one or more NATs. Hole punching technique generally fails when end-points behind NATs have a mapping behaviour of "address dependent mapping" or "address and port dependent mapping".

Typically, a TURN client is connected to a private network and to the public network through one or more NATs. A TURN server is situated on the public Internet. If a TURN client wants to communicate with peers residing behind one or more NATs or elsewhere in the public Internet, the client can use the TURN server as a relay to send packets to these peers and to receive packets from these peers. At first, the client needs to send TURN messages with an allocation request from its host transport address to the TURN server transport address. If the allocation is possible, the server replies with relayed transport address located at the TURN server. The relayed transport address is the transport address on the server that peers can use to have the server relay data to the client. When allocation is successful, the client can send application data indicating which peer the data is to be sent. Then, the TURN server will relay this data to the appropriate peer. The client sends the application data to the server inside a TURN message, the data is extracted from the TURN message and sent to the peer in a UDP datagram. In the reverse direction, a peer can send application data in a UDP datagram to the relayed transport address for the allocation. The server will encapsulate this data inside a TURN message and send it to the client along with an indication of which peer sent the data. A client that uses TURN should have some means to know the relayed transport address of its peers. One solution is to use rendezvous protocol [SKF08].

TURN uses UDP for communication between the server and the peer. It can also use TCP or Transport Layer Security(TLS) [DR06] over TCP for this purpose. For later cases, the server will convert between these transports and UDP transport while relaying data to the peer and from the peer. TURN supports TCP as some firewalls are configured to block UDP entirely but not TCP. TLS over TCP provides additional security properties besides digest authentication [MMR10] provided by TURN by default.

TURN server is very expensive in terms of bandwidth and requires high-bandwidth connection to the Internet. It incurs additional delay for media traffic. Therefore, it is recommended to use TURN server only when a direct communication path cannot be found.

3.4 Interactive Connectivity Establishment (ICE)

Interactive Connectivity Establishment (ICE) [Ros10] is a protocol that enables end-points to exploit multimedia communication protocol over UDP through middleboxes, such as NAT, firewall based on offer/answer model [RS02] of session negotiation. ICE also support TCP [RKLR12] for some media protocols, such as screen sharing, instant messaging, that need to run on top of TCP in the presence of NAT box.

ICE is responsible to provide a set of candidate transport addresses for each media stream. A candidate consists of IP address and port for a particular transport protocol (e.g., UDP, TCP). ICE uses STUN as a tool to validate these candidates. In Figure 3.1, it is shown that two end-points (ICE agent) can communicate indirectly by performing offer/answer protocol through web server or SIP server. The agents are capable of exchanging offer/answer messages (e.g., SDP) to set up a media session between two agents. This exchange will usually take place through a SIP server. Each ICE agent has a list of candidate IP addresses and ports to communicate with other agent. The candidate addresses are of three types [Ros10]:

Host Candidate These candidates are derived from directly attached network interface, e.g. Ethernet, Wifi. Host candidate can also be obtained trough tunnel mechanism, such as Virtual Private Network or Mobile IP.

Server Reflexive Address STUN provides support to obtain Server Reflexive Address that is on public side of a NAT.

Relayed Address This candidate is gathered from TURN server.

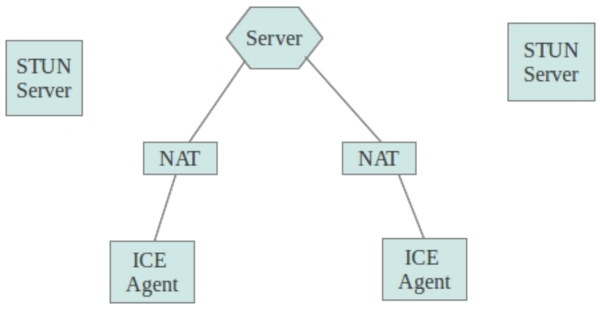


Figure 3.1: Simplified Network Topology for ICE.

If two agents are behind NAT, Host candidates are unlikely to be able to communicate directly. ICE needs to use STUN or TURN to gather suitable candidates. When ICE uses TURN, both Server Reflexive Address and Relayed Address are obtained. STUN only provides Server Reflexive Address. The relationship of these addresses are shown in Figure 3.2. When the ICE agent sends the TURN Allocate request from private IP address and port X:x, the NAT will create a binding X1′:x1′ which is called server reflexive candidate. On the arrival of the Allocate request at the TURN server, it allocates a port y from its local IP address Y, and generates an Allocate response, informing the agent of this relayed candidate Y:y. When ICE utilize STUN servers, the agent sends a STUN Binding request to its STUN server. The STUN server will respond the agent providing the server reflexive candidate X1′:x1′.

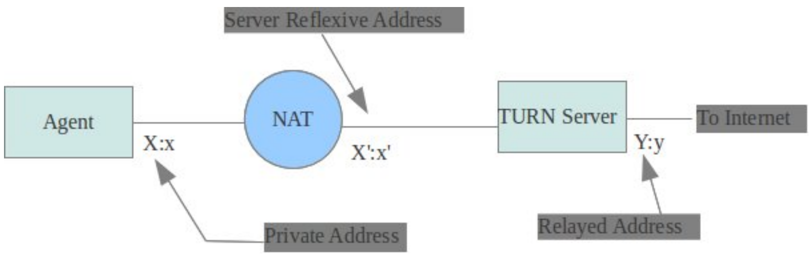


Figure 3.2: Candidate Address Relationship [Ros10].

ICE performs peer-peer connectivity checks [Ros10] for the communication between two end-points. Connectivity checks are required to discover a pair of candidates, one for host agent, and the other from peer agent, that will work. When the host agent has gathered all of its candidates, it sorts them in highest to lowest priority and sends them to the peer agent over the signaling channel. The priority algorithm is generally designed so that similar type of candidate addresses get similar priorities (e.g., more direct routes, that pass through fewer NATs, preferred over indirect ones passing through more NATs). When the peer agent receives the offer, it performs the same gathering process and responds with its own list of candidates. At the end of connectivity checks, each agent will have a complete list of both its candidates and its peer's candidates.

Transportation of multimedia messages through NAT using offer/answer protocol is difficult to operate, since those carry the IP addresses and ports of media sources and sinks within their messages. Therefore, if the server of multimedia sources is located behind NAT, it will create problem while traversing NAT [Sen02]. ICE helps in this case by providing a set of candidate transport addresses for each media stream. RTCWeb uses ICE for NAT traversal. ICE utilizes STUN or TURN server for network address translation.

4 Real Time Communication for Web Browsers

The current Internet is widely considered as a medium of real-time communication and interactive applications. Several proprietary solutions that facilitate direct interactive communication using audio, video, collaboration, games etc. already exist on the Internet. These solutions lack interoperability because of the requirement of non-standard extensions or plugins to work in the browsers. However, Internet Engineering Task Force (IETF) and World Wide Web Consortium (W3C) are working together on extending Web architecture with a desire to standardize a set of protocols so that interoperability can be achieved in real time communication between two compatible browsers.

4.1 Overview

The primal goal and vision of RTCWeb is to standardize a set of protocols to enable browser to browser communication using audio, video, and auxiliary data along the most direct possible path between clients without installing plugin in the browser [Alv12]. The communication may also be possible between browsers and other endpoints that are compatible with RTCWeb. The effort of RTCWeb consists of two parts: a protocol specification done by IETF and a JavaScript API specification done in W3C. The former standardization effort is referred as RTCWeb and the latter one as WebRTC. Another goal of RTCWeb/WebRTC is interoperability between protocol specification and the API specification in order to ensure that multiple products implementing a standard are able to work together providing a particular functionality to user. Moreover, the working group also considers security requirements for such communication.

Conventionally, services for the browser have been provided by plug-ins which need to be downloaded and installed separately from the browser. Consequently, this creates some drawbacks. Firstly, plug-ins are specific to browser and operating system, and not available universally. Secondly, it is the responsibility of user to aptly install them to work in the browser. RTCWeb architecture is intended to support communication between browsers without installing plug-ins. A general architecture of RTCWeb is shown in Figure 4.1. The communication path is composed of three components: signaling path to transport control information, media channel for media (i.e., audio, video) and data channel for non-media data (e.g. character screen position within an multiplayer HTML5 video game, text file, text chat). The communication path that goes through the web server is ascribed as signaling path. The direct communication path between browsers consists of two parts as shown in Figure 4.1: media channel and data channel.

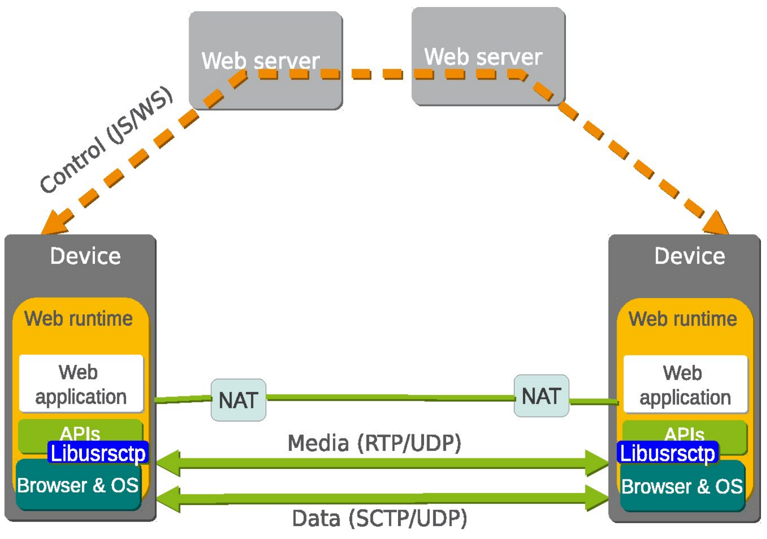


Figure 4.1: RTCWeb Architecture

Standard or proprietary signaling protocols can be used to establish, manage, and control the communication path between browsers or other RTCWeb compatible devices. The communication through the signaling path is in the form of XML-HttpRequest based or WebSocket-based communication, which is a web technology providing bidirectional communications over a single TCP connection. WebSocket is designed to be implemented in web servers and web browsers. Websocket, in contrast to TCP, enables a stream of messages instead of a stream of bytes. However, if the two servers are used by different entities, the signaling path should be agreed upon either by standardization or by any other way of agreement, such that both servers end up using same signaling protocol (e.g. SIP or subset of SIP, XMPP [SA04] etc.). The signaling message that goes through the web servers, can be modified or translated if required. A WebSocket sub-protocol [CMP12] for SIP transport is specified for bidirectional communication between clients and servers along with multimedia capabilities for audio and video sessions in web browsers.

In RTCWeb, the media data and non media data can be sent directly through the media channel and data channel, respectively. The communication path uses

PeerConnection[1] Interface. This interface uses Interactive Connectivity Establishment (ICE) [Ros10], Session Description protocol (SDP) [HJP06], Session Traversal Utilities (STUN) [RMMW08], and Traversal Using Relays around NAT (TURN) [MMR10] protocols in order to traverse legacy NAT and perform codec negotiation. In order to bootstrap peer-to-peer connection, one peer loads a page and exchange messages with other peer through the signaling path. Messages are sent to the server with a session identifier, and the server routes it to the other peer using the initiated session. Each peer has accounts with some Identity provider. This kind of identity service is common in Web environment such as OAuth [HL10], OpenID [RR06].

When the signaling channel is established, arbitrarily one peer initiates JavaScript callback to create PeerConnection object through which data can be delivered. While creating PeerConnection, it passes a configuration string containing information about whether STUN or TURN server will be used. As soon as PeerConnection has been created, one browser sends initial offer to the other peer. Upon receiving the initial offer, the other peer similarly creates PeerConnection object with a configuration string. Therefore, PeerConnection allows two users to communicate directly without the intervention of servers. The communication is coordinated via the signaling path provided by script in the page via the server, e.g. XMLHttpRequest.

The signaling in RTCWeb is envisioned to be such that the media plane will be fully specified and controlled by call establishment phase, and the signaling plane will reside to the application as much as possible. The idea is that different applications may use different protocols, such as SIP or Jingle [LBSA$^+$09] call signaling protocol, for signaling. The required information that needs to be negotiated during call setup is multimedia session description, which specify the necessary transport and media configuration strings to establish media plane.

The media negotiation will be performed using SDP offer/answer semantics that are used in Session Initiation Protocol (SIP) [RSC$^+$02]. The same semantics enable to build a signaling gateway between SIP and the RTCWeb media negotiation. RTCWeb offer/answer protocol(ROAP) [ROAP] has been proposed for media negotiation between browsers or other RTCWeb compatible devices. ROAP uses SDP offer/answer protocol [RS02] that enables RTCWeb browser to establish media sessions to another browser or a SIP device. In case of browser to SIP device communication, the signaling gateway is responsible for mapping signaling messages between ROAP and SIP.

[1]http://www.htmlrules.com/javascript/peer-connection-interface/index.html.

ROAP proposal has some limitations. First, this protocol is inflexible as the signaling state machine is embedded into the browser. Therefore, any modification required to session descriptions, or use of alternate state machine is difficult. Second, user may reload the web page randomly, leading to a problem if the state machine is being run at a server, the server can simply return the current state back down to the page and resume the call where it left off. If the state machine is run on the browser end, the state machine will be initialized again upon reloading the web page. But, it seems complicated to design the state machine to maintain the same state after reloading the web page.

Nevertheless, JavaScript Session Establishment Protocol (JSEP) [JU12] is proposed to consider the issues explained above. This protocol proposes to implement the signaling state machine into JavaScript. Therefore, the browser is almost out from the core signaling flow. This approach decouples ICE state machine from signaling. ICE remains in the browser, and only the browser is concerned about candidates and other transport information.

RTCWeb utilizes UDP for transportation of media data as well as non-media data. The Real-time Transport Protocol (RTP) [SCFJ03] is used to exchange audio and video data over UDP in RTCWeb. It also supports TCP in case of UDP is blocked by NAT boxes. The attractive feature of RTP is that it supports both unicast and group communication. RTP is composed of two parts: RTP for data transmission and RTCP for RTP control information. RTP and RTCP are flexible and extensible allowing an application to adapt extensions if existing mechanisms are not sufficient. RTCWeb requires a NAT traversal method to establish a data path between two end-points. ICE is used for NAT traversal with the aid of STUN or TURN server.

For data channel to transport non-media data, several proposals have been discussed in RTCWeb [TLJ12a]. The most common proposal is how ICE can be utilized to set up data connection as similar to RTP for media streams. It is also proposed to have a thin layer on top of UDP or DTLS to multiplex the data with other packets. The issues raised for non-media data are implementation maturity, congestion control and avoidance, high overhead and NAT traversal. A detailed picture regarding non-media data will be found in Chapter 5.

RTCWeb communications are directly governed by a web server that introduces new security threats [Res12]. The basic idea is that each browser in the end-points exposes some standardized JavaScript APIs which are used by web server to establish call between two browsers. Therefore, these JavaScript APIs can prompt a denial-of-service or other kind of attacks by malicious calling services. The security of media channel is another important consideration. Much discussion is going on in IETF community to select a suitable protocol to protect the media channel. The most likely solution seems to be Secure Real-Time transport protocol (SRTP) [BMN$^+$04]. SRTP does not have any key management protocol. Therefore, SRTP is to be used along with a key management protocol such as Datagram Transport Layer Security (DTLS) [RM06].

4.2 NAT traversal

RTCWeb clients are mostly web browsers and may be located behind NAT or firewall. Therefore, web browsers need to have native NAT traversal mechanisms without which the functionality of RTCWeb clients will be significantly limited.

RTCWeb uses RTP for media communication and RTCP to carry control signals for RTP. RTP and RTCP listen on separate UDP ports. However, symmetric RTP/RTCP [Win07] is required in RTCWeb to get rid of the issues of maintaining multiple NAT bindings, while traversing NAT or firewall. A device supports Symmetric RTP/RTCP if it selects, communicates, uses same IP address and port number for sending and receiving RTP/RTCP packets. It has been decided to use SCTP for non-media data in RTCWeb. New IP payloads, such as SCTP, DCCP and new TCP options, experience problems in NAT traversal, since NAT boxes do not know how to handle these new protocols. NAT boxes drop such packets with unknown transport protocols or even extension of known transport protocols, e.g. new TCP options. Therefore, it is required to impose some mechanisms to deal with SCTP.

Moreover, whenever a calling client wants to set up connection with its peer, it requires consent from the receiving client before starting data transmission. Therefore, web browsers should have some consent mechanisms to establish connection between RTCWeb clients. ICE negotiation can serve this purpose. To support ICE, client applications need to implement STUN or TURN. There should also be a mechanism in web browsers to configure the access STUN server. Presently, PeerConnection interface expose JS API that includes STUN server address and port number as a configuration string. Configuration string is passed as an argument to one JS API to create PeerConnection object.

Nevertheless, web browser vendors need to natively support ICE for connectivity requirements. There have been discussions among IETF researchers regarding the ICE implementation; whether ICE will be implemented in web browsers natively or within JavaScript library.

4.3 SCTP NAT Traversal

IP packets with unknown or too new transport protocol types such as SCTP, DCCP, are dropped while traversing NAT, since specialized code of network address translation for these new transport protocol has not yet been installed in most of the NATs. NAT traversal issue for SCTP is more complex when the association is multi-homed. It has been decided in IETF to use SCTP for RTCWeb data channel. In this section, various NAT traversal scenario [XSHT07] for SCTP will be discussed briefly.

SCTP packets can go through either single NAT as shown in Figure 4.2 or multiple NATs as shown in Figure 4.3. In single point scenario, all packets go through a single NAT box. Another variation can be to have multiple NATs on a single path. In the single point scenario, NAT box has to deal with all of the SCTP packets. In this case, end-points can be either single homed or multi-homed.

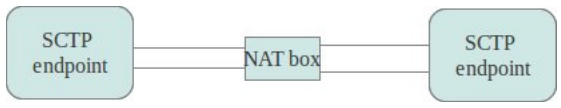


Figure 4.2: NAT Tranversal through Single Path [XSHT07].

In multiple point scenario, a fraction of total packets is passed through each NAT. This scenario is applied to multi-homed SCTP association. The existence of multiple NATs between end-points can preserve the benefits of path diversity of a multi-homed association for the entire path [XSHT07].

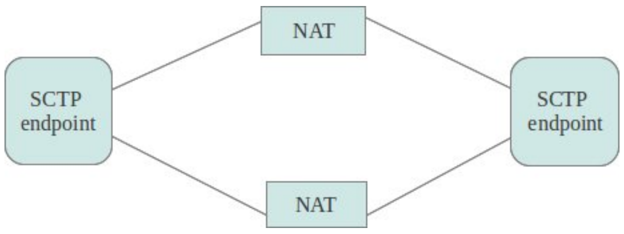


Figure 4.3: NAT Traversal through Multi-Path [XSHT07].

In both single point and multiple point traversal scenario, SCTP protocol should be integrated into the NAT box. Much work has been performed for TCP and UDP [SKF08]. However, the work to support SCTP is in its early stages. The following terms are defined to discuss NAT behaviour with SCTP.

Global-Address : Global-port: It denotes the external address that a host behind a NAT is trying to connect.

Local-Address : Local-Port: The internal/private address of the end-points behind a NAT.

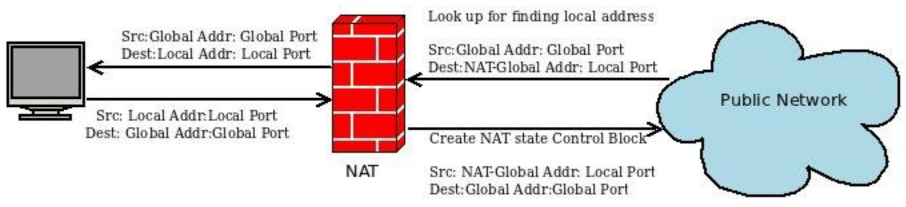
Nat-Global-Address : Nat-Global-Port: It defines the global address assigned to NAT box and the port number is translated from the Local-Port.

Local-Vtag: The verification tag (32bits) is used by the host behind a NAT for its communication. This tag must accompany any incoming SCTP packets for this association to the Local-Address.

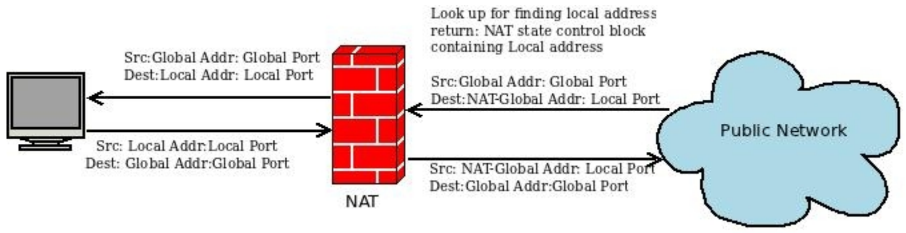
Remote-Vtag: This tag (32bits) is used by the remote host for its communication that must accompany any incoming SCTP packets for this association to the Global-Address.

In single point traversal scenario, the source address of the SCTP outgoing packet has to be replaced with NAT-Global-Address as shown in Figure 4.4. On the other hand, to deliver incoming packets to thr correct SCTP endpoint, NAT does a lookup to find the corresponding Local-Address that replaces the destination address. Therefore, NAT box has to maintain a look-up table of Global-Vtag, Global-Port, Global-Address, Local-Vtag, Local-Port and Local-Address. The look-up with the Global-Vtag, Global-Port, Global-Address, Local-Vtag and the Local-Port of an incoming packet returns the Local-Address.

However, some difficulties arise with an SCTP compatible NAT. First, when a SCTP end-point behind NAT box has to utilize SCTP for data communications, classical NAPT may change the SCTP port numbers while processing SCTP packets. Therefore, it leads to expensive re-computation of the transport layer checksum. Consequently, this creates significant computational burden to NAT box and requires substantial hardware support, whereas it can be done very efficiently for UDP and TCP. Secondly, multi-homed SCTP end-point has multiple addresses but only one single port number. Therefore, to support multihoming, NAT box needs to differentiate all the packets with respect to each association and perform appropriate port translation to redirect packets towards appropriate hosts.


Src:Global Addr: Global Port
Dest:Local Addr: Local Port
Src: Local Addr:Local Port
Dest: Global Addr:Global Port
NAT
Look up for finding local address
Src:Global Addr: Global Port
Dest:NAT-Global Addr: Local Port
Public Network
Create NAT state Control Block
Src: NAT-Global Addr: Local Port
Dest:Global Addr:Global Port

(a) SCTP INIT packets.


Look up for finding local address
return: NAT state control block
containing Local address
Src:Global Addr: Global Port
Dest:Local Addr: Local Port
Src:Global Addr: Global Port
Dest:NAT-Global Addr: Local Port
Public Network
Src: Local Addr:Local Port
Dest: Global Addr:Global Port
Src: NAT-Global Addr: Local Port
Dest:Global Addr:Global Port
NAT

(b) SCTP packets except INIT

Figure 4.4: Network Address Translation for SCTP packet.

To get rid of the difficulties stated above, one possible solution is to encapsulate SCTP packets into UDP packets. UDP encapsulation allows SCTP traffic to pass NAT boxes without SCTP supported NATs. Additionally, SCTP userland implementation does not require any special privileges to access IP layer directly. It should be noted that the IP addresses that are needed to change while traversing NAT must not be put into SCTP packet if UDP encapsulation is used. UDP encapsulation [TS11] i.e., tunnelling over UDP can be a good solution to traverse NAT boxes with new IP payloads. UDP encapsulation triggers SCTP if any packet is lost during transmission. The basic idea is that UDP encapsulated SCTP packets will look like UDP messaging. UDP does not provide any congestion control, but SCTP natively provides congestion control and avoidance. Therefore, UDP encapsulation of SCTP packet needs to use a remote UDP encapsulation port per destination address for each SCTP association and a local port for all of its incoming packets. The local port number is 9989 as assigned by IANA[2]. For this purpose, UDP header is inserted between IP header and SCTP header as shown in Figures 4.5 and 4.6.

At the other end, the receiver will truncate the UDP header to get the SCTP header and a look up is performed to find out the SCTP association the received

[2]http://www.ietf.org/assignments/port-numbers

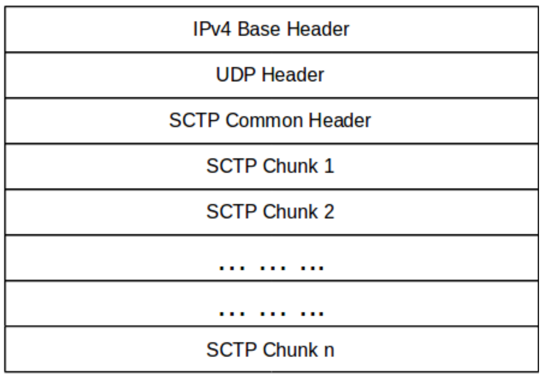


Figure 4.5: UDP encapsulation using IPv4 (originally from [TS11]).

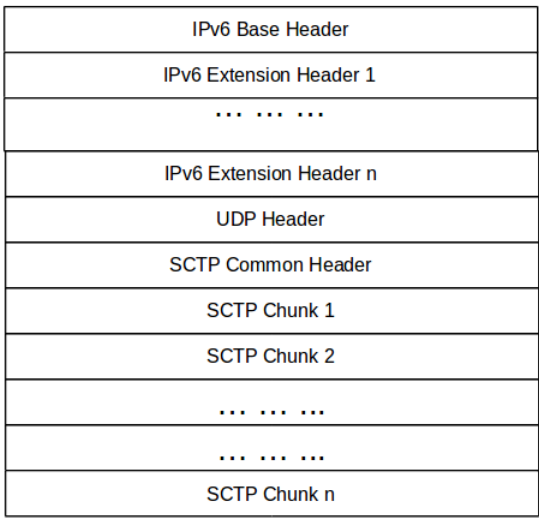


Figure 4.6: UDP encapsulation using IPv6(originally from [TS11]).

packet belongs to. The UDP port number is used as the encapsulation port for the destination address.

When SCTP end-point receives ICMP or ICMPv6 packets, it can not be possible to find out which SCTP packets trigger these ICMP or ICMPv6 packets, since there might not be enough bytes in the payload to identify the corresponding SCTP association. During encapsulation, SCTP endpoints must deduct the size of UDP header from path MTU.

4.4 Congestion Control

Congestion control is required for RTCWeb data transport. Congestion control for real time media data is difficult for several reasons. First, the encoding for media data cannot be changed quickly with response to varying bandwidth. Second, sending application may not be willing to reduce data rate required by the flow on which congestion is detected. Third, non-media data also competes for bandwidth and may create congestion that impacts media data.

In WebKit RTCWeb implementation, PeerConnection interface allows direct communication from application running in one browser to application running in another browser. PeerConnection implementation uses UDP for sending media or non-media data directly between browsers. Currently, PeerConnection does not include any congestion control mechanisms for media and data channel. Moreover, UDP does not ensure in-order delivery or reliability of user data. Therefore, the issue of how to integrate congestion control with current implementation of PeerConnection is still an open issue. This thesis work gives focus on data part and uses SCTP userland stack[3] for data channel congestion control.

[3]http://sctp.fh-muenster.de/sctp-user-land-stack.html

5 RTCWeb Data Channel Protocol Analysis

Several transport protocols such as TCP, UDP, RTP, SRTP, SCTP, DCCP etc., presently exist for communication of media and non-media data. A single protocol alone is not best suitable for browser to browser data communication satisfying all the requirements in the context of RTCWeb. UDP is most common transport protocol for real time data. Currently, RTP over UDP is used for real time data such as audio and video. UDP experiences limitations in terms of reliability, ordering, data integrity and especially, it does not provide any congestion control mechanisms. TCP is conservative in ordering, in-sequence delivery but provide some of those requirements. In this chapter, we discuss in succinct how to deliver non-media data, and what are the requirements for data channel in the context of RTCWeb.

5.1 Data Channel Requirements

Data channel offers usage for several applications such as chat, file transfer, real time game, shared whiteboard, shared document editing with real time media data. Some use-cases for data channel are presented in Appendix 3.

The requirements for RTCWeb data channel are as follows [TLJ12a].

- Data channel must allow multiple simultaneous streams (i.e., multiple data channel) at the same time.
- Both reliable and unreliable, ordered and unordered data transmission should be considered.
- There should be suitable congestion control mechanism either separately or in conjunction with the media streams so that data transport cannot create congestion on media streams.
- Potential prioritization on each data stream with respect to other streams as well as media streams.
- It should support confidentiality, integrity, and source authentication.
- There should be some means for NAT traversal.
- Data channel protocol should not rely on ICMP or ICMPv6 for path MTU discovery.

- The data protocol must not encrypt local IP addresses inside its protocol fields to preserve potentially private information.
- Unbounded messages must be supported at the application layer.

We conclude that no single protocol alone is able to meet all of the requirements. Therefore, a suitable protocol stack needs to be designed for data channel. In the next section, several alternatives will be discussed in brief.

5.2 Data Channel Protocol

RTCWeb/WebRTC community is working to design a generic transport protocol service that enables browsers to exchange data in a bidirectional peer to peer fashion. Presently, there is a general consensus in RTCWeb IETF community that RTCWeb data transport protocol stack should have a suitable transport protocol that will provide necessary and sufficient congestion control over UDP to overcome legacy NAT traversal issues. For confidentiality, authentication, integrity protection, the proposal is to use the transport protocol that provide congestion control over DTLS and DTLS will be implemented on top of UDP.

Data channel protocol stack should be designed in a way that will meet all of the requirements of RTCWeb presented in Section 5.1. The basic idea is that the protocol stack should provide congestion control and avoidance to data channel as well as sufficient protection for user data. Therefore, the stack will be like as shown in Figure 5.1. DTLS will be implemented on top of UDP and provide protection to user data. A suitable transport protocol will be on top of DTLS to serve congestion control. In the following discussion, several options for data channel will be discussed in brief.

Pseudo-TCP(reliable)+DCCP (unreliable)/DTLS/(ICE)UDP

One data channel protocol option can be TCP on top of DTLS or UDP. This option is shown in Figure 5.2. Pseudo-TCP is userspace TCP implementation. Pseudo-TCP can be attributed as a subset of the TCP stack to allow for reliable transport over non-reliable sockets (such as UDP). For unreliability requirement of RTCWeb, DCCP can be used on top of UDP. DTLS is responsible for confidentiality, integrity and source authentication for the upper layer payload. The basic idea is that TCP

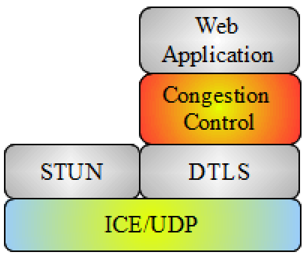


Figure 5.1: RTCWeb/WebRTC Basic Protocol Stack.

over TLS or Pseudo-TCP is for reliable traffic, and DCCP over DTLS is for unreliable traffic.

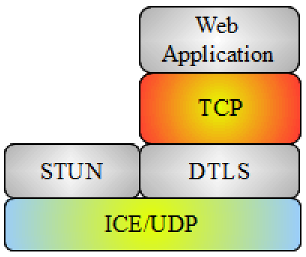


Figure 5.2: RTCWeb/WebRTC Protocol Stack using TCP.

There are several advantages of this protocol stack [TLJ12b]. First, TCP is well known and its congestion control mechanism is well understood. Second, userspace TCP implementation exists to run on top of DTLS. Third, multiplexing of datagram flows can also be supported with this approach by adding stream identifier to TCP header. The downside of this option is that TCP does not support unreliable data transfer and additional framing layer is required on top of TCP to provide the necessary datagram interface to the calling API. Additionally, stable version of DCCP userspace implementation does not exist at this point.

SCTP/DTLS/UDP(ICE)

Figure 5.3 represents the protocol stack of SCTP over DTLS over UDP. Presently, DTLS/UDP is specified in [RM06], but SCTP over DTLS has been only specified recently in an Internet draft [TSLJ12]. DTLS is usually implemented in userspace,

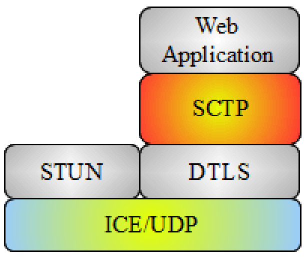


Figure 5.3: RTCWeb/WebRTC Basic Protocol Stack using SCTP over DTLS.

and hence, SCTP userland implementation is required to make this protocol stack deployable. Moreover, it is necessary to have SCTP userland stack due to lack of universal availability of SCTP in operating systems. The lower layer interface of an SCTP implementation needs to be adapted to differentiate between IPv4 or IPv6 (being connection-less)or DTLS (being connection-oriented).

Processing of incoming ICMP or ICMPv6 packets cannot be performed in SCTP layer, since there exist no means to identify the corresponding SCTP association. SCTP should not rely on ICMP or ICMPv6 for path MTU discovery. Therefore, DTLS should have support for path MTU discovery. It can be done by controlling Don't Fragment (DF) bit when the stack uses IPv4. DF bit is required to be set for path MTU discovery.

DTLS/SCTP/UDP(ICE)

In this alternative, SCTP association is performed over UDP. UDP encapsulation of SCTP is presented in IETF draft [TS11] and the DTLS over SCTP is delineated in [TSR11]. This protocol stack provides legacy NAT traversal support through UDP encapsulation as well as confidentiality, integrity and source authentication to SCTP payload using DTLS. SCTP control information is exchanged without encryption. This stack also provides multihoming feature of SCTP using ICE.

SCTP/DTLS vs DTLS/SCTP

Table 5.1 shows the feature of these two stack SCTP/DTLS and DTLS/SCTP with traditional transport protocols, TCP and UDP. SCTP natively supports multistreaming, ordered and unordered delivery of user data, reliability and partial reliability on data exchange. Both stacks support NAT traversal through ICE. DTLS provides the security and privacy for both DTLS/SCTP and SCTP/DTLS.

Table 5.1: Feature List of Transport Protocol or Protocol stack.

Features	**TCP**	**UDP**	**DTLS/SCTP/UDP**	**SCTP/DTLS/UDP**
Communication	Byte oriented	Minimal message oriented	Message oriented	Message oriented
Connection oriented	Yes	No	Yes	Yes
Reliability	Yes	No	Yes	Yes
Partial Reliability	No	N/A	Yes	Yes
Ordered delivery	yes	N/A	Yes	Yes
Unordered delivery	No	N/A	Yes	Yes
NAT traversal	Yes	Yes	Yes	Yes
Security and privacy	Sensitive to SYN attack	No	DTLS provides security	DTLS provides security
Congestion Control mechanisms	Yes	N/A	Yes	Yes
Multistreaming	No	No	Yes	Yes
Multihoming	Some level through multipath TCP	No	Yes	No
Kernel Implementation support	Yes	Yes	Yes	No
userspace implementation	Yes	Yes	Yes	Yes

For SCTP/DTLS, userspace SCTP is required to be implemented, since DTLS is typically implemented in userspace. Multihoming feature cannot be possible, since DTLS is used as lower layer in this stack, and it does not provide any address management to its upper layer. On the other hand, if SCTP is used as a lower layer, UDP encapsulation of SCTP will be used. It also supports both userspace and kernel implementations and multihoming feature. Multihoming feature will be achieved through ICE.

RTCWeb envisions that all the communication will be performed over a single connection. In order to facilitate the media channel, SCTP/DTLS/(ICE)UDP is more advantageous, since a DTLS connection can be reused for SRTP. DTLS will provide authentication keys for SRTP. In contrast, DTLS/SCTP would require to setup the SCTP association every time one client want to send data, since DTLS connection must not span multiple SCTP associations [TSR11]. Each SCTP packet contains Payload Protocol Identifier (PPID) that is used by upper layer to specify multiple protocols. The other end, therefore, demultiplexes the user messages based on the received PPID.

DCCP vs SCTP

Both DCCP and SCTP are suitable for real time communication services. There was discussion among IETF researchers for choosing DCCP or SCTP. DCCP provides unreliable data transfer with congestion control mechanism. Therefore, DCCP satisfies the unreliability requirements of RTCWeb, but for reliability issues, there should be some other mechanism on top of it. Typically, DCCP is implemented as part of operation system kernel. However, to support DCCP on top of DTLS, DCCP as part of user process is necessary to be implemented. In contrast, SCTP satisfies both reliability, unreliability requirements and recently, a very first version of SCTP userland stack has been implemented and released.

5.3 Data Channel Connection

Data channel is designed to provide a generic transport service allowing web browsers to exchange generic data in a bidirectional peer to peer fashion. Data channel can be defined as a bidirectional medium that supports in-sequence, out-of-sequence, or reliable and unreliable data transmission. A bidirectional channel is formed by two SCTP streams; one for outgoing and one for incoming data. A stream is defined as an unidirectional logical channel that exist within an SCTP association from one SCTP endpoint to another. Therefore, one incoming and one outgoing stream pair create one bidirectional data channel. User data is delivered to a particular stream either in order or unordered. It is noteworthy that the ordering is only preserved for those messages that are transferred on the same stream.

All negotiation between RTCWeb clients will be done over signaling channel. The basic idea behind the data channel connection is as follows:

- ICE negotiation will be performed first to gather IP addresses and port numbers that provide help to setup a bidirectional UDP channel.
- DTLS handshake will take place to setup a bidirectional UDP channel.
- SRTP can be used with key material derived from the DTLS connection.
- Now the SCTP handshake can be initiated running on top the the established DTLS connection.

SCTP association over DTLS will be established over signaling path using SDP. The protocol identifier required for this negotiation is defined and explained in [LC12]. The SCTP stack will just pass down the SCTP packet (without IP header) to DTLS layer.

As soon as SCTP association is established, the data channel can be created between two SCTP end-points using 3-way handshake as shown in Figure 5.4. The control messages for data channel negotiation are DATA_CHANNEL_OPEN_REQUEST, DATA_CHANNEL_OPEN_RESPONSE, and DATA_CHANNEL_ACK. It should be noted that data channel negotiation is performed over the established SCTP association. The SCTP association is created with a initial number of streams specified by the application (default value is 16). However, the default value can be extended by performing negotiation between end-points.

Data channels are closed by resetting the outgoing stream. Resetting a stream [SLT12] means to set the stream sequence numbers(SSN) of the stream to 'zero' and the application layer is notified accordingly that the reset has been performed. This feature allows the application to reuse the streams later if required. If an incoming stream is reset by the peer, a corresponding outgoing stream reset should be issued. When both incoming and outgoing streams of a channel are reset, the channel is closed.

`SCTP_RESET_STREAMS` option of SCTP allows the application to request the reset of incoming and/or outgoing streams.

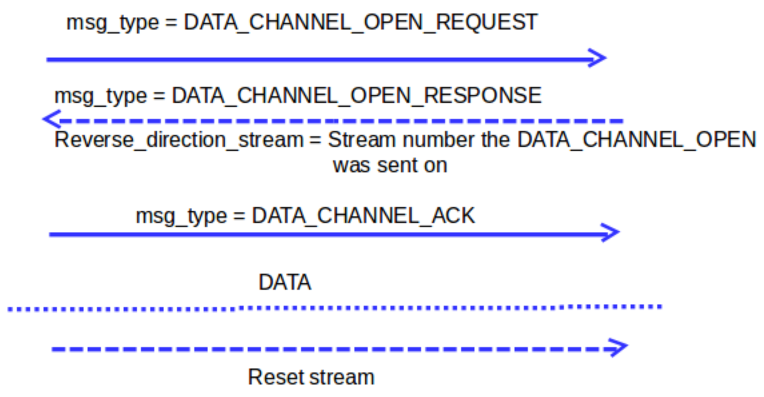

msg_type = DATA_CHANNEL_OPEN_REQUEST
msg_type = DATA_CHANNEL_OPEN_RESPONSE
Reverse_direction_stream = Stream number the DATA_CHANNEL_OPEN was sent on
msg_type = DATA_CHANNEL_ACK
DATA
Reset stream

Figure 5.4: Adding a data channel.

5.4 Congestion Control

SCTP performs congestion control based on per association. Therefore, all SCTP streams in an association share a single congestion window. Congestion control and avoidance mechanism for data channel is provided by SCTP. SCTP provides TCP-friendly congestion control. The congestion control mechanism can be modified to integrate media stream congestion control as well. Furthermore, multistreaming feature of SCTP will be utilized to create multiple data channels with different features, such as reliability or unreliability, partial reliability, and ordered or un-ordered delivery. SCTP also provides congestion control for large datagrams and PMTU-discovery. Fragmentation of large datagram is also possible if required.

6 Design and Experiment

In this chapter, We will build up a prototype for data channel in order to integrate the solution into WebKit. The prototype uses SCTP userland stack and GStreamer multimedia framework. We will also investigate the way Mozilla has integrated data channel protocol stack in the browser and how the data channel has been designed and implemented using SCTP userland stack.

6.1 WebKit

WebKit is an open source web browser engine. It started within Apple in 2001 as a fork of KHTML (KDE′s HTML Layout Engine) and KJS (KDE′s ECMAScript-JavaScript engine). It has evolved very fast and included support for HTML5. One of the major component of WebKit is WebCore that uses a number of objects to represent a web page in memory. Figure 6.1 represents a simplified version of

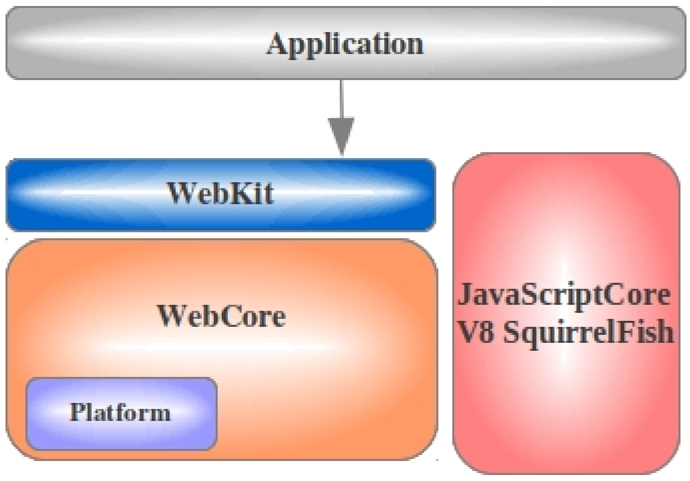


Figure 6.1: WebKit Architectural View

WebKit Architecture. Webkit architecture includes the following layers:

WebKit API layer that provides a full GObject-based API for programmatic manipulation of the Document Object Mode (DOM), making it easy for applications to control and modify content in embedded HTML views.

WebCore is a layout, network, multimedia, rendering, and Document Object Model (DOM short) library for HTML and SVG (Scalable Vector Graphics).

JavaScriptCore is a JavaScript Engine based on KJS. This framework is separate from WebCore and WebKit.

Platform This layer tempts to implement general-purpose platform specific operations.

6.2 GStreamer

GStreamer is an open source multimedia framework for creating streaming media applications. This framework is designed to help application programmer to easily deal with audio or video or both. It can also process any kind of data other than media data.

Basically, GStreamer is a plugin-based framework, where each plugin contains elements. Each element provides specific functionality such as reading/writing from/to files, encoding, decoding, filtering, or rendering data. A GStreamer pipeline can be created by linking some of these elements to perform a complex task.

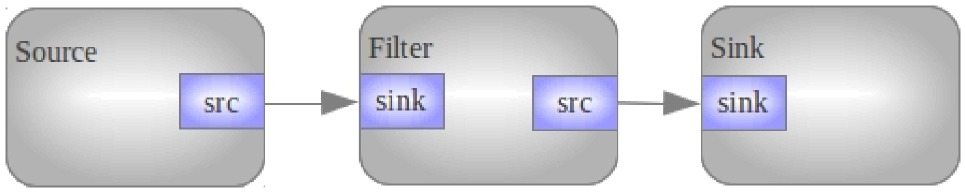


Figure 6.2: GStreamer with three linked elements.

Figure 6.2 shows a simple pipeline of three elements that are linked together. The output of first element will be used as input in the filter element. The Filter element will do something with the data buffer and forward it to next element. The first element is always a source that is used to push data buffer into a pipeline and sink element being last element is responsible for rendering data to the application. Buffers are the basic unit of data transfer in GStreamer. GstBuffer is the type of data passing buffer.

GStreamer has four element states: GST_STATE_NULL, GST_STATE_READY, GST_STATE_PAUSED, and GST_STATE_PLAYING. GST_STATE_NULL is the default state where no resources are allocated. In GST_STATE_READY state, an element allocates all of its global resources i.e., opening devices, allocating buffer and so on. In GST_STATE_PAUSED state, an element is allowed to modify a stream′s position, read and process data and such to prepare for playback as soon as state is changed to PLAYING. In this state, an element is not allowed to play the data which would make the clock run. In the PLAYING state, an element does similar functions as in the PAUSED state, except that the clock now runs.

6.3 Design and Implementation

This research started with the design shown in Figure 6.3. The basic idea is that the data from the WebCore would be fed into the GStreamer pipeline using Appsrc element and pass it to the next element SCTP that is responsible for providing congestion control functionality. SCTP element passes the data as SCTP packets

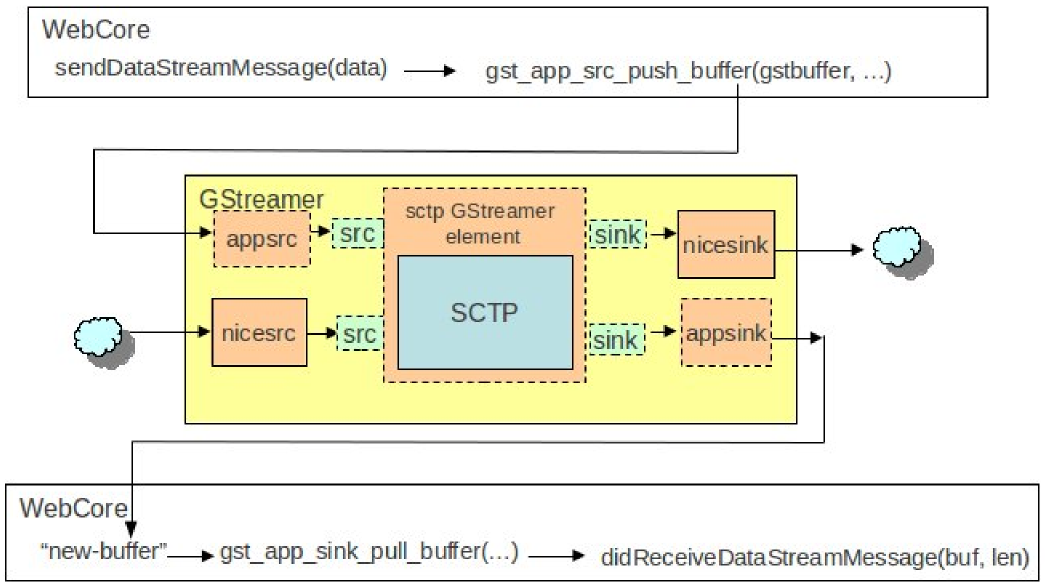

WebCore
sendDataStreamMessage(data)
gst_app_src_push_buffer(gstbuffer, ...)
GStreamer
appsrc
src
sctp GStreamer element
SCTP
sink
nicesink
nicesrc
src
sink
appsink
WebCore
"new-buffer"
gst_app_sink_pull_buffer(...)
didReceiveDataStreamMessage(buf, len)

Figure 6.3: Design for porting SCTP plugin with Congestion Control functionality.

to the nicesink element that renders the data to the network. Nicesrc is responsible for receiving data from the network and passes it to the SCTP element. Nicesink and nicesrc are implemented in libnice [4] library. However, the userspace SCTP

[4] http://nice.freedesktop.org/libnice/NiceAgent.html. ICE agent API implementation that take care of everything relating to ICE

implementation provides API that are dependant on IP and congestion control is performed over sending path. It needs major re-factoring of libusrsctp to make this design functional. Therefore, the design is simplified as shown in Figure 6.4.

The data from the WebCore will be fed into the GStreamer pipeline using Appsrc element and pass it to sctpsink element which is responsible for rendering data to the network. Sctpsrc element is responsible receiving data from the network and pass it to the WebCore via appsink element.

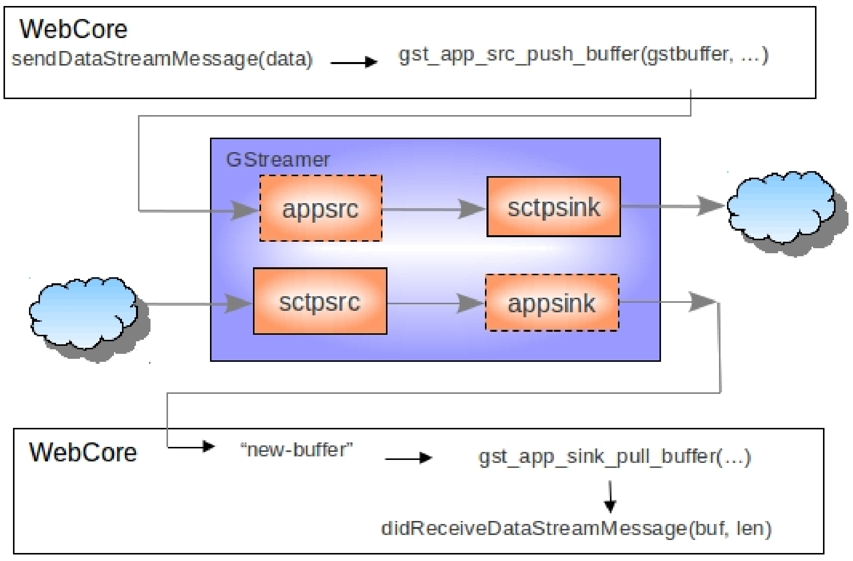

WebCore
sendDataStreamMessage(data)
gst_app_src_push_buffer(gstbuffer, ...)
GStreamer
appsrc
sctpsink
sctpsrc
appsink
WebCore
"new-buffer"
gst_app_sink_pull_buffer(...)
didReceiveDataStreamMessage(buf, len)

Figure 6.4: Design for porting SCTP plugin.

Appsrc element is used to push data into a GStreamer pipeline that provides external API for this purpose. Libgstapp library is required to be linked with appsrc to access the methods directly or by using appsrc action signals. Appsrc push data into pipeline by calling gst_app_src_push_buffer() method or by emitting the push buffer action signal. The buffer should be type of GstBuffer and before pushing data the following code segments are required:

```
buffer = gst_buffer_new();
GST_BUFFER_MALLOCDATA(buffer) = dataToBePushed;
GST_BUFFER_SIZE(buffer) = size of dataToBePushed;
GST_BUFFER_DATA(buffer) = GSTBUFFER_MALLOCDATA(buffer);
```

Appsink is used to provide methods in order to enable an application handling on the GStreamer data in a pipeline. Appsink retrieves the data buffers from a pipeline by calling gst_app_sink_pull_buffer() and gst_app_sink_pull_preroll() methods that block until a buffer is available in the sink or when the sink is closed or reaches End of Stream signals (EOS).

SCTP plugin contains two elements: sctpsrc and sctpsink. The sctpsink element consumes the data from a source element (e.g. Appsrc) and renders the data to the network. The sctpsrc element retrieves data from the network and transfer it to the next sink element for further action.

This implementation has used libusrsctp 0.9.0 version. The current version is 0.9.1. The work is still going on to release a stable version of this library. Libusrsctp is a userspace implementation of SCTP. This user-land stack is based on the FreeBSD kernel sources of SCTP and provides functionality enabling to perform as part of user process. It supports Windows, FreeBSD, Linux and Mac OS X. This library includes SCTP/IPv4 and SCTP/IPv6 and the UDP encapsulation variants.

In order to see the parameters used in SCTP plugin or to determine whether the plugin is working properly, gst-inspect command can be used. Obviously, GStreamer needs to be installed for this purpose. In this case, GStreamer-Core, GStreamer-Base-Plugins, and Gst-Plugins-Bad packages are required. GStreamer is available in many Linux package systems for a variety of distributions such as Debian, Red Hat, Ubuntu etc. A brief description of GStreamer plugin creation steps is presented in Appendix 2.

GStreamer provides two base classes that can be used to create a source element: GstBaseSrc provides the basic source functionality, and GstPushSrc is a non-byte exact source base-class. Pushsource base class itself derives from the base source class. Sctpsink element derives from GstBaseSink class and sctpsrc from GstPushSrc class. In order to integrate libusrsctp into this plugin, the configuration and Makefile.am file need to be changed so that these elements can use the APIs provided by libusrsctp.

Sink element is a special type of element in GStreamer that has to take care of preroll. Preroll is the process by which elements going into the GST_STATE_PAUSED state will have buffers ready after the state change i.e., they can start processing data immediately after going into the GST_STATE_PLAYING state. Sink elements can derive from GstBaseSink base-class, which does preroll and a few other utility functions automatically. To derive from base sink, we can use the following macro:

```
GST_BOILERPLATE_FULL (
GstSctpSink, gst_my_sink, GstBaseSink, GST_TYPE_BASE_SINK
);
```

Both sctpsink and sctpsrc element initialize the SCTP userland stack with a listening UDP port number as a parameter to the sctp_init() function and the remote port is set as a socket option, SCTP_REMOTE_UDP_ENCAPS_PORT, to the userspace_sctp_sockopt().

The congestion control and avoidance mechanism is done on the sending path i.e., `userspace_sctp _sendmsg()` function is invoked with data buffer to send out data. Congestion control is carried out while injecting data to the network. Sctpsink element uses this functionality.

The userland library for receiving messages in sctpsrc plugin can be used on two modes:

- For receiving messages a callback API is used: We need to register a callback which is called when a message can be delivered.
- For receiving messages a blocking API can be used by `userspace_sctp_recvm-sg()`. This puts the thread calling `userspace_sctp_recvmsg()` to sleep if there is no data. Much like the socket API.

The sctpsink and sctpsrc element initializes SCTP useland stack with UDP encapsulation properties to follow SCTP over UDP. It is done by setting SCTP_REMOTE _UDP_ENCAPS_PORT option as second argument and a pointer to struct sctp_ udpencaps is passed as third argument to the userspace_sctp_sockopt() function.

6.4 Result Analysis

The sending path in GStreamer solution includes the chain of two elements; appsrc and sctpsink. Another chain of sctpsrc and appsink is responsible receiving data from the network and transfer it to the application. Therefore, in order to enable bidirectional communication, two chains of this kind are required on each peer; one for sending and one for receiving. SCTP/UDP solutions have been implemented for data transmission.

The issue is whether GStreamer solution around SCTP is compatible with RTCWeb. To be compatible with RTCWeb, src/sink pair on each side should share the same socket to provide a bidirectional communication. For this purpose, there should be some way in GStreamer by which both elements can share the same socket. The possible solution may be to have the functionality of inter-element communication i.e., one element can pass some information to other element. Unfortunately, we have not found any mechanism for inter-element communication. GStreamer framework is not suitable with the current implementation of userland SCTP stack to add channel or remove channel on demand. The major re-factoring of SCTP implementation is required for including GStreamer into the WebKit framework.

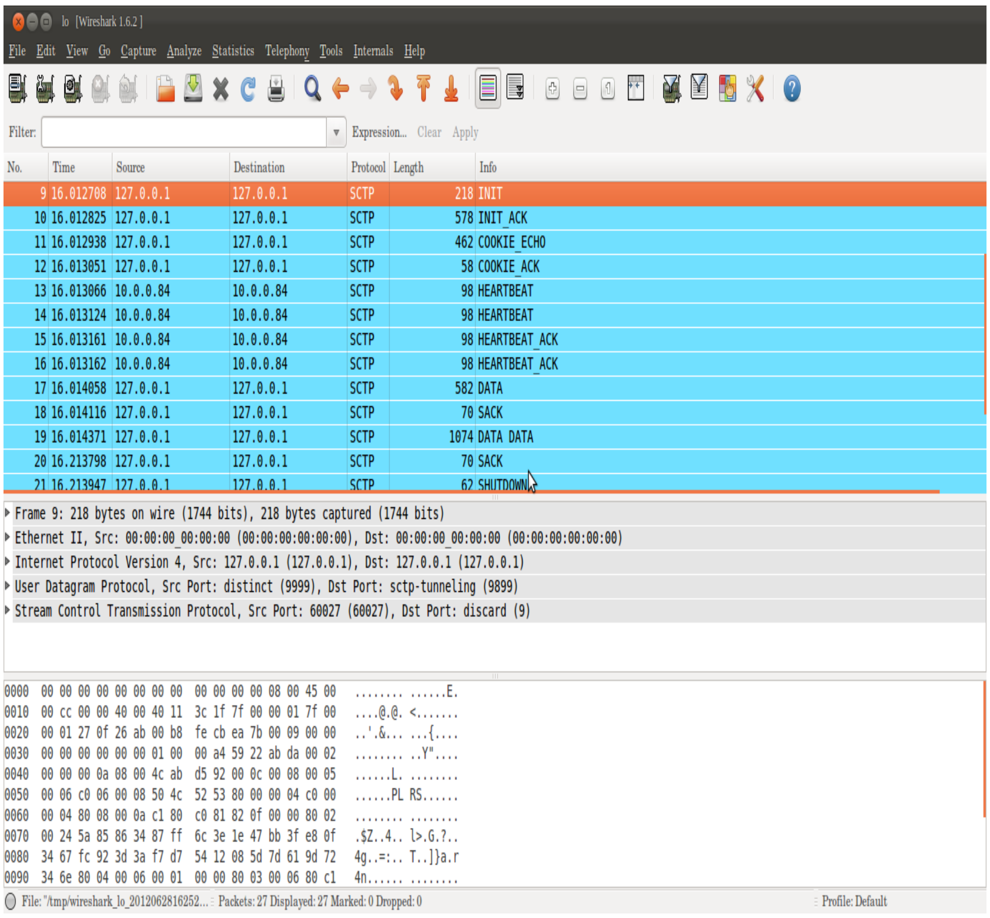
Discussion in the web community (Google, Mozilla) regarding GStreamer solution for data channel came into unanimity that GStreamer is not suitable medium to integrate SCTP into browser for the data channel. It will be better to integrate SCTP natively into browser. Mozilla has already integrated SCTP userland stack for data channel. This thesis work also investigated Mozilla architecture in order to know how the data channel is implemented, what properties are considered there, and finally performed some experiments on data channel JS API using Mozilla nightly builds which are only for testing purposes.

6.5 Data Channel in Mozilla

Recently, SCTP userland stack has been integrated in mozilla for a data channel solution. Data channel is part of network component of mozilla that enables the browser with data channel features in the context of RTCWeb. At present, the data channel features in Mozilla are the following:

- Bidirectional Communication.
- JavaScript API to create a data channel.
- Work is going on to support extending streams and stream reset.
- Data channel protocol stack SCTP/UDP. The vision is to implement SCTP/DTLS/(ICE)UDP.

In this thesis work, a simple chat is implemented and executed with Firefox Nightly build, which is for testing purposes. The snapshot is shown in Figure 6.6. The call set up is complicated and detailed procedure is presented in Appendix 1. The result realized in wireshark is shown in Figure 6.5.


lo [Wireshark 1.6.2]
File Edit View Go Capture Analyze Statistics Telephony Tools Internals Help
Filter:
Expression... Clear Apply

```
No.  Time       Source      Destination  Protocol Length Info
   9 16.012708  127.0.0.1   127.0.0.1    SCTP      218 INIT
  10 16.012825  127.0.0.1   127.0.0.1    SCTP      578 INIT_ACK
  11 16.012938  127.0.0.1   127.0.0.1    SCTP      462 COOKIE_ECHO
  12 16.013051  127.0.0.1   127.0.0.1    SCTP       58 COOKIE_ACK
  13 16.013066  10.0.0.84   10.0.0.84    SCTP       98 HEARTBEAT
  14 16.013124  10.0.0.84   10.0.0.84    SCTP       98 HEARTBEAT
  15 16.013161  10.0.0.84   10.0.0.84    SCTP       98 HEARTBEAT_ACK
  16 16.013162  10.0.0.84   10.0.0.84    SCTP       98 HEARTBEAT_ACK
  17 16.014058  127.0.0.1   127.0.0.1    SCTP      582 DATA
  18 16.014116  127.0.0.1   127.0.0.1    SCTP       70 SACK
  19 16.014371  127.0.0.1   127.0.0.1    SCTP     1074 DATA DATA
  20 16.213798  127.0.0.1   127.0.0.1    SCTP       70 SACK
  21 16.213947  127.0.0.1   127.0.0.1    SCTP       62 SHUTDOWN
```

```
Frame 9: 218 bytes on wire (1744 bits), 218 bytes captured (1744 bits)
Ethernet II, Src: 00:00:00_00:00:00 (00:00:00:00:00:00), Dst: 00:00:00_00:00:00 (00:00:00:00:00:00)
Internet Protocol Version 4, Src: 127.0.0.1 (127.0.0.1), Dst: 127.0.0.1 (127.0.0.1)
User Datagram Protocol, Src Port: distinct (9999), Dst Port: sctp-tunneling (9899)
Stream Control Transmission Protocol, Src Port: 60027 (60027), Dst Port: discard (9)
```

```
0000  00 00 00 00 00 00 00 00  00 00 00 00 08 00 45 00   ........ ......E.
0010  00 cc 00 00 40 00 40 11  3c 1f 7f 00 00 01 7f 00   ....@.@. <.......
0020  00 01 27 0f 26 ab 00 b8  fe cb ea 7b 00 09 00 00   ..'.&... ...{....
0030  00 00 00 00 00 00 01 00  00 a4 59 22 ab da 00 02   ........ ..Y"....
0040  00 00 00 0a 08 00 4c ab  d5 92 00 0c 00 08 00 05   ......L. ........
0050  00 06 c0 06 00 08 50 4c  52 53 80 00 00 04 c0 00   ......PL RS......
0060  00 04 80 08 00 0a c1 80  c0 81 82 0f 00 00 80 02   ........ ........
0070  00 24 5a 85 86 34 87 ff  6c 3e 1e 47 bb 3f e8 0f   .$Z..4.. l>.G.?..
0080  34 67 fc 92 3d 3a f7 d7  54 12 08 5d 7d 61 9d 72   4g..=:.. T..]}a.r
0090  34 6e 80 04 00 06 00 01  00 00 80 03 00 06 80 c1   4n...... ........
```

```
File: "/tmp/wireshark_lo_2012062816252... Packets: 27 Displayed: 27 Marked: 0 Dropped: 0    Profile: Default
```

Figure 6.5: Wireshark output while chatting on Mozilla browsers.

We used Node.js for web server and two firefox browser instances are required. Now, we need to do the following:

In one browser at left in Figure 6.6, Browse to `http://localhost:3000/datachan` URL will show`http://host:3000/listen/A/B`

Therefore, this browser will be listening for connections from other browsers.

Other browser at the right should browse to the following link:
`http://host:3000/connect/B/A/my_ip_address`
Note that A and B are reversed from what you see in the listener

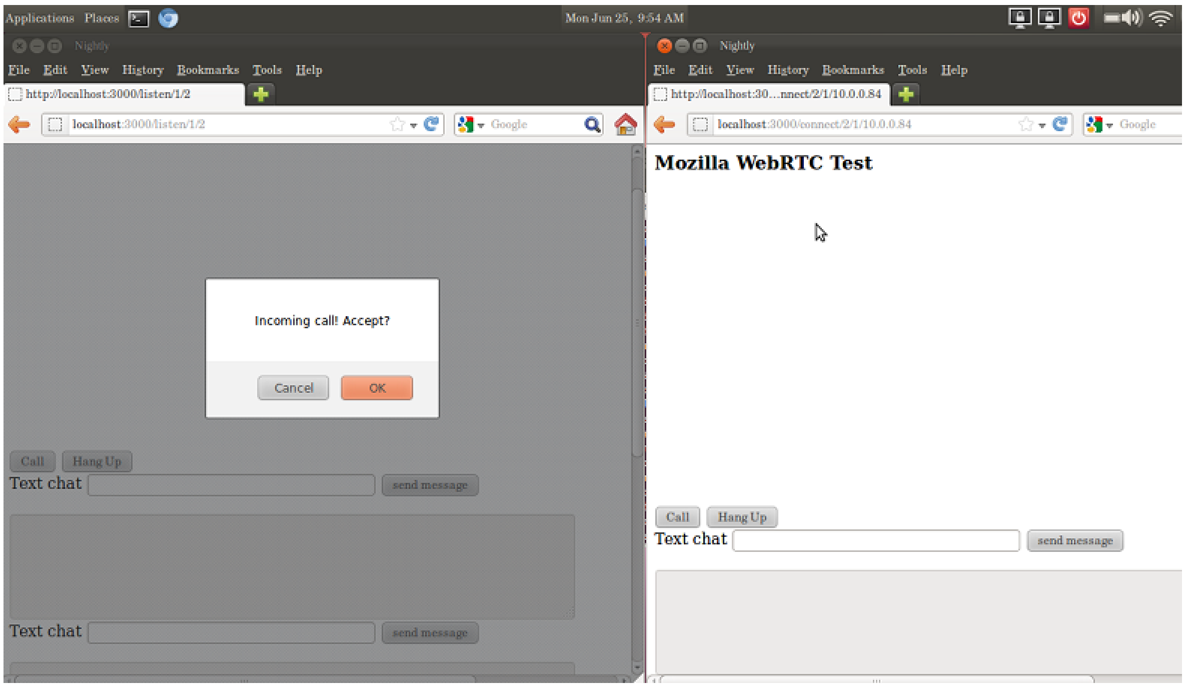


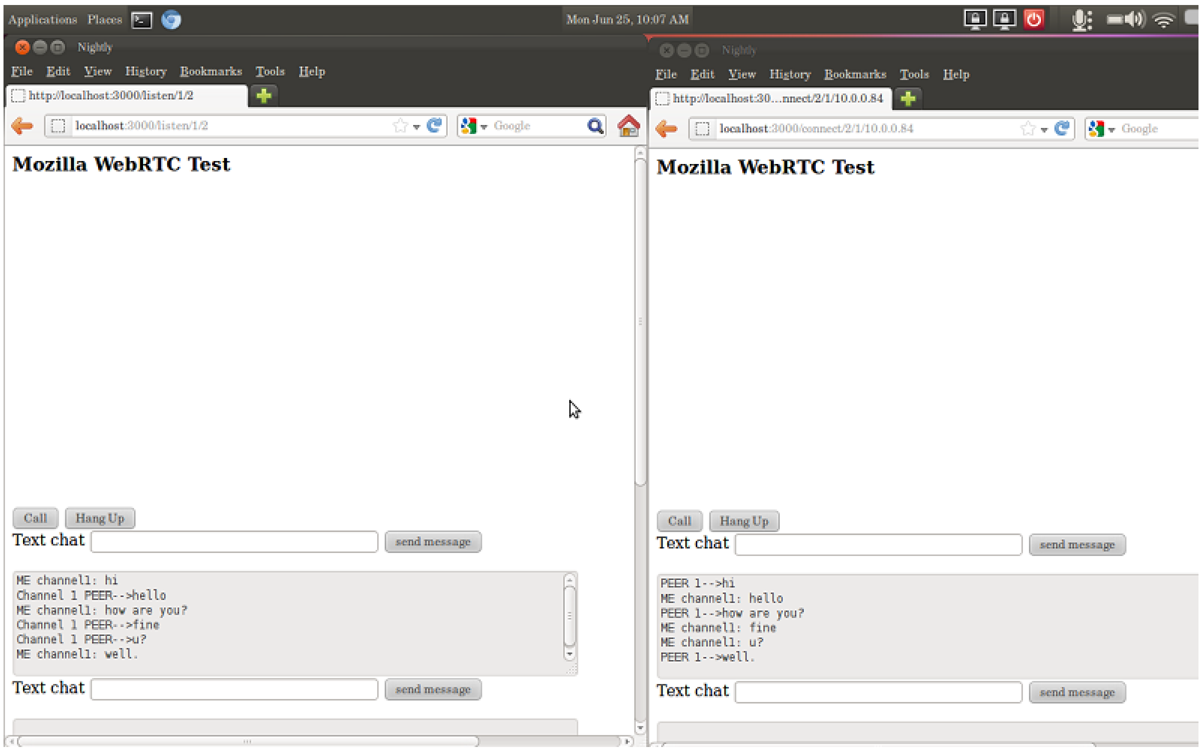
Figure 6.6: Bidirectional chat between two browsers.

If everything is going smooth, then 'Ringing' will be heard. When one browser presses 'accept' button, two browsers are ready for exchanging text data over data channel. The call negotiation is done over signaling channel. The snapshot for data exchange is presented in Figure 6.7.

Present implementation allows the data channel to open before the call is accepted, since the DTLS has not yet been integrated. It includes the 3-way handshake and allows immediate sending.

6.6 Challenges

At the beginning of this thesis work, we analysed the WebKit source code to find out the portion that is responsible for transportation of non-media data in Peer-


Applications
Places
Mon Jun 25, 10:07 AM
Nightly
File
Edit
View
History
Bookmarks
Tools
Help
http://localhost:3000/listen/1/2
localhost:3000/listen/1/2
http://localhost:30...nnect/2/1/10.0.0.84
localhost:3000/connect/2/1/10.0.0.84
Google
Call
Hang Up
send message

Mozilla WebRTC Test

Text chat

```
ME channel1: hi
Channel 1 PEER-->hello
ME channel1: how are you?
Channel 1 PEER-->fine
Channel 1 PEER-->u?
ME channel1: well.
```

Text chat

Mozilla WebRTC Test

Text chat

```
PEER 1-->hi
ME channel1: hello
PEER 1-->how are you?
ME channel1: fine
ME channel1: u?
PEER 1-->well.
```

Text chat

Figure 6.7: Bidirectional chat between two browsers.

Connection. The lack of proper documentation was the principal hindrance of this thesis work.

This thesis work has been performed with a very beginning unstable version of SCTP userland stack implementation. There is still no documentation written for this library. GStreamer basically is intended for multimedia applications, though it supports any kind of data. The documentation is not matured enough for non-media data application.

6.7 Critique

Though SCTP/DTLS provides congestion control, and message authentication, it experiences significant overhead during handshaking period. Theoretically, DTLS handshake message can be quite large and up to 2^{24} - 1 bytes (pragmatically many kilobytes) [RM06]. On the other hand, UDP datagrams are often less than 1500 bytes. In order to mitigate this limitation, and to run DTLS over UDP, DTLS handshake message may need to be fragmented over several DTLS records.

SCTP usually uses multihoming feature. SCTP endpoint uses different transport address if it detects congestion on using one particular address. DTLS does not support multihoming feature of SCTP, since DTLS does not have any address management mechanism for upper layer.

ICE will be used for gathering candidate addresses for transporting data over UDP. Here, the point is that ICE can change candidates unbeknownst to upper layer like DTLS or SCTP. This issues need to be taken care of while implementing the stack for data channel. Therefore, SCTP stack needs to be bound to an abstract data channel API, not to UDP transport to facilitate creating a DTLS shim under SCTP layer i.e., SCTP stack needs to be IP agnostic.

DTLS will be used as lower layer so that SRTP can also reuse the DTLS connection. The key issue here is that the maximum_lifetime parameter in the SRTP protection profile limits the duration to use each encryption and authentication key [RM06]. When this limit exceeds, a new DTLS session should be performed to replace the previous key.

Application using SCTP as part of a user process needs special privileges to access IP layer directly. Multihoming feature of SCTP leads to some limitation, since userspace process has no control over which interface user messages will be delivered. Additionally, if application crashes, the clients are left in a undefined state and experiences delay to recover if the application is restarted.

It is clearly understood that many messages other than actual user data need to be exchanged to set up the connection. SCTP itself supports data channel characteristics (e.g. ordered or unordered delivery, reliability or unreliability) to be controlled dynamically per chunk of transmitted data. Therefore, to create new channels and shut down one to accommodate different characteristics seems redundant. Additionally, every new channel requires 3-way handshake messages. It may create some delay for sending actual data. If a user message can also be carried with handshake messages, it may be advantageous reducing delays. However, this characteristics can be carried out by SCTP block of data as discussed in Section 5.3. It may help to reduce redundant exchange of information. JS API could specify these characteristics option to promote SCTP to use these options without any negotiation.

Userland SCTP implementation provides TCP friendly congestion control mechanism. It will be working fine if there is no real-time sessions going on. However, it would be really problematic with such congestion control mechanism even for sending or receiving a slight amount of data while any audio/video session is up especially in mobile networks where user buffer may be bloated easily. Therefore, it is necessary to balance bandwidth between media and data channels so as to avoid data channels causing congestion that might deteriorate the performance of media channels. It is possible to impose a bandwidth control manager controlled by main application and congestion code that could limit the data channels.

In real time communication the data channel may be often idle or low volume for long periods of time (e.g. instant messaging). Many applications may want to have data channel connection alive just in case they need to send some traffic. The SCTP over DTLS stack may lead a problem of having data channel keep alive for a long time due to power consumption. Moreover, frequent set up and shut down of data channel will cause extra overhead and introduce delay for sending the first message. Low volume situation may utilize websockets or HTTP instead of data channel to resolve this problem.

7 Conclusions and Future Work

IETF and W3C are working together on extending the web architecture. In particular, RTCWeb project aims to allow browsers to natively support interactive peer to peer communications and real time data collaboration. The fundamental goal of RTCWeb is to specify a set of protocols that support transportation of media data, such as audio, video, and non-media data along the most direct possible path between RTCWeb clients.

RTCWeb includes three components: media (i.e., audio, video), data, and control information (i.e., signaling). A short description of RTCWeb architecture has been presented in Chapter 4. For media channel, the amount of traffic, that can be generated, is known beforehand because of the codec and the profile chosen during call establishment, whereas data channel may experiences huge amount of data depending on the web user that may lead to congestion on the networks. To overcome this problem, we need a suitable congestion control for the data channel. Moreover, a suitable congestion control mechanism helps data channel to fairly compete with media channel.

Data channel is interesting area in RTCWeb that opens possibilities of future innovations in the Web arena. The possibility of transporting any kind of data will offer a web developer to create (or recreate) all kinds of applications such as P2P applications, gaming with RTCWeb/HTML5. In theory, it may even be possible to invent a new codec and transport audio and video on top of the data channel.

This thesis has discussed how the data channel could be designed for non-media data in the context of RTCWeb so that it will provide suitable congestion control mechanism for data channel, and security, authentication, integrity, and privacy of application data. Existing transport protocols such as TCP, UDP, RTP, SRTP, SCTP, DCCP, alone cannot meet all of the requirements of data channel. In chapter 5, the possible solutions for the protocol stack has been outlined briefly. IETF RTCWeb community have reached the consensus to use SCTP/DTLS/(ICE)UDP for data channel. SCTP will provide congestion control for each data channel performed based on each stream. DTLS will be responsible for providing security, confidentiality, and source authentication. RTCWeb clients are mostly web browsers and may be located behind NAT or firewall. Therefore, web browsers need to have native NAT traversal mechanisms. ICE with the help of STUN or TURN will provide NAT traversal solution.

In RTCWeb, PeerConnection implementation uses RTP over UDP for media data and UDP tunnelling for non-media data. The current implementation for transportation of media and non-media data has not included any congestion control on the end-points. How to integrate congestion control and avoidance mechanisms with current implementation of PeerConnection is still an open issue. This thesis focused on data part and selected SCTP userland stack implementation that supports SCTP over UDP with the goal to integrate it into WebKit. PeerConnection has utilized GStreamer for real time communication. In order to allow a smooth integration of the implementation within WebKit, we implemented GStreamer plugins that are able to send and receive data using SCTP over UDP.

There were two possibilities to integrate SCTP into WebKit; one to integrate natively and another one is using GStreamer plugin. GStreamer framework has been used for the transportation of media data. Therefore, this work has given effort to integrate SCTP within WebKit using GStreamer multimedia framework. In Chapter 6, the design for using SCTP around GStremer has been shown and discussed. Recently, Mozilla has integrated SCTP userland stack natively into the browser for data channel. As a proof of concept, we downloaded Mozilla source code and executed it with Firefox nightly build. Two instances of Mozilla browser can communicate through data channel. Currently, Mozilla browser uses SCTP/UDP for data channel. The vision is to implement SCTP/DTLS/(ICE)UDP.

Future Work

Userspace SCTP implementation has not yet been released as a stable version. In current implementation, all SCTP API is dependent on IP layer. The next step of userspace SCTP implementation will be to make SCTP stack IP agnostic i.e., this could support lower layers other than UDP or IPv4 or IPv6 (e.g., DTLS). This thesis work suggests to integrate SCTP natively instead of with GStreamer. Due to unavailability of stable version of libusrsctp and the limitation of time for this work, it was difficult to integrate SCTP natively in the WebKit. A thorough investigation of data channel integration into Mozilla browser concludes that the major functionality of data channel could be performed using userspace SCTP implementation. Future work could be to give similar efforts for WebKit.

In future work, it would be useful to investigate the impact, benefits and performance gain with data channel protocol stack; how it will have impact on media channels in terms of bandwidth consumption. This can be done by having both media channels and multiple data channels up, and analyse how the bandwidth is consumed by different channels. Transmission delay measurement can also be performed to realize how well the protocol stack behaves.

8 References

AAI02 Alamgir, R., Atiquzzaman, M. and Ivancic, W., Effect of Congestion Control on the Performance of TCP and SCTP over Satellite Networks. *NASA Earth Science Technology Conf*, June, 2002.

AFP02 Allman, M., Floyd, S. and Partridge, C., Increasing TCP's Initial Window. RFC 3390, October, 2002.

AJ07 Audet, F. and Jennings, C., Network Address Translation (NAT) Behavioral Requirements for Unicast UDP. RFC 4787, January, 2007.

AJ12 Alvestrand, H. and Jesup, R., Congestion Control Requirements for Real Time Media. (work in progress) March, 4, 2012. `http://tools.ietf.org/html/draft-jesup-rtp-congestion-reqs-00`. [last accessed in 14.6.2012]

Alv12 Alvestrand, H., Overview: Real Time Protocols for Brower-based Applications. (work in progress), June, 2012. `http://tools.ietf.org/html/draft-ietf-rtcweb-overview-04`. [last accessed in 19.8.2012]

APB09 Allman, M., Paxson, V. and Blanton, E., TCP congestion control. RFC 5681, September, 2009.

Bag11 Bagnulo, M., Threat Analysis for TCP Extensions for Multipath Operation with Multiple Addresses. RFC 6181, March, 2011.

BAW$^+$12 Blanton, E., Allman, M., Wang, L., Jarvinen, I., Kojo, M. and Nishida, Y., A Conservative Loss Recovery Algorithm Based on Selective Acknowledgment (SACK) for TCP. RFC 6675, August, 2012.

Bel10 Bellis, R., DNS Transport over TCP-Implementation Requirements. RFC 5966, August, 2010.

BMN$^+$04 Baugher, M., McGrew, D., Naslund, M., Carrara, E. and Norrman, K., The Secure Real-time Transport Protocol (SRTP). RFC 3711, March, 2004.

BPSK97 Balakrishnan, H., Padmanabhan, V., Seshan, S. and Katz, R., A comparison of mechanisms for improving TCP performance over wireless links. *Networking, IEEE/ACM Transactions on*, 5,6(1997), pages 756–769.

CMP12	Castillo, I., Millan, J. and Pascual, V., The Websocket Protocol as a Transport for the Session Initiation Protocol (SIP). (work in progress), April 6, 2012. `http://tools.ietf.org/html/draft-ibc-sipcore-sip-websocket-02`. [last accessed in 14.6.2012]
DH98	Deering, S. and Hinden, R., Internet Protocol, version 6 (IPv6) specification. RFC 2460, December, 1998.
DR06	Dierks, T. and Rescorla, E., The Transport Layer Security (TLS) Protocol. RFC 4346, April, 2006.
Edd07	Eddy, W., TCP SYN Flooding Attacks and Common Mitigations. RFC 4987, August, 2007.
FK06	Floyd, S. and Kohler, E., Profile for Datagram Congestion Control Protocol (DCCP) congestion control ID 2: TCP-like congestion control. RFC 4341, March, 2006.
FK09	Floyd, S. and Kohler, E., Profile for Datagram Congestion Control Protocol (DCCP) congestion control ID 4: TCP-friendly rate control for small packets (TFRC-SP). RFC 5622, August, 2009.
FKP06	Floyd, S., Kohler, E. and Padhye, J., Profile for Datagram Congestion Control Protocol (DCCP) congestion control ID 3: TCP-friendly rate control (TFRC). RFC 4342, March, 2006.
HFGN12	Henderson, T., Floyd, S., Gurtov, A. and Nishida, Y., The NewReno modification to TCP's fast recovery algorithm. RFC 6582, April, 2012.
HFPW03	Handley, M., Floyd, S., Padhye, J. and Widmer, J., TCP friendly rate control (TFRC): Protocol specification. RFC 3448, January, 2003.
HJP06	Handley, M., Jacobson, V. and Perkins, C., SDP: Session Description Protocol. RFC 4566, July, 2006.
HL10	Hammer-Lahav, E., The OAuth 1.0 Protocol, RFC 5849, April, 2010.
Jac88	Jacobson, V., Congestion avoidance and control. *ACM SIGCOMM Computer Communication Review*, volume 18. ACM, 1988, pages 314–329.
Jac90	Jacobson, V., Modified TCP congestion avoidance algorithm. Note sent to end2end interest mailing list, April 30, 1990.

JB11 Jennings, C. and Bran, C., RTCWeb Non-media Data Transport Requirements. (work in progress), July 4, 2011. `http://tools.ietf.org/html/draft-cbran-rtcweb-data-00`. [last accessed in 14.6.2012]

JRUJ00 Jennings, C., Rosenberg, J., Uberti, J. and Jesup, R., RTCWeb Offer/Answer Protocol (ROAP). (work in progress) October, 2000. `http://tools.ietf.org/html/draft-jennings-rtcweb-signaling-01`. [last accessed in 28.6.2012]

JU12 Jennings, C. and Uberti, J., Javascript Session Establishment Protocol. (work in progress) August 19, 2012. `http://tools.ietf.org/html/draft-uberti-rtcweb-jsep-02`. [last accessed in 19.7.2012]

KA98 Kent, S. and Atkinson, R., Security Architecture for the Internet Protocol. RFC 2401, November, 1998.

KHF06 Kohler, E., Handley, M. and Floyd, F., Datagram Congestion Control Protocol (DCCP). RFC 4340, March, 2006.

KS99 Karn, P. and Simpson, W., Photuris: Session-key Management Protocol. RFC 2522, March, 1999.

LBSA$^+$09 Ludwig, S., Beda, J., Saint-Andre, P., McQueen, R., Egan, S. and Hildebrand, J., Jingle. XMPP Standards Foundation, December, 2009.

LC12 Loreto, S. and Camarillo, G., Stream Control Transmission Protocol (SCTP)-based Media Transport in the Session Description Protocol (SDP). (work in progress) July, 31, 2012. `http://tools.ietf.org/html/draft-ietf-mmusic-sctp-sdp-00`. [last accessed in 10.7.2012]

MMFR96 Mathis, M., Mahdavi, J., Floyd, S. and Romanow, A., TCP selective acknowledgement options. RFC 2018, October, 1996.

MMR10 Mahy, R., Matthews, P. and Rosenberg, J., Traversal Using Relays around NAT (TURN): Relay Extensions to Session Traversal Utilities for NAT (STUN). RFC 5766, April, 2010.

MNTW01 Miller, B., Nixon, T., Tai, C. and Wood, M., Home networking with universal plug and play. *Communications Magazine, IEEE*, 39,12(2001), pages 104–109.

NBBB98 Nichols, K., Black, D., Blake, S. and Baker, F., Definition of the differentiated services field (DS field) in the IPv4 and IPv6 headers. RFC 2474, December, 1998

Per02 Perkins, C., Mobile IP. *Communications Magazine, IEEE*, 40,5(2002), pages 66–82.

Pos80 Postel, J., User Datagram Protocol. RFC 768, August 28, 1980.

Pos81 Postel, J., Internet Protocol. RFC 791, September, 1981.

Pos97 Postel, J., Transmission Control Protocol. RFC 793, September, 1997.

Res12 Rescorla, E., Security Considerations for RTC-Web. (work in progress) July, 16, 2012. `http://tools.ietf.org/html/draft-ietf-rtcweb-security-arch-03`. [last accessed in 19.7.2012]

RKLR12 Rosenberg, J., Keranen, A., Lowekamp, B. and Roach, A., TCP Candidates with Interactive Connectivity Establishment (ICE). RFC 6544, March, 2012.

RM06 Rescorla, E. and Modadugu, N., Datagram Transport Layer Security. RFC 4347, April, 2006.

RMMW08 Rosenberg, J., Mahy, R., Matthews, P. and Wing, D., Session Traversal Utilities for NAT (STUN). RFC 5389, October, 2008.

Ros10 Rosenberg, J., Interactive Connectivity Establishment (ICE): A protocol for Network Address Translator (NAT) Traversal for Offer/Answer Protocols. RFC 5245, April, 2010.

RR06 Recordon, D. and Reed, D., OpenID 2.0: a platform for user-centric identity management. *Proceedings of the second ACM workshop on Digital identity management.* ACM, 2006, pages 11–16.

RS02 Rosenberg, J. and Schulzrinne, H., An offer/answer model with Session Description Protocol (SDP). RFC 3264, June, 2002.

RSC$^+$02 Rosenberg, J., Schulzrinne, H., Camarillo, G., Johnston, A., Peterson, J., Sparks, R., Handley, M. and Schooler, E., SIP: Session Initiation Protocol. RFC 3261, June, 2002.

RWHM08 Rosenberg, J., Weinberger, J., Huitema, C. and Mahy, R., Simple Traversal of User Datagram Protocol (UDP) Through Network Address Translator (NATs). RFC 3489, March, 2008

SA04 Saint-Andre, P., Extensible Messaging and Presence Protocol (XMPP): Core. RFC 3920, October, 2004.

Sav06 Savola, P., MTU and Fragmentation Issues with In-the-Network Tunneling. RFC 4459, April, 2006.

SCFJ03 Schulzrinne, H., Casner, S., Frederick, R. and Jacobson, V., RTP: A Transport Protocol for Real-Time Applications. RFC 3550, July, 2003.

SE01 Srisuresh, P. and Egevang, K., Traditional IP Network Address Translator (traditional NAT). RFC 3022, January, 2001.

Sen02 Senie, D., Network Address Translator (NAT)-friendly Application Design Guidelines. RFC 3235, January, 2002.

SH99 Srisuresh, P. and Holdrege, M., IP Network Address Translator (NAT) Terminology and Considerations. RFC 2663, August, 1999

SKF08 Srisuresh, P., Kegel, D. and Ford, B., State of Peer-to-Peer (P2P) communication across network address translators (NATs). RFC 5128, March, 2008.

SLT12 Stewart, R., Lei, P. and Tuexen, M., Stream Control Transmission Protocol (SCTP) Stream Reconfiguration. RFC 6525, February, 2012.

SXM^{+}07 Stewart, R., Xie, Q., Morneault, K., Sharp, C., Schwarzbauer, H., Taylor, T., Rytina, I., Kalla, M., Zhang, L. and Paxson, V., Stream Control Transmission Protocol (SCTP). RFC 4960, September, 2007.

TLJ12a Tuexen, M., Loreto, S. and Jesup, R., RTCWeb Datagram Connection. (work in progress) March, 6, 2012. `http://tools.ietf.org/html/draft-ietf-rtcweb-data-channel-00`. [last accessed in 14.6.2012]

TLJ12b Tuexen, M., Loreto, S. and Jesup, R., WebRTC Data Channel Protocol. (work in progress) June, 9, 2012. `http://tools.ietf.org/id/draft-jesup-rtcweb-data-protocol-01.txt`. [last accessed in 29.7.2012]

TS11 Tuexen, M. and Stewart, R., UDP Encapsulation of SCTP packets. (work in progress) October, 29, 2011. `http://tools.ietf.org/html/draft-ietf-tsvwg-sctp-udp-encaps-01`. [last accessed in 14.7.2012]

TSLJ12 Tuexen, M., Stewart, R., Loreto, S. and Jesup, R., DTLS Encapsulation of SCTP Packets for RTCWeb. (work in progress), July, 16, 2012. `http://tools.ietf.org/html/draft-tuexen-tsvwg-sctp-dtls-encaps-01`. [last accessed on 19.7.2012]

TSR11 Tuexen, M., Seggelmann, R. and Rescorla, E., Datagram Transport Layer Security (DTLS) for Stream Control Transmission Protocol (SCTP). RFC 6083, January, 2011.

Win07 Wing, D., Symmetric RTP/RTP Control Protocol (RTCP). RFC 4961, July, 2007.

XSHT07 Xie, Q., Stewart, R., Holdrege, M. and Tuexen, M., SCTP NAT Traversal Considerations. (work in progress), November, 17, 2007. `https://tools.ietf.org/html/draft-xie-behave-sctp-nat-cons-03`. [last accessed on 10.7.2012]

Appendix 1. Data Channel in Mozilla

A Checkout Mozilla Source Code

If you want to be familiar with Mozilla source code and Mozilla software architecture, you can check the following link:

`http://jollyrogerelakiri.blogspot.com/2011/05/conceptual-architecture-of-firefox.html`

B Download source code

- hg clone `http://hg.mozilla.org/projects/alder`

  (wait...)

- cd alder

- hg update -r paris_demo –clean

Now, we have mozilla source code. To execute it from terminal, we need to create our own .mozconfig file. The content will be like as follows:

```
# Import the stock config for building the browser (Firefox)
. topsrcdir/browser/config/mozconfig

# Define where build files should go. This places them in the directory
# "obj-ff-dbg" under the current source directory
mk_add_options MOZ_OBJDIR=@TOPSRCDIR@/obj-ff-dbg
# -s makes builds quieter by default
# -j4 allows 4 tasks to run in parallel. Set the number to be the amount of
# cores in your machine. 4 is a good number.
mk_add_options MOZ_MAKE_FLAGS="-s -j4"

# Enable debug builds
ac_add_options –enable-debug
```

```
mk_add_options MOZ_OBJDIR=@TOPSRCDIR@/obj-@CONFIG_GUESS
@-debug

# Turn off compiler optimization. This will make applications run slower,
# will allow you to debug the applications under a debugger, like GDB.
ac_add_options –disable-optimize
CC=clang
CXX=clang++
#mk_add_options AUTOCONF=/usr/local/Cellar/autoconf213/2.13/bin/auto
conf213
mk_add_options MOZ_MAKE_FLAGS="-j2"
ac_add_options –enable-webrtc
ac_add_options –disable-crashreporter
#ac_add_options –disable-strip
ac_add_options –with-ccache=/usr/bin/ccache
ac_add_options –enable-debugger-info-modules
```

C Compilation

Run the following commands in a shell to install the needed tools:

- `sudo apt-get build-dep firefox`
- `sudo apt-get install mercurial libasound2-dev libcurl4-openssl-dev libnotify-dev libxt-dev libiw-dev mesa-common-dev autoconf-2.13 yasm uuid`

If you face any problem or want to know more information, you can read the instructions from the link `https://developer.mozilla.org/En/Developer_Guide/Build_Instructions`.

Data Channel execution

- `cd /alder/obj-i686-pc-linux-gnu-debug/dist/bin/`

- We can test data channel between two browsers on the same machine, using ”./ `firefox -profilemanager -no-remote`”

N.B: If there is no web cam on the machine, then you need to copy a 176 × 144 (QCIF) raw YUV video to ”camera.yuv” in the directory that firefox is run from. This will substitute for a camera feed for one of the browsers (both if the machine has no web cam).

Data Channel simple demo

At this point, we have two instances of Mozilla browser. We used Node.js for web-server. Now, we need to do the following:

In one browser, Browse to `http://localhost:3000/datachan`
URL will show`http://host:3000/listen/A/B`

Therefore, this browser will be listening for connections from other browsers.

Other browser should browse to `http://host:3000/connect/B/A/my_ip_address`
Note that N and M are reversed from what you see in the listener

If everything is going smooth, then ′Ringing′ will be heard. If one browser accept the call, then these are ready for chatting over data channel.

Appendix 2. Data Channel Use Cases

D Use Cases for Unreliable Data Channels

Use case 1 A real-time game where position and object state information is delivered through unreliable data channels. At that moment, there may be reliable data channels as well or inactive media channels or no media channels.

Use case 2 Non-critical state (e.g. Mute state) updates about a client in a video chat or conference is sent via unreliable data channels.

E Use Cases for Reliable Data Channels

Use case 3 Critical state information such as control information of a real time game would be send over reliable data channel. Such a game may not include any media channel, or inactive media channels at any given time, or may only be added due to in-game actions.

Use case 4 Exchange of files between clients while chatting. This could be datagrams or streaming. File transfer may be of large numbers, sequential or parallel (e.g. sharing a folder of images or a directory of files.)

Use case 5 Text chatting while ongoing audio or video conference among multiple people.

Use case 6 Renegotiation of the set of media streams or data channels in the PeerConnection.

Use case 7 A browser may use data channels of a PeerConnection to exchange HTTP/HTTPS requests and data in order to avoid local internet filtering or monitoring.

Appendix 3. GStreamer Plugin Creation Step

Creation Step

The first step is to download a copy of gst-template git module that provides an important tool and source code template for a basic GStreamer plugin. We can check out the template module by the following command :

```
$ git clone git //anongit.freedesktop.orggstreamergst-template.git
```

This command will download a series of files and directories into gst-template. The source code for the template can be found in gst-template/gst-plugin/directory. In order to create the plugin, the following commands are required :

```
$ cd gst-template/gst-plugin/src

$ ../tools/make_element MyPlugin
```

The make-element tool will create two files: gstexamplefilter.c and gstexample-filter.h. Then one needs to change the Makefile.am to use the new filenames. After that the project can be built and installed using the following command:

```
$ ./autogen.sh

 $ sudo make

$ sudo make install
```

The userland library for receiving messages in sctpsrc plugin can be used on two modes: non-blocking and blocking mode. The mode is currently selected at compile time. The default is mode non-blocking. For the blocking mode, the following command is required:

```
./configure -disable-callback-api

make clean all
```