Master's Degree in Communications and Computer Networks Engineering

Politecnico di Torino

1859

Master's Degree Thesis

Benefits of IMS Data Channels for Voice Enrichment in 4G/5G: A Simulation-Based Approach

Supervisor:
Prof. Carla Fabiana Chiasserini

Candidate:
Michela Salvadori

External Supervisor:
Paolo Belloni

July 2025

Abstract

As mobile communication becomes more advanced and focused on user experience, there is growing demand for services that enrich voice and video calls in real time such as transcription, translation, emotion detection and file sharing. Traditional networks, though reliable, cannot guarantee the performance required for these new features.

This thesis focuses on a promising technology for future mobile services: the IMS Data Channel. It allows real-time exchange of data (like text, files or emotional feedback) during a voice or video call. The IMS Data Channel is an extension of the IP Multimedia Subsystem (IMS), which telecom operators already use to manage services over 4G, 5G and Wi-Fi. With this architecture, advanced services can run directly inside the call without using external apps, offering better control, security and quality.

This thesis is structured into three main parts. The first part provides a theoretical overview of how IMS Data Channel works, based on international standards. It introduces the main components involved and explains how secure, real-time data sessions are established and controlled. The focus is on how telecom operators can offer smart services during calls, while maintaining reliability and quality of service, thanks to a standardized, operator-managed infrastructure. The second part implements a real-time simulation of use cases using WebRTC and Artificial Intelligent services. The test environment is based on the Janus WebRTC Gateway and integrates Amazon services such as Transcribe, Translate and Comprehend. A Python Flask server acts as middleware, connecting the WebRTC clients to Amazon Web Services via software Application Programming Interfaces for real-time data exchange. In the simulated scenario, user A speaks or sends text messages in Italian. User B receives real-time English transcription, translated audio and text, as well as visual feedback based on sentiment analysis. The third part investigates the system's behavior under unstable network conditions, which are common in best-effort internet environments. Controlled degradations such as jitter, artificial congestion and packet loss are introduced to evaluate the impact on real-time performance. Metrics such as Round-Trip Time (RTT), time to first transcription and packet loss are collected and analyzed under various conditions. Results show that best-effort networks fail to guarantee stable performance.

Unlike WebRTC, which relies on public internet conditions and cannot ensure consistent performance, the IMS Data Channel operates within the operator's managed network and follows international standards designed to guarantee quality of service. By supporting features like resource reservation and traffic prioritization, it maintains low latency and service stability even under stress. These capabilities make the IMS Data Channel a promising candidate to support next generation of Artificial Intelligent enhanced communication services.

Acknowledgements

I would like to express my deepest gratitude to all those who have supported me throughout this journey and made this achievement possible.

I am especially grateful to Professor Carla Fabiana Chiasserini for her guidance, constructive feedback and valuable suggestions throughout the development of this work. Her guidance helped in shaping the direction and structure of the thesis.

I am also deeply thankful to my colleague Paolo Belloni, with whom I have the privilege of working. His continued encouragement and thoughtful advice have been invaluable in helping me move forward, especially in defining the scope and clarity of this research.

A special thank you goes to my friend Claudia, with whom I've shared moments of joy and sacrifices and a celebratory drink after each exam.

I warmly thank my partner Jean-Claude and his daughter Chloé for their patience and support during this demanding period. I am aware of the time I took away from you and I am deeply grateful for your understanding, love and patience.

To my parents, thank you for always standing by my side and for teaching me, through your example, the values of resilience and perseverance.

Lastly, I wish to thank my younger self, the little girl who once dreamed of becoming an engineer. We did it. After all these years, a circle is now complete and I am proud of who I have become. *What is meant to be, will be.*

Table of Contents

List of Figures

List of Tables

List of Code Listings

Acronyms

3GPP	3rd Generation Partnership Project
AI	Artificial Intelligence
AR	Augmented Reality
AS	Application Server
AWS	Amazon Web Services
DCAR	Data Channel Application Repository
DCAS	Data Channel Application Server
DCSF	Data Channel Signaling Function
DTLS	Datagram Transport Layer Security
GSMA	Global System for Mobile Communications Association
HSS	Home Subscriber Server
ICE	Interactive Connectivity Establishment
IETF	Internet Engineering Task Force
IMS	IP Multimedia Subsystem
IMS AS	IMS Application Server
IoT	Internet of Things
M2M	Machine-to-Machine Communications
MEC	Multi-access Edge Computing
MF	Media Function
mMTC	Massive Machine-Type Communications
MRF	Media Resource Function
MTSI	Multimedia Telephony Service for IMS
NAT	Network Address Translation
NEF	Network Exposure Function
NG-RTC	Next Generation Real-Time Communication
P-CSCF	Proxy Call Session Control Function

PCRF	Policy and Charging Rules Function
QoS	Quality of Service
RFC	Request For Comments
SBI	Service-Based Interface
SCTP	Stream Control Transmission Protocol
SDP	Session Description Protocol
SDK	Software Development Kit
SIP	Session Initiation Protocol
S-CSCF	Serving Call Session Control Function
UE	User Equipment
URLLC	Ultra-Reliable Low-Latency Communication
VoLTE	Voice over LTE
ViLTE	Video over LTE
VR	Virtual Reality
WebRTC	Web Real-Time Communication
WSL	Windows Subsystem for Linux
XR	Extended Reality

Chapter 1

Introduction

Overview of IMS

The IP Multimedia Subsystem (IMS) is an architectural framework designed to provide multimedia services over IP-based networks. Initially introduced to support real-time communication services such as voice and video over LTE (VoLTE and ViLTE) in 4G networks, IMS has progressively evolved to meet the demands of modern telecommunications. With the introduction of 5G, IMS plays a crucial role in enabling a wide range of services beyond traditional voice, including enhanced video conferencing, messaging and the integration of emerging technologies such as Augmented Reality (AR). IMS ensures interoperability across different access networks, including 4G, 5G and Wi-Fi, while maintaining high quality of service, scalability and seamless session continuity. This evolution makes IMS a key part of next-generation communication systems, linking telecommunications with artificial intelligent and machine learning algorithms. This allows for more interactive and personalized services that better meet the needs and expectations of the customer.

Importance of Data Channels in IMS

The integration of Data Channels into IMS networks represents a significant advancement in enhancing multimedia communication services. Data Channels, based on WebRTC protocols, enable real-time transmission of any type of data alongside traditional voice and video streams within the IMS framework. This capability not only extends the flexibility of IMS, but also facilitates the development of new interactive services such as file sharing, augmented reality (AR) applications and real-time data synchronization during calls. By supporting seamless and efficient data exchange over IP, Data Channels improve the overall user experience, enabling richer, more immersive communication scenarios. In addition, the introduction of Data Channels in IMS allows operators to leverage existing infrastructure to deliver innovative 5G services, ensuring scalability, quality of service (QoS) and interoperability across different network environments, including 4G, 5G and Wi-Fi. This positions Data Channels as a key enabler for the next generation of multimedia communication and service delivery in modern network architectures.

Thesis Objectives

The objective of this thesis is to provide a comprehensive analysis of the IMS Data Channel architecture, to address the key challenges associated with its deployment and to explore its potential integration with machine learning algorithms to enable intelligent and real-time services in next-generation networks. The thesis is structured into three main areas of focus, with the following specific goals:

- Analyze the IMS Data Channel architecture, detailing its main components, reference points, protocol stack and integration within the broader IMS framework.
- Identify and discuss the main challenges related to the deployment of IMS Data Channels, with particular attention to Quality of Service (QoS) management, security and session control.
- Simulate representative use cases involving real-time data exchange over a WebRTC-based environment, using the Janus WebRTC Gateway as a flexible testing platform. The simulated system is integrated with Amazon Web Services (AWS) through AWS SDKs and APIs, enabling the implementation and evaluation of advanced scenarios such as:
  - Real-time speech transcription;
  - Real-time speech-to-text translation;
  - Audio sentiment analysis;
  - Text translation;
  - Text sentiment analysis.

  AWS Transcribe Streaming SDK is used for real-time audio transcription, while the boto3 library (API) is employed for text translation and sentiment analysis services (Translate and Comprehend). These experiments aim to demonstrate how IMS Data Channels could enhance user experience when combined with AI-driven cloud services.
- Simulate network degradations typically encountered in best-effort environments, such as jitter, congestion and packet loss, assessing their impact on real-time audio and text communication. The results highlight the limitations of best-effort communication and the benefits of IMS Data Channels when combined with QoS mechanisms.

Chapter 2

IMS Data Channels: Concept and Standardization

Definition

IMS Data Channels are an innovative feature of the IP Multimedia Subsystem architecture, enabling real-time data transfer alongside traditional voice and video streams. These channels support advanced applications by allowing the simultaneous exchange of multiple data types, enhancing multimedia communication experiences within IMS sessions.

Defined by standards from the 3rd Generation Partnership Project (3GPP) and the Internet Engineering Task Force (IETF), IMS Data Channels leverage protocols such as RFC 8831 and WebRTC specifications. These protocols provide the technical foundation for secure, bidirectional and reliable data transmission, ensuring seamless integration with existing media streams of IMS.

The flexibility of IMS Data Channels makes them ideal for scenarios requiring concurrent data exchange, such as live transcription and translation, real-time collaboration, remote medical assistance, telehealth and augmented reality. Key documents, such as NG.134-v3.0-1 and ETSI TS 24.186, provide detailed insights into the role of IMS Data Channels in enabling these advanced services. They outline how the WebRTC Data Channel protocol stack facilitates the integration of innovative functionalities within the robust signaling and media handling framework of IMS.

Standards Specifications

IMS Data Channels align with established 3GPP, IETF and GSMA standards, ensuring interoperability, security and efficient multimedia session management. The following key specifications define their role within the IMS ecosystem:

- **RFC 8831 (IETF):** Specifies the use of Stream Control Transmission Protocol (SCTP) over Datagram Transport Layer Security (DTLS) for secure, reliable and real-time data exchange. SCTP's message-oriented approach is particularly suited for managing different data types, while DTLS ensures encryption and data integrity.
- **TS 23.228 Annex AC (3GPP):** Defines the architecture and procedures of the IP Multimedia Subsystem, providing the foundation for managing multimedia sessions, including the integration of Data Channels. This specification addresses key aspects

such as session management, quality of service (QoS) and security, ensuring seamless operation of Data Channels within the IMS environment.

- **TS 26.114 (3GPP):** Defines procedures for managing multimedia telephony services within IMS, including the integration of Data Channels alongside other media types.
- **TS 26.264 (3GPP):** Adds support for augmented reality (AR) applications, detailing mechanisms for incorporating AR components into IMS sessions.
- **NG.134-v3.0-1 (GSMA):** Highlights how IMS Data Channels enable interactive services like remote assistance and AR in both consumer and enterprise environments.
- **NG.129-v1.0 (GSMA):** Focuses on the value chain for Data Channel-enabled services, defining roles and interactions among operators, device manufacturers and developers.
- **3GPP Release 18:** Introduces significant improvements to IMS Data Channels, particularly for Augmented Reality (AR) applications, by defining real-time AR communication procedures in TS 26.264. These updates take advantage of 5G's ultra-low latency and high-bandwidth capabilities, ensuring seamless AR integration within IMS sessions. Although IMS Data Channels implement secure transport through DTLS encryption, full end-to-end encryption and standardized application-level data integrity mechanisms are still lacking. This limitation, especially critical for privacy-sensitive applications, has led some device manufacturers to hesitate in fully adopting IMS Data Channels pending further enhancements in security frameworks.
- **3GPP Release 19:** Aims to further enhance IMS Data Channels to support XR applications, including immersive video, spatial audio and avatar-based interactions.

Possible Real-World Applications of IMS Data Channels

The introduction of 5G has positioned IMS Data Channels as an essential enabler for next-generation communication services, addressing requirements for low latency, high throughput and robust security. Advanced 5G capabilities, such as network slicing and enhanced Quality of Service (QoS) handling, make it an ideal environment for deploying real-time applications over IMS Data Channels. Features like massive machine-type communications (mMTC) and ultra-reliable low-latency communication (URLLC) align with the data requirements of IMS Data Channels, particularly for use cases requiring reliable performance and responsiveness.

IMS Data Channels are designed to operate over IP-based networks, including IMS deployments in 4G, 5G and Wi-Fi. Their protocol interactions and signaling procedures are specified in 3GPP TS 24.186, which defines session setup, capability negotiation and resource allocation within IMS. This enables seamless integration of Data Channels within IMS-based multimedia services, supporting advanced applications.

Moreover, IMS Data Channels can be enhanced by integrating AI-driven services, enabling intelligent speech recognition, real-time translation, sentiment analysis and predictive analytics. This allows for more personalized, automated and context-aware interactions, improving applications such as real-time collaboration, remote assistance and interactive AR/VR experiences. The combination of AI and IMS Data Channels creates new opportunities for smart customer service, accessibility features for people with disabilities and advanced decision-making across IoT, healthcare and other real-time critical domains.

Building on these capabilities, various representative use cases are presented in the following subsections.

2.3.1 Real-Time Speech Transcription and Translation

IMS Data Channels enable real-time speech-to-text transcription and automatic translation, facilitating communication in multilingual environments and improving accessibility for people with hearing loss.

Applications:

- **Accessibility for the deaf and hard of hearing:** Real-time transcription of voice calls into text, allowing users with hearing loss to follow conversations seamlessly.
- **Multilingual customer service:** Automatic translation of conversations in real time, enabling businesses such as hotels, restaurants and travel agencies to interact with customers who speak different languages.
- **Sentiment Analysis for Enhanced Interactions:** Integration with AI-powered sentiment analysis allows businesses to detect emotions in customer interactions, adjusting tone and responses accordingly to improve service quality and customer satisfaction.

2.3.2 Smart Customer Service and Remote Assistance

IMS Data Channels can improve customer service by enabling features such as file sharing, real-time AR-assisted support and synchronized visual guidance.

2.3.3 Collaboration and Remote Productivity

IMS Data Channels facilitate real-time data exchange for enhanced multimedia conferencing, interactive document editing and screen sharing.

2.3.4 Immersive Entertainment and AR/VR

IMS Data Channels integrate AR/VR services into IMS, enabling interactive gaming, real-time training and augmented live events.

2.3.5 IoT and M2M Communications

IMS Data Channels enable real-time data exchange between IoT devices, supporting smart home automation, industrial IoT and predictive maintenance. While traditional IoT communication protocols are widely used, IMS Data Channels provide additional advantages in scenarios where IoT data needs to be integrated with IMS-based voice and video services.

Key Advantages of IMS Data Channels:

- **Seamless integration with IMS voice and video services:** IoT devices can transmit diagnostic data while simultaneously establishing a voice/video session through IMS.
- **Real-time synchronization of multimedia and sensor data:** an industrial IoT sensor can detect an anomaly and automatically initiate an IMS video call to a technician while sending sensor readings over a Data Channel.
- **Operator-controlled QoS and security:** Unlike cloud-based IoT solutions, IMS Data Channels operate within carrier-managed networks, ensuring end-to-end security and predictable Quality of Service (QoS).

2.3.6 Healthcare and Remote Diagnostics

IMS Data Channels can enhance telemedicine by enabling real-time diagnostics, remote patient monitoring and interactive medical consultations. While existing solutions (e.g., WebRTC or cloud-based services) already provide telehealth capabilities, IMS Data Channels offer unique advantages in environments requiring synchronous multimedia and data exchange under operator-controlled QoS.

Key Advantages of IMS Data Channels:

- **Simultaneous voice, video and medical data transmission:** A doctor in a remote consultation can receive live electrocardiogram readings or vital signs data while interacting with the patient over an IMS-based video call.
- **operator-controlled security and reliability:** IMS ensures secure, low-latency transmission of sensitive medical data.
- **Guaranteed QoS for real-time medical applications:** IMS Data Channels leverage network-managed prioritization for critical healthcare services, ensuring smooth data transmission even in congested networks.

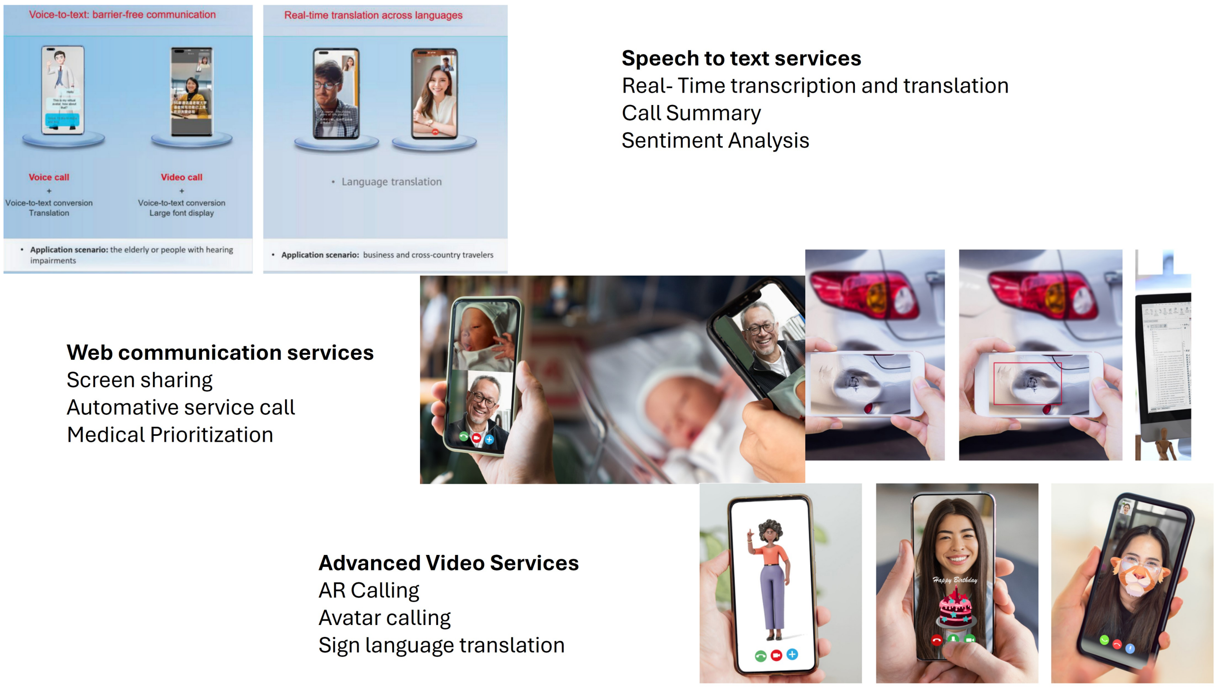

Voice-to-text: barrier-free communication
Voice call
+
Voice-to-text conversion
Translation
Video call
+
Voice-to-text conversion
Large font display
Application scenario: the elderly or people with hearing impairments
Real-time translation across languages
Language translation
Application scenario: business and cross-country travelers
Speech to text services
Real- Time transcription and translation
Call Summary
Sentiment Analysis
Web communication services
Screen sharing
Automative service call
Medical Prioritization
Advanced Video Services
AR Calling
Avatar calling
Sign language translation

Figure 1: Possible IMS Data Channel use cases

Chapter 3

Architecture of IMS Data Channels

This chapter explores the architecture of IMS Data Channels, including the essential components, the main interfaces, the protocol stack and QoS mechanisms within IMS environments. Standards such as NG.134-v3.0-1, NG.129-v1.0, ETSI IMSDC, TS 23.228, TS 29.175 and TS 29.330 are referenced to highlight the guidelines and requirements.

IMS Data Channel Reference Architecture

In March 2024, CT#103 (103rd meeting of the 3GPP Core Network and Terminals (CT) Technical Specification Group) finalized two important technical specifications, TS 29.175 and TS 29.330, completing the Release 18 work on NG-RTC (Next Generation Real-Time Communication). NG-RTC is a new framework for real-time communication services built on the IMS architecture. It combines advanced features like the IMS Data Channel, AI-driven media processing and Service-Based Interface (SBI) frameworks. These innovations enable richer and more interactive experiences, such as ultra-high-definition intelligent calling.

The IMS Data Channel, introduced in Release 16, is a flexible mechanism for transmitting data within IMS. It acts as a content-agnostic tunnel, offering several key advantages over traditional data channels used in EPC/5GC networks:

- **Seamless integration with IMS sessions:** The IMS Data Channel is set up as part of an IMS session (e.g., a voice or video call). This makes it easy to link data with the session, enabling advanced voice and video features.
- **User identity correlation:** The IMS Data Channel is tied to the authenticated identity of the user, simplifying how applications access user-specific information.
- **Quality of Service (QoS) guarantees:** Just like IMS audio and video streams, the IMS Data Channel supports dedicated bearers, ensuring reliable data transmission with guaranteed performance.

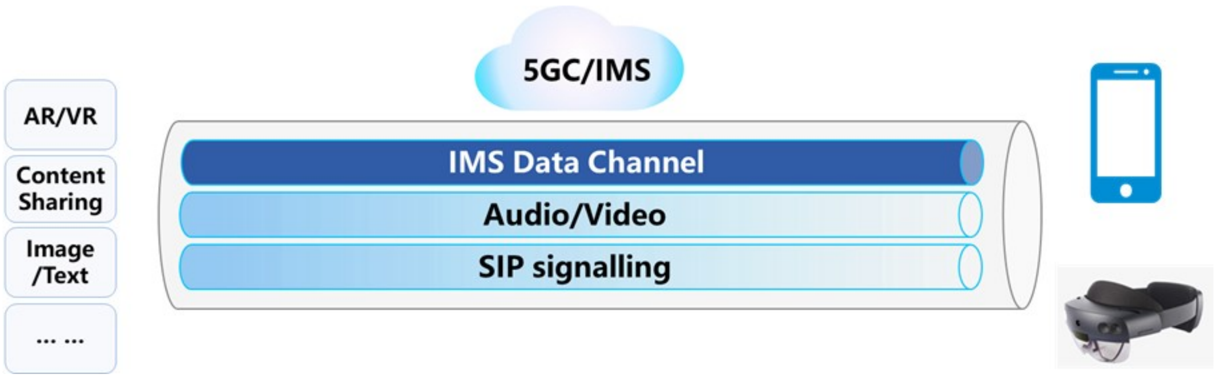


Figure 2: IMS Data Channel bearers

3.1.1 IMS Enhancements for Data Channel Integration

To facilitate the IMS Data Channel based applications, 3GPP has enhanced the IMS architecture by standardizing several new network functions in TS23.228:

- **Media Function (MF) / Media Resource Function (MRF)**: Acts as the media plane function for the IMS Data Channel, terminating IMS Data Channels from the UE (User Equipment) and establishing further IMS Data Channels towards the DC Application Server or another Media Function. The MF and MRF provide the media resource management and forwarding of data channel media traffic. The MF and MRF provide the following functionalities:
  - The MF and MRF manage the data channel media resources (bootstrap and application data channel resources, if applicable) under the control of the IMS AS;
  - The MF and MRF terminate the bootstrap data channel from the UE and forward HTTP traffic between the UE and DCSF (Data Channel Signalling Function) via MDC1 (Media Data Channel 1);
  - The MF and MRF may anchor the application data channel in P2P (Peer-to-Peer) scenarios, if required, and forward application data traffic from/to the UEs;
  - The MF and MRF relay traffic on the A2P/P2A (Application-to-Peer/Peer-to-application) data channels between the UE and the DC Application Server via MDC2 (Media Data Channel 2).
- **Data Channel Signaling Function (DCSF)**: The DCSF is the signalling control function that provides data channel control logic. The DCSF subscribes to IMS session events from the IMS Application Server (AS) and instructs the Media Function to perform media plane handling (e.g., IMS DC forwarding and media processing). The DCSF is not involved in SIP signalling. The DCSF supports the following functionalities:
  - receives event reports from the IMS AS and decides whether data channel service is allowed to be provided during the IMS session;

  - manages bootstrap data channel and (if applicable) application data channel resources at the MF or MRF via the IMS AS;
  - supports HTTP web server functionality to download data channel applications (bootstrapping) via MF and/or MRF to the UE based on UE subscription.
  - downloads data channel applications from the Data Channel Application Repository;
  - interacts with NEF for data channel capability exposure via N33 in 5G networks;
  - interacts with the DC Application Server for DC resource control via DC4/DC3, and for traffic forwarding via MDC3/MDC2 (Media Data Channel 3/Media Data Channel 2).
  - interacts with HSS (Home Subscriber Server) for retrieving and storing DCSF service specific data via N72/Sc.

- **DC Application Server** : Refers to an application server that leverages network capabilities to provide service logic for both individual subscribers and enterprise users. While it does not function as a dedicated network element, it acts as a service platform enabling applications that rely on network infrastructure

- **DC Application Repository (DCAR)**: Serves as a repository from which the UE can download data channel applications which are retrieved by the DCSF and needed for specific service logic. Applications do not need to be pre-installed on the UE but can be downloaded and launched during runtime. The UE accesses applications from DCAR indirectly via the DCSF and MF/MRF.

- **IMS AS**: The IMS AS is enhanced to support the following functionalities:
  - The IMS AS interacts with the DCSF via DC1 for event notifications;
  - The IMS AS receives the data channel control instructions from the DCSF and accordingly interacts with the MF via DC2 or with MRF via Mr'/Cr for data channel media resource management;
  - The IMS AS interacts with HSS for retrieving and storing DC enhanced IMS AS service specific data via N71/Sh.

- **S-CSCF**: The S-CSCF is enhanced to support the following functionalities:
  - The S-CSCF includes a Feature-Caps header field indicating its data channel capability in the 200 OK response to the initial and any subsequent REGISTER request from UE.

- **HSS**: In order to support data channel service, the HSS is additionally enhanced with the following functionalities:
  - stores IMS Data Channel subscription data as transparent data.
  - interacts with DCSF during service data retrieval by DCSF via N72/Sc.
  - interacts with IMS AS during service data retrieval by IMS AS via N71/Sh.

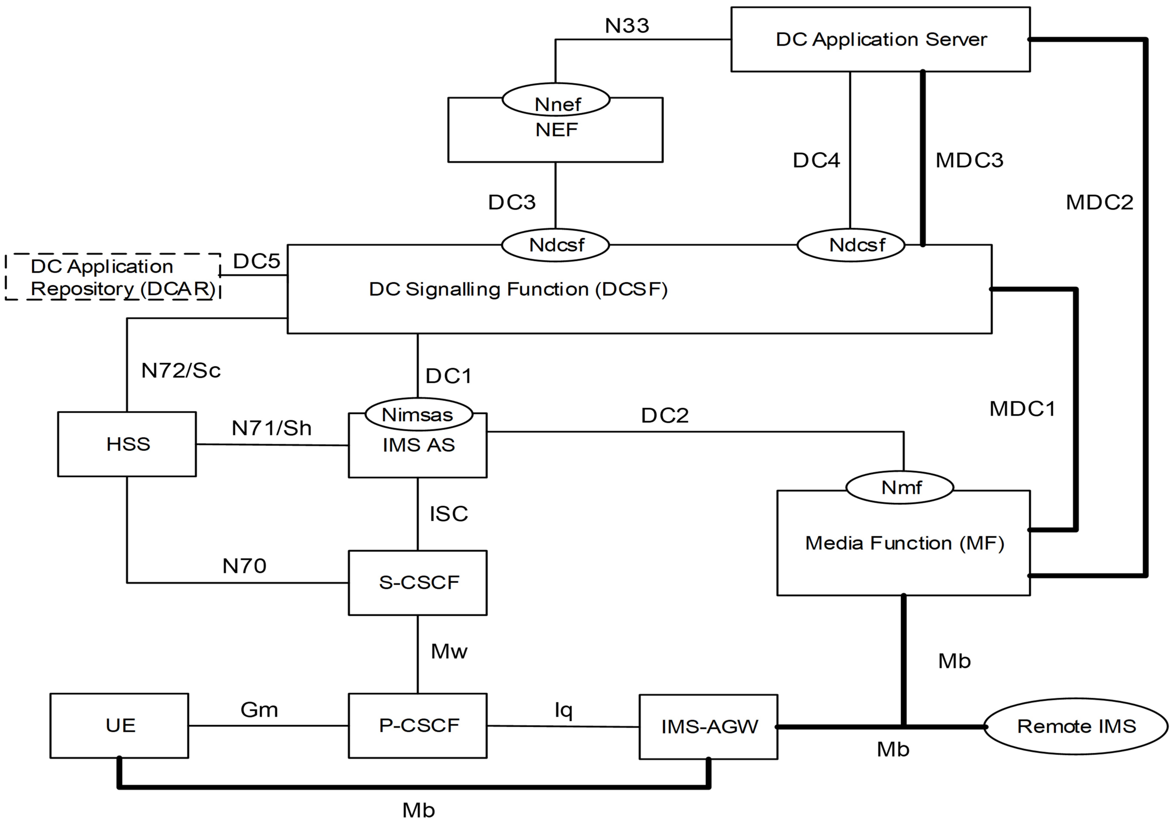


Figure 3: TS23.228 - Architecture option of IMS supporting DC usage with MF

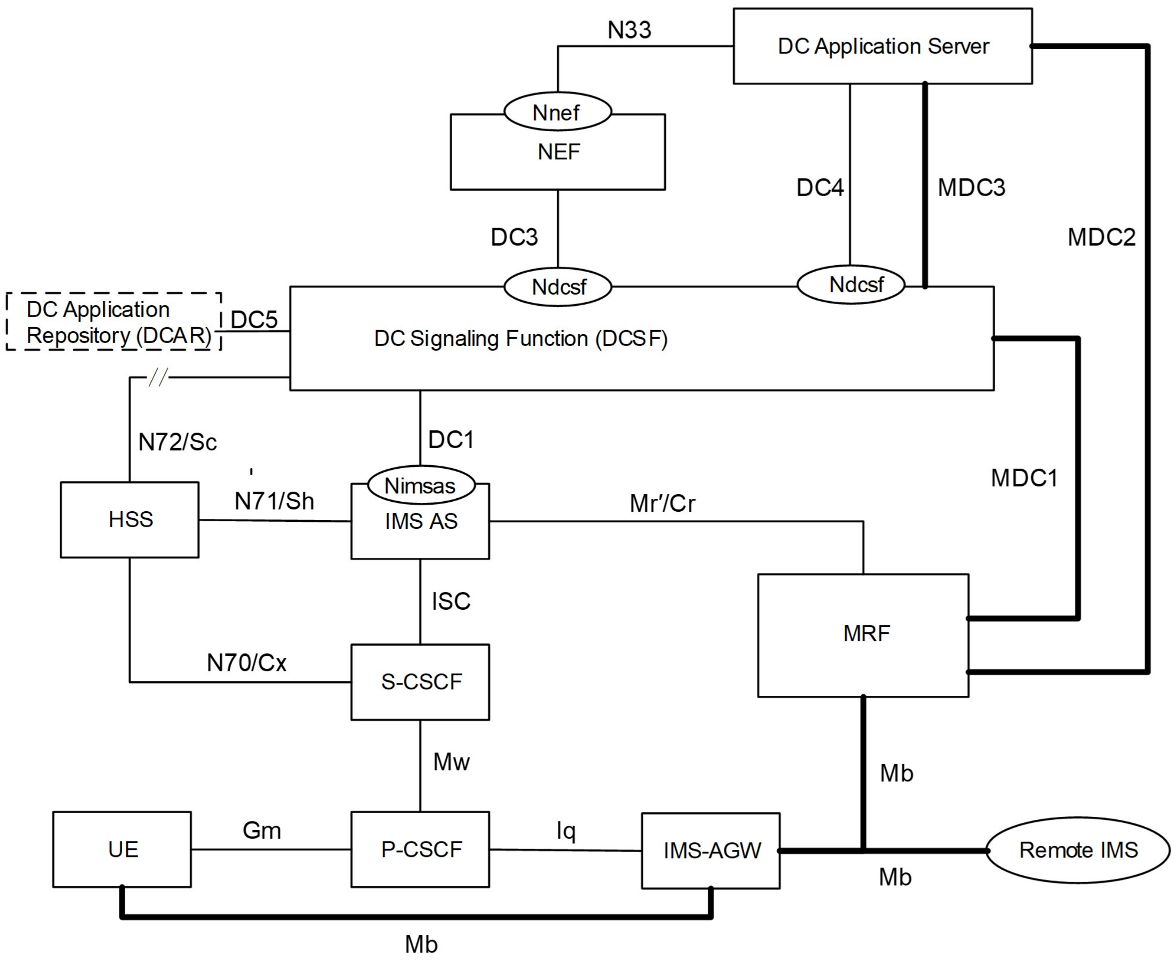


Figure 4: TS23.228 - Architecture option of IMS supporting DC usage with MRF

3.1.2 Reference Points

The following reference points are defined in TS23.228 to support the data channel service in IMS:

- **DC1:** Reference point between the DCSF and the IMS AS.
- **DC2:** Reference point between the IMS AS and MF.
- **DC3:** Reference point between the DCSF and NEF.
- **DC4:** Reference point between the DCSF and DC Application Server.
- **DC5:** Reference point between the DCSF and DCAR.
- **N72/Sc:** Reference point between the DCSF and HSS.

The following reference points are updated to support data channel signaling control in IMS:

- **N70/Cx/Dx:** Reference point between the CSCF and HSS.
- **N71/Sh:** Reference point between the IMS AS and HSS.

The following reference points are defined for data channel media handling:

- **MDC1:** Reference point for transport of data channel media between data channel media function (either MF or MRF) and DCSF.
- **MDC2:** Reference point for transport of data channel media between data channel media function (either MF or MRF) and DC Application Server.
- **MDC3:** Reference point for transport of data channel media between DCSF and DC Application Server.

The following reference point is updated to support data channel media handling:

- **Mr'/Cr:** SIP-based reference point between IMS AS and MRF.

Some of the above interfaces are SBI (Service Based Interfaces) interfaces that have been introduced as enhancement to IMS architecture for IMS Data Channel. IMS has been using SIP protocol for most of the interfaces, the strict state machine behind SIP protocol does not fit well the various new services emerging rapidly nowadays. A new SBI (Nimsas) has been defined for IMS AS, by consuming the APIs over Nimsas the DCSF can subscribe to the IMS session events, flexible service triggering is achieved via such mechanism. Furthermore, the DCSF exposes APIs towards DC applications via Ndcsf interface, the application servers can therefore dynamically discover and invoke the APIs provided by the network. All intentions behind the introduction of SBI into IMS is to make the real-time communication network more open and flexible.

IMS Data Channel Protocol Stack

The IMS Data Channel Protocol Stack follows a layered architecture, designed to ensure secure, flexible and real-time data transmission within IMS environments. This stack is built upon the WebRTC framework and has been extended to meet the demanding requirements of the IP Multimedia Subsystem (IMS), including:

- **High security**: End-to-end encryption using DTLS (Datagram Transport Layer Security) to protect data integrity and confidentiality.
- **Quality of service (QoS) management**: Unlike standard WebRTC, IMS enforces strict QoS mechanisms to ensure consistent performance across various network conditions.
- **Interoperability with 4G, 5G and different operators**: Seamless integration with mobile networks and IMS core functionalities, requiring standardized protocols such as SIP (Session Initiation Protocol) and SDP (Session Description Protocol)

Each layer has a specific role, from handling data transport and security to managing sessions, allowing IMS Data Channels to support various applications.

In particular, the IMS Data Channel protocol stack relies on two core transport protocols, SCTP (Stream Control Transmission Protocol) and DTLS (Datagram Transport Layer Security), to ensure secure and reliable data transmission. Additionally, the SIP (Session Initiation Protocol) and SDP (Session Description Protocol) protocols operate in the control plane, managing the signaling, session setup and media negotiation required for establishing and maintaining Data Channels.

The following sections provide a detailed description of these key protocols and their roles within the IMS Data Channel framework.

3.2.1 SCTP over DTLS

At the transport layer of the IMS Data Channel protocol stack, SCTP (Stream Control Transmission Protocol) runs over DTLS (Datagram Transport Layer Security). This combination provides a communication framework that is both secure and reliable, which is essential for supporting real-time, multimedia-oriented data exchange in IMS environments.

- **SCTP (Stream Control Transmission Protocol)**: As defined in RFC 8831, 3GPP TS 26.114 and adopted in NG.134-v3.0-1, SCTP is a message-based transport protocol that improves flexibility and reliability compared to traditional stream-based protocols like TCP. SCTP transmits data in discrete chunks rather than as a continuous stream, making it more efficient for multimedia services that need to deliver independent messages, such as chat, file sharing or augmented reality updates.

- **Reliable and Partially Reliable Data Transfer**: SCTP supports both fully reliable and partially reliable modes. This allows applications to choose between guaranteed delivery and reduced latency, depending on their requirements. For example, while file transfers may need full reliability, real-time AR or interactive messaging may benefit from faster delivery even if occasional data loss occurs.
- **Multistreaming and Head-of-Line Blocking Reduction**: SCTP enables multistreaming, allowing multiple independent logical streams within a single association. This prevents Head-of-Line (HoL) blocking, a situation where the delay or loss of one packet blocks the delivery of all following packets, even if they arrive on time. With SCTP, each stream operates independently, ensuring that a delay in one data flow does not impact others, which is critical for real-time responsiveness in IMS applications.

- **DTLS (Datagram Transport Layer Security)**: DTLS, specified by the IETF and integrated into 3GPP TS 26.114, provides encryption and data integrity for SCTP, securing IMS Data Channels against various threats. This ensures that sensitive data is protected throughout transmission.
  - **Data Confidentiality and Integrity**: DTLS applies end-to-end encryption to all data transmitted over SCTP, preventing unauthorized access. This is crucial for IMS applications handling sensitive data, such as medical records.
  - **Low-Latency Security**: Unlike traditional security mechanisms that may introduce delays, DTLS is optimized for real-time applications, maintaining security without compromising performance.
  - **Replay Protection and Authentication**: DTLS includes mechanisms for replay protection, preventing attackers from retransmitting intercepted packets to manipulate communication. Each message is uniquely marked, ensuring it is processed only once. Additionally, DTLS provides mutual authentication, verifying the identities of both the sender and receiver to prevent unauthorized access.

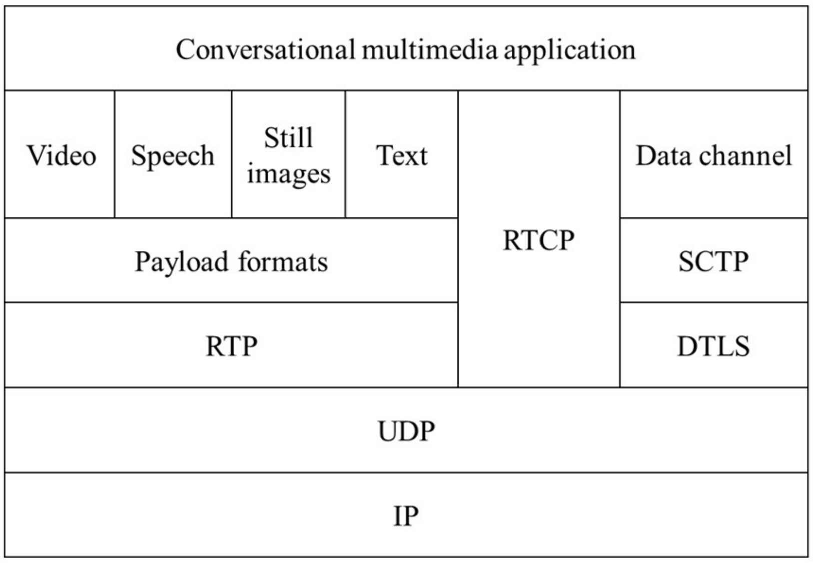


Figure 5: User plane protocol stack for a Data Channel client defined in 3GPP TS26.114

3.2.2 SIP and SDP Extensions

The Session Initiation Protocol (SIP) and Session Description Protocol (SDP) are essential components for managing sessions in the IP Multimedia Subsystem, particularly for supporting IMS Data Channels. SIP and SDP handle the signaling and media setup required to establish, modify and terminate sessions, providing IMS Data Channels with flexibility to adapt dynamically to network conditions and user needs. These protocols facilitate the establishment, negotiation and modification of Data Channels, enabling IMS applications to adapt to changing network conditions or user requirements. The extensions introduced in 3GPP TS 24.229, TS 26.114, ETSI IMSDC, enhance SIP and SDP to accommodate the unique demands of IMS Data Channels, including real-time data exchange, low latency and secure multimedia communication.

- **Session Initiation Protocol (SIP)**: SIP serves as the primary signaling protocol within IMS, handling session initiation, modification and termination. For IMS Data Channels, SIP extensions allow for:
  - **Channel Setup and Media Negotiation**: SIP initiates the session and negotiates media capabilities between endpoints. It enables the connected devices to negotiate parameters, including supported codecs, transport protocols (e.g., SCTP over DTLS) and encryption settings.
  - **Dynamic Session Modification**: SIP enables dynamic session modifications, allowing the addition, removal or adjustment of Data Channels during an active session. For example, an IMS voice or video call can be enhanced by adding a Data Channel for file sharing or displaying real-time speech to text transcription, real-time translation and sentiment analysis. SIP ensures that these Data Channels can be seamlessly incorporated without interrupting the existing session.
  - **Quality of Service (QoS) Signaling**: SIP signaling in IMS includes extensions for QoS negotiation, which allows Data Channels to request specific bandwidth and priority levels according to the application's needs.
- **Session Description Protocol (SDP)**: Embedded within SIP messages, SDP provides comprehensive descriptions of the media types, codecs, transport protocols and other parameters involved in establishing and managing IMS sessions. In the context of IMS Data Channels, SDP plays a crucial role in enabling precise media negotiation, ensuring interoperability and providing flexibility across different applications. This allows for the adaptation of session parameters in real time to support varying network conditions and application-specific requirements.
  - **Media and Channel Parameter Negotiation**: SDP facilitates the negotiation of channel-specific parameters, including transport protocols (such as SCTP over DTLS), reliability configurations and Quality of Service (QoS) requirements. This flexibility is fundamental for IMS Data Channels, as it allows applications to choose configurations that best suit their specific needs. For instance, data channels can be configured for full reliability, making them ideal for applications that require error-free data transfer (e.g., file transfers). Alternatively, for real-time applications like AR, data channels can be set up

with partial reliability to prioritize low latency over complete data accuracy, supporting responsive and interactive experiences.

- **Codec and Application Layer Configuration**: SDP enables devices to negotiate codecs that are optimized for IMS applications, as outlined in 3GPP TS 26.264, which includes Enhanced Voice Services (EVS) tailored for AR and XR experiences. By allowing devices to dynamically adjust codec settings—such as bitrate, resolution and compression level—SDP ensures that media quality is maintained while meeting the application's latency and bandwidth requirements. This capability is especially valuable for immersive applications like AR and XR, where high-quality, low-latency audio and video are essential to user experience.
- **Security Parameters**: SDP also supports the negotiation of security-related parameters, including encryption methods such as DTLS (Datagram Transport Layer Security). This negotiation is critical to ensure that both endpoints agree on the security configurations before data transmission begins, establishing an encrypted and authenticated channel. Such a security layer is indispensable for applications handling sensitive data, as it provides end-to-end encryption to protect against unauthorized interception and access. By enabling these security negotiations, SDP contributes to the overall integrity and confidentiality of IMS Data Channels, making them suitable for applications requiring robust data protection.

IMS Data Channel Setup

The setup of an IMS Data Channel follows a structured procedure that ensures secure and efficient session establishment, resource allocation and application delivery. This section outlines the key steps and entities involved in the setup process.

3.3.1 Session Initiation and Configuration

The UE may initiate an IMS Data Channel concurrently with an IMS audio/video session or upgrade an existing session through a re-INVITE message. While simultaneous setup allows enhanced multimedia interaction from the start, it may slightly increase the initial session setup time.

The Home Public Land Mobile Network (HPLMN) can configure the UE via device management mechanisms or UICC (Universal Integrated Circuit Card) to define whether and when to request a Data Channel. If no explicit configuration is provided by the network, the UE may autonomously decide whether to initiate a Data Channel request, as long as IMS Data Channel support was confirmed by the network during the registration process.

If the UE is not authorized for IMS Data Channels, the IMS Application Server discards the data channel request, allowing only the audio/video session to proceed.

3.3.2 Bootstrap Process Overview

The bootstrap phase is essential for establishing a control channel that facilitates application download and configuration. The overall steps are:

1. **Session Request.** The UE sends a SIP INVITE containing an SDP offer that specifies intent to establish a data channel.
2. **Authorization.** The IMS AS validates the request by querying the Home Subscriber Server (HSS) to verify subscription status and policy compliance.
3. **Resource Allocation.** The IMS AS interacts with the Data Channel Signaling Function (DCSF), which coordinates with the Policy and Charging Rules Function (PCRF) via the P-CSCF to reserve necessary resources.
4. **Application Provisioning.**
    - The Data Channel Application Repository (DCAR) stores application packages (e.g., HTML/JavaScript).
    - The Data Channel Application Server (DCAS) retrieves and prepares the application content for delivery.
5. **Media Function Configuration.** The DCSF instructs the IMS AS to engage either the Media Function (MF) or Media Resource Function (MRF), which acts as a proxy for application delivery via HTTP over data channels.

6. **Stream ID and URL Assignment.** The DCSF assigns a unique Stream ID and a session-specific URL, communicated back through the IMS AS to the MF/MRF.

7. **Application Request.** The UE sends an HTTP GET over the data channel to retrieve application content. The MF/MRF intercepts, modifies and forwards the request to the DCAS.

8. **Content Delivery and Session Finalization.** The DCAS responds with the application resources. Once downloaded, the IMS AS finalizes the setup, enabling full functionality of the IMS Data Channel.

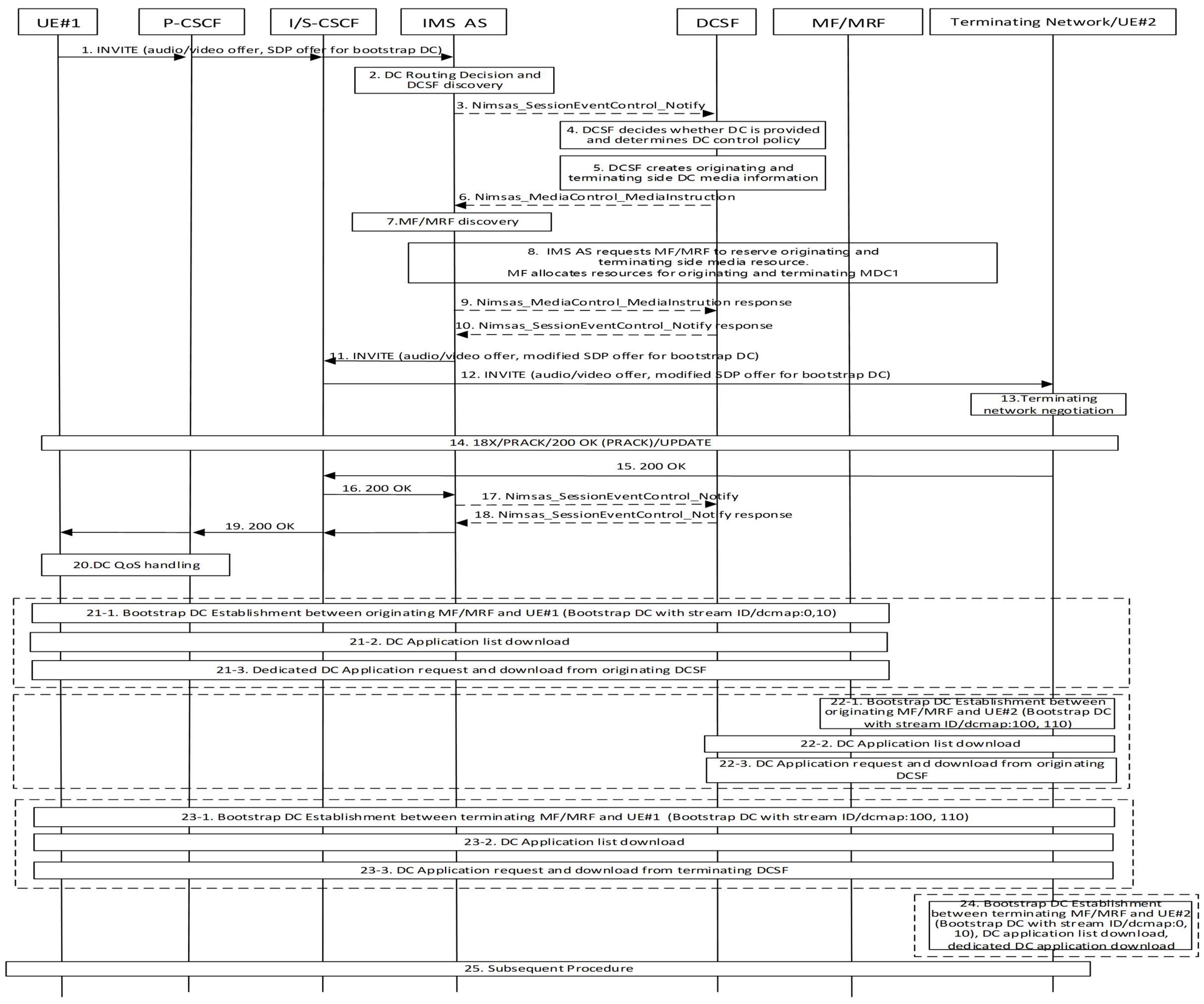

UE#1
P-CSCF
I/S-CSCF
IMS AS
DCSF
MF/MRF
Terminating Network/UE#2
1. INVITE (audio/video offer, SDP offer for bootstrap DC)
2. DC Routing Decision and DCSF discovery
3. Nimsas_SessionEventControl_Notify
4. DCSF decides whether DC is provided and determines DC control policy
5. DCSF creates originating and terminating side DC media information
6. Nimsas_MediaControl_MediaInstruction
7.MF/MRF discovery
8. IMS AS requests MF/MRF to reserve originating and terminating side media resource. MF allocates resources for originating and terminating MDC1
9. Nimsas_MediaControl_MediaInstrution response
10. Nimsas_SessionEventControl_Notify response
11. INVITE (audio/video offer, modified SDP offer for bootstrap DC)
12. INVITE (audio/video offer, modified SDP offer for bootstrap DC)
13.Terminating network negotiation
14. 18X/PRACK/200 OK (PRACK)/UPDATE
15. 200 OK
16. 200 OK
17. Nimsas_SessionEventControl_Notify
18. Nimsas_SessionEventControl_Notify response
19. 200 OK
20.DC QoS handling
21-1. Bootstrap DC Establishment between originating MF/MRF and UE#1 (Bootstrap DC with stream ID/dcmap:0,10)
21-2. DC Application list download
21-3. Dedicated DC Application request and download from originating DCSF
22-1. Bootstrap DC Establishment between originating MF/MRF and UE#2 (Bootstrap DC with stream ID/dcmap:100, 110)
22-2. DC Application list download
22-3. DC Application request and download from originating DCSF
23-1. Bootstrap DC Establishment between terminating MF/MRF and UE#1 (Bootstrap DC with stream ID/dcmap:100, 110)
23-2. DC Application list download
23-3. DC Application request and download from terminating DCSF
24. Bootstrap DC Establishment between terminating MF/MRF and UE#2 (Bootstrap DC with stream ID/dcmap:0, 10), DC application list download, dedicated DC application download
25. Subsequent Procedure

Figure 6: Bootstrap Data Channel setup signaling procedure (TS 23.228)

3.3.3 Client and Application Support

A DCMTSI client is an MTSI-capable UE that supports IMS Data Channels. It signals its capability using the +sip.app-subtype media feature tag set to webrtc-datachannel, as per RFC 5688.

Applications retrieved over the bootstrap data channel are delivered securely through MF/MRF intermediaries. The process includes:

- Content hosted in DCAR.
- Policy enforcement and retrieval via DCAS.
- HTTP session managed through MF/MRF acting as intermediary proxy.

3.3.4 SDP and ICE Considerations

SDP is central to session negotiation, enabling the configuration of multiple data channels within an IMS session. Each data channel is associated with a unique Stream ID and mapped through specific SDP attributes.

Multiple data channels must respect a global session bandwidth limit. Each channel is mapped via SDP with unique parameters and may require individual media descriptions if different transport parameters are used.

To simplify NAT traversal, DCMTSI clients typically use ICE Lite, which relies on local (host) candidates. In IMS controlled networks, full ICE/STUN/TURN mechanisms are generally unnecessary.

Stream IDs below 1000 are reserved for bootstrap purposes. The DCMTSI client uses HTTP over data channels to load GUI or application logic. Host information in headers is omitted to ensure compatibility with IMS routing policies.

If unintended HTTP requests are received on a bootstrap channel, the client removes the associated SDP mapping, closing the stream to prevent unauthorized use.

3.3.5 Device Architecture Overview

The figure below illustrates the architecture of a device designed to interact with IMS Data Channels.

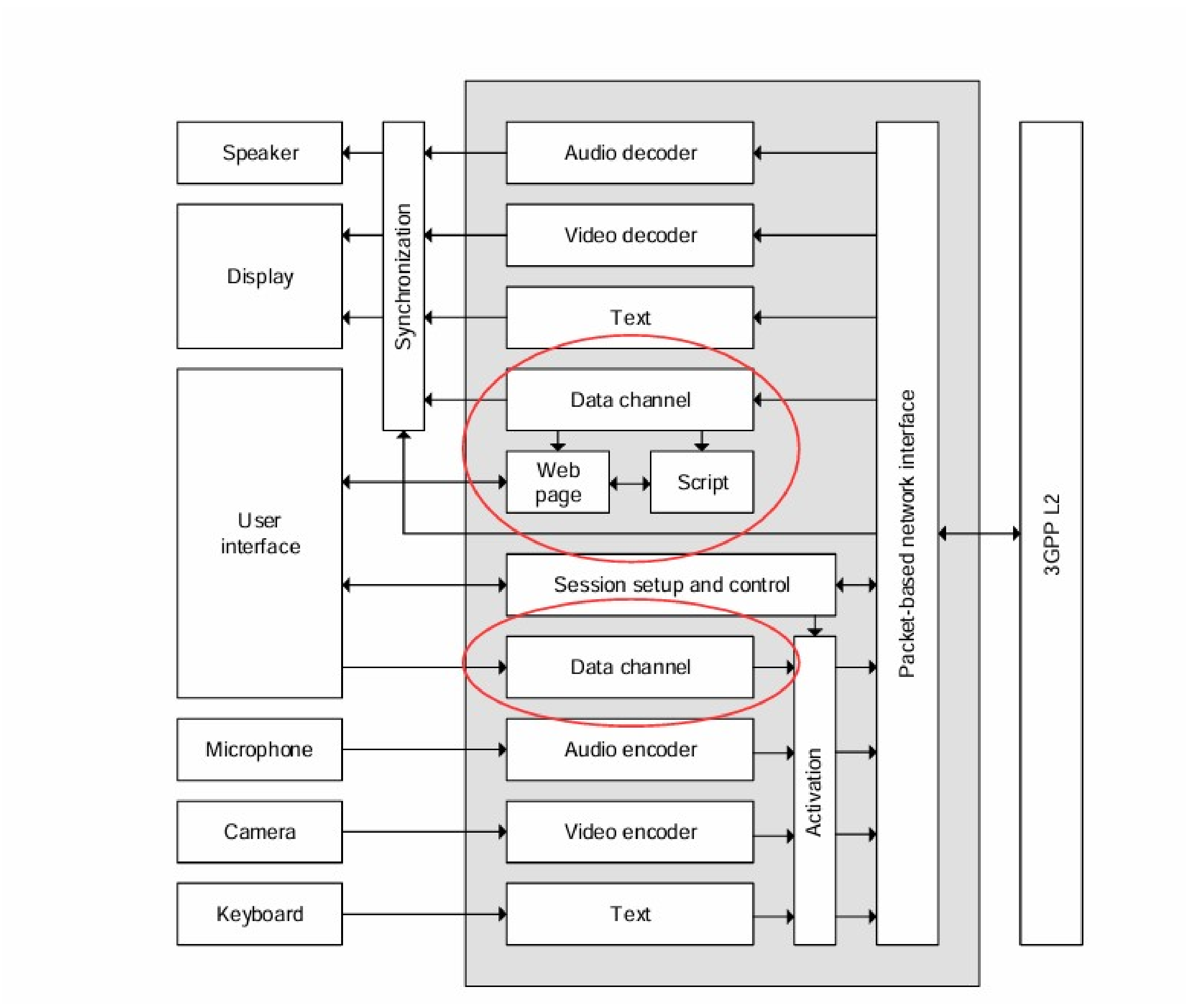


Figure 7: Device components supporting IMS Data Channels

3.3.6 Application Workflow and Interaction in Data Channels

A data channel application workflow involves the following steps, detailed in Figure 8:

1. **Application Storage and Upload**: Applications are uploaded to the network by authorized parties and stored in a repository (DCAR) within the IMS network.
2. **Application Retrieval**: During the session, the application is retrieved from the repository DCAR by the DCAS. The DCAS manages updates and interactive functionalities.
3. **Application Delivery**: Content is sent through bootstrap data channels to the local UE A and, in parallel, to the remote UE B. Additional data channels can be established for direct communication between UEs A and B. Data transmission starts only after mutual instantiation confirmation between peers.

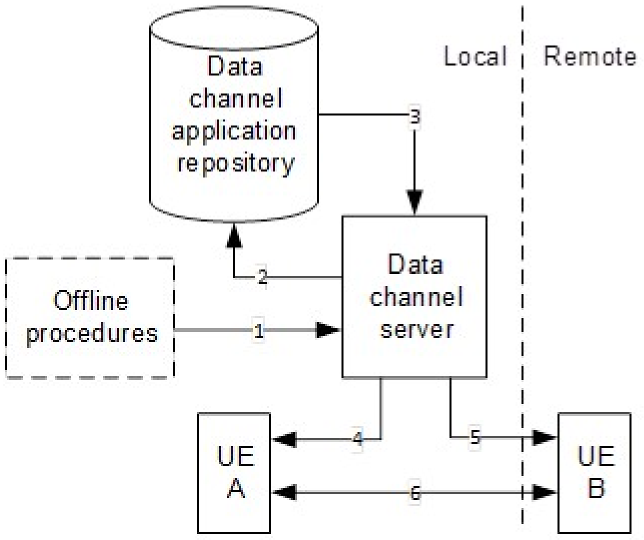


Figure 8: Data Channel Workflow

Applications retrieved in this manner are securely stored, associated with specific session requirements and mapped to unique Stream IDs for interaction. The interaction is dynamic, enabling multi-source application handling as shown in Table 3.1.

Stream ID	Content Source
0	Local network provider
10	Local user
100	Remote network provider
110	Remote user

Table 3.1: Bootstrap Data Channel Content Sources - TS26.114

Data channel providers include the local UE user, authorized network parties, remote users and remote network providers. Each stream ID corresponds to a unique content source and is listed with the a=dcmap attribute.

Application-Specific SDP Attributes

Two specific attributes are defined to support application data channels:

- `a=3gpp-bdc-used-by`: Specifies the party (either ”sender” or ”receiver”) using the bootstrap data channel. This attribute helps distinguish bootstrap channel descriptions in SDP offers and answers, ensuring proper session termination for each party.
- `a=3gpp-req-app`: Used when adding application data channels, this attribute specifies the requesting application via a req-app-id. The attribute may include an adc-stream-id-endpoint parameter to differentiate endpoints for server or UE communication. This configuration allows applications to manage data channels specifically associated with unique identifiers.

Session Control and QoS Handling in SDP

IMS sessions using data channels adhere to the SDP offer-answer model from RFC 3264, allowing for the dynamic addition or removal of media components without affecting ongoing media lines. In sessions with multiple media types, the a=3gpp-qos-hint attribute facilitates QoS management for application data channels. Adjustments can be made based on strict or relaxed latency requirements for data flows and if necessary, modifications can be applied to meet changing conditions.

Receiving SDP Answer and Channel Setup

When a DCMTSI client receives an SDP answer, it checks the a=dcmap attributes to see which stream IDs have been accepted for data channels. If a stream ID that was offered is missing in the answer, the corresponding data channel is considered rejected. The req-app-id parameter helps link each data channel to a specific application, allowing precise control over which data belongs to which service.

By using SDP in this way, IMS Data Channels can manage multiple types of data traffic, support application loading and maintain quality of service. This ensures that data is delivered reliably and that the system remains flexible, even in complex or high-demand IMS scenarios.

QoS and Flow Management in 4G and 5G

IMS Data Channels are designed to leverage QoS mechanisms in both 4G and 5G networks, ensuring that data transmission meets application-specific performance requirements. While both networks provide QoS, 5G offers enhanced capabilities, such as network slicing and more granular QoS flows.

- **4G LTE QoS Integration**: In 4G LTE, QoS is managed using bearers that provide various levels of service based on data requirements. IMS Data Channels can request dedicated bearers for prioritized data handling, ensuring optimal performance for high-priority traffic.
  - **Guaranteed Bit Rate (GBR) and Non-GBR Bearers**: 4G LTE supports GBR bearers for applications that need consistent bandwidth (e.g., voice calls) and non-GBR bearers for applications with flexible requirements. IMS Data Channels can use GBR bearers to ensure reliable delivery of critical data, such as interactive media content.
  - **Dynamic QoS Adjustment**: SIP signaling in IMS enables Data Channels to adapt QoS settings dynamically based on real-time network conditions. For instance, if network congestion occurs, a Data Channel for AR overlays can reduce its bit rate to maintain low latency.
- **5G QoS Framework**: The 5G network introduces advanced Quality of Service (QoS) mechanisms, specifically designed to meet the different requirements of modern applications. Key features, such as network slicing and fine-grained QoS flows, allow the network to prioritize and manage data traffic with greater flexibility and precision. IMS Data Channels benefit significantly from these features, enabling applications that require ultra-low latency and high reliability.
  - **Network Slicing**: Network slicing in 5G allows operators to create multiple, virtualized network segments (or ”slices”) on a shared physical infrastructure. Each slice can be configured with specific resources and QoS policies tailored to the needs of a particular application or service type. One key slice type in 5G is URLLC (Ultra-Reliable Low-Latency Communication), specifically designed for applications that require extremely low latency and high reliability. This slice is ideal for AR/XR applications and other real-time interactive services. IMS Data Channels can leverage URLLC slices to ensure that latency-sensitive applications maintain consistent, responsive performance. For example, in a remote assistance scenario using AR, a URLLC slice provides the necessary resources to maintain ultra-low latency and prevent delays, making the user experience more seamless and stable.
  - **QoS Flow Control**: In 5G networks, QoS flow control enables a more detailed level of management for data streams by allowing the network to assign specific priority levels and resource allocations to different types of data within a single session. Unlike the less flexible QoS handling in previous generations, 5G’s QoS flows can dynamically adjust to the needs of various data streams, providing each with the resources required for optimal performance. This capability

allows IMS Data Channels to prioritize certain data streams over others within the same session. For example, during an IMS session involving both AR data and standard messaging, the AR data stream can be assigned a higher priority to ensure timely delivery, while less time-sensitive data like text messaging can be given lower priority. For applications requiring real-time interaction, such as live AR annotations in a video call, assigning these data flows the highest QoS priority minimizes latency and maintains data integrity, ensuring that users receive high-quality service even in network-congested situations.

Chapter 4

Demonstrating potential applications of IMS Data Channels using WebRTC technology

WebRTC (Web Real-Time Communication) is an open-source technology that enables real-time audio, video and data communication directly between browsers or applications without the need for dedicated intermediary servers. It is designed to support interactive communication features such as voice calls, video calls and data exchange with low latency and high efficiency, leveraging internet protocols and standards.

Key Features of WebRTC

- **Peer-to-peer communication:** Enables direct connections between two endpoints (browsers, devices or applications) for data transfer without a centralized server.
- **Standardization:** Built on open standards defined by World Wide Web Consortium and IETF, ensuring interoperability across different browsers and devices.
- **Protocols used:**
  - **ICE (Interactive Connectivity Establishment):** For negotiating connections through firewalls and NAT.
  - **SRTP (Secure Real-Time Protocol):** Ensures secure audio/video communication.
  - **DTLS (Datagram Transport Layer Security):** Secures data communication.
- **Integrated APIs:** WebRTC provides JavaScript APIs (for the web) that allow developers to add features such as:
  - Audio and video streaming (media streams).
  - Real-time data exchange (RTCDataChannel).
  - Peer-to-peer connection management (RTCPeerConnection).
- **Common Applications:**
  - Video calls and conferences (e.g., Google Meet, Zoom).
  - Instant messaging services with file transfer.

- Real-time multiplayer online games.
- IoT applications for video streaming or remote control.

Use Case Simulation with WebRTC

Since IMS Data Channels are not yet widely available and adopted in the European markets, simulating their use cases through WebRTC makes it possible to showcase potential application scenarios that could be implemented with IMS Data Channels in the future.

4.2.1 Exploration of IMS Data Channel Potential

WebRTC offers a highly flexible environment for experimenting with features such as video call, real-time communication, text and, most importantly, the integration of advanced machine learning algorithms provided by modern cloud platforms like AWS (Amazon) and GCP (Google Cloud Platform).

These simulations allow for practical exploration of how IMS Data Channels can support and enhance such intelligent capabilities within IMS networks, enabling smarter, more adaptive services powered by cloud-based AI.

The integration of IMS Data Channels introduces significant benefits for the end user, enhancing communication experiences through more dynamic and efficient service access:

- **Advanced Voice Services:** Enhancing voice communication with real-time contextual information, translations and on-screen information, making the interaction richer and more engaging.
- **Multi-Network Accessibility:** Seamless operation across 4G, 5G and Wi-Fi networks, ensuring service continuity and consistent quality regardless of the user's network environment.
- **Guaranteed Quality of Service:** By leveraging the IMS framework, Data Channels benefit from QoS-aware resource management, ensuring reliable performance for real-time applications even under varying network conditions.
- **Simplified User Interaction:** Advanced features are automatically activated during sessions via bootstrapping mechanisms, eliminating the need for manual setup or application downloads. This approach reduces complexity and makes services more accessible and user-friendly.

This work aims to demonstrate advanced voice service use cases by simulating them through a WebRTC-based environment, integrated with cloud-based machine learning services offered by AWS. The simulation includes real-time functionalities such as speech transcription, speech-to-text translation and sentiment analysis. While IMS Data Channels are not yet widely adopted, the use cases implemented with WebRTC are designed to showcase how such services could be mapped onto the IMS architecture in the future opening new perspectives for the evolution of voice communication.

Janus WebRTC Gateway

For the simulation, the Janus WebRTC Gateway was chosen due to its strong capabilities in building and testing advanced real-time communication scenarios. Its flexible architecture and native support for WebRTC protocols made it an ideal platform to explore potential use cases and their integration with machine learning algorithms.

- Janus is specifically designed to support WebRTC protocols, which form the basis for IMS Data Channel functionality. This made it an effective tool for simulating IMS Data Channel capabilities in a controlled environment. In this project, Janus was used to implement real-time speech transcription, speech and text translation, voice and text sentiment analysis showcasing how advanced voice services with ML algorithms interactions can be delivered through data channels.
- Janus's modular architecture, with plugins like TextRoom, AudioBridge and VideoCall provided the flexibility required for my research. This design allowed me to customize and extend the Janus Gateway to suit my specific thesis requirements, such as integrating APIs and SDKs for cloud-based machine learning services.
- Janus is lightweight, making it easy to deploy and manage within my development environment. The availability of pre-built Docker images simplified the setup process.
- Janus enabled me to simulate realistic communication scenarios, such as data exchange and interactive applications, using WebRTC data channels.

While IMS Data Channels are not yet widely implemented, Janus provides a practical way to model their potential. By simulating use cases on Janus, it is possible to explore how IMS Data Channels could enhance voice communication, ensure seamless operation across different network types (such as 4G, 5G and Wi-Fi) and improve user experience by automatically loading application features during a session—without requiring manual downloads or setup.

Environment setup simulation

4.4.1 Ubuntu Environment via WSL

The entire simulation environment was managed using Ubuntu running within the Windows Subsystem for Linux (WSL). This setup provides a fully functional Linux environment directly on top of a Windows system, allowing seamless interaction with native Linux tools and Docker containers.

Ubuntu was used for several tasks in the simulation workflow:

- **Launching the Janus WebRTC Gateway:** The Janus Docker container was managed from the Ubuntu shell, using standard Docker commands to start, stop and configure the gateway and expose the necessary ports.
- **Serving the WebRTC Demo Files:** The demo files were served to the browser by running a lightweight HTTP server with the command:

  ```
  python3 -m http.server 8080
  ```

  This made the Janus WebRTC demos accessible via http://localhost:8080.
- **Running the Flask Server:** The Python script responsible for handling audio and text communication with AWS (including real-time transcription, translation and sentiment analysis) was executed directly within the Ubuntu environment, using the following command:

  ```
  ./full_videocall_rt_comprehend_final.py
  ```

  This script continuously listened for WebSocket and REST requests from the client interface and forwarded them to the relevant AWS APIs and SDKs.

Using WSL with Ubuntu provided a highly flexible and efficient environment for development and testing. It allowed the full stack including Janus, the Flask backend and the HTTP server to be executed in a Unix-like system, while still being accessible from the browser running in the host Windows environment.

4.4.2 Janus WebRTC Gateway configuration and organization

The Janus WebRTC Gateway was configured locally using Docker and the demo files were served via a local HTTP server.

The Janus WebRTC Gateway demo files, which allow testing of Janus features such as WebRTC data channels, were obtained directly from the Docker container rather than cloning the GitHub repository. These files are pre-installed in the /usr/local/share/janus/demos directory within the Janus Docker image. The following steps were taken to access and organize these files:

- The Janus Docker image was pulled from Docker Hub using:

  ```
  docker pull canyan/janus-gateway:latest
  ```

- The Docker container janus was created and launched.
- The demo files (.js, .html and .css files) were copied from the container to the local directory `/Tesi/html` for easier access:

  ```
  docker cp janus:/usr/local/share/janus/demos ~/Tesi/html
  ```

4.4.3 Setting Up the Janus WebRTC Gateway

To deploy the Janus WebRTC Gateway, the following steps were carried out:

1. Any previously running Janus containers were stopped and removed:

   ```
   docker stop janus
   docker rm janus
   ```

2. A new Janus container was started using the official Docker image:

   ```
   docker run -d --name janus \
      -p 8088:8088 \
      -p 8188:8188 \
      -p 10000-10200:10000-10200/udp \
      canyan/janus-gateway:latest
   ```

The ports used in the container serve the following purposes:

- **8088 (REST API):** Used for HTTP-based REST API interactions with the Janus Gateway for session management
- **8188 (WebSocket):** Enables WebSocket communication for plugins such as the VideoCall for real-time data exchange.
- **10000-10200/UDP (Media Traffic):** Handles audio, video and data streams exchanged during WebRTC sessions using the UDP protocol for low-latency transmission.

Note: Although multiple ports are exposed when running the Janus Gateway container, only port 8188 is directly used in this project. All signaling between the browser-based WebRTC client and the Janus server, such as session creation, plugin attachment, call setup and SDP exchange, is handled through this WebSocket port. The UDP port range 10000–10200 is used automatically by the browser's internal WebRTC engine to transmit media streams (audio, video and data). This process happens in the background without requiring manual configuration. Port 8088, which is reserved for REST API communication with Janus, is configured but not used in this simulation. Both WebRTC clients (Client A and Client B) run on the same machine, which simplifies network routing and avoids issues related to NAT traversal.

4.4.4 Serving the Demo Files Locally

Web browsers enforce strict security rules, preventing WebRTC and WebSocket-based pages from working properly when opened directly as local files. To comply with these security policies, a local HTTP server was started to serve the demo files:

```
cd ~/Tesi/html
python3 -m http.server 8080
```

This command maps the local directory /Tesi/html to the URL http://localhost:8080, making the demo files accessible via HTTP.

4.4.5 Accessing the Demo Files in the Browser

The demo files can be accessed in the browser using the local HTTP server:

- **Homepage:** http://localhost:8080
- **VideoCall demo:** http://localhost:8080/videocalltest.html

When a demo is opened in the browser:

- The browser loads HTML, JavaScript and CSS files from the local HTTP server.
- WebSocket communication with the Janus Gateway occurs via port 8188.
- WebRTC media streams are exchanged over the UDP ports 10000-10200.

This setup enables full testing of WebRTC features such as video calls, data messaging and integration with external AI services.

Use Case: Video Call with Integrated Machine Learning AWS Services

This use case demonstrates a comprehensive WebRTC-based simulation of advanced voice and text communication services, enhanced through the integration of AWS machine learning capabilities. The implementation is based on the Janus WebRTC Gateway and extends the default video call demo to support features such as real-time audio transcription, audio and text chat translation, audio and text sentiment analysis.

The setup simulates a call between two clients (from Client A to Client B) running on the same machine. In order to avoid echo and audio feedback, custom modifications in the videocalltest.js file were made to disable audio element used by Client B:

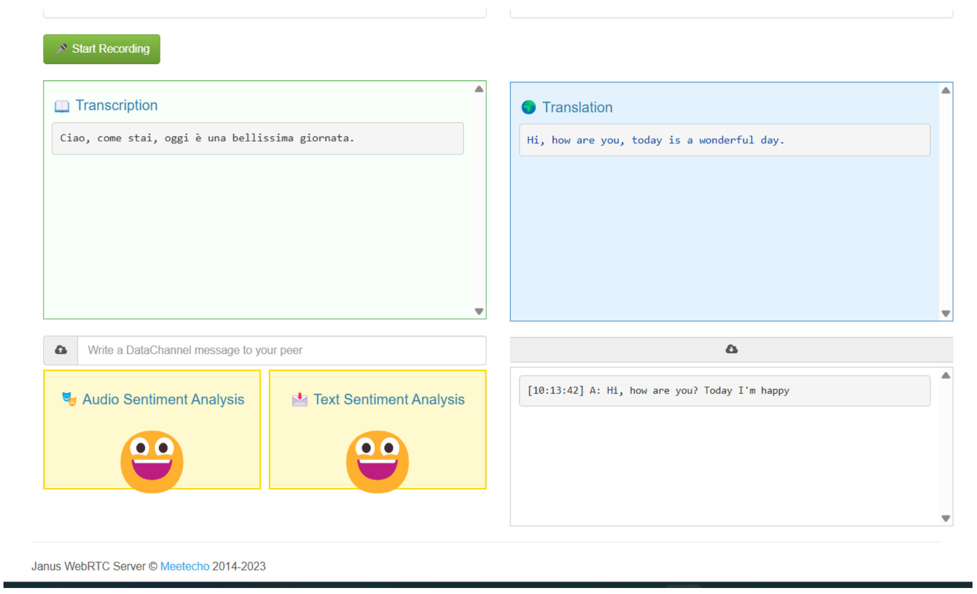
```
if (myusername === "B") {
    console.log("Fully disabling audio for Client B to avoid echo (both incoming stream and HTML element).");

    let remoteStream = videocall.webrtcStuff.pc.getReceivers().find(r => r.track && r.track.kind === "audio");
    if (remoteStream && remoteStream.track) {
        remoteStream.track.stop();
        console.log("Incoming audio track for Client B has been fully stopped.");
    }

    let audioElement = document.querySelector("#peervideo0");
    if (audioElement) {
        audioElement.muted = true;
        audioElement.volume = 0;
        audioElement.srcObject = null;
        console.log("HTML media element on Client B has been muted and detached from the stream.");
    }
}
```

Listing 4.1: Client B audio stream fully disabled to avoid echo

In addition to the backend logic, several important updates were made to the user interface in the videocalltest.html file:

- Two new buttons, Start Recording and Stop Recording, were added to the HTML layout to allow the user to control when real-time transcription begins and ends.
- A dedicated section was added to display the original transcription and its translated version side by side. These are updated live via WebSocket events from the Flask server.

- Custom visual blocks were implemented to display sentiment analysis results using emoji. The interface includes two boxes: one for audio sentiment and one for text sentiment, each updated independently.
- Additional logic was added to handle WebSocket events for partial transcriptions, final transcriptions and sentiment updates, ensuring that the interface remains responsive during the entire call.



```
Ciao, come stai, oggi è una bellissima giornata.
```

```
Hi, how are you, today is a wonderful day.
```

```
[10:13:42] A: Hi, how are you? Today I'm happy
```

Figure 9: Custom user interface of the Video Call demo with live transcription, translation and sentiment analysis

4.5.1 Implemented functionalities

The enhanced demo addresses the following key functionalities:

- Real-time speech-to-text transcription using Amazon Transcribe.
- Real-time audio and text translation from Italian to English using Amazon Translate.
- Real-time audio and text sentiment analysis using Amazon Comprehend.

4.5.2 System Components and Architecture

- **Janus WebRTC Gateway:** Used to establish video/audio/dataChannel sessions. Deployed via Docker with REST and WebSocket interfaces.
- **WebRTC Clients (A and B):** Accessed via the browser at `http://localhost:8080/videocalltest.html`, each with a unique username.
- **Flask Server:** Acts as a middleware between the WebRTC clients and AWS services. It provides a WebSocket interface for real-time audio streaming to Amazon Transcribe and a REST API for text translation and sentiment analysis using AWS Translate and Comprehend.
- **AWS Services:** Includes Amazon Transcribe (streaming) accessed via SDK for python, Translate and Comprehend APIs accessed via boto3.

4.5.3 Flask Server Implementation

The Flask server plays a central role in the simulation architecture by acting as an intelligent middleware between the WebRTC clients and the AWS machine learning services. It is responsible for handling both real-time audio processing and HTTP-based text message analysis. The Python script used in this simulation integrates Flask, Flask-SocketIO, Amazon Transcribe Streaming SDK and the AWS Boto3 for Translate and Comprehend.

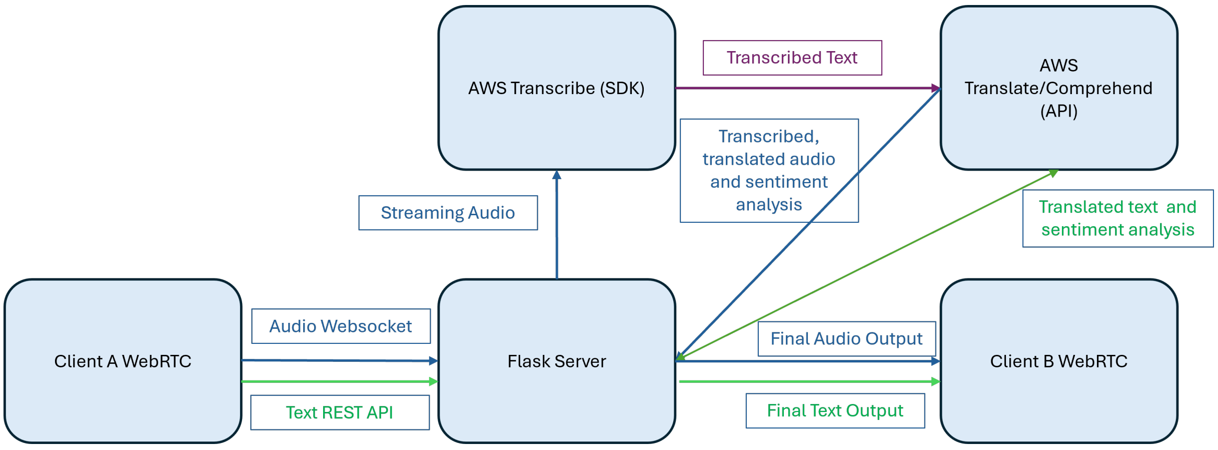


Figure 10: Flask server architecture acting as middleware between WebRTC clients and AWS services.

At the beginning of the script, the required Python modules are imported, including boto3 for accessing AWS services, asyncio for asynchronous processing and flask, flask_cors and flask_socketio for handling web and WebSocket communication. The AWS region is explicitly set to eu-central-1 and the necessary AWS clients for Transcribe, Translate and Comprehend are initialized.

A Flask application is configured to use SocketIO, enabling real-time two-way communication between clients and the server. Cross-Origin Resource Sharing (CORS) is enabled to allow browser-based WebRTC clients to communicate with the server.

Audio Transcription, Translation and Sentiment Analysis

The server implements a WebSocket event listener receive_audio(data) that is triggered whenever audio chunks are sent from the client. These chunks are buffered until they reach a duration of approximately 50 milliseconds (the minimum recommended by AWS for real-time processing) and are then inserted into a queue for asynchronous processing.

The transcription process is handled by the asynchronous function transcribe_stream(), which initiates a streaming session with Amazon Transcribe through start_stream_transcription(). Audio data is continuously read from the queue and sent to the AWS service. The server receives both partial and final transcription results, which are processed through a custom event handler MyEventHandler, derived from TranscriptResultStreamHandler.

When a partial transcription result is available:

- The text is immediately translated using Amazon Translate.
- The original and translated text are emitted via WebSocket to the client using the event partial_transcription, allowing the user interface to display real-time results.

When a final transcription result is received:

- The full sentence is stored in a buffer.
- The sentence is translated from Italian to English.
- The result is sent to the client using the event final_transcription.
- The translated text is also analyzed by Amazon Comprehend to classify the sentiment (e.g. positive, negative, neutral).
- The sentiment result is converted into a visual emoji and sent to the client using the event sentiment_result.

The transcription session is terminated either when the user clicks the "Stop Recording" button or automatically after a period of inactivity. To ensure that Amazon Transcribe closes the stream cleanly and produces final results, 10 silent audio chunks are added before termination.

Text-Based Translation and Sentiment Analysis

For chat messages sent via the text interface, the Flask server exposes a REST endpoint /analyze, which accepts HTTP POST requests with a JSON payload containing the following fields:

- `text`: the original message content.
- `source_language`: language code of the input text.
- `target_language`: desired language for translation.

The server performs the following actions:

1. Translates the message using the AWS Translate API.
2. Performs sentiment analysis on the translated message using Amazon Comprehend.
3. Sends the sentiment result back to the client as a WebSocket event named text_sentiment_result.
4. Returns a JSON response containing both the translated text and the detected sentiment.

Server Launch

The server is launched on http://0.0.0.0:5000 to accept connections from local WebRTC clients. The startup command is:

```
./full_videocall_rt_comprehend_final.py
```

This server handles both real-time and asynchronous workflows, enabling seamless integration of voice and text services with AWS machine learning APIs.

4.5.4 Client-Side Real-Time Audio Processing and WebSocket Communication

The logic for capturing, processing and transmitting audio data is entirely managed by the videocalltest.js script, which was originally provided as part of the Janus WebRTC Gateway demo and has been modified to support the functionalities required by this project.

The client-side enhancements introduced in this simulation include microphone stream acquisition, real-time audio processing using the AudioWorklet API, conversion to a 16-bit PCM format compatible with AWS Transcribe and transmission to a Flask server over WebSocket for real-time transcription, translation and sentiment analysis.

Microphone Access and Stream Initialization

When the user clicks the “Start Recording” button, the startRecording() function is triggered. This function requests access to the user’s microphone using the WebRTC API navigator.mediaDevices.getUserMedia().

The stream is configured with the following parameters:

- Sample rate: 16 kHz
- Channel count: mono
- Echo cancellation: enabled
- Noise suppression: enabled
- Automatic gain control: enabled

This configuration ensures that the input signal is optimized for speech recognition.

AudioWorklet Processing and Audio Conversion

The captured audio stream is routed through a custom AudioWorkletNode, which runs in a dedicated audio thread for efficient real-time processing. This node executes a module defined in processor.js, responsible for converting the floating-point audio samples provided by the browser into 16-bit PCM format. This format is required by Amazon Transcribe for audio streaming.

Browsers typically output audio samples as 32-bit floating-point values (Float32) in the range [-1.0, +1.0]. However, Amazon Transcribe expects audio in signed 16-bit linear PCM format (Int16), with values ranging from -32768 to +32767.

To fix this gap, the processor.js module performs the following steps for each audio frame:

1. Clips the floating-point values to ensure they remain within [-1, +1].

2. Scales each sample to the 16-bit integer range.
3. Converts the array to an Int16Array and sends it to the main thread via postMessage().

Once received by the main JavaScript thread, the PCM buffer is sent to the Flask server over WebSocket using the Socket.IO client. The Flask server processes the audio using Amazon Transcribe, Amazon Translate and returns real-time transcription/translation and sentiment results.

The following code shows the actual implementation of processor.js:

```javascript
class MyAudioProcessor extends AudioWorkletProcessor {
    process(inputs, outputs, parameters) {
        const input = inputs[0];  // First input channel
        if (input && input.length > 0) {
            const channelData = input[0]; // Mono channel assumption
            const int16Buffer = new Int16Array(channelData.length);

            // Convert audio data from Float32 to PCM 16-bit
            for (let i = 0; i < channelData.length; i++) {
                let s = Math.max(-1, Math.min(1, channelData[i])); // Clipping
                int16Buffer[i] = s < 0 ? s * 32768 : s * 32767;
            }

            // Send the buffer to the main thread as ArrayBuffer
            this.port.postMessage(int16Buffer.buffer, [int16Buffer.buffer]);
        }
        return true;
    }
}

registerProcessor('my-audio-processor', MyAudioProcessor);
```

Listing 4.2: AudioWorklet processor converting Float32 to PCM 16-bit

WebSocket Connection to Flask

A continuous WebSocket connection to the Flask server is established at page load using Socket.IO:

```
let socket = io("http://127.0.0.1:5000");
```

This connection is used to transmit small chunks of microphone audio data to the Flask server in real time:

```
socket.emit("audio", buffer.buffer);
```

WebSocket was chosen over HTTP because it supports low-latency and full-duplex communication. This means that the client can continuously stream audio to the server while simultaneously receiving transcription and translation updates in real time, which is essential for interactive, real-time applications like speech recognition and feedback.

Server Feedback Handling

As the Flask server receives and processes the audio (via AWS Transcribe, Translate and Comprehend), it emits WebSocket events back to the client interface. The client handles:

- **partial_transcription:** displaying in-progress transcriptions.
- **final_transcription:** showing the complete sentence and translation.
- **sentiment_result:** displaying an emoji that reflects the detected emotion based on key words.
- **text_sentiment_result:** used for chat messages, showing both translation and sentiment analysis.

These events update specific boxes in the user interface in real time, allowing a highly interactive experience for the user.

Chapter 5

Audio Simulation Tests and Evaluation Metrics

This section explains how the simulation tests were organized and which metrics were collected to evaluate the quality of the user experience during real-time audio communication.

The goal of these tests is to simulate different levels of degradation in a best-effort network, which represents the default behavior of the WebRTC-based environment used in this simulation and to understand how the absence of QoS mechanisms impacts overall system performance and perceived user experience.

The tests aim to explore how increased latency, jitter and packet loss affect the real-time processing chain, especially when compared to what could be expected in an IMS Data Channel environment, where QoS is guaranteed.

To make the results consistent and comparable across all tests, the same pre-recorded audio was used every time:

Ciao, come stai? Questa è una frase di test. Oggi è una bella giornata.

Because everything happens in real time, it wasn't possible to fully automate the tests. Each test was run manually and every scenario was repeated five times to get average values and reduce the impact of any unexpected fluctuations.

In best-effort networks, where data packets are forwarded without guaranteed delivery times, jitter and congestion delay are among the most frequent and impactful factors that can degrade real-time communication services.

Jitter (Variable Delay). Jitter refers to the variation in packet arrival times. In the context of this simulation, jitter is introduced as a random fluctuation of delay every 5 audio chunks, simulating real-world scenarios where packets arrive inconsistently due to:

- Dynamic routing decisions
- Buffering at intermediate network nodes

Values in the range of 10–25 ms (light jitter) and 40–100 ms (severe jitter) were selected to reflect real-world conditions in congested wireless or cellular networks, where jitter can easily exceed 50 ms without QoS enforcement.

Congestion Delay (Buffered Queuing Delay). Congestion delay occurs when packets accumulate in buffers at intermediate routers during high traffic load, causing noticeable delays before forwarding. In this simulation, fixed delays (150–400 ms and 400–600 ms) were added periodically to simulate:

- Latency spikes caused by full buffers
- Delays caused by heavy traffic on upload or download connections

These values reflect network conditions often observed in best-effort environments during peak hours or in situations where multiple users contend for limited upstream capacity.

The audio simulation tests were organized into four groups of scenarios, each focusing on a specific aspect of system behavior under different network and configuration conditions. The goal was to progressively introduce controlled degradations such as jitter, congestion and packet loss while observing their impact on latency, transcription performance and overall user experience.

- **Scenario A** – Reference tests under ideal conditions: no jitter, no packet loss and no congestion. The only parameter varied across sub-scenarios is the chunk size, in order to establish a baseline comparison across configurations:
  - **A1** – Chunk size: 50 ms
  - **A2** – Chunk size: 30 ms
  - **A3** – Chunk size: 100 ms
  - **A4** – Chunk size: 150 ms
  - **A5** – Chunk size: 200 ms

  The chunk size defines the duration (in milliseconds) of each audio segment transmitted to the transcription algorithm. Smaller chunk sizes enable lower latency and faster feedback, which are desirable in real-time communication systems. However, they also increase the number of packets sent and the associated processing overhead. Larger chunk sizes reduce packet frequency but may increase end-to-end delay and reduce interactivity.

  According to AWS Transcribe Streaming best practices, the recommended chunk size range is between 50 ms and 250 ms, with 50 ms being a common choice for real-time streaming scenarios.

- **Scenario B** – Chunk size variation combined with two levels of simulated jitter. A random delay was applied every 5 audio chunks to simulate unstable delivery typical of best-effort networks.

  To simulate jitter effects every 5 audio chunks, the following logic was added to the server Flask:

```
    # Chunk counter initialization
    if "chunk_counter" not in receive_audio.__dict__:
        receive_audio.chunk_counter = 0

    while len(audio_buffer) >= chunk_size_bytes:
        receive_audio.chunk_counter += 1

        # Simulate jitter: apply random delay every 5 chunks
        if receive_audio.chunk_counter % 5 == 0:
            jitter_delay = random.uniform(0.01, 0.025)
            time.sleep(jitter_delay)
            log_event(f"Simulated jitter delay: {int(jitter_delay * 1000)}
 ms")

        audio_queue.put(audio_buffer[:chunk_size_bytes])
        audio_buffer = audio_buffer[chunk_size_bytes:]

```

Listing 5.1: Simulated jitter delay every 5 chunks

Chunk Size	**Jitter 10–25 ms (Light)**	**Jitter 40–100 ms (High)**
50 ms	B1	B5
100 ms	B2	B6
150 ms	B3	B7
200 ms	B4	B8

- **Scenario C** – Chunk size combined with two levels of congestion delay, introduced every 5 audio chunks to simulate queue accumulation in overloaded network nodes.

  To emulate congestion in overloaded networks, the Flask server introduces fixed queuing delays every 5 audio chunks. This simulates latency spikes typically caused by buffer accumulation or transmission bottlenecks:

```
# Simulate congestion every 5 chunks
if receive_audio.chunk_counter % 5 == 0:
    congestion_delay = np.random.uniform(0.15, 0.4)
    time.sleep(congestion_delay)
    log_event(f"Simulated congestion delay: {int(congestion_delay * 1000)} ms")

audio_queue.put(audio_buffer[:chunk_size_bytes])
audio_buffer = audio_buffer[chunk_size_bytes:]

```

Listing 5.2: Simulated congestion delay every 5 chunks

Chunk Size	**Delay 150–400 ms (Light)**	**Delay 400–600 ms (Severe)**
50 ms	C1	C5
100 ms	C2	C6
150 ms	C3	C7
200 ms	C4	C8

- **Scenario D** – Chunk size combined with different levels of packet loss. Audio chunks were randomly dropped based on predefined loss probabilities to emulate unreliable delivery.

  To simulate packet loss, a random number is generated for each audio chunk. If this number is below a predefined threshold (e.g., 10%), the chunk is discarded and not sent to the transcription engine. This models real-world scenarios where network unreliability leads to missing packets.

```
    drop_probability = 0.1  # packet loss simulation
    if random.random() < drop_probability:
        receive_audio.dropped_chunks += 1
        log_event(f"DROPPED chunk #{receive_audio.chunk_counter}")
        audio_buffer = audio_buffer[chunk_size_bytes:]
        continue

```

Listing 5.3: Simulated packet loss logic for Scenario D

  - **D1** – Chunk 50 ms, ∼10% loss
  - **D2** – Chunk 100 ms, ∼10% loss
  - **D3** – Chunk 150 ms, ∼10% loss
  - **D4** – Chunk 200 ms, ∼10% loss
  - **D5** – Chunk 50 ms, ∼15% loss
  - **D6** – Chunk 200 ms, ∼15% loss
  - **D7** – Chunk 50 ms, ∼20% loss

During the tests, chunk sizes from 50 ms to 200 ms were used to explore the trade-offs between latency, responsiveness and accuracy under various conditions. Only for use case A chunk size = 30 ms has been also investigated.

Each test was repeated five times and the average values were used for comparison.

Audio Events and Collected Metrics

During each test, a series of key events were logged on the server side (Flask server), with precise timestamps recorded into CSV files. This event tracking enabled a detailed analysis of system behavior and allowed for the calculation of latency and reliability metrics under various network conditions.

The following events were captured during the real-time audio processing workflow:

- **Speech Start (first audio chunk received)** – Represents the arrival of the first audio chunk at the server and the official start of the test.
- **First Partial Transcription Received** – Indicates when AWS Transcribe produced the first partial transcription result, providing an early feedback of the system responsiveness.
- **Full Transcription Received** – Represents the reception of the complete transcribed sentence from AWS Transcribe.
- **Translation Completed** – The translated version of the sentence has been generated via AWS Translate.
- **Sentiment Analysis Completed** – AWS Comprehend has analyzed the translated sentence and produced the detected sentiment.
- **Audio Round-Trip Latency** – Measures the time between the reception of the first audio chunk and the completion of the sentiment analysis, representing the end-to-end processing delay.
- **Audio Recording Stopped by User** – Indicates the manual stopping of audio recording on the client side.
- **End of Audio Stream (last chunk sent)** – Represents the moment when all audio chunks, including final silence, have been transmitted to the server.
- **Transcription Session Ended** – Signals the final closure of the transcription, translation and sentiment processing session.

From these logged events, the following metrics were extracted and computed for each test:

- **Audio Round-Trip Time (RTT)** – Measures the time elapsed between the reception of the first audio chunk and the completion of the sentiment analysis, capturing the full end-to-end processing delay perceived by the user.
- **Time to First Partial Transcription** – Time required for AWS Transcribe to produce the first partial transcription, providing an early indication of system responsiveness.
- **Number of Chunks Sent and Dropped** – Allows evaluating system robustness under packet loss conditions and correlating delivery reliability with transcription accuracy.

Scenario A: Impact of Chunk Size under Ideal Conditions

Scenario A was designed to serve as a baseline reference for all subsequent experimental scenarios. No artificial network degradations such as jitter, packet loss or congestion delay were introduced. This allowed an analysis of how the system behaves when only the chunk size parameter is varied under ideal transmission conditions.

In real-time audio streaming, the chunk size defines how much audio is collected before being packetized and sent for processing. Smaller chunks enable faster initial feedback but increase packetization overhead and system workload. On the other hand, larger chunks reduce packet frequency but introduce additional buffering delays, impacting interactivity.

In this scenario, five different chunk sizes were tested: 30 ms, 50 ms, 100 ms, 150 ms, and 200 ms considering that for the Streaming Transcription AWS suggests a chunk size between 50 ms and 200 ms. In addition to this range also chunk size = 30 ms has been tested. Two critical metrics were monitored during each test:

- **First Partial (ms)** – Time elapsed between the reception of the first audio chunk and the reception of the first partial transcription from AWS Transcribe. It represents the initial responsiveness of the system.
- **Audio Round-Trip Time (RTT) (ms)** – Time from the reception of the first audio chunk to the arrival of the final sentiment analysis result after transcription, translation and sentiment classification. It captures the full end-to-end latency perceived by the user.

The results, shown in Figure 11, show important trends:

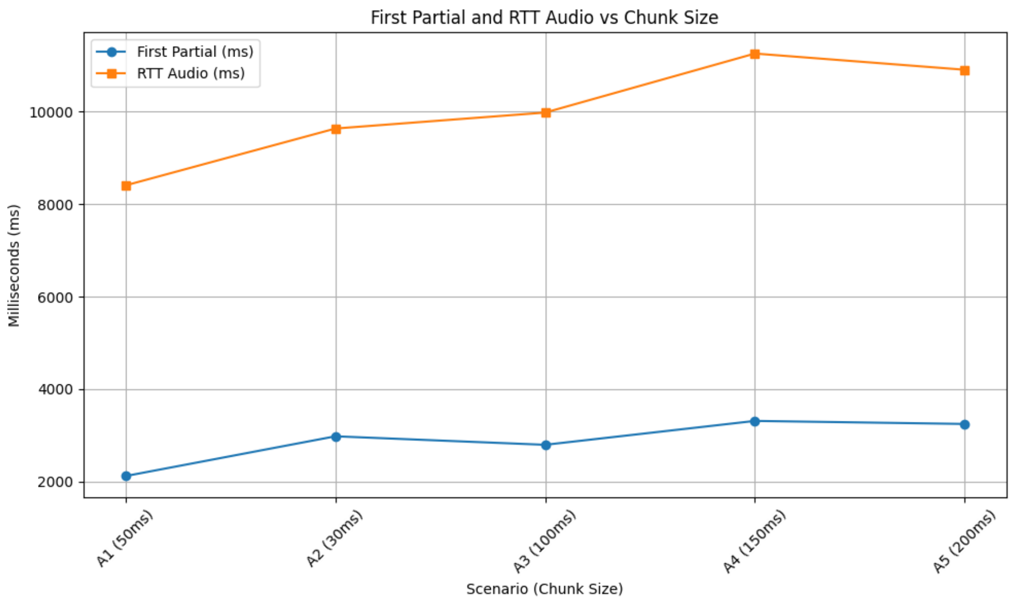


Figure 11: First Partial and RTT Audio as a function of Chunk Size in Scenario A (Ideal Conditions)

5.2.1 Impact on First Partial Latency

The first partial latency is highly sensitive to the chunk size. With a 50 ms chunk size, the system achieved the lowest first partial latency, delivering initial transcription feedback and consequently initial translation rapidly. Reducing the chunk size to 30 ms unexpectedly led to worse latency instead of an improvement.

This degradation is explained by the overhead introduced by very small chunks: sending too many packets in quick succession puts excessive load on both the local system and the cloud services. Each packet transmission triggers processing overhead (WebSocket framing, network I/O, buffer management) and increases scheduling pressure on the transcription engine. AWS Transcribe is optimized for chunk sizes between 50 ms and 200 ms and operating outside this range (as with 30 ms) results in inefficiencies and degraded performance.

Increasing the chunk size beyond 50 ms (to 100 ms, 150 ms, and 200 ms) progressively worsened the first partial latency. Larger chunks mean the system needs to accumulate more audio before transmitting anything, introducing an initial buffering delay. Therefore, the first partial result is delayed simply because the first audio packet is sent later.

5.2.2 Impact on Audio Round-Trip Time (RTT)

The overall Audio RTT showed a similar trend, but with a few differences:

- As the chunk size increased from 50 ms to 150 ms, the RTT also increased significantly, showing that larger chunks not only delay the start of transcription but also propagate delays throughout the full processing chain.
- With a chunk size of 200 ms, a slight decrease in RTT was observed compared to 150 ms, but the performance remained worse than at 50 ms.

This behavior can be explained by considering that larger chunks cause initial buffering delays, but may also reduce the number of packets sent overall, slightly mitigating system load at extreme chunk sizes like 200 ms. Nevertheless, the overall user-perceived responsiveness remains lower compared to the 50 ms configuration.

5.2.3 Conclusions on Chunk Size Optimization

The combination of overhead management and buffering delay explains why a chunk size of 50 ms proves to be the best choice under ideal network conditions in this simulation environment. It minimizes both the time needed to receive the first partial transcription, translation and the total round-trip latency, resulting in the best overall user experience.

On the other hand, the 30 ms configuration, although initially intended to improve responsiveness, actually degraded performance. This happened because it operated outside AWS's recommended range and introduced excessive packetization overhead, making the system less efficient.

These conclusions will serve as a reference for evaluating the system behavior under degraded network conditions, as analyzed in the following scenarios.

Scenario B: Jitter Impact on Latency and System Stability

The results shown in Figure 12 illustrate how the introduction of jitter in a best-effort network affects the performance of real-time audio communication. Both light jitter (10–25 ms) and high jitter (40–100 ms) were simulated under different chunk size configurations and the corresponding First Partial Latency and Audio Round-Trip Time (RTT) were measured.

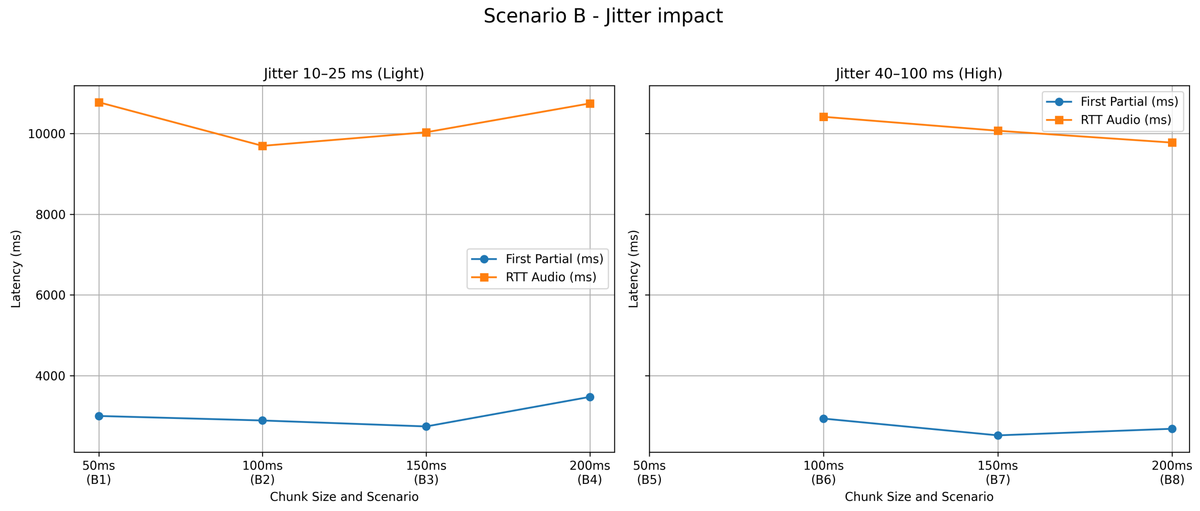


Figure 12: Scenario B – Latency vs Chunk Size under Light and High Jitter Conditions

5.3.1 Impact of Light Jitter (10–25 ms)

When a moderate jitter level was introduced, the system remained relatively robust and stable. In particular:

- First Partial Latency showed a slight decreasing trend as the chunk size increased from 50 ms to 150 ms, suggesting that larger chunks help absorb minor delay variations, making the transmission more stable toward AWS Transcribe.
- However, when reaching 200 ms, both the First Partial Latency and the RTT Audio increased again. This is likely because the buffering delays introduced by the larger chunks became more significant than the benefits gained from absorbing jitter.
- In terms of accuracy, the system performed very well under light jitter: all tests achieved 100% success except for chunk size 100 ms (B2), which showed an 80% success rate.

These results indicate that minor delay fluctuations typical of moderately loaded best-effort networks can be tolerated without major degradation in service quality, especially with appropriate chunk size tuning.

5.3.2 Impact of High Jitter (40–100 ms)

When a much higher jitter range was introduced, both latency performance and accuracy were negatively affected:

- First Partial Latency remained consistently higher across all chunk sizes compared to the light jitter case. This indicates that severe delay variability compromises the system's ability to provide rapid initial transcription feedback.
- RTT Audio also showed a general increase, confirming that significant jitter affects not only the start of the processing chain but also the full round-trip experience, from transcription to sentiment analysis.
- Chunk sizes between 100 ms and 150 ms provided a slightly better performance than 200 ms even under high jitter conditions. They were large enough to absorb most delay variations without introducing the excessive buffering delays seen with 200 ms chunks, thus preserving better responsiveness.
- Accuracy also dropped under these conditions: chunk sizes of 150 ms (B7) and 200 ms (B8) achieved only 80% success rates, compared to the 100% success rate observed under lighter jitter.

5.3.3 Critical Failure Scenario: Chunk Size 50 ms with High Jitter (use case B5)

A particularly critical failure was observed for the combination of small chunk size and high jitter. In Scenario B5 (chunk size 50 ms with high jitter), the system consistently failed: the accuracy was 0% across all tests. AWS Transcribe was unable to reconstruct a coherent transcription stream. As a result, no meaningful RTT Audio measurements were obtained, and all attempts were classified as failures (NOK). Furthermore, in every test, AWS truncated the audio early, closing the transcription session before the full sentence was completed. Consequently, no complete transcription, no translation and no sentiment analysis results were produced.

This behavior led to a severely degraded user experience, highlighting how fragile real-time services can become under extreme jitter when packet sizes are too small and Quality of Service (QoS) mechanisms are absent, as typical in degraded best-effort networks.

5.3.4 Best Configurations under Jitter Conditions

Based on the collected data:

- Under **light jitter** (10–25 ms), chunk sizes between 100 ms and 150 ms proved to be the most effective, providing a good balance between responsiveness and resilience against moderate network fluctuations.

- Under **high jitter** (40–100 ms), a chunk size of 100 ms delivered the best robustness, achieving 100% success across all tests. However, this improvement in reliability came at the cost of increased latency: both the First Partial Latency and the Audio Round-Trip Time (RTT) were higher compared to higher chunk sizes. Therefore, in scenarios where reliability is prioritized over ultra-low latency, 100 ms can be considered the optimal setting.
- Still under **high jitter** conditions, larger chunk sizes such as 150 ms and 200 ms led to a slight drop in accuracy (down to 80%), but latency actually improved compared to 100 ms. This suggests that while increasing the chunk size beyond 100 ms reduces the sensitivity to jitter and improves delay, it may also introduce synchronization issues or partial packet loss that slightly affect the reliability of full transcription and translation.

Operating with very small chunks (such as 50 ms) under high jitter is strongly discouraged unless additional network controls like jitter buffers or prioritization are available.

Scenario C: Impact of Congestion Delay on Latency and System Stability

The results shown in Figure 13 illustrate the impact of simulated congestion delays on real-time audio communication performance. Both moderate congestion (150–400 ms) and severe congestion (400–600 ms) were analyzed, evaluating First Partial Latency, Audio Round-Trip Time (RTT) and system accuracy across different chunk sizes.

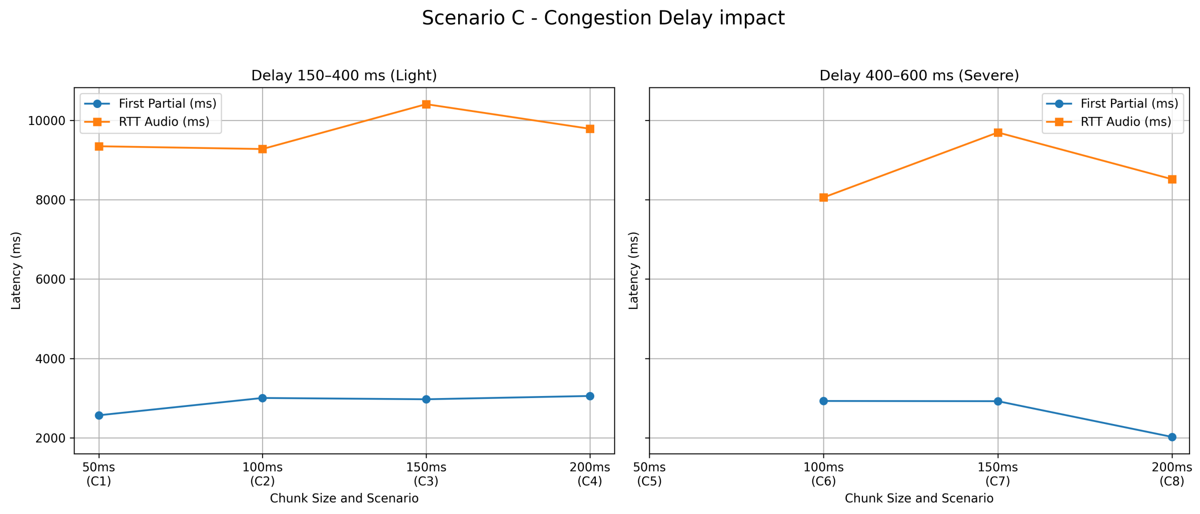


Figure 13: Scenario C – Latency vs Chunk Size under Simulated Congestion Delay Conditions

5.4.1 Impact of Moderate Congestion Delay (150–400 ms)

Under moderate congestion conditions:

- **First Partial Latency** exhibited a slight increase moving from a chunk size of 50 ms to 100 ms, followed by a stable trend across 100–200 ms. This suggests that the system can tolerate moderate delays without major penalties on initial feedback timing.
- **RTT Audio** showed a bell-shaped curve: it was lowest at 100 ms, peaked at 150 ms and then decreased at 200 ms. This behavior indicates that intermediate chunk sizes (100–150 ms) balance buffering and delay accumulation, while very large chunks (200 ms) start to reduce the overall transmission overhead.
- **Accuracy** remained generally high:
  - C1: 100% OK
  - C2: 60% OK
  - C3: 80% OK
  - C4: 100% OK

Only in C2 a notable degradation was observed, possibly caused by sporadic delivery issues under congestion.

These results demonstrate that moderate congestion levels can still be effectively managed by properly tuning the chunk size.

5.4.2 Impact of Severe Congestion Delay (400–600 ms)

With increased congestion:

- **First Partial Latency** remained relatively constant between 100 ms and 150 ms, but significantly improved when increasing the chunk size to 200 ms, demonstrating the effectiveness of larger buffering in stabilizing the initial feedback.
- **RTT Audio** was highest at 100 ms, slightly decreased at 150 ms and further improved at 200 ms. This confirms that smaller chunks become highly sensitive to severe congestion, while larger chunks mitigate the effect of burst delays.
- **Accuracy** was more heavily impacted:
  - C5: 0% OK (AWS closed sessions prematurely, resulting in dropped sentences)
  - C6: 60% OK
  - C7: 100% OK
  - C8: 80% OK

  Scenario C5 represents a catastrophic case where the combination of small chunk size (50 ms) and high congestion delay led to total session failures.

5.4.3 Critical Observations and Best Practices

Based on the analysis:

- Under **moderate congestion** (150–400 ms), chunk sizes between 100 ms and 150 ms offer the best trade-off between latency and accuracy, maintaining a good level of service even with minor fluctuations.
- Under **severe congestion** (400–600 ms), larger chunk sizes, particularly 200 ms, provide the most robust performance by reducing the sensitivity to packet delivery variations, despite slightly increasing the buffering delay.
- Small chunk sizes (50 ms) under severe congestion should be strictly avoided, as they resulted in complete session failures due to excessive fragmentation and packet arrival disorder.
- In highly degraded network scenarios, prioritizing reliability over ultra-low latency becomes crucial, and adjusting the chunk size is an effective strategy to maintain service quality.

Scenario D: Impact of Packet Loss on Accuracy and System Stability

The results shown in Figure 14 illustrate how different levels of packet loss affect the system's ability to correctly process real-time audio communication. Unlike previous tests focused on latency, this evaluation prioritized system accuracy and successful transcription and translation since packet loss introduces semantic distortions, word drops or even early session closures.

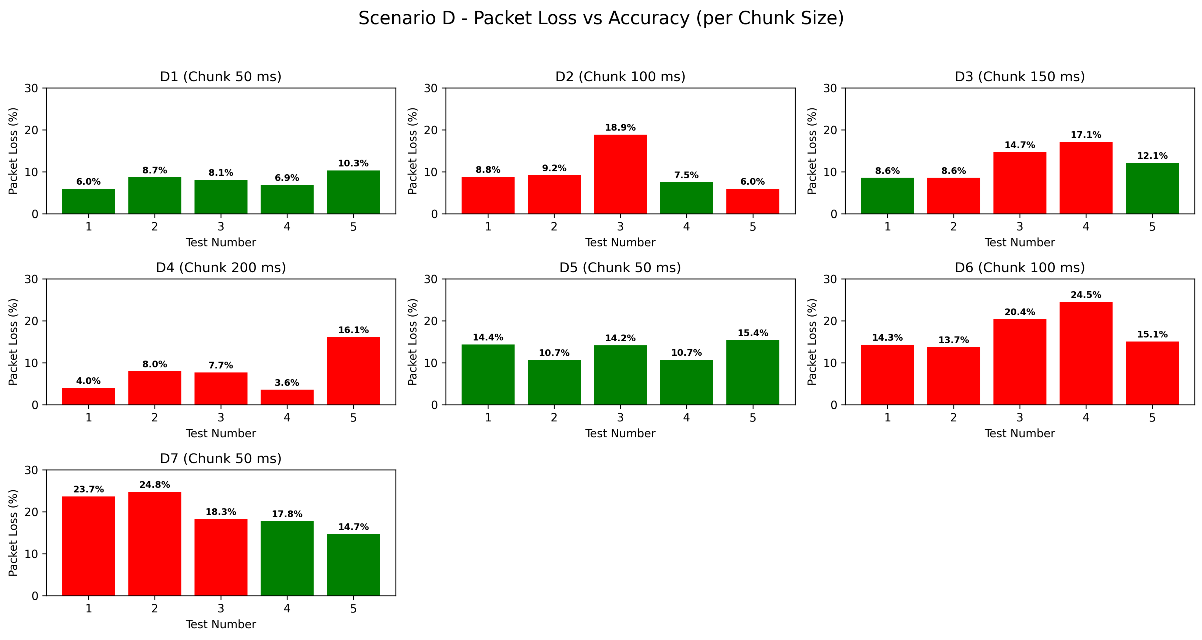


Figure 14: Scenario D – Packet Loss vs Accuracy (per Chunk Size)

5.5.1 Detailed Results per Scenario

- **D1 (Chunk 50 ms)**:
  All tests were successful (100% OK) despite packet loss values ranging between 6% and 10.3%. This configuration demonstrated excellent robustness under moderate packet loss conditions.

- **D2 (Chunk 100 ms)**:
  Tests 1, 2, 3, and 5 resulted in NOK outcomes, even with relatively low packet loss (starting from 6%). Only Test 4 (packet loss 7.5%) was successful. This indicates that D2 is highly sensitive to even low packet loss.

- **D3 (Chunk 150 ms)**:
  Tests 2, 3, and 4 failed (NOK) already at packet loss values around 8.6%. Interestingly, Test 1 was successful despite having a packet loss similar to that of Test 2 and Test 5 also succeeded even though it showed a higher packet loss than Test 2. This suggests that, under this configuration, the system's behavior was somewhat inconsistent, with moderate packet loss generally degrading reliability, but with occasional successful deliveries even at comparable or slightly worse conditions.

- **D4 (Chunk 200 ms)**:
  All tests (1 to 5) failed (NOK) despite very low packet loss levels, starting as low as 3.6%. This confirms that very large chunk sizes cannot compensate for packet loss and tend to worsen the quality of transcription and translation processes.

- **D5 (Chunk 50 ms)**:
  All tests were successful (100% OK) even with packet loss values up to 15.4%. This configuration proved to be very resilient to moderate levels of packet loss.

- **D6 (Chunk 100 ms)**:
  All tests failed (100% NOK) with packet loss levels ranging from 13.7% to 24.5%. This indicates that when the network degradation becomes severe, chunk sizes around 100 ms are not sufficient to ensure robustness.

- **D7 (Chunk 50 ms)**:
  Tests 1, 2, and 3 failed (NOK) under packet loss between 18.3% and 24.8%. Tests 4 and 5 were still successful with lower packet loss (around 14.7%–17.8%). This suggests that small chunk sizes (50 ms) can tolerate moderate packet loss but are vulnerable beyond approximately 18%.

5.5.2 Critical Observations and Best Practices

- Chunk sizes of 50 ms (as in D1 and D5) demonstrated the best overall resilience against packet loss up to 15%.
- Intermediate chunk sizes (100 ms and 150 ms) showed a higher sensitivity to packet loss, already degrading at levels as low as 6–8%.
- Larger chunk sizes (200 ms) proved ineffective in mitigating packet loss: even minimal losses resulted in failures.
- High packet loss rates (above 15%) severely impact the system regardless of chunk size, confirming that above this threshold additional recovery mechanisms would be necessary.
- In scenarios with unstable networks and packet loss above 10–15%, maintaining smaller chunk sizes (e.g. 50 ms) appears to be the most effective strategy to preserve service quality. Smaller chunks have the intrinsic advantage that, if a packet is lost, only a limited portion of the audio is affected. This allows AWS Transcribe to still reconstruct a coherent transcription. Conversely, losing larger chunks (e.g. 200 ms) results in the loss of more significant portions of speech, making recovery much more difficult and leading to severe degradation in transcription and translation quality.

Chapter 6

Text Simulation Tests and Evaluation Metrics

This section explains how the simulation tests were organized and which metrics were collected to evaluate the quality of the user experience during real-time text communication.

The goal of these tests is to simulate different levels of degradation in a best-effort network, which represents the default behavior of the WebRTC-based environment used in this simulation and to understand how the absence of Quality of Service (QoS) mechanisms impacts overall system performance and perceived user experience.

The tests aim to explore how increased latency and packet loss affect the real-time processing chain, especially when compared to what could be expected in an IMS Data Channel environment, where QoS is guaranteed.

To make the results consistent and comparable across all tests, the same text message was used every time:

Ciao, come stai? Oggi è una bella giornata. Questo è un messaggio di test che contiene diverse parole per simulare una normale comunicazione via chat.

The text simulation tests were organized into three groups of scenarios, each focusing on a specific aspect of system behavior under different network and configuration conditions:

- **Scenario A** – Reference tests under ideal conditions: no artificial delays and no packet loss.
- **Scenario B** – Tests with the introduction of fixed artificial delays to simulate network latency. The following fixed delays were applied:
  - **B1** – 200 ms delay
  - **B2** – 300 ms delay
  - **B3** – 400 ms delay
  - **B4** – 500 ms delay
  - **B5** – 600 ms delay
- **Scenario C** – Tests with artificial packet loss randomly applied to words within the transmitted message, with loss percentages varying between 4% and 28%.

For consistency, each test in Scenarios A and B (B1 to B5) was repeated five times, while Scenario C tests were repeated eight times to ensure an evaluation under different packet loss conditions.

Text Events and Collected Metrics

During each test, a series of key events were logged on the server side (Flask server), with precise timestamps recorded into CSV files. This event tracking enabled a detailed analysis of system behavior and allowed for the calculation of latency and reliability metrics under various network conditions.

The following events were captured during the real-time text processing workflow:

- **Text Message Received from Client** – Shows the reception of a text message from the client, marking the start of the processing sequence.
- **Text Start Translation** – Marks the beginning of the translation process via AWS Translate.
- **Text Translation Completed** – Represents the successful completion of the text translation.
- **Text Start Sentiment Analysis** – Marks the beginning of the sentiment analysis using AWS Comprehend.
- **Text Sentiment Analysis Completed** – Shows the successful detection of the sentiment of the translated text.
- **Text Round-Trip Latency** – Measures the time elapsed between the reception of the original text and the completion of sentiment analysis, capturing the end-to-end processing delay perceived by the user.

From these logged events, the following metrics were extracted and computed for each test:

- **Text Translation Time** – Measures the time taken to translate the received text message.
- **Text Sentiment Analysis Time** – Measures the time taken to complete the sentiment analysis after translation.
- **Text Round-Trip Time (RTT)** – Measures the total elapsed time between the reception of the text and the availability of the sentiment analysis result.
- **Number of Words Lost (only in Scenario C)** – In Scenario C, additional metrics were collected to evaluate the degradation caused by artificial packet loss, including the number of words lost and the corresponding percentage of loss.

Scenario A and Scenario B: Effect of Fixed Artificial Delays

In this set of experiments, the impact of artificial fixed delays on text communication was evaluated.

The main objective was to analyze whether increased delay could impact the real-time usability and responsiveness of the system without causing semantic or accuracy issues.

The results, summarized in Figure 15, show that:

- The overall Round Trip Time (RTT) shows fluctuations rather than a purely linear increase with the artificial fixed delay. A significant latency spike is observed at 300 ms.
- Translation time is generally stable, but a noticeable peak at 300 ms suggests sensitivity to specific network or processing conditions, rather than a smooth delay propagation.
- Sentiment analysis time remains comparatively stable, showing minor variations across different delay scenarios.
- This behavior indicates that while artificial delay impacts RTT, the internal processing times for translation and sentiment analysis are influenced also by other factors such as processing load and API internal buffering.
- No errors or semantic distortions were observed in any of the tests, confirming that while delay degrades responsiveness, it does not affect the translation or sentiment analysis accuracy.

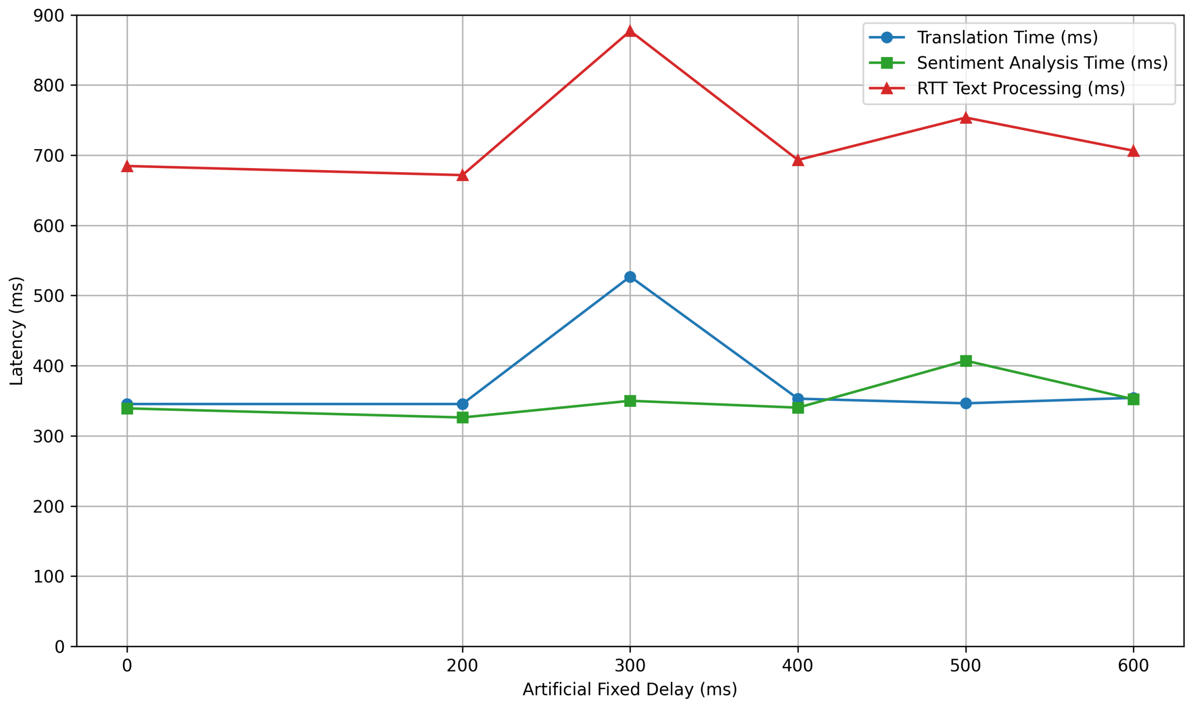


Figure 15: Impact of Artificial Fixed Delay on Text Processing Latency

Scenario C: Impact of Packet Loss on Real-Time Text Communication

The results presented in this section analyze how packet loss affects the accuracy and the emotional coherence of real-time text communication over a best-effort network.

Unlike previous latency-focused scenarios (Scenario A and Scenario B), here the main focus is the semantic and the sentiment analysis integrity of the transmitted message rather than transmission speed.

Table 6.1 summarizes the detailed results obtained during the experiments, where different levels of random word loss were artificially introduced.

Test	**Words Lost**	**Loss (%)**	**Translated text without packet loss**	**Translated text with packet loss**
1	7	28%	Hi, how are you? Today is a beautiful day. This is a test message that contains several words to simulate normal chat communication	Hi, are you? A nice day. This is a test that contains several words to simulate normal communication via
2	4	16%	Hi, how are you? Today is a beautiful day. This is a test message that contains several words to simulate normal chat communication	Hi, how are you? Today is a nice day. A test that contains several words to simulate normal chat communication
3	2	8%	Hi, how are you? Today is a beautiful day. This is a test message that contains several words to simulate normal chat communication	How are you? Today is a beautiful day. This is a test message that contains several words to simulate normal chat communication
4	2	8%	Hi, how are you? Today is a beautiful day. This is a test message that contains several words to simulate normal chat communication	Hi, how are you? Today is a beautiful day. This is a test message that contains words to simulate normal communication via
5	1	4%	Hi, how are you? Today is a beautiful day. This is a test message that contains several words to simulate normal chat communication	Hi, how are you? Today is a day. This is a test message that contains several words to simulate normal chat communication
6	2	8%	Hi, how are you? Today is a beautiful day. This is a test message that contains several words to simulate normal chat communication	Hi, how is today a nice day. This is a test message that contains several words to simulate a normal chat message
7	6	24%	Hi, how are you? Today is a beautiful day. This is a test message that contains several words to simulate normal chat communication	Hi, how are you? Today is a beautiful day. This is a test that contains several words to simulate a
8	6	24%	Hi, how are you? Today is a beautiful day. This is a test message that contains several words to simulate normal chat communication	Hi, how are you? Today is a beautiful day. This is a test message, several words for chat communication.

Table 6.1: Scenario C – Impact of Packet Loss on Text Communication

6.3.1 Critical Observations and Best Practices

- Even low packet loss rates (4%–8%) immediately degraded the transmitted sentences, either by removing essential words or modifying their meaning.
- Unlike audio streaming, where minor packet losses may still allow partial reconstruction, in real-time text transmission the loss of individual words leads to semantic distortions.
- Sentiment analysis accuracy was also affected: the absence of emotionally relevant words (such as adjectives like *beautiful*, *happy* or *sad*) sometimes caused incorrect sentiment detection.
- These findings confirm that text communications over best-effort channels are extremely sensitive to packet loss: even minimal degradation can compromise both the understanding and the sentiment interpretation of the message.

Chapter 7

Conclusions

This work provided a comprehensive exploration of the challenges posed by best-effort network degradations on real-time audio and text communications in a WebRTC-based environment integrated with cloud-based machine learning services such as AWS Transcribe, Translate and Comprehend.

Impact on Audio Communication

- Increased jitter and congestion led to higher latencies in audio transmission and processing, often causing significant delays in transcription and translation. In severe cases, session instabilities occurred, with partial or even complete failures in transcription and sentiment analysis.
- Small chunk sizes (e.g., 50 ms) minimized latency and improved responsiveness under ideal network conditions. However, they showed sensitivity under network degradation such as high jitter and packet loss, leading to increased instability and processing errors.
- Packet loss beyond 10–15% levels dramatically affected the audio communication chain, making real-time transcription unreliable and disrupting downstream services such as translation and sentiment analysis.

Impact on Text Communication

- Artificial network delays (fixed additional latencies) increased the round-trip time for text processing but did not significantly affect the accuracy of translations or sentiment analysis, even up to 600 ms.
- In contrast, even relatively low rates of word loss (4%–8%) immediately compromised the semantic integrity of transmitted text. Essential parts of the message were lost, leading to distortions in meaning and inaccuracies in sentiment detection.
- Text-based communications demonstrated critical vulnerability to packet loss: unlike audio streams, where partial recovery is sometimes possible, the loss of individual words severely affected both comprehension and emotional interpretation.

These results highlight the limitations of best-effort networks for supporting high-quality, real-time communication services, particularly when complex processing chains such as real-time transcription, translation and sentiment analysis are involved.

Advantages of IMS Data Channel Architectures

Compared to best-effort Internet connections, where transmission quality cannot be guaranteed, the IMS Data Channel framework defined by 3GPP introduces significant enhancements to address these limitations:

- **Guaranteed Quality of Service (QoS)**: IMS Data Channels leverage standardized QoS frameworks in both 4G and 5G to ensure predictable performance. In LTE networks, the QoS Class Identifier (QCI) mechanism assigns specific traffic classes with fixed characteristics. For example, QCI 1 is used for conversational voice with a packet delay budget of 100 ms and packet loss rate of 10^{-2}. In 5G, this is replaced by the more flexible 5G QoS Identifier (5QI). For instance, 5QI 1 corresponds to conversational voice (same as QCI 1), and 5QI 2 is assigned to conversational video with a delay budget of 150 ms and a tighter packet loss constraint (10^{-3}). These frameworks enable IMS Data Channels to operate with consistent latency, jitter and reliability across both technologies.
- **Dedicated Bearers and Network Slicing**: IMS supports the allocation of dedicated network resources (dedicated bearers in LTE and QoS flows in 5G), and enables service-specific segmentation of the network through network slicing. This ensures that communication quality remains stable even under congested or heavily loaded conditions.
- **Session Control and Dynamic Resource Reservation**: Using the SIP protocol and IMS Application Servers, IMS enables advanced session control and dynamic reservation of network resources. This ensures that applications receive guaranteed bandwidth and latency profiles before transmission begins, reducing the risk of mid-session degradation.

By incorporating these features, IMS Data Channels significantly mitigate or even eliminate the challenges typically observed in best-effort networks, ensuring faster, more reliable and higher-quality communication experiences.

Beyond improvements in transmission reliability, the location where machine learning processing takes place also plays a crucial role in enhancing user experience.

In the simulation developed, services such as real-time transcription, translation and sentiment analysis were executed remotely via cloud-based platforms. However, in use cases demanding ultra-low latency, the use of distant cloud servers introduces additional transmission delays and increases dependence on external network conditions.

To further optimize performance, shifting critical AI processing closer to end-users using technologies like 5G Multi-Access Edge Computing (MEC) would provide substantial benefits:

- **Latency Reduction**: Bringing processing closer to the network edge significantly shortens transmission paths, reducing end-to-end delays and enabling near real-time responses.

- **Reliability Enhancement**: Localized processing reduces dependency on wide-area Internet connectivity, improving service stability.
- **Bandwidth Optimization**: Processing part of the data locally reduces the amount of information that needs to travel across the network, preserving bandwidth for critical services.
- **Enhanced User Experience**: Faster and more stable services improve the experience for users, especially for applications that involve real-time interaction, emotional analysis, and other AI-based functions.

Thus, IMS Data Channels, complemented by edge-based machine learning integration, represent a critical foundation for delivering the next generation of intelligent and real-time multimedia services.

Final Remarks

The results of this work confirm that best-effort networks are not suitable for supporting advanced, real-time communication services using data channels, especially when artificial intelligence is part of the service.

By using IMS Data Channel architectures, which guarantee network quality through mechanisms like QCI in 4G and 5QI in 5G, and by moving machine learning functions closer to the users with edge computing, it becomes possible to offer high-quality real-time communication services.

The combination of IMS Data Channels and AI processing at MEC is a key factor for enabling the next generation of intelligent and real-time mobile services providing reliable, fast and emotionally aware experiences for users.

Chapter A

Appendix: Source Code

This appendix includes the source code developed and used during the simulation experiments described in this thesis.

Real-Time Transcription Server

```python
#!/usr/bin/env python3
import os
import queue
import asyncio
import time
import boto3
import sys
import numpy as np
from flask import Flask, request, jsonify
from flask_cors import CORS
from flask_socketio import SocketIO
from amazon_transcribe.client import TranscribeStreamingClient
from amazon_transcribe.handlers import TranscriptResultStreamHandler
from amazon_transcribe.model import TranscriptEvent

# AWS Region and Clients
AWS_REGION = "eu-central-1"
transcribe_client = TranscribeStreamingClient(region=AWS_REGION)
translate_client = boto3.client(service_name="translate", region_name=
    AWS_REGION, use_ssl=True)
comprehend_client = boto3.client("comprehend", region_name=AWS_REGION)

# Flask WebSocket Setup
app = Flask(__name__)
CORS(app)
socketio = SocketIO(app, cors_allowed_origins="*", async_mode="threading")

# Audio Processing Variables
audio_queue = queue.Queue()
transcription_started = False
sample_rate = 16000
audio_buffer = b""
transcription_text = ""
final_transcription_received = False
chunk_size_ms = 50  # Recommended chunk size for real-time streaming
```

```python
chunk_size_bytes = int((chunk_size_ms / 1000) * sample_rate * 2)

last_full_sentence = ""  # Store the last full sentence before sending to translation

## AUDIO: Transcription and Sentiment Analysis
class MyEventHandler(TranscriptResultStreamHandler):
    async def handle_transcript_event(self, transcript_event: TranscriptEvent):
        "Handles incoming transcription results and immediately translates partial texts."
        global transcription_text, final_transcription_received, last_full_sentence
        try:
            results = transcript_event.transcript.results
            for result in results:
                text = result.alternatives[0].transcript.strip()

                if text:
                    if result.is_partial:
                        translated_partial = translate_text(text, "it", "en")
                        socketio.emit("partial_transcription", {"original": text, "translated": translated_partial})
                    else:
                        transcription_text += " " + text
                        last_full_sentence = text
                        final_transcription_received = True

                        translated_full = translate_text(last_full_sentence, "it", "en")
                        socketio.emit("final_transcription", {"original": last_full_sentence, "translated": translated_full})

                        # **AUDIO Sentiment Analysis**
                        sentiment, _ = analyze_sentiment(last_full_sentence, "it")
                        sentiment_emoji = sentiment_to_emoji(sentiment)
                        socketio.emit("sentiment_result", {"sentiment": sentiment_emoji, "client": "B"})

        except Exception as e:
            print(f" Error processing transcript event: {e}")

async def transcribe_stream():
    "Handles streaming audio to Amazon Transcribe."
    try:
        stream = await transcribe_client.start_stream_transcription(
            language_code="it-IT",
            media_sample_rate_hz=sample_rate,
            media_encoding="pcm",
        )
        handler = MyEventHandler(stream.output_stream)
        asyncio.create_task(handler.handle_events())

        while True:
            chunk = await asyncio.get_event_loop().run_in_executor(None, audio_queue.get)
            if chunk is None:
                await stream.input_stream.end_stream()
                break
            await stream.input_stream.send_audio_event(audio_chunk=chunk)

    except Exception as e:
        print(f" Transcription stream error: {e}")

def background_transcription():
    "Starts transcription in a background task."
    loop = asyncio.new_event_loop()
    asyncio.set_event_loop(loop)
    loop.run_until_complete(transcribe_stream())

@socketio.on("audio")
def receive_audio(data):
    """Receives audio data and queues it for transcription."""
    global transcription_started, audio_buffer
    try:
        audio_buffer += data
        if not transcription_started:
            transcription_started = True
            socketio.start_background_task(target=background_transcription)

        while len(audio_buffer) >= chunk_size_bytes:
            audio_queue.put(audio_buffer[:chunk_size_bytes])
            audio_buffer = audio_buffer[chunk_size_bytes:]

    except Exception as e:
        print(f"Error processing audio: {e}")

@socketio.on("stop_audio")
def stop_audio():
    "Handles stopping the audio stream and finalizing transcription."
    global transcription_started, final_transcription_received

    if audio_buffer:
        audio_queue.put(audio_buffer)

    silence_chunk = np.zeros(int(sample_rate * 2 * 0.2), dtype=np.int16).tobytes()
    for _ in range(10):
        audio_queue.put(silence_chunk)
        time.sleep(0.2)

    audio_queue.put(None)

    wait_time = 0
    while not final_transcription_received and wait_time < 15:
        time.sleep(1)
        wait_time += 1

    print("Transcription session closed successfully.")
    transcription_started = False

## TEXT: Translation and Sentiment Analysis

@app.route('/analyze', methods=['POST'])
def analyze_message():
    "Handles text translation and sentiment analysis."
    data = request.json
    text = data.get('text')
    source_lang = data.get('source_language', 'auto')
    target_lang = data.get('target_language', 'en')

    if not text:
        return jsonify({"error": "Il messaggio  vuoto"}), 400

    translated_text = translate_text(text, source_lang, target_lang)

    if not translated_text:
        return jsonify({"error": "Errore durante la traduzione"}), 500

    sentiment, confidence_scores = analyze_sentiment(translated_text,
   target_lang)

    # Text sentiment analysis via WebSocket
    socketio.emit("text_sentiment_result", {"sentiment": sentiment_to_emoji(
   sentiment)})

    return jsonify({
        "translation": translated_text,
        "sentiment": sentiment,
        "confidence_scores": confidence_scores
    })


## **Support Functions**
def translate_text(text, source_lang, target_lang):
    "Translates text from source_lang to target_lang using AWS Translate."
    try:
        response = translate_client.translate_text(
            Text=text,
            SourceLanguageCode=source_lang,
            TargetLanguageCode=target_lang
        )
        return response["TranslatedText"]
```

```python
    except Exception as e:
        print(f" Error in translation: {e}")
        return "[Translation Failed]"

def analyze_sentiment(text, language):
    "Analyzes the sentiment of a text using AWS Comprehend."
    try:
        response = comprehend_client.detect_sentiment(
            Text=text,
            LanguageCode=language
        )
        return response['Sentiment'], response['SentimentScore']
    except Exception as e:
        print(f" Error in sentiment analysis: {e}")
        return "UNKNOWN", {}

def sentiment_to_emoji(sentiment):
    "Converts sentiment text to an emoji."
    return {"POSITIVE": "", "NEGATIVE": "", "NEUTRAL": "", "MIXED": ""}.get(sentiment, "")

## Server Launch
if __name__ == "__main__":
    print("\n Starting Flask WebSocket & API server on http://0.0.0.0:5000")
    socketio.run(app, host="0.0.0.0", port=5000, allow_unsafe_werkzeug=True)
```

Listing A.1: full_videocall_rt_comprehend_final.py – Flask server with real-time streaming, translation and sentiment analysis via AWS

Custom WebRTC Client Interface

```html
<!DOCTYPE html>
<html xmlns="http://www.w3.org/1999/xhtml">
<head>
<meta charset="utf-8">
<meta name="viewport" content="width=device-width, initial-scale=1.0"/>
<meta http-equiv="Content-Type" content="text/html; charset=UTF-8" />
<title>Janus WebRTC Server (multistream): Video Call Demo</title>
<script type="text/javascript" src="https://cdnjs.cloudflare.com/ajax/libs/webrtc-adapter/8.2.2/adapter.min.js" ></script>
<script type="text/javascript" src="https://cdnjs.cloudflare.com/ajax/libs/jquery/1.9.1/jquery.min.js" ></script>
<script type="text/javascript" src="https://cdnjs.cloudflare.com/ajax/libs/jquery.blockUI/2.70/jquery.blockUI.min.js" ></script>
<script type="text/javascript" src="https://cdnjs.cloudflare.com/ajax/libs/twitter-bootstrap/3.4.1/js/bootstrap.min.js"></script>
<script type="text/javascript" src="https://cdnjs.cloudflare.com/ajax/libs/bootbox.js/5.4.0/bootbox.min.js"></script>
<script type="text/javascript" src="https://cdnjs.cloudflare.com/ajax/libs/spin.js/2.3.2/spin.min.js"></script>
```

```
<script type="text/javascript" src="https://cdnjs.cloudflare.com/ajax/libs/
    toastr.js/2.1.4/toastr.min.js"></script>
<script src="https://cdnjs.cloudflare.com/ajax/libs/socket.io/4.7.2/socket.io.
    js"></script>
<script type="text/javascript" src="settings.js" ></script>
<script type="text/javascript" src="janus.js" ></script>
<script type="text/javascript" src="videocalltest.js"></script>
<script>
    $(function() {
        $(".navbar-static-top").load("navbar.html", function() {
            $(".navbar-static-top li.dropdown").addClass("active");
            $(".navbar-static-top a[href='videocalltest.html']").parent().
    addClass("active");
        });
        $(".footer").load("footer.html");
    });
</script>
<link rel="stylesheet" href="https://cdnjs.cloudflare.com/ajax/libs/bootswatch
    /3.4.0/cerulean/bootstrap.min.css" type="text/css"/>
<link rel="stylesheet" href="css/demo.css" type="text/css"/>
<link rel="stylesheet" href="https://cdnjs.cloudflare.com/ajax/libs/font-
    awesome/4.7.0/css/font-awesome.min.css" type="text/css"/>
<link rel="stylesheet" href="https://cdnjs.cloudflare.com/ajax/libs/toastr.js
    /2.1.4/toastr.min.css"/>
</head>
<body>

<a href="https://github.com/meetecho/janus-gateway"><img style="position:
    absolute; top: 0; left: 0; border: 0; z-index: 1001;" src="
    forkme_left_darkblue_121621.png" alt="Fork me on GitHub"></a>

<nav class="navbar navbar-default navbar-static-top">
</nav>

<div class="container">
    <div class="row">
        <div class="col-md-12">
            <div class="page-header">
                <h1>Plugin Demo: Video Call
                    <button class="btn btn-default" autocomplete="off" id="
    start">Start</button>
                </h1>
            </div>
            <div class="container" id="details">
                <div class="row">
                    <div class="col-md-12">
                        <h3>Demo details</h3>
                        <p>This Video Call demo is basically an example of how
    you can achieve a
                        scenario like the famous AppRTC demo but with media
    flowing through Janus. It
```

```
                    basically is an extension to the Echo Test demo, where in this case the media
                    packets and statistics are forwarded between the two involved peers.</p>
                    <p>Using the demo is simple. Just choose a simple username to register
                    at the plugin, and then either call another user (provided you know
                    which username was picked) or share your username with a friend and
                    wait for a call. At that point, you'll be in a video call with the
                    remote peer, and you'll have the same controls the Echo Test demo
                    provides to try and control the media: that is, a button to mute/unmute
                    your audio and video, and a knob to try and limit your bandwidth. If
                    the browser supports it, you'll also get a view of the bandwidth
                    currently used by your peer for the video stream.</p>
                    <p>If you're interested in testing how simulcasting can be used within
                    the context of this sample videocall application, just pass the
                    <code>?simulcast=true</code> query string to the url of this page and
                    reload it. If you're using a browser that does support simulcasting
                    (Chrome or Firefox) and the call will end up using VP8, you'll
                    send multiple qualities of the video you're capturing. Notice that
                    simulcasting will only occur if the browser thinks there is enough
                    bandwidth, so you'll have to play with the Bandwidth selector to
                    increase it. New buttons to play with the feature will automatically
                    appear for your peer; at the same time, if your peer enabled simulcasting
                    new buttons will appear for you when watching the remote stream. Notice that
                    no simulcast support is needed for watching, only for publishing.</p>
                    <p>A very simple chat based on Data Channels is available as well:
                    just use the text area under your local video to send messages
                    to your peer. Incoming messages will be displayed below the
                    remote video instead.</p>
                    <p>Press the <code>Start</code> button above to launch the demo.</p>
                </div>
            </div>
        </div>
        <div class="container hide" id="videocall">
            <div class="row">
                <div class="col-md-12">
                    <div class="col-md-6 container hide" id="login">
                        <div class="input-group margin-bottom-sm">
                            <span class="input-group-addon"><i class="fa fa-user fa-fw"></i></span>
                            <input class="form-control" type="text" placeholder="Choose a username" autocomplete="off" id="username" onkeypress="return checkEnter(this, event);" />
                        </div>
                        <button class="btn btn-success margin-bottom-sm" autocomplete="off" id="register">Register</button> <span class="hide label label-info" id="youok"></span>
                    </div>
                    <div class="col-md-6 container hide" id="phone">
                        <div class="input-group margin-bottom-sm">
                            <span class="input-group-addon"><i class="fa fa-phone fa-fw"></i></span>
                            <input class="form-control" type="text" placeholder="Who should we call?" autocomplete="off" id="peer" onkeypress="return checkEnter(this, event);" />
                        </div>
                        <button class="btn btn-success margin-bottom-sm" autocomplete="off" id="call">Call</button>
                    </div>
                </div>
            <div/>
            <div id="videos" class="hide">
                <div class="col-md-6">
                    <div class="panel panel-default">
                        <div class="panel-heading">
                            <h3 class="panel-title">Local Stream
                                <div class="btn-group btn-group-xs pull-right hide">
                                    <button class="btn btn-danger" autocomplete="off" id="toggleaudio">Disable audio</button>
                                    <button class="btn btn-danger" autocomplete="off" id="togglevideo">Disable video</button>
                                    <div class="btn-group btn-group-xs">
                                        <button autocomplete="off" id="bitrateset" class="btn btn-primary dropdown-toggle" data-toggle="dropdown">
                                            Bandwidth<span class="caret"></span>
                                        </button>
```

```html
                                        <ul id="bitrate" class="dropdown-menu" role="menu">
                                            <li><a href="#" id="0">No limit</a></li>
                                            <li><a href="#" id="128">Cap to 128kbit</a></li>
                                            <li><a href="#" id="256">Cap to 256kbit</a></li>
                                            <li><a href="#" id="512">Cap to 512kbit</a></li>
                                            <li><a href="#" id="1024">Cap to 1mbit</a></li>
                                            <li><a href="#" id="1500">Cap to 1.5mbit</a></li>
                                            <li><a href="#" id="2000">Cap to 2mbit</a></li>
                                        </ul>
                                    </div>
                                </div>
                            </h3>
                        </div>
                        <div class="panel-body" id="videoleft"></div>
                    </div>
                    <!--  START/STOP REGISTRATION BUTTONS -->
                    <div style="margin-top: 10px;margin-bottom: 20px;">
                            <button id="startRecording" class="btn btn-success"> Start Recording</button>
                            <button id="stopRecording" class="btn btn-danger hide"> Stop Recording</button>
                    </div>

                    <!--  TRANSCRIPTION BOX -->
                    <div id="transcription-container" style="border: 1px solid #4CAF50; padding: 10px; height: 300px; overflow-y: scroll; background-color: #f9fff9; margin-bottom: 20px;">
                            <h4> Transcription</h4>
                            <pre id="originalText" style="margin: 0; font-size: 14px;"></pre>
                    </div>



                    <div class="input-group margin-bottom-sm">
                        <span class="input-group-addon"><i class="fa fa-cloud-upload fa-fw"></i></span>
                        <input class="form-control" type="text" placeholder="Write a DataChannel message to your peer" autocomplete="off" id="datasend" onkeypress="return checkEnter(this, event);" disabled />
                    </div>


```

```
          <div style="display: flex; justify-content: space-between; gap: 10px; margin-bottom: 20px;">

              <!--  AUDIO SENTIMENT ANALYSIS -->
              <div id="sentiment-container" style="flex: 1; border: 2px solid #FFD700; padding: 15px; height: 150px; text-align: center; background-color: #FFFACD;">
                <h4> Audio Sentiment Analysis</h4>
                <div id="sentimentEmoji" style="font-size: 70px;"></div>
              </div>

              <!--  TEXT SENTIMENT ANALYSIS -->
            <div id="text-sentiment-container" style="flex: 1; border: 2px solid #FFD700; padding: 15px; height: 150px; text-align: center; background-color: #FFFACD;">
                <h4> Text Sentiment Analysis</h4>
                <div id="textSentimentEmoji" style="font-size: 70px;"></div>
              </div>

          </div>



        </div>
        <div class="col-md-6">
          <div class="panel panel-default">
            <div class="panel-heading">
              <h3 class="panel-title">Remote Stream <span class="label label-info hide" id="callee"></span> <span class="label label-primary hide" id="curres"></span> <span class="label label-info hide" id="curbitrate"></span></h3>
            </div>
            <div class="panel-body" id="videoright"></div>
          </div>

          <!--  TRANSLATION BOX (BELOW REMOTE STREAM) -->

          <div id="translation-container" style="border: 1px solid #2E86C1; padding: 10px; height: 300px; overflow-y: scroll; background-color: #e3f2fd; margin-bottom: 20px; margin-top: 80px;">
              <h4> Translation</h4>
              <pre id="translatedText" style="margin: 0; font-size: 14px; color: #0D47A1;"></pre>
          </div>


          <div class="input-group margin-bottom-sm">
              <span class="input-group-addon"><i class="fa fa-cloud-download fa-fw"></i></span>
```

```
                        </div>

                        <!--  CHAT BOX BELOW REMOTE STREAM -->
                        <div id="chat-container" style="border: 1px solid #ccc;
    padding: 10px; height: 200px; overflow-y: scroll;">
                                <pre id="datarecv" style="margin: 0;"></pre>
                        </div>





                    </div>
                </div>
            </div>
        </div>
    </div>

    <hr>
    <div class="footer">
    </div>
</div>

</body>
</html>
```

Listing A.2: videocalltest.html – Customized WebRTC interface with transcription and translation panels

JavaScript Custom Audio Processing and WebSocket Integration

The following Javascript contains the main functions that were added to the default Janus demo in order to support real-time audio streaming, audio worklet processing and machine learning integration via WebSocket.

```


function doCall() {
    // Call someone
    $('#peer').attr('disabled', true);
    $('#call').attr('disabled', true).unbind('click');
    let username = $('#peer').val();
    if(username === "") {
        bootbox.alert("Insert a username to call (e.g., pluto)");
        $('#peer').removeAttr('disabled');
        $('#call').removeAttr('disabled').click(doCall);
        return;
    }
    if(/[^a-zA-Z0-9]/.test(username)) {
        bootbox.alert('Input is not alphanumeric');
        $('#peer').removeAttr('disabled').val("");
        $('#call').removeAttr('disabled').click(doCall);
        return;
    }
    // Call this user
    videocall.createOffer(
        {
            // We want bidirectional audio and video, plus data channels
            tracks: [
                { type: 'audio', capture: true, recv: true },
                { type: 'video', capture: true, recv: true, simulcast:
    doSimulcast },
                { type: 'data' },
            ],
            success: function(jsep) {
                Janus.debug("Got SDP!", jsep);
                let body = { request: "call", username: $('#peer').val() };
                videocall.send({ message: body, jsep: jsep });
            },
            error: function(error) {
                Janus.error("WebRTC error...", error);
                bootbox.alert("WebRTC error... " + error.message);
            }
        });
}

function doHangup() {
    // Hangup a call
    $('#call').attr('disabled', true).unbind('click');
    let hangup = { request: "hangup" };
    videocall.send({ message: hangup });
    videocall.hangup();
    yourusername = null;
}

// Global variables for audio management
let audioContext;
let source;
let workletNode;
let stream;
let socket = io("http://127.0.0.1:5000"); // WebSocket initialized once

socket.on("connect", function() {
    console.log("WebSocket connected successfully!");
});

let recording = false;
window.lastBuffer = null; // To avoid sending duplicate audio packets

// Function to start recording
```

```
async function startRecording() {
    if (recording) {
        console.log(" Recording is already active!");
        return;
    }

    console.log("Starting recording, WebSocket is ready...");
    recording = true;

    try {
        const audioStream = await navigator.mediaDevices.getUserMedia({
            audio: {
                sampleRate: 16000,
                channelCount: 1,
                echoCancellation: true,
                noiseSuppression: true,
                autoGainControl: true
            }
        });

        console.log("Microphone capture started...");
        audioContext = new AudioContext({ sampleRate: 16000 });

        if (audioContext.state === "suspended") {
            await audioContext.resume();
            console.log("AudioContext resumed.");
        }

        source = audioContext.createMediaStreamSource(audioStream);
        stream = audioStream;

        // Load the AudioWorklet module
        try {
            await audioContext.audioWorklet.addModule('processor.js');
            console.log("AudioWorklet loaded successfully!");
        } catch (e) {
            console.error("Error loading AudioWorklet:", e);
            stopRecording();
            return;
        }

        await new Promise(resolve => setTimeout(resolve, 100)); // Ensure readiness

        // Create and connect the AudioWorkletNode
        try {
            workletNode = new AudioWorkletNode(audioContext, 'my-audio-processor');
            console.log("AudioWorkletNode created successfully.");
        } catch (e) {
            console.error("Error creating AudioWorkletNode:", e);
            stopRecording();
            return;
        }

        source.connect(workletNode);

        // Handle audio chunks from AudioWorklet
        workletNode.port.onmessage = function(event) {
            let buffer = new Int16Array(event.data);

            let isSilent = buffer.every(sample => Math.abs(sample) < 100);
            if (isSilent) {
                console.log("Ignoring silent audio.");
                return;
            }

            if (window.lastBuffer && buffer.length === window.lastBuffer.length
    &&
                buffer.every((val, i) => val === window.lastBuffer[i])) {
                console.log("Duplicate audio packet ignored.");
                return;
            }

            window.lastBuffer = buffer.slice();

            if (socket) {
                socket.emit("audio", buffer.buffer);
            }
        };

        // Show the stop button once Worklet is ready
        $('#stopRecording').removeClass('hide').show();
        console.log("Audio recording is active.");

    } catch (err) {
        console.error("Error accessing microphone:", err);
        stopRecording();
    }
}

// Function to stop recording
async function stopRecording() {
    if (!recording) return;

    console.log("Stopping audio capture...");
    recording = false;

    try {
        if (workletNode) workletNode.disconnect();
        if (source) source.disconnect();
        if (stream) stream.getTracks().forEach(track => track.stop());
        if (audioContext) await audioContext.close();

        if (socket) {
            console.log("Sending stop_audio to Flask");
            socket.emit("stop_audio");
        }

    } catch (err) {
        console.error("Error during recording shutdown:", err);
    }

    setTimeout(() => {
        $('#stopRecording').addClass('hide');
        console.log("Recording stopped.");
    }, 500);
}

// Assign event listeners to HTML buttons
$(document).ready(function() {
    $('#startRecording').click(startRecording);
    $('#stopRecording').click(stopRecording);
});

// WebSocket: handle partial transcription updates
socket.on("partial_transcription", function(data) {
    let originalTextDiv = document.getElementById("originalText");
    let translatedTextDiv = document.getElementById("translatedText");

    if (data.original.trim() !== "") {
        originalTextDiv.innerText = data.original;
    }
    if (data.translated.trim() !== "") {
        translatedTextDiv.innerText = data.translated;
    }

    originalTextDiv.scrollTop = originalTextDiv.scrollHeight;
    translatedTextDiv.scrollTop = translatedTextDiv.scrollHeight;
});

// WebSocket: handle final transcription and translation
socket.on("final_transcription", function(data) {
    let originalTextDiv = document.getElementById("originalText");
    let translatedTextDiv = document.getElementById("translatedText");

    if (originalTextDiv.innerText !== data.original) {
        originalTextDiv.innerText = data.original;
    }
    if (translatedTextDiv.innerText !== data.translated) {
        translatedTextDiv.innerText = data.translated;
    }

    originalTextDiv.scrollTop = originalTextDiv.scrollHeight;
    translatedTextDiv.scrollTop = translatedTextDiv.scrollHeight;
});

// WebSocket: handle audio sentiment result (only for client B)
socket.on("sentiment_result", function(data) {
    if (data.client === "B") {
        let sentimentDiv = document.getElementById("sentimentEmoji");
        sentimentDiv.innerHTML = `<h1 style="font-size: 80px;">${data.sentiment
    }</h1>`;
    }
});

// WebSocket: handle sentiment result for text messages
socket.on("text_sentiment_result", function(data) {
    console.log("Text Sentiment received:", data.sentiment);
    let textSentimentDiv = document.getElementById("textSentimentEmoji");
    textSentimentDiv.innerHTML = `<h1 style="font-size: 80px;">${data.sentiment
    }</h1>`;
});

// Function to send a message and process translation and sentiment
function sendData() {
    let dataInput = $('#datasend');
    let data = dataInput.val().trim();

    if (data === "") {
        bootbox.alert('Please enter a message to send via DataChannel.');
        return;
    }

    let currentTime = new Date().toLocaleTimeString();
    let chatBox = $('#datarecv');
    let localUsername = myusername;

    // Step 1: Show original message in chat
    chatBox.append(`<div>[${currentTime}] <b>You:</b> ${data}</div>`);
    chatBox.parent().scrollTop(chatBox.parent()[0].scrollHeight);

    // Clear input field using multiple methods for reliability
    dataInput.val('');
    document.getElementById("datasend").value = '';
    dataInput.trigger("input");
    setTimeout(() => {
        dataInput.val('');
        document.getElementById("datasend").value = '';
    }, 100);

    // Step 2: Send text to Flask for translation and sentiment analysis
    let sourceLang = (localUsername === "A") ? 'it' : 'en';
    let targetLang = (localUsername === "A") ? 'en' : 'it';

    $.ajax({
        url: 'http://127.0.0.1:5000/analyze',
        method: 'POST',
        contentType: 'application/json',
        data: JSON.stringify({
            text: data,
            source_language: sourceLang,
            target_language: targetLang
        }),
        success: function(response) {
            let translatedText = response.translation || 'Translation
    unavailable';
            let sentiment = response.sentiment || '';

            // Send sentiment to WebSocket for UI update
            socket.emit("text_sentiment_result", { sentiment: sentiment });

            // Step 3: Send translated message to peer via DataChannel
            videocall.data({
                text: JSON.stringify({
                    sender: localUsername,
                    text: translatedText
                }),
                error: function(reason) {
                    console.error('Error during message send:', reason);
                    bootbox.alert('Error during message send.');
                },
                success: function() {
                    console.log('Message sent successfully.');
                }
            });
        },
        error: function() {
            console.error('Error communicating with Flask.');
            bootbox.alert('Error communicating with Flask.');
        }
    });
}
```

Listing A.3: videocalltest.js - Custom AudioWorklet integration and WebSocket logic

Bibliography

[1] 3GPP TS 23.228, *IP Multimedia Subsystem (IMS); Stage 2*, Release 18.

[2] 3GPP TS 26.114, *IP Multimedia Subsystem (IMS); Multimedia Telephony; Media Handling and Interaction*, Release 18.

[3] 3GPP TS 26.264, *IP Multimedia Subsystem (IMS); Audio-Visual Profile for Augmented Reality*, Release 18.

[4] IETF RFC 8831, *WebRTC Data Channels*, January 2021.

[5] IETF RFC 5688, *Session Initiation Protocol (SIP) Media Feature Tag for MIME Application Subtypes*, November 2009.

[6] IETF RFC 3264, *An Offer/Answer Model with the Session Description Protocol (SDP)*, June 2002.

[7] GSMA NG.134 v3.0, *IMS Data Channel Requirements and Architecture*, 2024.

[8] GSMA NG.129 v1.0, *IMS Data Channel Ecosystem and Use Cases*, 2022.

[9] GSMA Foundry, *Delivering Real-Time Translation*, Case Study, May 2023.

[10] GSMA Foundry, *5G New Calling: Revolutionising the Communication Services Landscape*, September 2023.

[11] ETSI TS 24.186, *IP Multimedia Subsystem (IMS); Stage 3 Specification for IMS Data Channel Services*, Release 17.

[12] Amazon Web Services, *Amazon Transcribe Developer Guide*, 2024.

[13] Amazon Web Services, *Amazon Comprehend Developer Guide*, 2024.

[14] Amazon Web Services, *Amazon Translate Developer Guide*, 2024.